Toward Automated Network Management and Operations

by

Xu Chen

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2010

To my family.

ACKNOWLEDGEMENTS

First and foremost, I'd like to thank my advisor Professor Morley Mao. I cannot ask for a better advisor. Her insightful comments and the ability to connect the dots (and memorizing all the dots in the first place) never cease to amaze and educate me. Being the unanimous all-time winner of the fierce competition on “who leaves the CSE building the latest every day”, she is the hardest-working person that I know. She devotes so much of her time and energy to her students, and always holds our best interest in mind. Morley has been my role model and always will be.

Internships are very essential pieces of my PhD. I'd like to sincerely thank Dr. Jacobus Van der Merwe for his wise and sincere suggestions, and his enthusiastic attitude towards building real systems. He picked me up when I was an inexperienced second-year grad student, and never complained about my silliness throughout the years. The collaboration with him led to most of the work in this thesis. I am forever grateful for his guidance. I'd also like to thank Dr. Ming Zhang for hosting me in MSR, during which he taught me so much about designing, building and evaluating network systems. Dr. Yun Mao helped tremendously on the COOLAID project, which was only possible because of his masterful Python hacking skills.

I am thankful to Professor Jason Flinn, Robert Dick, Scott Malhke and Michael Bailey for serving on my thesis committee. Their thoughtful advice and suggestions helped improve this dissertation. Jason is a great lecturer, and I am glad that I took his Advanced Operating Systems course, which changed significantly how I view system research. Robert is a great thinker (and fast talker), and the discussion with him always intrigues me. Scott is the coolest professor that I know, and I can only wish to handle everything at ease like him. Michael has always been helpful in guiding my work. His experience has great influence on many of my projects.

I'd like to thank all my RobustNet group members. In particular, Ying Zhang has always been a great friend, who kindly offers me help and suggestions in both work and life. I am grateful to Joseph Xu, Eugene Chai, Bin Liu, Haixiong Wang, Ye Du and many others, who spent so much wonderful time with me and always were there when I needed them the most. Hats off to my fellow interns in MSR-112/3001 and FP-103: you made those summers fun.

Finally, I am forever grateful to my parents, who brought me into this world, gave me the best possible education, and are always there for me. My wife Wei has tolerated me so much and given me immense support and joy, making this journey possible and colorful. This dissertation, and everything else, is dedicated to them.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER I

Introduction

Network management plays a fundamental role in the operation and well-being of today's networks. Despite this importance, network management operations still largely rely on fairly rudimentary technologies: network changes are mostly manually performed via archaic, low-level command line interfaces (CLIs). As computer networks become larger-scale, more complex and more dynamic, human operators are increasingly short-handed. To make matters worse, network operations are usually driven by tight deadlines, making human errors more likely. As a result, network misconfigurations are common, causing profound negative impact to the global infrastructure and high-value network services. In 1997, a misconfiguration by operators of Florida Internet Exchange (AS 7007) resulted in an extended period of disruption throughout the Internet [2]. A more recent example is a BGP misconfiguration by an ISP in Pakistan that blocked access to YouTube globally for two hours in 2008 [6].

The earliest attempts to better handle network management operations originated from network operators themselves, by composing and executing various home-brew scripts. Over time, these scripts become increasingly sophisticated and widely used with every operator maintaining his/her own "arsenal". Unfortunately, these scripts are ad-hoc and task-centric, and thus require significant manual coordination and domain expertise to accomplish more complicated operations and prevent network-wide side-effects. Furthermore their development and refinement require significant expertise and involve much trial-and-error. From the research community, a variety of automation systems are proposed. Notably, a template-driven approach is used to automatically generate router configurations [53]. However, such systems do not handle the ordering and dependencies in network operations, where configuration changes must be guided by dynamic and network-wide conditions. Generating the templates is also non-trivial to begin with. To address the overwhelming complexities and increasingly higher demand for reliable networks, better automation systems are required to assist network management operations by reducing operator workload and preventing undesired network misconfigurations.

To improve the limited automation support in network management operations, we propose two automation systems. The first system allows the composition and execution of automated network operations, and at the same time integrates network-wide checks to prevent misconfigurations. The second system facilitates more automated network configurations and misconfiguration prevention, while minimizing operator workload. We contend that the key to achieving both automated and correct network operations is the use of formal abstractions to capture domain knowledge to guide management operations and integrate network-wide property checks.

The rest of this chapter is organized as follows. We first describe the key challenges in network management in general and performing management operations in particular. We then discuss two important issues in building and evaluating network management systems. The following section then gives a brief overview of our approach to addressing the challenges and achieving the requirements. We conclude this chapter with a discussion over the scope of this dissertation and limitations.

1.1 Challenges in Network Management

This section presents an overview of the challenges involved to better familiarize the readers with the pressing concerns in network management.

1.1.1 Managing modern networks

Network management operations are challenging because modern networks are complex, large-scale and highly dynamic.

Complexity in network management: Modern computer networks are expected to deliver a diverse set of services, some of which have competing goals, on a distributed collection of heterogeneous devices. Best-effort IP transit is the most basic network functionality, while the increasingly popular VPN service has to satisfy stringent service-level agreements when connecting multiple office sites of large enterprises. To realize these services, a service provider must manage a diverse set of hardware devices, including routers, switches, firewalls, *etc.*, which feature different software designs and expose heterogeneous management interfaces. For example, Cisco alone has released more than a dozen major versions of their router firmware, not to mention other major vendors, like Juniper and Avici.

Managing a network is partly about enabling and maintaining the high-level network services and functionalities through the correct low-level configurations to satisfy the service-level agreements with users and customers. Such activities require operators to understand voluminous configuration manuals and apply the knowledge to specific network setups. To fulfill a network change, operators have to translate the high-level goal into low-level implementation in terms of the lines of configurations to modify — a process that is becoming increasingly difficult.

A particular complication stems from the fact that network services and features are commonly dependent on each other, thus enabling a single feature may in fact require the creation and maintenance of several other services [1]. These dependencies are usually verbally described in documentation and impose a steep learning curve. Not surprisingly, questions like "Why is my VPN service not configured correctly?" frequently appear on network management related forums and mailing lists [9].

[1]For example, as we later show in Figure 4.4, on commodity routers (*e.g.,* from Cisco or Juniper), setting up a VPLS (Virtual Private LAN Service, a form of VPN to provide layer-2 connectivity) depends on establishing LSPs (Label-Switching Paths) and BGP signaling, while LSPs in turn depend on an MPLS- and RSVP-enabled network, and BGP signaling further relies on other factors, *e.g.,* a properly configured IGP.

Scale in network management: Modern networks are massive in size. It is common for a service provider network to contain thousands of network devices in its core network alone, with each device requiring hundreds or thousands of lines of configuration. Even though the provided services and design goals of a network are embedded in these configuration files, distilling how a network functions, *i.e.,* building a network-wide view, is very challenging. In particular, understanding each feature requires not only inspecting individual device configurations but also reasoning about the distributed protocol execution logic, *e.g.,* how the routes propagate through OSPF within a network. Furthermore, the protocols and features in a networks are usually dependent on each other, as we explained previously. As networks continue to expand in scale and complexity, the capacity of network operators to maintain such a network-wide view is seriously challenged, *e.g.,* a single line of configuration modification can result in network-wide functionality change. On the other hand, such network-wide understanding is extremely important to correctly performing network operations, *e.g.,* enabling a new service without impacting existing ones. Mistakes in answering the questions like “What services might be impacted if I shut down router A’s loopback interface?” might lead to network-wide outages.

Dynamics in network management: Modern networks are increasingly dynamic. On the one hand, much of the network evolution is driven by growth demands and feature requests, *e.g.,* adding core or edge routers to handle more customers with higher throughput and more diverse services. These device introduction, upgrade and re-purpose activities are commonly performed in today’s large networks, and they need to be handled correctly and efficiently to ensure continuous service delivery. On the other hand, various network events happen spontaneously, *e.g.,* device failures and a variety of network attacks. The ability to mitigate these events while minimizing user impacts is crucial for the success of service provider networks.

However, such network dynamics impose significant challenges for network management. Network operators have to overcome the difficulty in quickly building and indeed rebuilding network-wide views to understand the impact of such events, and navigate through the complexity in network configurations to identify ways to maintain the correct functions of network services.

In particular, each network may feature several high-level properties that should be enforced. For example, all routers must form a BGP full-mesh. Such policies may be violated during these dynamic network events, *e.g.,* adding a new router violates the full-mesh property. Because these properties are usually not explicitly documented, deriving them from the network configurations is difficult and involves non-trivial reverse-engineering. Taking a step further to enforce them is even more difficult, given how quickly the network changes. The emerging trend of network virtualization further raises the bar [98], as the device inventory of a virtualized network can be dynamically allocated and the topology can be easily modified.

These properties lead to two important but sometimes conflicting concerns about *correctness* and *timeliness*. On the one hand, network operations should be correct, meaning that they achieve the desired outcome without causing undesired side-effects, *e.g.,* to perform a router maintenance with minimal impact on traffic. On the other hand, network operations should be performed in a timely fashion to adapt to the dynamic changes. These two requirements are conflicting because of the network operators in the loop. To ensure correctness, it takes human operators a long time to reason about and verify the network-wide properties of complex networks. To ensure timeliness, operators are forced to rush management tasks or rely on primitive automation methods, such as scripts, oftentimes violating network-wide properties due to insufficient, if any, reasoning support. These properties are confirmed by the network operator community [26], calling for automated solutions that are "more reliable, easier to maintain, and easier to scale".

1.1.2 Network misconfigurations

Misconfigurations occur all the time, not only in computer systems [39, 64] and networks [81], but also in other reliable systems [48]. Not surprisingly, the major contributing factor is human operators, who are responsible for 20-70% of system failures [78].

In network management, there are two types of misconfiguration. The first type renders a related service to be completely non-functional, *e.g.,* a network loses the capability to exchange network packets with a particular neighboring network. The second type renders functional but degraded services, *e.g.,* significant traffic rate reduction over a network link. The latter type is usually caused by the inappropriate use of resource, *e.g.,* a sub-optimal routing design may route too much traffic on a single link, thus causing high loss rate. Indeed, these misconfiguration instances are under the category of performance management. We discuss the scope of this thesis in Section 1.4.

A common goal of network operators is to minimize the impact of misconfigurations, which depends on a variety of factors, such as network size, time of day, problem duration, *etc.* One extreme is to *prevent* misconfigurations from happening, rather than identifying them during *a-posteriori* analyses [54]. For production services, this is highly desirable, but not always possible because of dynamic network changes. For example, a correctly functioning network might become problematic when physical devices fail, because the assumptions on network conditions based on which the configurations were developed no longer hold. In these scenarios, operators are required to *fix* misconfiguration as quickly as possible, to shorten the duration of the production network in non-functional or partially functional states. Similarly, it is preferable if operators can quickly fix misconfigurations during service set up phase, otherwise they would eventually lead to prolonged service realization time and reduced revenue. Unfortunately, modern networks are large-scale and complex, as we previously discussed, thus preventing and fixing misconfigurations are extremely challenging for network operators.

1.2 Building management support systems

This dissertation investigates building systems that help improve correctness and timeliness in network management. We advocate the approach of capturing domain knowledge using formal abstractions for the later integration into operational logic.

We choose this approach because significant evidence suggests that the current management practice relies heavily on human operators, thus time-consuming and error-prone [81]. Our systems can make direct impact on how networks are managed, unlike other proposed solutions that are supplementary to the current management operations and thus require operators' manual consultant and constant involvement.

Correctness and timeliness are chosen as the main evaluation metrics for our systems. Unfortunately, modern ISPs or any large networks are closed in nature. As a result, it is impossible for us to make a direct comparison against the actual management systems and practices of those networks. Nevertheless, we can infer their management approaches by distilling various sources of information, *e.g.,* operational logs, management recipes, and some disastrous events that actually happened in the past, based on which we extract short-comings of the current best practice. We then show that our systems can overcome those short-comings in performing real management tasks.

The inadequacies of current network management and operational procedures have been widely recognized [81], and many solutions have been proposed. Yet most network operators still tend to stay with primitive command line interfaces and their home-brew scripts. For the rest of this section, we explain two main concerns caused this state of affairs.

1.2.1 Choosing the right abstraction

One of the biggest challenges in network management automation is to find the right level of abstraction. What is needed, on the one hand, is the ability to abstractly describe operational goals without getting bogged down in the minutiae of how to achieve those goals. On the other hand, because of the sophistication of modern networking equipment, the ability to fine tune the details of how an operational task is performed is in fact critical to achieving the desired effect. This inherent tension is exacerbated by the fact that network equipment vendors, driven in part by feature requests from operators, have allowed network configuration languages to evolve into arcane sets of low-level commands. Operators therefore have to become accustomed to designing and reasoning about their networks in the same low-level nomenclature, resulting in significant resistance to evolving to new network management paradigms.

We believe that a key requirement for the success of a management system is to use the right level of abstraction that is both close enough to current management approach, thus enable quick adoption, general enough to capture the complexity of existing approaches, and powerful enough to automate and augment them.

Existing supports are usually not sufficient. Scripts are easily adopted, but they lack the sophistication to capture network complexity. Configuration generation tools [53, 63] cannot handle dynamic network changes. On the other hand, many clean-slate proposals [33, 67] deviate drastically from current approach, thus limiting possibility for adoption.

1.2.2 Evaluating proposed designs

For any network management system or framework, a key challenge is evaluation. Many of the existing works were evaluated through small-scale simulation or device emulation with a few lab machines. This is because network changes are fundamentally difficult to make, as modern networks are inherently shared and multi-service in nature. As a result, any change to the network has the potential to negatively impact existing services. A tier-one ISP would certainly not adopt a research management system overnight without significant lab testing. However, the gap between lab equipment and realistic network environments further raises the bar to the introduction of any new network functionalities.

We believe that the need for a new network testing environment is imminent. In particular, we need a platform where new management practices can be tested with real devices and realistic network environments. Such a platform also must not impact other services in the production network.

1.3 Contributions

This dissertation describes a series of efforts to understand, automate, augment and evaluate network operations. First, we performed one of the largest-scale studies over the operational log of a tier-one ISP network [45]. We characterized the complexity of network management, including the diversity of customer requirements and the ordering and dependencies in performed operations. A key observation of our study was that, underneath the complex nature, there are significant patterns in terms of network states and operations executed therein. We were able to use a Deterministic Finite Automaton (DFA) model to capture the dynamics in network operations. An important insight we gathered from this analysis was that network status checks are essential to the progression of network operations, motivating the rest of our work.

Second, we describe *Active Documents* [46], which is based on a Petri-Net model for capturing and automating network management operations. The key idea behind Active Documents is to explicitly model network status checks and execution logic in management operations. As a result, we are able to not only automate network operations by executing Active Documents (achieving timeliness), but also prevent misconfigurations by integrating various additional network-wide reasoning steps to guide operational procedures (improving correctness). We present the design, implementation and evaluation of a system, called *PACMAN*, which assists in the creation of Active Documents and uses them to automate a variety of common network management tasks.

Third, we take a top-down approach to model a network as a database, where network operations are performed via a database-like interface [44]. The main difference from previous bottom-up approaches, like PACMAN, is that under such an abstraction we are able to formally capture domain knowledge of network management using a declarative language. This fundamentally shifts the complex nature of network management away from network operators, as the domain knowledge required can be provided as declarative rules by device vendors and network experts, and seamlessly integrated under the same framework to offer new management primitives, such as configuration automation, misconfiguration prevention, network property enforcement under dynamic conditions, *etc.* We describe the design, implementation and evaluation of *COOLAID*, a system that fully realizes a database interface and is capable of managing large-scale networks.

Both PACMAN and COOLAID expose straightforward interfaces for operators to use, thus facilitate quick adoption. PACMAN allows operators to reason and perform network operations at task-level, and at the same time automates network-wide checks and decision logic without a continuous and tedious manual involvement. COOLAID exposes a simple database interface, while hiding and automating complicated network-wide reasoning behind simple table manipulation operations.

Finally, to address the problem that network management systems are usually inadequately evaluated, we build a realistic, distributed and shared infrastructure, called *ShadowNet* [47] to facilitate the evaluation of the new network management practices in particular and network services and features in general. The key technical effort involves a framework that manages a distributed set of virtualization-capable network devices, creates user-specified network specifications on off-the-shelf devices, and enforces isolation to the production network and fair resource sharing among different users. Both PACMAN and COOLAID were partly evaluated on ShadowNet.

Therefore, my thesis is:

Operations in network management are inherently and increasingly challenging to be handled correctly and timely. With the abstractions like DFA, Petri-Net, and database, we can build new systems that formally capture domain knowledge, automate network management operations while enforcing network-wide properties, and reduce human involvement. With network virtualization, we can better evaluate these new systems and allow faster network evolution.

1.4 Scope and Limitations

Unlike many existing works that focus on static network snapshots, our work emphasizes mostly on the *dynamic* aspect of network management. In particular, our goal is firstly to understand how operational networks are changed on a daily basis by network operators to fulfill a variety of tasks, such as service introduction, attack mitigation and fault diagnosis and resolution. Unlike systems that generate configuration for networks [53, 63], we emphasize on how such configurations are integrated into an existing network, usually using well-designed steps and procedures. We also take a more systematic approach to abstract the protocol dependencies thus generate configuration through actively reasoning about the network status rather than simply populating templates [53]. Unlike systems that identify misconfigurations in configuration snapshots [55, 85], we study how misconfigurations are introduced during network operations and try to actively prevent this from happening. In this sense, our work is complementary to these existing works, as we provide the means for them to express the related domain knowledge using our formal abstractions and further seamlessly integrate into operational logic.

With respect to many clean slate proposals [34, 67], our proposed solutions are considered dirty-slate, as we focus on the existing best practice in network management and how to build automation systems based on existing infrastructure support.

Network management is a very broad concept, and we do not claim to provide a panacea for all problems. According to the definition of International Organization for Standards (ISO), network management has five sub-areas, namely configuration management, fault management, security management, performance management, and accounting management. Our work focuses on configuration management, or more generically all commands executed via the operational interface of network elements. Indeed these are the primary means through which most network operational tasks, *e.g.,* service realization, planned maintenance, fault diagnosis and recovery, performance monitoring and capacity planning, are performed.

In configuration management, COOLAID provides support to automate the generation of network configurations. In particular COOLAID prevents misconfigurations through network-wide reasoning and automates the configuration changes to adapt to dynamic network events. More fundamentally, COOLAID's concept of allowing domain knowledge to be captured by declarative rules potentially revolutionizes the way networks are managed by alleviating the reasoning burden on network operators. PACMAN deals with issues in realizing configuration changes, where network status must be explicitly checked and reasoned to guide the progression of network operations. Both PACMAN and COOLAID integrate domain knowledge into operational logic, in particular, allowing various studies identifying misconfiguration and optimizing network traffic

In fault management, PACMAN is capable of automating fault diagnosis and fault recovery operations. COOLAID can potentially perform more detailed cross-protocol fault diagnosis by tracing the view generation process and compare against network status tables. However, there are a variety of network faults that cannot be handled by PACMAN nor COOLAID. In particular, we do not handle the network faults that are probabilistic or only identifiable through large-scale statistical analysis [89].

Similarly for performance, accounting and security management, our systems can handle the automation of the operations performed in these areas. However, we do not handle problems like performance optimization or security vulnerability assessment. Nevertheless, because of the database abstraction and declarative language used in COOLAID, such analyses and methods can be easily implemented under COOLAID and thus used to guide network operations.

CHAPTER II

A DFA-view of Network Operations

This dissertation begins with an effort to *understand* existing network management practices. The goal is to deepen the understanding of existing approaches by establishing formal models to capture the dynamics in management operations. The knowledge ultimately can guide us to design better management tools and systems.

Managing IP networks is increasingly challenging due to diverse protocols, growing application requirements and primitive support from network devices. The state of the art in network configuration management tends to be template-driven and device-centric [53]. These approaches usually focus exclusively on the configuration aspects of network management, *i.e.,* the task of generating the persistent configuration files that dictate the behavior of the network elements making up the network. Indeed, much of our understanding of network management stems from analyses of such **static** configuration files obtained from operational networks [55, 56, 84].

However, network operators seldom re-generate configurations for the whole network. Instead, they make incremental changes to a network, transiting the configurations from one steady state to another for achieving the desired operational goals. For example, adding or removing a customer is mostly done on an edge router, and handling DDoS attacks usually involves re-routing a portion of the traffic to scrubber boxes rather than changing the whole routing settings. Indeed, most of the transient or long-term misconfigurations causing network disruptions are introduced when the networks are operated on in this manner.

In this chapter, we analyze the complete network operational logs in terms of all the commands executed via router CLIs, building a *dynamic* view of how the network configurations are modified in particular and how the network operations in general are performed over time. A particular challenge is that the command logs present the lowest level of details about network operations. Our solution is to build an interface-centric view of the operational logs, by correlating the commands executed to an interface via exploiting the referential relationships in network configuration. We choose this abstraction because interfaces the most common operational unit. For example, on the edge of a network, an interface is directly connected to a customer, thus the operations applied on the interface is corresponding to the management task associated to that customer. We can easily extend to a group of interfaces, for example a pair of backbone interfaces that form a backbone link, or a group of customer-facing interfaces that connect to different sites of the same enterprise.

After correlating commands to the related interfaces, we extract sequential patterns, representing combinations of commands that are executed together to fulfill different management tasks. From the data we also observe that in addition to commands that lead to *persistent* configuration changes, virtually all management tasks also involve *status-checking* commands that do not change the configuration, but allow the operator to verify network states, which largely determines the follow-up actions. Such operations are critical in operational procedures, but largely ignored by existing studies.

Finally, we model network management operations using automatically generated deterministic finite automata (DFA), where a state represents the configured behavior of an interface and an edge indicates the operations performed on the interface either to fulfill a specific task or to check the status of the network. The DFA model not only allows us to capture common management tasks, but also gives us the ordering and dependency information among those tasks. Containing the information about the temporal progression of network management under different network conditions, the DFA model provides a dynamic view of how large networks are managed today. Based on this understanding, better tools for automating network management can be built. We argue that composing DFAs is a better network management abstraction, which enables operators to reason about the operational state of the network.

This rest of this chapter is organized as follows. Section 2.2 describes the data sources and the initial processing steps taken to facilitate later analysis. In Section 2.3, we present the main analysis and results. The key method is to exploit the referential relationships to associate commands related to individual interfaces. Section 2.4 describes how the DFA model is used to capture network dynamics and the potential usages of such a model. Finally, section 2.5 summarizes this chapter.

2.1 Motivation for Using a DFA Model

DFA stands for deterministic finite automaton [71], which consists of a finite set of states and transitions that depend on input symbols at each step. A number of factors imply that DFA is a good abstraction to model network operations:

1. A network usually transits from one state to another after a series of management operations are performed on it.

2. The state of the network limits the possible operations to be performed.

3. There are a limited number of network states and operations.

(1) is clearly true when the operations make configuration changes, so that the resulting network functionality differs from before. Even when the operations are for checking network status only, (1) is also true from the operator or management systems' perspective. Usually, a status-checking operation would reveal certain properties of the network, *e.g.,* testing if a customer interface is properly configured, or if a backbone interface still has traffic flowing through. The result of these checking operations would change the view of the network. (2) is also true because there is usually ordering and dependency among the operations. For example, an operator cannot shut down a backbone interface (an operation), unless she confirms that no traffic is flowing through that link (a state). Our later analysis results confirm this

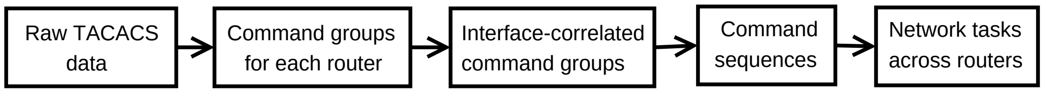

Raw TACACS data
Command groups for each router
Interface-correlated command groups
Command sequences
Network tasks across routers

Figure 2.1: The overall processing diagram

claim. (3) is not suitable when considering a whole network, because the configuration states are clearly exponential. However, the number of states is quite limited when we switch to an interface-centric view, as we explain in detail later.

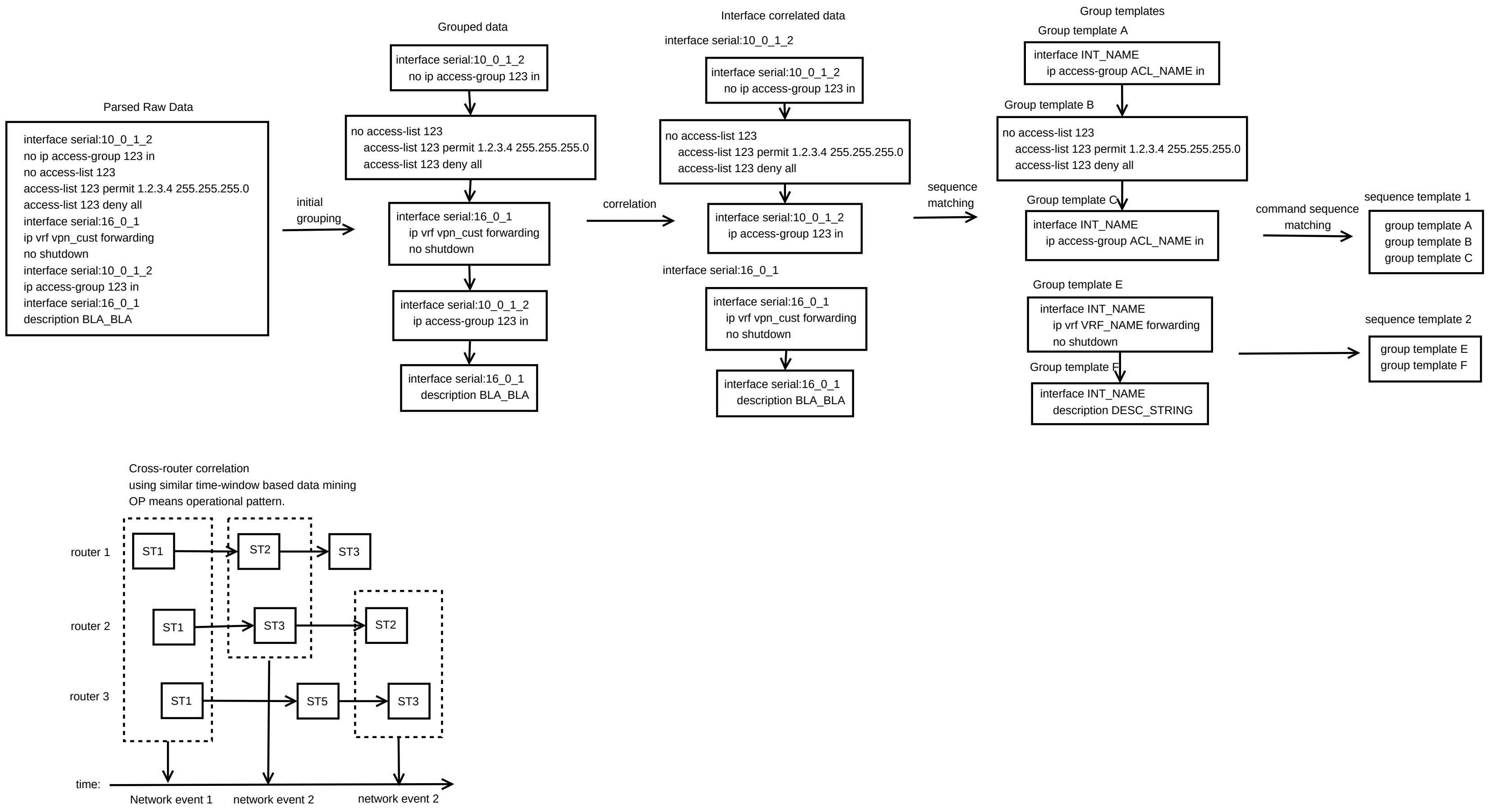
2.2 Analysis Methodology

In this section we describe the data sources and the data processing required to perform our analysis to distill current operational practices from low-level logging information. The analysis steps are illustrated in Figure 2.1, while Figure 2.2 gives a real example of how a sequence of TACACS logs are processed.

2.2.1 Data Sources

We use three data sources that are commonly available in well-managed networks in our study. The main data source is TACACS logs, containing the commands executed on the routers across the ISP network. TACACS is a remote authentication protocol used by network devices to communicate with authentication servers and to audit whether a user has access to a certain router and has sufficient privilege to execute a command. As such, the TACACS logs contain three types of records for login requests (authentication), privilege escalation (authorization) and commands executed (accounting). We are particularly interested in the accounting entries, each of which corresponds to a command executed. Note that TACACS logs are emitted from individual routers to a few centralized servers, thus the logs in rare occasions can be incomplete or corrupted due to network problems. Nevertheless, we found TACACS to be a reliable source.

The second data source we used is the daily configuration snapshots of all the



```
interface serial:10_0_1_2
no ip access-group 123 in
no access-list 123
access-list 123 permit 1.2.3.4 255.255.255.0
access-list 123 deny all
interface serial:16_0_1
ip vrf vpn_cust forwarding
no shutdown
interface serial:10_0_1_2
ip access-group 123 in
interface serial:16_0_1
description BLA_BLA
```

```
interface serial:10_0_1_2
   no ip access-group 123 in

no access-list 123
   access-list 123 permit 1.2.3.4 255.255.255.0
   access-list 123 deny all

interface serial:16_0_1
   ip vrf vpn_cust forwarding
   no shutdown

interface serial:10_0_1_2
   ip access-group 123 in

interface serial:16_0_1
   description BLA_BLA
```

```
interface serial:10_0_1_2

interface serial:10_0_1_2
   no ip access-group 123 in

no access-list 123
   access-list 123 permit 1.2.3.4 255.255.255.0
   access-list 123 deny all

interface serial:10_0_1_2
   ip access-group 123 in

interface serial:16_0_1

interface serial:16_0_1
   ip vrf vpn_cust forwarding
   no shutdown

interface serial:16_0_1
   description BLA_BLA
```

```
interface INT_NAME
   ip access-group ACL_NAME in

no access-list 123
   access-list 123 permit 1.2.3.4 255.255.255.0
   access-list 123 deny all

interface INT_NAME
   ip access-group ACL_NAME in

interface INT_NAME
   ip vrf VRF_NAME forwarding
   no shutdown

interface INT_NAME
   description DESC_STRING
```

Figure 2.2: Data processing example

routers in the ISP network. We use it as a reference point for the configuration changes. Since TACACS data capture how the router configuration is modified across time, combining these two data sources gives us a continuous view of how the configuration of the network evolves over time.

Finally, we make use of network-specific databases [41, 56], which store information regarding physical setups and network configurations, to augment the first two data sources. Specifically, because we take an interface-centric view for our analysis, we use this data to indicate the *role* of an interface in the network, *e.g.,* backbone link versus customer link *etc.* This data source helps us to distinguish different operational targets, and derive common operations on each type.

2.2.2 Data Pre-Processing

We first pre-process the raw TACACS log with the purpose of cleansing the textual data for later analysis that tries to form structures and correlate them. The steps involved are detailed below. Each accounting log entry contains several useful fields, including `username`, `router-id`, `terminal`, `timestamp`, `task-id`, `command`, *etc.*

Ordering commands sequentially: The raw TACACS logs are in the form of large text files consisting of the records for all the routers saved by multiple TACACS servers. The `task-id` field is a monotonically increasing counter inside each router, which can be used to uniquely identify an executed command. We first separate the commands executed on different routers, according to the `router-id` field. For log entries about a particular router, we sort the commands based on both the associated `timestamp` and the `task-id` fields, since `task-id` is initialized to zero when the router is reset to original factory default settings by command `reload`.

Extracting login sessions: There can be multiple operators logged into one router and each operator can have multiple simultaneous login sessions. We first extract out the commands which have the same `username` and `terminal` pair and then demarcate them according to special entries that flag the creation and termination of login sessions. In the end, we infer the login session boundaries and the commands executed within each session.

Differentiating command types: Commands that inspect router status are commonly performed, and they do not change a router's configuration or running status. We call them *status-checking* commands, *e.g.,* `show running-config`. The other type of commands modifies the configuration or behavior of a router, thus we call them *persistent* commands. The status-checking commands usually serve as the purpose of condition checking in the current network management practice to determine whether and how to proceed with the next configuration step, which we will discuss in detail later.

Parsing commands: We developed a parser for processing configuration commands. In particular, we extract the command itself and the parameters used through regular expression matching. For example, `int serial0/1:0` will be matched to `interface INT_NAME`.

2.2.3 Command Group Generation

In router configuration, a complication is that the same command can have different meanings when executed under two execution *contexts*. For example, the same `shut down` command can be used to either disable an interface or stop a BGP session, depending on the context. Each context is similar to a sub-category under the whole configuration, representing a particular aspect, *e.g.,* a particular interface, a BGP session, or an access control list. We thus group the commands executed consecutively within the same context to be a **command group**, as they are likely to be modified for the same purpose.

Each configuration category has its own context-switch commands, with which a sequence of commands within a login session are chopped into command groups. For example, once an operator logs into a router, she is under a normal context, under which only status-checking commands are allowed. Once `configure terminal` is executed, she switches to a configuration context, under which the configuration modifications are performed. To configure the interface `serial1/0:0`, she needs to

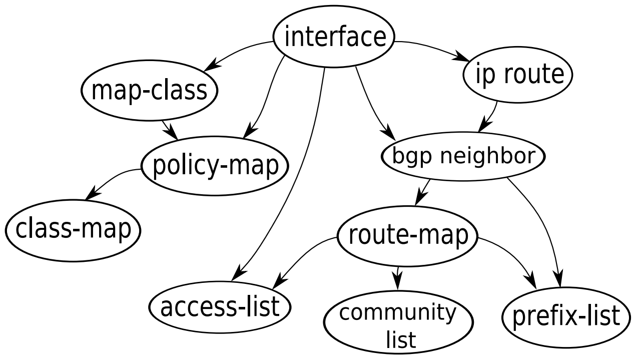


Figure 2.3: Correlation among command groups

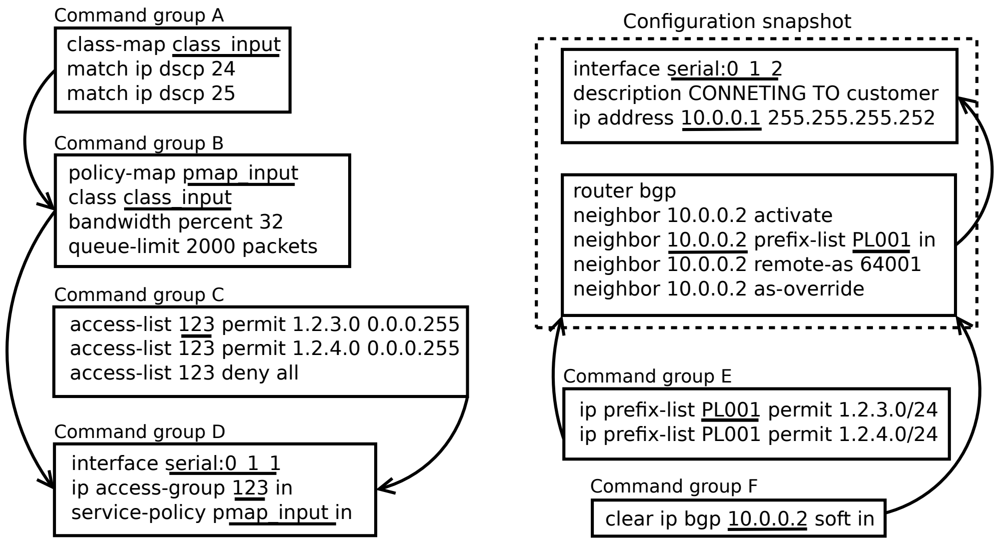

Command group A
class-map class_input
match ip dscp 24
match ip dscp 25
Command group B
policy-map pmap_input
class class_input
bandwidth percent 32
queue-limit 2000 packets
Command group C
access-list 123 permit 1.2.3.0 0.0.0.255
access-list 123 permit 1.2.4.0 0.0.0.255
access-list 123 deny all
Command group D
interface serial:0_1_1
ip access-group 123 in
service-policy pmap_input in
Configuration snapshot
interface serial:0_1_2
description CONNETING TO customer
ip address 10.0.0.1 255.255.255.252
router bgp
neighbor 10.0.0.2 activate
neighbor 10.0.0.2 prefix-list PL001 in
neighbor 10.0.0.2 remote-as 64001
neighbor 10.0.0.2 as-override
Command group E
ip prefix-list PL001 permit 1.2.3.0/24
ip prefix-list PL001 permit 1.2.4.0/24
Command group F
clear ip bgp 10.0.0.2 soft in

Figure 2.4: Interface correlation example

execute `interface serial1/0:0` first, to switch to an interface configuration context.

2.2.4 Interface Correlation

The dependencies within router configurations have previously been studied by Feldman *et al.* [56] to check static configuration errors, *e.g.,* broken references. In our work, we extend this referential relationship to identify correlated configuration command groups. We define command groups A and B to be **correlated**, if one command in A contains a variable name which is exactly group B's name. For example an interface group with command `ip access-group 123 in` is correlated with `access-list 123` command group, because the interface is configured to apply the ACL. Figure 2.3 shows some possible correlations across different command groups. For example, an interface configuration might reference a `policy-map` for traffic differentiating, while the `policy-map` must reference several `class-map`s, which define the traffic classes. The dependency is directly derived from command syntax, which is well-understood.

We developed an algorithm to correlate command groups to related interfaces. This **interface correlation** is trivial for interface command groups. Conceptually, we want to understand which interface is directly impacted by a given command group. The key idea is to find a chain of correlation relationships that would eventually link a command group to an interface. We first identify correlations within TACACS log only. For example, in Figure 2.4, command group A-D are executed within the same login session. We can correlate group C to D, because of the ACL setup, and A to D through B. If a command group cannot directly be linked to an interface, we further look at network configuration. In the same figure, group E and F are linked to an interface connected to a BGP-speaking customer, with the help of the configuration in the snapshots. Note that one command group can be correlated to multiple interfaces. For instance, modifying an ACL that is used by two interfaces.

2.2.5 Command Sequence Extraction

Command groups that are correlated to one interface are usually executed together in fixed patterns, which we define to be a **command sequence**. In Figure 2.4, command group E and F represent an interesting pattern: changing an *ip prefix-list* for a BGP session followed by a BGP session reset. This is a typical operation for effecting routing policy changes.

We first canonicalize the commands executed by replacing parameters with generic place-holders, and then transform command groups to group templates. We exclude commands that are hardware-specific in favor of generating a small number of templates. We combine the group templates that are always executed together as a sequence template. As a result, a sequence of commands that match a sequence template is considered as a command sequence.

2.2.6 Network-wide Event Correlation

Managing large networks sometimes requires configuring multiple routers simultaneously. For example, inter-domain traffic engineering usually needs to change the BGP routing policy of two or more BGP sessions between two ASes. We perform a network-wide timing correlation to extract the sequence templates that are executed on different interfaces within a short time window. These correlated events give us ideas on how network-wide operations are performed.

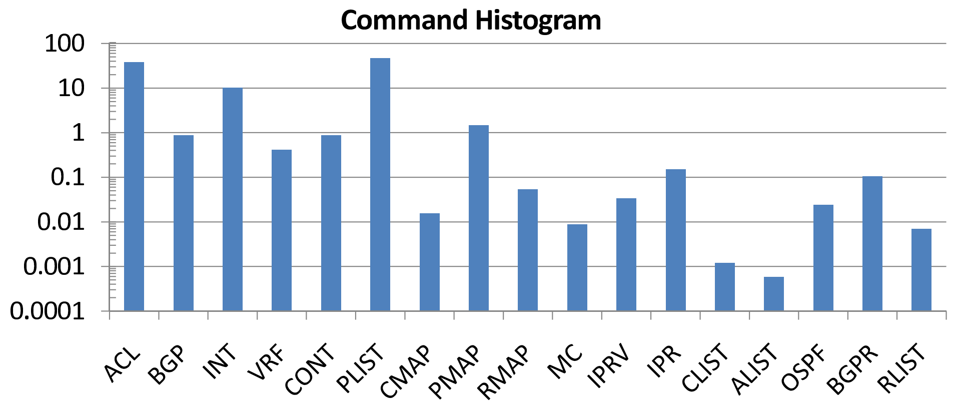
2.3 TACACS analysis results

We analyzed a four-month TACACS log from a Tier-1 ISP network. This reflects the major operations performed in a large-scale, operational and realistic network. We here report our main findings, from low-level command group statistics to high-level network-wide events. Our current implementation focuses on Cisco routers, although our overall approach is also applicable to other configuration syntax. Our parser is capable of parsing over 98% of the persistent commands[1], and all status-checking commands observed can be parsed.

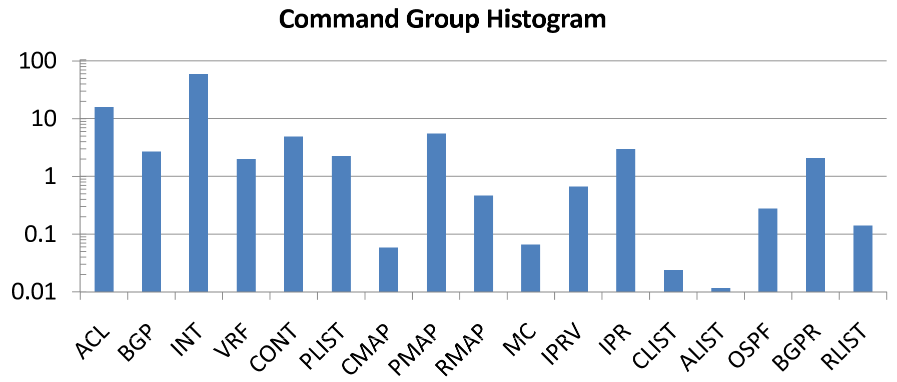
2.3.1 Command Groups

Figure 2.5 shows the histogram of the number of commands and the number of command groups observed in the data respectively, broken down by the command group categories shown in Table 2.1. The Y-axis is the log-scaled percentage value. In Figure 2.5(a), we can see that the majority of the commands executed on routers are ACL (39%) and PLIST (47%), which correspond to access-lists that filter packets

[1]The remaining 2% of persistent commands are all hardware-specific commands related to specific interface types in the provider network and are not germane to the focus of our study.



(a) The breakdown of configuration commands



(b) The breakdown of configuration command groups

Figure 2.5: TACACS data analysis results

ACL	`access-list` group and `ip access-list` group which define ACLs
BGP	`router bgp` group, which define BGP sessions
INT	`interface` group, defining interface configurations
VRF	`ip vrf` group, which defines VRF profiles
CONT	`controller` group, which defines controller setups
PLIST	`ip prefix-list` group, which defines prefix-list that is used to filter routes
CMAP	`class-map` group, which defines a class of packets
PMAP	`policy-map` group, which defines the traffic-shaping policy of certain classes of packets
RMAP	`route-map` group, which defines how to manipulate certain routing messages
MC	`map-class` group, which encapsulates policy-maps and can be directly applied to interfaces
IPRV	`ip route vrf` group, defining VRF static routes
IPR	`ip route` group, defining static routes
CLIST	`ip community-list` group, which defines filters based on community values for route-maps
ALIST	`ip as-path access-list` group, which defines filters based on as-path for route-maps
OSPF	`router ospf` group, defining OSPF routing process
BGPR	`clear ip bgp` group, which resets BGP sessions
RLIST	`ip receive list` group, which uses an ACL to filter received packets of the router

Table 2.1: TACACS command group types

and prefix-lists that filter BGP routes. This is expected as access-lists and prefix-lists are the first line of defense for a network's data plane and control plane respectively, effectively guarding against source spoofing (often used in DoS attacks [59]) and wrong route injection (common practice for prefix hijacking [82] and a common cause for routing misconfiguration.) These command groups tend to have many entries, especially for a prefix-list that filters routes from a peering ISP, which can contain thousands of entries. Interface groups contribute to the third largest number of commands and around 60% of the total number of groups. As shown in Figure 2.5(b) the interface components are the most frequently modified in network management. This is also expected as many planned and unplanned events may trigger interface related operations. For example, all the customer provisioning operations must involve configuring the corresponding customer-facing interfaces, while backbone links are usually operated for traffic engineering and link maintenance. This partly motivates our later attempt to use an interface-centric view. The second and third largest number of groups come from access-list (16%) and policy-map (5.6%). The importance of policy-map also emerges, since it is used intensively to guarantee QoS.

2.3.2 Interface Correlation

After correlating configuration command groups to interfaces, we have formed another level of abstraction of the TACACS data, *i.e.,* the configuration commands that are executed in order with respect to a single interface. Our interface-centric correlation is very effective: over 99% of the TACACS commands are successfully correlated to one or more interfaces. Our algorithm is not able to correlate a command group like policy-map which is defined but not referenced by any interfaces — in this case, either we should delete such a policy-map, or it means some interfaces are using some other policy-maps by mistake.

We further analyzed the break-down of the commands executed on different types of customer-facing interfaces — connecting to VPN, static-route and BGP neighbor. The results show that the operations performed on these interfaces are dramatically

Description	Num. of occurrences	Pctg of the total commands
ACL MOD	58825	36.6%
PREFIX MOD	2677	42.2%
STATIC INIT	10761	6.23%
VRF INIT	4156	1.3%
VRF ENABLE	3982	1.34%
BGP MOD	1795	4.77%
STATIC ENABLE	5605	3.30%
BGP ENABLE	464	0.36%
RECV MOD	638	0.55%
TOTAL	91447	96.7%

Table 2.2: Major command sequences

different: 1) VPN customers are more likely to have policy-maps applied, which enforce a higher QoS; 2) over 80% of the commands related to BGP neighbors are prefix-list modifications, indicating routing policy changes characterized by allowed prefixes to exchange; 3) over 80% of the commands related to static-route customers are ACL modifications, probably associated to new address allocation.

2.3.3 Command Sequences

Table 2.2 shows the result for the command sequences we extracted. Command sequence is an interesting level of abstraction for understanding current network management. Upon manual inspection, a command sequence can be translated to a specific task performed on a router related to a specific interface. For edge routers in particular, managing such an interface is conceptually equivalent to managing the customer site connected to the interface. Note that these nine command sequences cover almost 97% of the commands executed during our study period, representing the majority of the network tasks being performed:

- Provision new interfaces neighbor or customer: "STATIC INIT" is a sequence of commands that initializes an interface (consisting of a controller group that allocates the sub-interface, an access-list group, and an interface group). The sequence of “VRF INIT” is similar to “STATIC INIT”, but with additional MPLS VPN related setup.

- Enable or tear down neighbors: “STATIC ENABLE”, “BGP ENABLE” and “VRF ENABLE” correspond to the sequences that finalize the configuration for an interface that connects to static-route customer, BGP neighbor or VPN customer respectively. There are four types of short but important command sequences not shown in the table - “SHUT DOWN” the interface (`shutdown`), “BRING UP” the interface (`no shutdown`), “REMOVE INT” by deleting the interface configuration and de-allocating the sub-interface if necessary, and “DIAG INT” for diagnosing problematic interfaces.

- **Handling network changes:** “ACL MOD” is an event in which the ACLs of the interface are modified, while “PREFIX MOD” means modifying the prefix-list used by a BGP session, followed by a BGP session reset. “BGP MOD” is the event in which the prefix-list of the BGP session and the ACL of the correlated interface are modified at the same time.

2.3.4 Network-wide Event Correlation

When extracting command sequences, we also know the interface being configured and the times when the configuration change happened. We did a preliminary study by counting the number of appearances of two events happening together within five-minute time windows. The most frequently occurring pair is the BGP policy change of two peering links connecting to one neighboring ISP. On one interface the ACL and prefix-list are modified such that a few prefixes are no longer allowed to transit the link, while the opposite is performed on another interface. The traffic data near those events shown in Figure 2.6 clearly shows the combined effect of shifting traffic from one link to another.

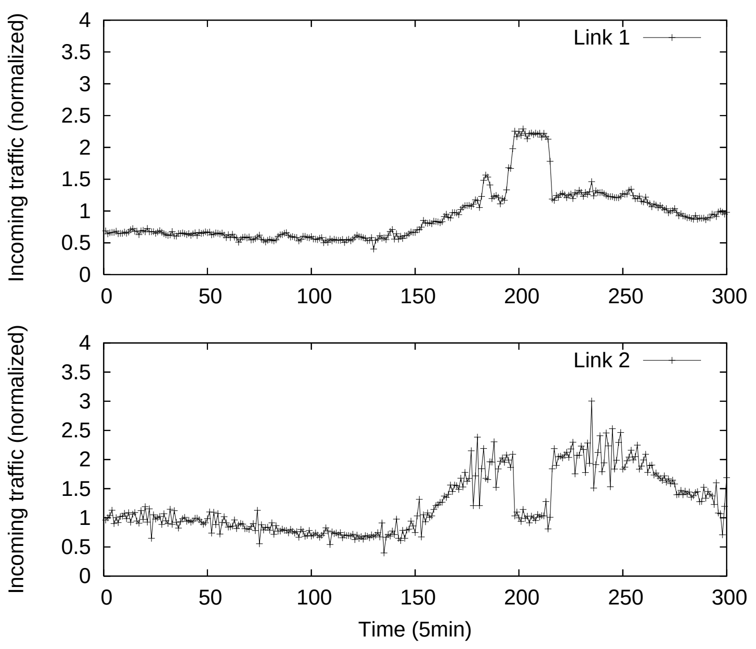


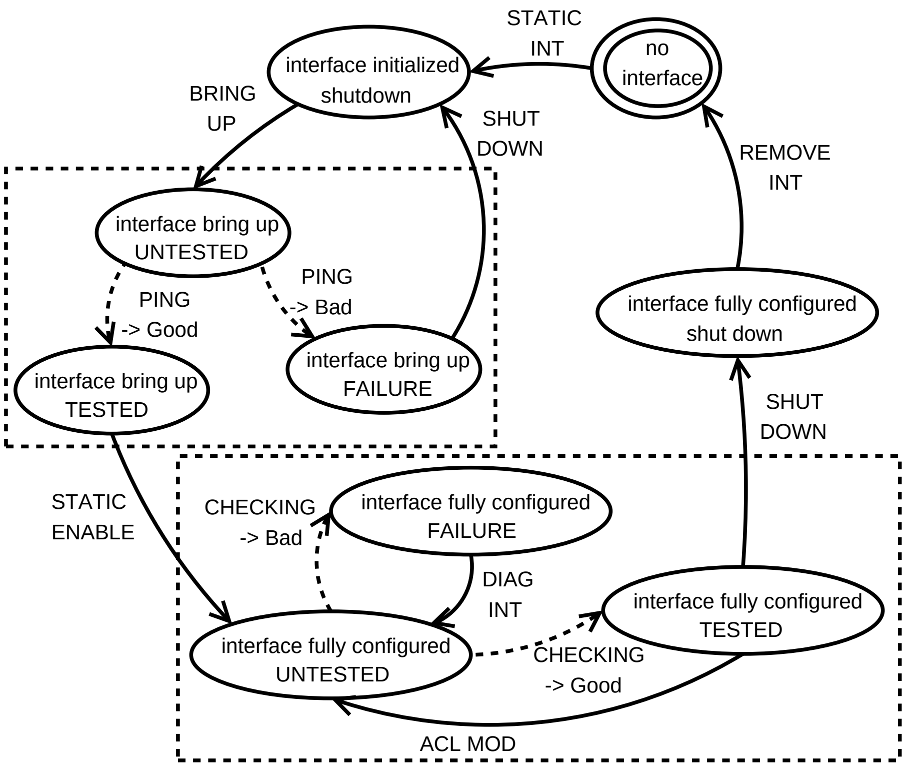
Figure 2.6: Correlated events across network

2.4 Modeling Network Operations Using DFAs

In this section, we propose to use DFAs to model dynamic network operations. We first explain how DFAs are used, and then discuss how they are generated and used in practice.

2.4.1 Example DFA

We use a DFA to model the complete life cycle of an interface abstraction, as illustrated in Figure 2.7. In this model, a state represents a particular configuration scenario and the perceived network running-status. An edge corresponds to an operation can either change the configuration or change the perceived network status, thus transiting from one state to another. In the example, all interface starts with the empty state "no interface", indicating that the interface is not even configured on the router. An interface initialization operation (edge `STATIC INIT`) would preliminarily configure the basic setups for the interface, but the interface is still in a shut down state. Only when the `BRING UP` operation is performed does the interface become operational. The states in the boxes share the same configuration, but are



STATIC INT
no interface
interface initialized shutdown
BRING UP
SHUT DOWN
REMOVE INT
interface bring up UNTESTED
PING -> Good
PING -> Bad
interface bring up TESTED
interface bring up FAILURE
interface fully configured shut down
SHUT DOWN
STATIC ENABLE
CHECKING -> Bad
interface fully configured FAILURE
DIAG INT
interface fully configured TESTED
interface fully configured UNTESTED
CHECKING -> Good
ACL MOD

Figure 2.7: Sample DFA for static-routed interface

characterized by different perceived status. For example, after being brought up, the interface is first considered to be "UNTESTED", meaning the operator has no idea if it works or not. In this state, a ping check is usually performed, such that a ping pass would transit the interface to a "TESTED" state, while a ping fail transits to "FAILURE" state.

2.4.2 Generating DFAs

In Section 2.2.5, we described how command sequences and sequence templates are generated. The sequence templates are here used as the edges in the DFA.

We generate the DFA states by analyzing network configurations. Given a router configuration at any time, we can find all the components that are correlated to a specific interface, using the same correlation algorithm described in Section 2.2.4. All these correlated configuration groups, which we denote to be **interface-correlated configuration** of an interface, exactly define how this interface *should* work. We define a **configuration state** to be one possible interface-correlated configuration.

Ideally, we should reconstruct how the router configuration change over time, and identify the individual configuration states for all interfaces during steady states. This requires a network configuration snapshot as a starting point and the complete configuration changes afterward. Since the operation log cannot guarantee completeness (delivered from individual routers via UDP to a set of collectors), we approximate this process by analyzing daily configuration snapshot data. The assumption is that for most interfaces are in a steady state when a snapshot is taken. We ignore those configuration snapshots if the corresponding router was operated on around the time the snapshots were taken, since the interfaces could be in an intermediate state. After extracting configuration states for different interfaces, each consisting of several command groups, we canonicalize them by ignoring the variable names and parameters of commands to facilitate comparisons. We denote a canonicalized form of a configuration state to be a **state template**. This process is fully automated given our parser. Interestingly, we found that a very small number of state templates can capture all the configuration states, confirming our previous assumptions. A state template is then considered as a state in the DFA. In Figure 2.7, the lower-cased words within a state is the high-level description of that state template.

In our design, each edge of the DFA corresponds to either a persistent command sequence that changes the configuration or running-status, or a status-checking command sequence that helps determine the actual running-state of the interface. Persistent sequences and status-checking sequences are marked respectively as firm and dotted arrows in Figure 2.7.

Finally, we need to stitch the states and edges together. We start with state templates and persistent sequence templates only. We combine the configuration snapshots and TACACS logs. If the configuration state of an interface transited from template S_A to S_B, while only sequence template E_C was performed in between, we then connect the two states with that edge. This process is automated and results presented for verification. It is possible that we see two or more persistent sequences after a status check in the same configuration state. In this case, it means the status check has established different understandings of the network. Note that, TACACS data is unable to capture the results of status-checking commands. We thus manually infer those states using domain knowledge. Chapter III discusses a system that allows such decision logic be formally captured. Here, we manually split the state related to the configuration state to multiple sub-states, like the ones within the dashed box.

Because of potential data loss in TACACS data, we are unable to evaluate precisely how well the DFA abstraction captures operational procedures. Yet, we found that the overwhelming majority of the sequences of operations can be accepted by at least one of the DFAs we generate. Should there be any violations, they either indicate data loss in TACACS or an operation that is performed out-of-context, meaning that such operations may not be appropriate given the network status. Given the sensitivity of such analysis, we are unable to provide further explanation.

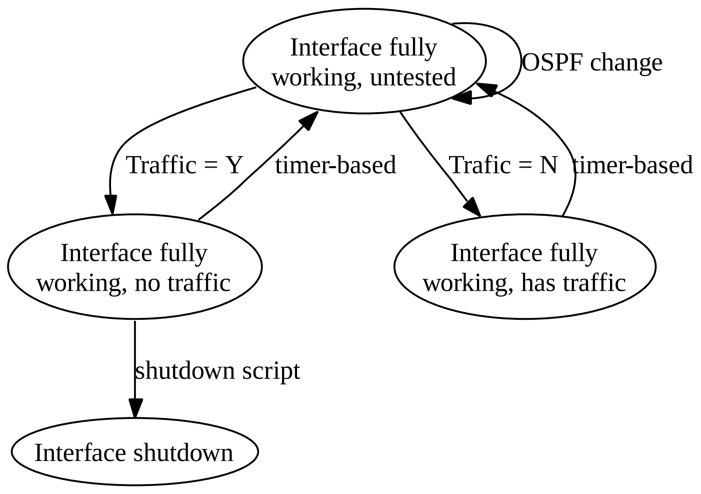

Interface fully working, untested
OSPF change
Traffic = Y
timer-based
Trafic = N
timer-based
Interface fully working, no traffic
Interface fully working, has traffic
shutdown script
Interface shutdown

Figure 2.8: Avoiding invalid transitions using a DFA.

2.4.3 Using DFAs

The DFA model that we developed not only provides a way to visualize all the operations related to a particular interface, but also gives much more information about the temporal progression in network management. In particular, a DFA concisely captures the possible operations that *can* be performed under a specific network state. This state awareness is missing in the current configlet-based approach [53, 63], as it relies on operators to manually stay on top of the network states and apply appropriate operations. Similarly, operators need to manually understand network states before executing any scripts to ensure correctness.

We can apply the derived DFAs to assist in network management automation in the following ways. The persistent command sequences (outgoing edges) of each state define exactly what can be done given a particular state of the interface, guiding the next step in configuration. The condition checking can be automatically performed to immediately verify previous execution results and trigger the subsequent persistent configuration changes. By encoding possible network running-status into a group of states with the same configuration state, we can achieve more sophisticated network management automation using the DFA model. For example, in Figure 2.8, we have two states, “Interface fully-configured, has traffic going through during the last minute", "Interface fully-configured, no traffic going through during the last minute". We can then clearly specify that the command sequence which shuts down the link can only be executed in the latter state. This can prevent undesired traffic disruption and ensure that the network maintains a healthy running state.

2.4.4 Limitations of DFAs

While being a succinct model for capturing interface related management operations, the DFA model has its limitations. First, it is derived from operational logs, thus cannot handle new devices and new procedures. The PACMAN system overcomes this problem by introducing a recording system to allow new operational procedures be accurately modeled (Section 3.3.) Second, it does not model concurrency among interfaces well. A DFA can be used to regulate and automate the operations performed on a single interface, but large-scale network operations in many cases require touching multiple interfaces and have cross-interface dependencies. As such, we need a better model that is capable of capturing and orchestrating network-wide concurrency. The PACMAN system addresses this problem by extending the DFA model to a Petri-Net model, which explicitly models concurrency.

2.5 Summary

In this chapter, we analyzed a TACACS data source consisting of all the commands executed on many routers within a Tier-1 ISP network. Starting from low-level raw data abstracted to high-level correlations, we developed a way to summarize the high-level network operations that are performed on the network. This is the first concrete study revealing the dynamic network management activities in real networks.

Today's network management is greatly eased by automatically generated configlets which can be translated from high-level policies and then applied to the routers directly. However, we found that configlets are usually applied to the routers through a sequence of carefully designed steps, which are usually causally dependent on each other or on the network's running-status. We developed a DFA model to characterize such dynamics in network management, which we believe is an important step towards automated network management.

CHAPTER III

A Petri-net Model for Network Automation

This chapter describes a system, PACMAN, for *automating* and *augmenting* network management operations. The key intuition is to use a Petri-Net model to capture network operations in a bottom-up fashion.

Despite the existing efforts of building the so-called "holistic" network management systems from both research and industry, the lingua franca of network operations continue to be libraries of *method of procedure (MOP)* documents. MOPs are text documents that describe the procedures to follow in order to realize specific operational tasks. There are two main sources of MOPs. First, device vendors usually provide manuals regarding how their products can be configured and operated to realize different functionalities [17, 73], and similarly how diagnostic and recovery procedures should be performed when network problems occur [11]. Second, experts in service providers explain in prose and configuration excerpts the best practice on network service realization and operational procedures. For example, carefully designed steps are commonly described on performing planned device maintenance without impacting existing network services. Currently, MOP documents are literally stored as libraries of text descriptions that are meant for human consumption, rather than the purpose of automation. As a result, manual operations following these MOPs are inherently limited by human operators' ability to consume and carry out the knowledge described.

In modern network operational environments, parts of MOP-defined procedures are typically automated to a limited extent. For example, configuration actions could be performed by scripts that push configlets to network elements [38, 53, 63]. However, for the most part, network operations still require human operators to verify the result of actions and to navigate through the logic involved in operational procedures, as we have shown by analyzing operational logs from a tier-1 ISP (Section 3.1). This is especially true in terms of being cognizant of possible interactions among different operational procedures and understanding the network-wide impact of such actions. For example, multiple backbone link maintenance operations could partition the network backbone causing undesired dis connectivity periods. Indeed, sophisticated tools have been developed to help operators understand the possible impact of their actions [57]; however, operators typically consult these tools independently and then use the information they provide to manually "close the loop" to perform operational tasks. In other words, such tools are not fully integrated into the process of network operations.

Towards this end, in this chapter we present our work on the *PACMAN* system, a Platform for Automated and Controlled network operations and configuration MANagement. Our work builds on two basic observations related to the elements contained in MOP documents. First, MOP document structure presents a natural way for operators to think and reason about operational activities. Second, the logic embedded in the design of these procedures represent the expert knowledge of MOP designers to ensure that high-level operational goals and conceptual designs are met, while minimizing unwanted side effects of operational actions. At a high level, PACMAN maintains these desirable properties by allowing experts to define operational procedures as before with one significant difference: The procedures defined in the PACMAN framework are not static documents meant for human consumption, but instead *active* method of procedures, or simply *active documents (AD)*, meant for execution in the PACMAN framework. As we explain in detail later, active documents follow a Petri-Net model to capture all the critical elements required in the network management procedures and described in MOP documents, forming a fundamental building block for the automation of network operations. ADs enable the complete, repeated, programmatic and automated execution of low-level management tasks, but more importantly enable the construction of more sophisticated tasks. Specifically, simple active documents can be *composed* and executed, with network-wide *policies* enforced. The policies also fall nicely into the Petri-Net model and are programmable to realize network-wide management objectives, *e.g.,* prevent network partitioning. In so doing, PACMAN raises the level of abstraction as the high-level operational goal becomes part of the composed task, thus being enforced in an automated fashion, eliminating the need for operators to be continually concerned about and actively involved in *how* to carry out a goal amongst multiple tasks. Furthermore, the PACMAN framework allows easy interaction with external tools to enable sophisticated decision making to be naturally integrated into the network operational loop.

In this work, we make the following contributions:

- We analyze method-of-procedure documents from an operational network to extract the *network management primitives* associated with network operations.
- We introduce *active documents* as a concise, composable, and machine-executable representation of the actions, conditions and workflow logic that operators perform during network management tasks.
- We present the design and implementation of the *PACMAN* framework, an execution environment that automates the execution of active documents.
- We bridge the gap between current operational practices and our automated environment with the AD *creation framework*, a set of tools which allow operators to work in their native idiom to easily generate active documents.
- We demonstrate the effectiveness of the framework using several case studies of common configuration tasks such as fault diagnosis, link maintenance, and a more complicated task of IGP migration.

3.1 Management Primitives in Current Practice

To understand current best practice to extract the fundamental primitives and requirements for performing network management operations in large networks, we analyze real method of procedure (MOP) documents from a Tier-1 ISP and a major router vendor. The ISP's MOPs are propiatary, but an example of a MOP from the router vendor can be found online [11]. These documents cover a wide variety of network management tasks, including customer provisioning, backbone link and customer access link migration, software/hardware upgrade, troubleshooting VPN, *etc.*

MOP documents are essentially instruction manuals for performing specific management tasks. They are usually modularized, consisting of multiple sub-tasks or steps. For example, a BGP customer provisioning MOP from the ISP contains three steps of link setup, IP setup and BGP session setup, where each step involves configuration changes on a router and verification of the resulting network running status. A fault diagnosis MOP [11] often contains a sequence of network checks to perform, such as `ping`, `show bgp`, and an instruction on how to interpret and act upon different check results. These observations are consistent with our previous DFA study.

At a micro level, we categorize the fundamental network management primitives that make up the MOPs as follows:

Configuration changing: Most of the management tasks involve configuration modification, which directly leads to network device behavior change. For example, configure a BGP session, change OSPF link metric *etc.*

Status acquiring: Network status information is crucial for the progression of network operations. Two types of status are usually obtained: static information, such as configuration, hardware components; dynamic information, such as BGP session states, routing tables. The acquired information can either be stored for future use or processed immediately.

Status processing: Status information is evaluated in a variety of ways, for example, check router configuration to identify OSPF-enabled interfaces, verify if a routing table contains a specific route, or even compare the current BGP peer list with previous captured list. Based on the evaluation, different next steps may be taken.

External synchronization: Explicit synchronization with other parties, including field operators, centralized decision engine, *etc.*, is very common. The operator can either notify an external party indicating operational progress or wait on external parties for their progress update, for example, wait for a field operator to finish an on-site physical upgrade.

The lack of automation also manifests as the fact that these primitives need to be *composed* together manually. We identify the following composition mechanisms (or workflow logic):

Sequential: This most basic composition simply perform one sub-task after another. It is useful for stitching many stand-alone operations into a complex operation.

If-else split: The purpose of status processing is to choose different subsequent sub-tasks based on an if-else logic. For example, for different OS versions or interface types, the configuration to change could be different.

Parallel split: In some cases, the operator is required to work on multiple devices at the same time. In other cases, a monitoring sub-task is spawned on the side. For example, creating a new terminal session to launch a continuous ping to monitor delay and jitter of a potentially impacted path.

Iterative processing: Operators may need to process one element at a time, until there is no such element left. For example, to identify all interfaces with IS-IS configured, and disable them one by one.

Wrapper: Predefined "head" and "tail" sub-tasks can be used to wrap around other sub-tasks. For example, saving the configuration and running status before and after the operation for later verification.

Sequential composition is the easiest to automate, but due to the frequent occurrence of other cases, the majority of network management tasks cannot simply be represented as a sequential flow. Other composition mechanisms are almost always handled by human operators.

At a macro level, the descriptive nature of MOPs dictates that the realization of management tasks have to rely on human operators, who consume the MOPs and carry them out either manually or via limited automation, resulting in a process that is known to be time-consuming and error-prone. Based on our analysis, only configuration changing and network status acquiring are automated, to a limited extent, through automated tools. The decision logic, for example reasoning about network status to determine the proper next step, is usually described in high-level terms and almost always left for human operators to realize. This deficit calls for an automatable representation of workflow logic, which we integrate into our active document design.

MOP documents are by necessity limited in scope, typically focusing on a specific operational task, with little visibility into the network-wide impact of the task or its interaction with other tasks. In the case of multiple concurrent tasks, the burden of avoiding undesired network states based on reasoning about global network status usually falls on human operators, who may not be able to correctly perform such reasoning due to knowledge deficit or resource constraint. The same problem is valid for the DFA model, as the execution of each DFA would progress the operation on an interface, while little support is available to coordinate the progression of multiple interfaces. This motivates us to design active documents that are composable and capable for network-aware policy enforcement. On the other hand, while tools are available to show how some operational tasks (e.g., costing out a link) might impact the network [57], the interaction with such tools is currently largely left to operations personal.

3.1.1 Motivation for Using a Petri-Net Model

We argue that a Petri-Net model is suitable for abstracting network management operations.

First, Petri-Net is known to be good for abstracting and automating workflow logic [70]. Indeed many support systems are made based on the Petri-Net model [29, 92, 117]. Within the context of network management, the execution logic of Petri-Net naturally maps to that of network management tasks: all the management primitives identified above can be captured by the two types of nodes in the Petri-Net model.

Second, Petri-Net explicitly handles concurrency. It has been used to model and reason about multi-threaded programs [79]. For us, instead of modeling individual interfaces as DFAs and worry about the coordination of DFA progressions, we can use Petri-Net to explicitly regulate the execution of different parts of the network.

Third, Petri-Net is a graph-based model, containing nodes and edges. As we show later in Section 3.2.2, it is fairly easy to combine multiple graph representations together to realize the composition mechanisms mentioned above.

Fourth, Petri-Net allows imposing higher level control logic to regular the execution of the modeled workflow. In a recent work, Wang *et al.* proposed to eliminate deadlocks in multi-threaded programs through modifying the Petri-Net model of the corresponding programs [115]. In our work, we show that this model can help us enforce network-wide properties.

3.2 The PACMAN Framework

PACMAN is motivated by the fact that automation is limited in current network management. We tackle this problem by introducing a new abstraction that incorporates all required operational primitives and compositional mechanisms identified from MOP documents, allowing natural absorption and formal representation of the domain knowledge and full automation of the network operations. As a step further, the abstractions allow multiple tasks to be independently specified but automated simultaneously with global awareness seamlessly imposed without additional manual involvement; therefore, they overcome the task-centric nature of MOP documents. To the contrary, traditional scripts are usually ad-hoc and do not allow systematic composition, coordination across multiple executions, and integration of

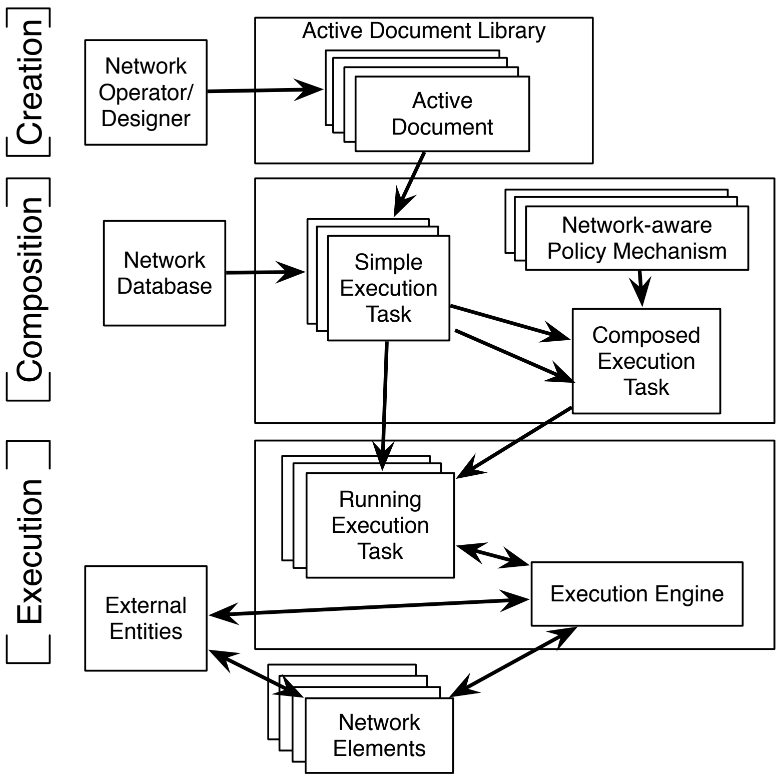


Figure 3.1: The PACMAN framework

network awareness.

PACMAN framework is conceptually divided into three aspects, namely creation, composition and execution, as shown in Figure 3.1.

Creation is a crucial step which allows network operators or experts to create instances of our abstraction, named *Active Document*, to describe and further enable the automation of the workflow of network management tasks in a form that is accurate, extensible, and composable. Unlike MOPs that are used to *guide* human operators, active documents can be *executed* on different networks to fulfill different management tasks. Compared to traditional scripts that at best automate the generation and modification of configuration on network devices, an active document also encapsulates the *logical reasoning* that guides the workflow of a management task. This capability enables full automation of network management tasks, minimizing human involvement. We describe the details regarding active documents in Section 3.2.1. Active document models the primitives and composition mechanisms derived from MOP documents in a straightforward fashion, thus can be created by anyone who understands these documents, enabling our framework to quickly absorb the expert knowledge from existing MOP documents to form our own *active document library*. To illustrate this conversion process, a framework to enable semi-automated active document creation is described in §3.3. Note that once an AD is generated, it can be re-used and combined with other ADs for repeated and controlled executions.

Composition is the step when specific management tasks are created from the abstract active documents. An *execution task* can be generated in multiple ways: (i) a simple execution task is generated by selecting an AD from the AD library and assigning proper parameters either manually or from external network databases; (ii) a composed execution task is created from one or more execution tasks following a proper composition mechanism; (iii) a composed execution task can be further enhanced by adding network-aware policies, each of which regulates the execution flow by reasoning about global network conditions, automatically guarantee that the execution of a collection of task-centric operations would not violate network-wide constraints. Note that such policies are pre-defined and can be selected and automatically imposed during composition. These mechanisms provide the means by which operators can start small and simple (developing task-centric ADs) yet achieve automation of network-wide coordination. Existing coordination and policy enforcement in management operations is usually either undocumented or written in high-level terms in MOP documents, due to the complexity involved. It is mostly done by skilled human operators, *e.g.,* who can decide when to execute which script so that traffic shift is minimized. PACMAN, on the other hand, provides a fully automated solution, enabled by the flexible and generic active document design.

Execution is the stage in which the management operations described by the execution tasks are actually effected onto the physical network with the support provided by an *execution engine*. We envision each network to have such an execution engine, which carries out the execution tasks in a fully automated fashion, achieving the goal of each task by reproducing the workflow and decision logic embedded in the ADs. Illustrated in Figure 3.1, as a result of running the execution tasks, the execution engine interfaces with devices in the network to perform the configuration change specified by the execution task, obtain various types of network status, and carry out the embedded reasoning logic. The execution engine also interacts with entities external to the PACMAN framework. As shown in the figure, these external entities might also interact with the network. Examples might include stand-alone network monitoring tools, or an on-site operator that is signaled that the network has been readied for the replacement of a router linecard, or some other manual operational task. The execution engine is also responsible for scheduling multiple tasks to run concurrently, providing failure handling support, *etc.*, moving closer to its goal of minimizing human involvement.

Finally, we note that the relationship among active documents, the execution engine, and running execution tasks is analogous to that among program binaries, the operating system, and running processes. Similar to what an operating system does, the execution engine provides the running environment to the execution tasks, ensuring the correct and automated task execution according to the AD. We now explain these three aspects in detail in the following sections.

Node type	API Call Name	Functionality
Action	`CommitConfigDelta()`	Commit a configuration change to a target device.
	`NotifyEntity()`	Send messages to external entities.
Condition	`QueryDeviceStatus()`	Obtain physical device status information.
	`QueryEntity()`	Obtain information from external entities.
	`QueryExecutionState()`	Obtain execution task running status.
	`TaskSucceed()`, `TaskFail()`	Notify execution engine that task has succeeded or failed

Table 3.1: API calls supported by the execution engine

We now consider each of the PACMAN components in detail.

3.2.1 Active Documents

In a nutshell, active documents follow a Petri-Net model [93], whose graph representation encodes the required network management primitives and allows flexible composition mechanisms described in Section 3.1. Like a program binary, an active document can be executed on the network with sufficient input parameters.

Elements: Petri nets are bipartite directed graphs containing two types of nodes: places, shown as circles, and transitions, shown as bars. We associate key management primitives to these two types of nodes: *Action* activities (corresponding to bars) include configuration or state modification and external notification; *Condition* activities (corresponding to circles) are status acquiring followed by status processing. We abstract receiving information from external parties as a type of status acquiring as well. This functional division keeps our AD model simple without compromising its functionality. The edges between nodes encapsulates the workflow of active documents, as we describe next.

Execution: When executed, a node in the graph effects the corresponding type of activity embedded. For example, an action node may emit a configuration change that adds a BGP neighbor setup on a router, while a condition node may check the

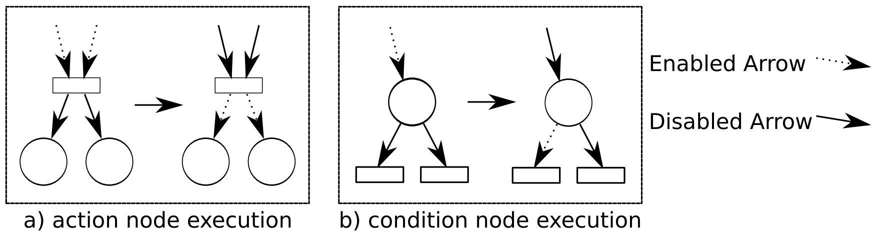


Figure 3.2: Active document node execution

BGP neighbor status and verify if the session is established. The execution of action and condition nodes may result in calling a set of APIs provided by the execution engine, which will be described later, to interact with devices or external parties, as shown in Table 3.1. For example, an action node should call `CommitConfigDelta()` by specifying the target router and configuration to change.

The progression ordering and dependency among these activities is modeled as the arrows between nodes. An arrow from node a to node b represents a happen-after relationship during execution. The basic execution mechanisms of active documents are shown in Figure 3.2. Each arrow is marked as either *enabled* or *disabled*.[1] An action node is executed only if *all* of its incoming arrows are enabled. After execution, all incoming arrows of the action node are changed to disabled, while all outgoing arrows are marked as enabled (shown in Figure 3.2-a). A condition node is executed if *one* of its incoming arrows are enabled. After executing, one of the enabled incoming arrows is switched to disabled, and *only one* of outgoing arrows is enabled based on the status processing result performed of the condition activity (shown in Figure 3.2-b). By using the structural elements [111] shown in Figure 3.3, Petri net is capable of modeling generic and complex workflows, fully covering the compositional mechanisms from MOP documents. These paradigms are capable of handling all the composition mechanisms we discussed in §3.1. We show how to design ADs to enable the automation of different realistic network management tasks in §3.4.

To allow reuse, the activities associated with the nodes in an AD are stored as templates. For example, a condition node that checks BGP session establishment would

[1]Petri net executes by passing tokens between places through transitions, which is equivalent to enabling and disabling arrows in AD execution.

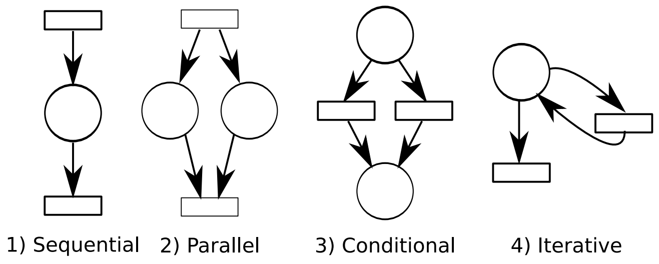
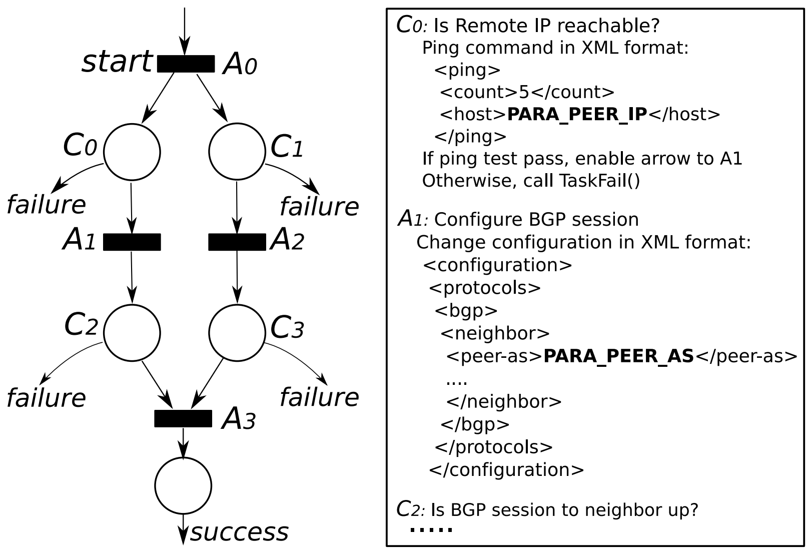


Figure 3.3: Active document design paradigms



```
C0: Is Remote IP reachable?
    Ping command in XML format:
      <ping>
       <count>5</count>
       <host>PARA_PEER_IP</host>
      </ping>
    If ping test pass, enable arrow to A1
    Otherwise, call TaskFail()

A1: Configure BGP session
   Change configuration in XML format:
    <configuration>
     <protocols>
      <bgp>
       <neighbor>
        <peer-as>PARA_PEER_AS</peer-as>
        ....
        </neighbor>
       </bgp>
      </protocols>
     </configuration>

C2: Is BGP session to neighbor up?
  .....
```

Figure 3.4: An example active document

be specified as “Check BGP session to `PARA_PEER_IP` on router `PARA_TARGET_DEVICE`”, where the two placeholders are replaced during execution with actual values specified in the execution task.

Besides choosing a follow-up action, an executed condition node can decide that the management task has succeeded or failed. In these cases, the API functions `TaskSucceed()` and `TaskFail()` are called respectively, similar to `exit()` statement in C programs. The execution engine stops the execution task and handles the failure if `TaskFail()` is called.

Example: Figure 3.4 shows an active document that can be used to set up a BGP session between two routers. Action A_0 is not performing any activities, except to create two parallel branches to operate on two routers. Conditions C_0 and C_1 launch a `ping` on both routers to see if the other end is reachable. The task fails if either `ping` fails. Otherwise, actions A_1 and A_2 are executed to add the actual BGP neighbor configurations on both routers. Then, conditions C_2 and C_3 check if the configured BGP session is up on both ends; only if both condition checks succeed, can the action in A_3 be executed. A dummy action node is used at the head and a dummy condition node at the tail of the active document, if necessary. For example, the bottom condition node in the example does not perform any activities but directly calls `TaskSucceed()`.

3.2.2 Execution Task Composition

While active documents describe the workflow of management tasks in the abstract, simple execution tasks are used to specify specific instantiations of ADs by replacing all template placeholders with appropriate parameter assignments, as shown in Figure 3.1. For example, an active document to configure an IP address on a particular interface of a router needs the parameters of router IP address, interface name and IP address to set. The parameters are usually generated from external network related databases [41, 53]. A practical concern is that the database could be out-of-sync with the actual network state, which is a problem for existing management

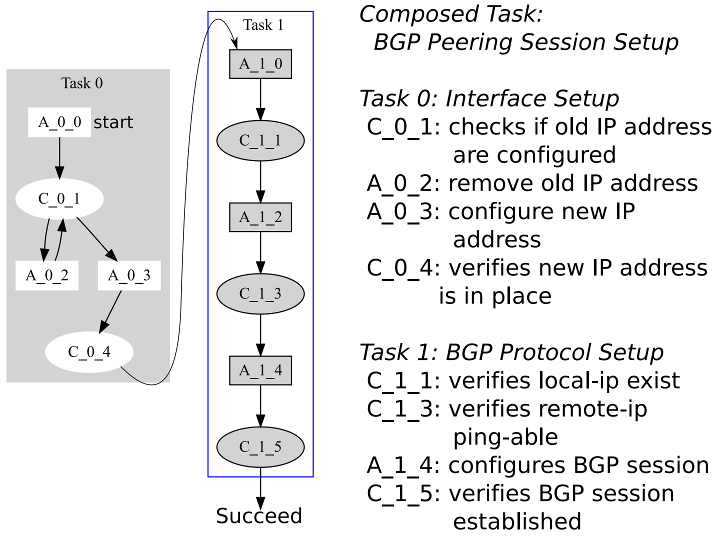

Task 0
A_0_0
start
C_0_1
A_0_2
A_0_3
C_0_4
Task 1
A_1_0
C_1_1
A_1_2
C_1_3
A_1_4
C_1_5
Succeed
Composed Task: BGP Peering Session Setup
Task 0: Interface Setup
C_0_1: checks if old IP address are configured
A_0_2: remove old IP address
A_0_3: configure new IP address
C_0_4: verifies new IP address is in place
Task 1: BGP Protocol Setup
C_1_1: verifies local-ip exist
C_1_3: verifies remote-ip ping-able
A_1_4: configures BGP session
C_1_5: verifies BGP session established

Figure 3.5: Sequential task composition

methods as well. To alleviate the potential negative impact, ADs can be designed to always perform in-sync checks at the start of execution.

We describe three composition mechanisms for generating composed execution tasks in detail next.

Sequential: Figure 3.5 shows how we apply a *sequential ordering* onto two tasks (failures are not shown for simplicity), namely the link setup task as Task 0 and the BGP configure task as Task 1, resulting in a "BGP peering session setup" task as the composed task. A strict ordering is enforced: a task is only executed when the previous task succeeds; the composed task succeeds if the last task succeeds. Note that node `C_0_4` originally calls `TaskSucceed()` if task succeeds. This API call is replaced with an edge pointing to `A_1_0` to stitch the two tasks together.

Meta structure: Figure 3.6 shows an example meta structure for automating operations before, during and after an execution task. The composed task starts by taking a snapshot of the running status of the target device. At the same time, a loop structure (shown on the right) is used to continuously monitor network running status via `ping` or dedicated traffic generation engine. If a network disruption is detected, `TaskFail()` is called to perform roll back immediately. When the wrapped task finishes, another snapshot of the network status is taken and compared with the

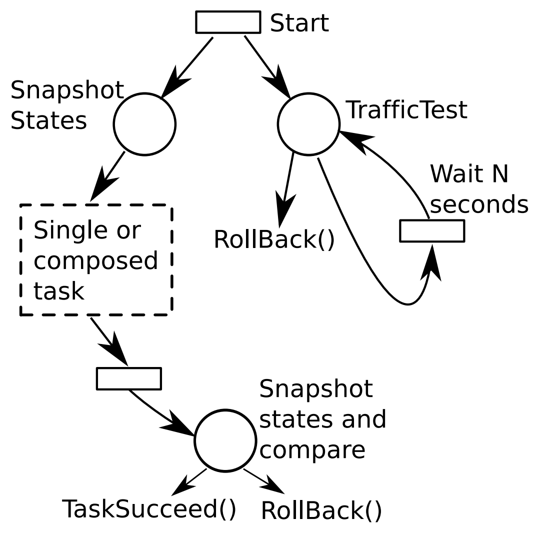


Figure 3.6: Wrapper construct for concurrent traffic disruption detection and state diffing

previous one. Failure is reported if certain criteria are not met, *e.g.,* some BGP sessions fail to establish. This is particularly useful for supporting software or hardware upgrade tasks.

Parallel with policy enforced: Figure 3.7 shows how several tasks are composed to execute in parallel but with a network-aware policy enforced. Each dotted box contains the original AD of each task for composition (only one action node and one condition node are shown for simplicity). A *policy condition node* (or policy node) is added to point to every action node in each task. Once imposed, the policy node becomes an additional condition to satisfy for each action node, thus it can embed a network-aware decision logic that goes beyond individual tasks. We show later how generic policies, like "prevent network partitioning" and "prevent link overloading", can be implemented. Policy nodes are usually written by network experts and can be directly used to regulate generic execution tasks. This further lowers the bar for AD creation, as existing policies can be applied to carry out the more complicated decision logic.

The policy node does *not* simply serialize the actions in each tasks. There are multiple arrows pointing from the dummy start action node to the policy node. This effectively adds multiple enabled arrows to the policy node, so that the policy node

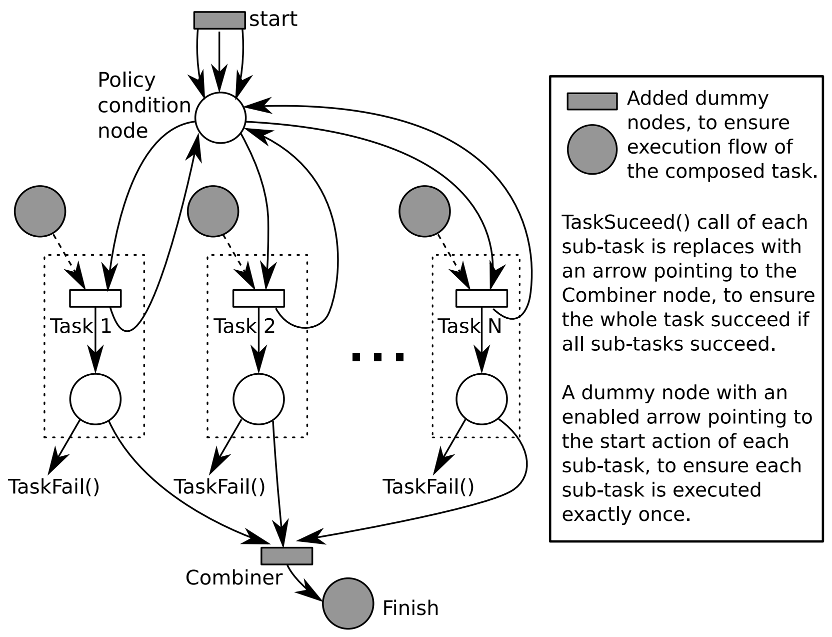

start
Policy condition node
Task 1
Task 2
Task N
TaskFail()
TaskFail()
TaskFail()
Combiner
Finish
Added dummy nodes, to ensure execution flow of the composed task.
TaskSuceed() call of each sub-task is replaces with an arrow pointing to the Combiner node, to ensure the whole task succeed if all sub-tasks succeed.
A dummy node with an enabled arrow pointing to the start action of each sub-task, to ensure each sub-task is executed exactly once.

Figure 3.7: Policy enforcement in parallel composed tasks

does not need to wait for an action to finish before enabling another action, allowing multiple action nodes to be executed concurrently, if permitted by the policy. As shown in Figure 3.7, when the action node is done, it would enable the added arrow pointing back to the policy node, such that the policy node can launch again to select the next action node to run, if there is any.

The active document design together with the sophisticated composition support completes the picture of PACMAN's capability for fulfilling automated network management. It fully satisfies the requirements of automating MOP documents, but also goes beyond that by imposing network awareness without additional manual work.

3.2.3 Execution Engine

An execution engine runs all execution tasks like separate programs, by providing three main functions:

Provide execution environment: Like an operating system, the execution engine allows each running execution task to interact with physical devices or external entities through a set of API calls. The execution state of an execution task is maintained as a collection of enabled arrows by the execution engine. To start an execution task, a dummy condition node is added with an enabled outgoing arrow pointing to the start action node. This effectively allows the start action node to activate the whole execution task. The enabled arrows for each execution task is updated after a node execution finishes. An execution task finishes by calling `TaskSucceed()`.

Handle API calls: To support the most common `CommitConfigDelta()` and `QueryDeviceStatus()` calls, the configuration delta or status query template is first parametrized, based on the input parameters to the execution task, and then fed into the proper device, which is usually indicated in the input parameters as well. If the configuration change is accepted by the device, the API call is done. A configuration delta may not be accepted by the target device for various reasons, *e.g.,* command syntax error, missing reference links, or device errors. In these cases, `TaskFail()` is called by the execution engine for the execution task. `NotifyEntity()` and `QueryEntity()` are invoked based on the node specification. The message or query should be parametrized as well. `QueryExecutionState()` returns the list of enabled arrows and the current nodes to the calling condition node, mostly used in policy nodes that need to reason about the execution state.

Handle failures: An execution task fails if `TaskFail()` is called. The execution task is stopped immediately, and a snapshot of the execution status is taken, which consists of the result for `QueryExecutingState()` along with the condition node that reports the failure. These are recorded for future manual inspection. The execution engine allows different failure mitigation strategies. By default, the effect of the whole execution task is rolled back. To support this, the execution engine maintains an execution history for each task. The rollback action is done by undoing all the configuration changes made based on history information. If external entities are notified in any form, revoke messages are sent. Other mitigation strategies may be also be used, such as no-rollback, partial rollback or redo.

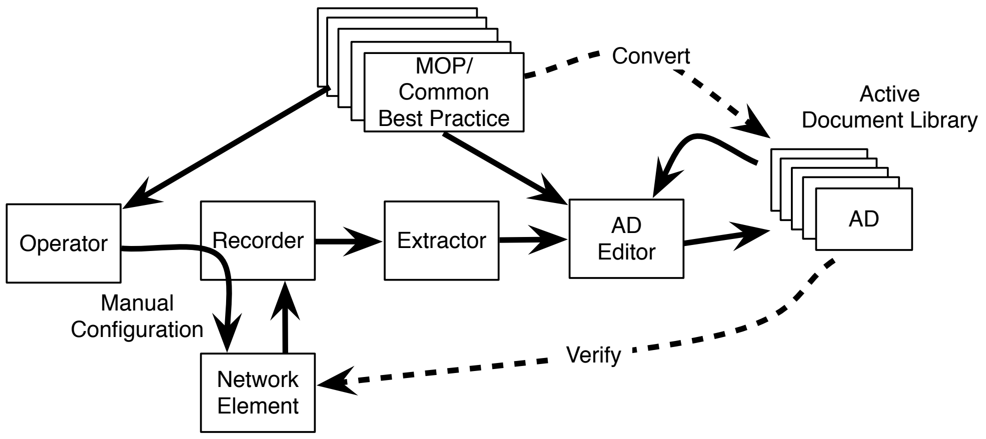


Figure 3.8: AD Creation Framework

3.3 Creating Active Documents

In Section 3.2 we abstractly described the creation of active documents and their derived execution tasks. We now describe a practical framework to assist the rapid creation of ADs. This AD creation framework is depicted in Figure 3.8. There are two requirements for building this creation framework: i) high usability, which can lead to quick adoption; ii) high expressiveness, so that the generated ADs describe management tasks accurately. The creation can be assisted from network operational logs, like TACACS. As shown in Chapter II, we can indeed extract out performed command sequences. Such creation framework, however, is still necessary, because operational logs may not capture all the necessary information, for example the results of status checking commands. An AD is created via the following three steps, allowing quick transformation from MOPs to ADs, forming an AD library:

Task observation: An operator or AD designer, guided by the MOP documents or with a common best practice in mind, performs a network management task on a set of network elements, typically in a testbed environment, for example, ShadowNet as we describe in Chapter V. Operators directly operate on a set of target devices using their familiar tools and mechanisms, *e.g.,* spawn multiple SSH sessions to CLI, while we use a *recorder* to transparently capture the full interaction. We record: i)

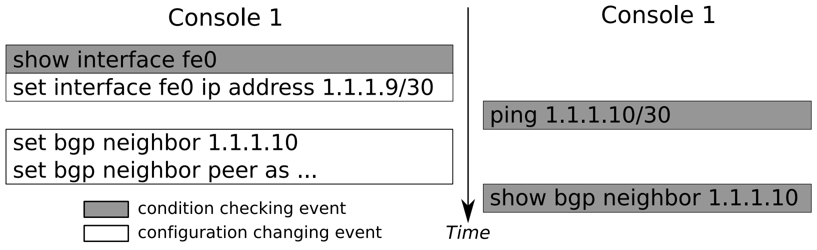

Console 1
show interface fe0
set interface fe0 ip address 1.1.1.9/30
set bgp neighbor 1.1.1.10
set bgp neighbor peer as ...
condition checking event
configuration changing event
Time
Console 1
ping 1.1.1.10/30
show bgp neighbor 1.1.1.10

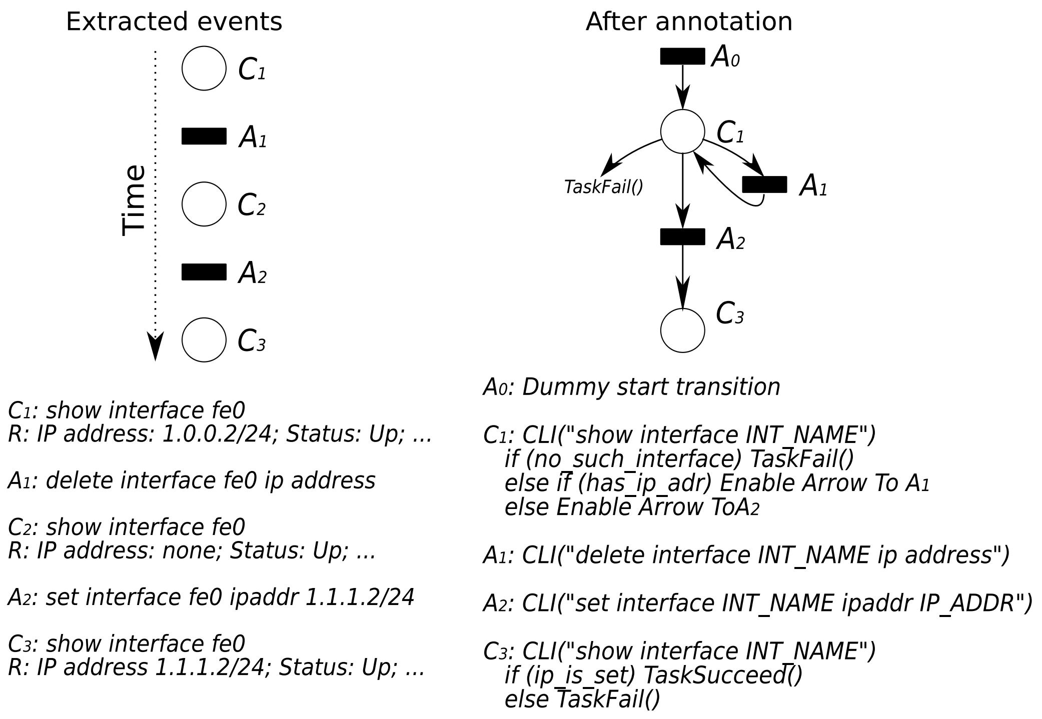
Figure 3.9: Event extraction from console log

performed activities through CLI, *e.g.,* modify configuration, change protocol state (such as BGP session reset), acquire network status; ii) device response and internal states, *e.g.,* displayed network status, emitted SNMP trap messages and device log messages. The recordings are tagged with timestamps.

Event extraction: Action and condition activities are extracted from the recordings automatically by an *extractor*. Contiguous configuration changes are grouped together as a single event, as long as there are no condition activities in the middle, as shown in Figure 3.9. Similarly, repetitive condition checks with the same result are combined. When device status is inspected in the CLI, we correlate logs from other sources, such as SNMP or device log message, to augment the condition event. That is, we allow the operator to specify the decision logic based on those information sources as well.

Operator annotation: The events extracted from previous step are presented as action or condition nodes in an *AD editor*, pending operators' annotation to complete the AD generation. The operator has to specify i) the parameters that are specific to tasks, so that we abstract them as placeholders for future re-use, ii) the workflow logic by drawing arrows between nodes, *e.g.,* identify two parallel branches, iii) the information source and decision logic in each condition node, iv) external synchronization events, since they are not recorded, v) additional events for hypothetical scenarios, *e.g.,* failure detection (condition) and response (action).

Figure 3.10 shows an example of operator annotation. The left side shows the processed CLI log by events generation. C indicates a condition checking, followed by the response (R) from the device; A indicates an action performed by the user. On



```
C1: show interface fe0
R: IP address: 1.0.0.2/24; Status: Up; ...

A1: delete interface fe0 ip address

C2: show interface fe0
R: IP address: none; Status: Up; ...

A2: set interface fe0 ipaddr 1.1.1.2/24

C3: show interface fe0
R: IP address 1.1.1.2/24; Status: Up; ...
```

```
A0: Dummy start transition

C1: CLI("show interface INT_NAME")
   if (no_such_interface) TaskFail()
   else if (has_ip_adr) Enable Arrow To A1
   else Enable Arrow ToA2

A1: CLI("delete interface INT_NAME ip address")

A2: CLI("set interface INT_NAME ipaddr IP_ADDR")

C3: CLI("show interface INT_NAME")
   if (ip_is_set) TaskSucceed()
   else TaskFail()
```

Figure 3.10: Example of operator annotation

the right side, the annotation result is shown: all IP addresses and interface names are replaced by generic placeholders, such as `INT_NAME`; the nodes are connected by arrows, indicating execution flow; for each condition nodes, the actual decision process is formally specified in the form of a sequence of `if-then-else` statements, which process the retrieved status information and determines a follow-up action. The same framework for annotation can be used to directly create or modify ADs, *e.g.,* by a skilled designer. This creation process only needs to be done once, as the generated ADs can be re-used and composed in the future to fulfill similar or even more complex tasks.

One limitation of this creation framework is that the recorded AD reflects the operations on a *fixed* amount of devices. As such, tasks like "to enable IS-IS on *all* routers" cannot be captured by a single AD, because the number of routers in the creation environment does not match that in the production environment. To overcome this problem, it is advised that the operators create ADs that are smaller and more specialized, *e.g.,* "to enable IS-IS on a single router", and use the composition

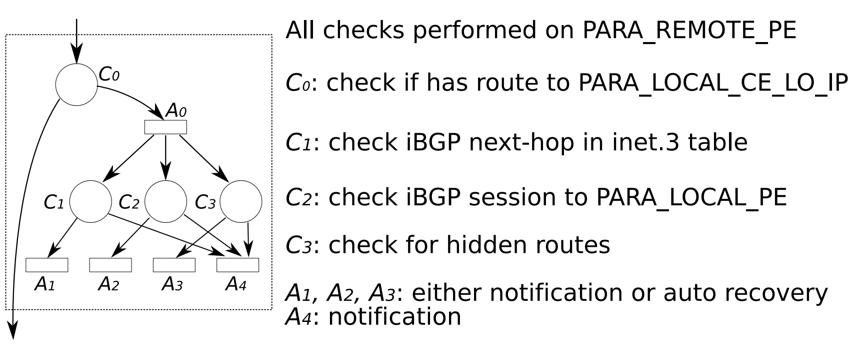

All checks performed on PARA_REMOTE_PE
C_0: check if has route to PARA_LOCAL_CE_LO_IP
C_1: check iBGP next-hop in inet.3 table
C_2: check iBGP session to PARA_LOCAL_PE
C_3: check for hidden routes
A_1, A_2, A_3: either notification or auto recovery
A_4: notification
C_0
A_0
C_1
C_2
C_3
A_1
A_2
A_3
A_4

Figure 3.11: Layer-3 VPN diagnostic AD

mechanisms to stitch multiple ADs together. For more sophisticated tasks, *e.g.,* "operate on all the routers that meet a certain criterion", the AD designer can design ADs to operate on one device, while encoding the selection criteria into the beginning of the AD, so that the operator can simply compose an execution task that works on all routers. Another solution is to leverage on external databases to determine the set of devices to operate on.

3.4 Case studies

In this section, we use several realistic examples to show how active documents are used to perform complex yet automated network operations in the PACMAN framework. Since a quantitative measurement of improvement is hard, we qualitatively evaluate the benefit of PACMAN comparing to existing approaches.

3.4.1 Fault Diagnosis

Active document is an ideal candidate for automating the fault diagnosis process. Condition nodes can be used to retrieve relevant information from various devices and then reason about the symptom. The outgoing arrows of condition nodes correspond to different diagnosis results and may lead to additional steps.

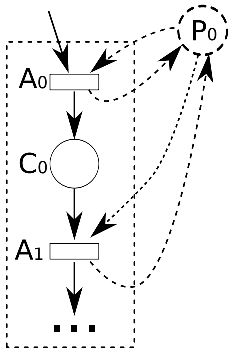
Figure 3.11 shows a portion of an active document that is used to diagnose layer-3 VPN connectivity. This AD is converted from a MOP provided by a major router vendor [11]. The whole diagnosis procedure checks multiple routers to see if VPN routes can properly propagate from a local customer edge (CE) router, through the local provider edge (PE) router, and reach the remote PE router and remote CE. The example shows the portion that diagnoses if routes propagate correctly from local PE to remote PE. C_0 logs into the remote PE router to check if the loopback IP address of the local CE router is seen on its layer-3 VPN routing table. If true, it means that routes from the local CE correctly propagate to remote PE, thus this portion of AD can be bypassed. Otherwise, A_0 is executed to spawn multiple tests to further diagnose the problem: *e.g.,* C_2 checks on remote PE if the iBGP session to the local PE is properly established; if not, the problem is found, leading to the execution of A_2 which either starts another sub-task to automatically fix the BGP session or calls `NotifyEntity()` to contact an operator about the diagnosis result.

The flexible composition capability provided by active documents allows network-wide fault detection, fault diagnosis and fault recovery in a closed loop by stitching appropriate ADs, reducing human involvement significantly, as such composition is only done once and can be reused in the future. The state of the art in automated fault diagnosis relies on router vendor support [10] to execute diagnosis scripts automatically when certain condition is met. This support is limited to a single device, while PACMAN can easily correlate and reason about status from devices across the network.

3.4.2 Link Maintenance

Figure 3.12 shows a planned maintenance task with enhancement by applying a network-aware policy. The dashed box contains part of the original active document: action node A_0 increases the OSPF metric of the target link to cost it out; A_1 brings the link down by changing configuration and, at the same time, notifies field operators, signaling them to start the on-site maintenance on related physical interfaces. The link bring-up procedure is similar thus ignored for brevity.

This task involves OSPF weight change and interface shut down thus has the



FO: Field Operator

A0: CommitConfigDelta(...) // OSPF link cost-out

C0: Dummy node

A1: CommitConfigDelta(...) // shut down interface
NotifyEntity("FO", "proceed with PARA_INT_NAME ")

P0: delay action that will cause predicted link overload,
delay shut down a link when there is still traffic

Figure 3.12: Planned maintenance AD

potential of negatively impacting live traffic. Current solutions rely on operators to manually predict and avoid negative impact, a usually slow and unreliable process, which is particularly undesired for such tasks with stringent requirements on timing and reliability. In PACMAN, we can impose a policy node, like P_0, to enforce a high-level policy that automates a network-aware decision process for minimizing traffic disruption. P_0 is composed with the original AD with added arrows pointing to all action nodes. In essence, P_0 is a condition node that reasons about network-wide states, such as traffic demand matrix, existing OSPF weights, *etc.*, and makes decisions by enabling the arrows to appropriate action nodes. In effect, P_0 will not allow A_0 (OSPF cost-out) to proceed, unless the estimated traffic shift caused by A_0 would not overload other links; P_0 will not allow A_1 (interface shut down), unless i) the routing has converged and ii) indeed no traffic is flowing through the link. Given the composition capability, P_0 can be used to regulate arbitrary tasks without additional manual work. This is especially useful for carrying out simultaneous maintenance tasks, which are hard to coordinate by operators and may cause significant network downtime, *e.g.,* a partitioned network.

Besides using network-aware policy control, this maintenance job can also take advantage of external reasoning platforms, such as a traffic engineering planner [57]. For example, C_0 can query the planner if it is permitted to shut down the interface. This allows PACMAN to take full advantage of existing infrastructures.

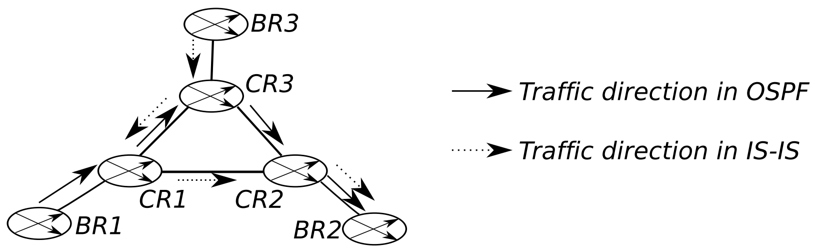


Figure 3.13: A simplified ISP Backbone

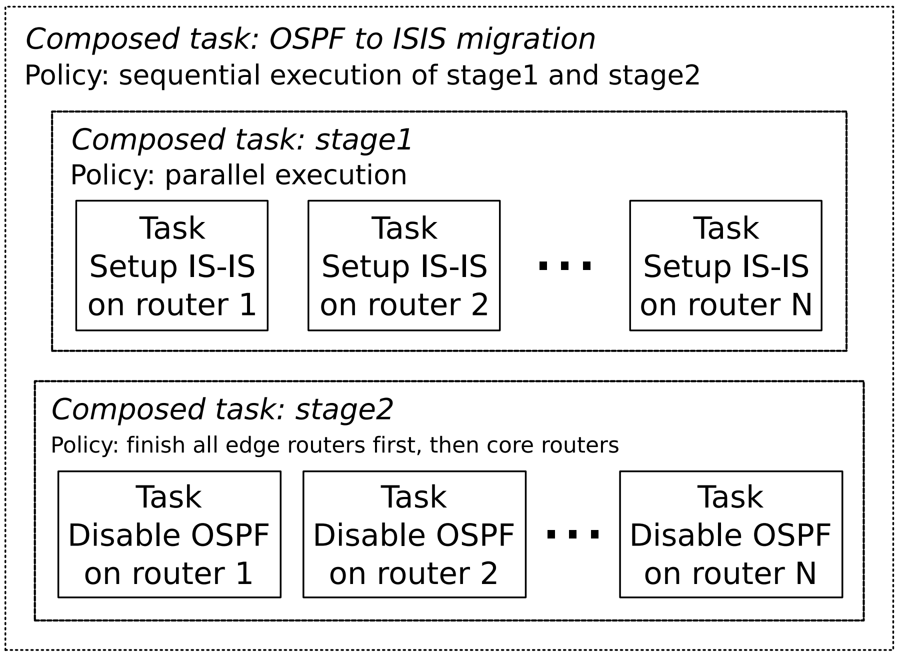

Composed task: OSPF to ISIS migration
Policy: sequential execution of stage1 and stage2
Composed task: stage1
Policy: parallel execution
Task Setup IS-IS on router 1
Task Setup IS-IS on router 2
...
Task Setup IS-IS on router N
Composed task: stage2
Policy: finish all edge routers first, then core routers
Task Disable OSPF on router 1
Task Disable OSPF on router 2
...
Task Disable OSPF on router N

Figure 3.14: Task design for OSPF to IS-IS migration

3.4.3 IGP Migration

Many ISP networks have performed IGP migration for a variety of reasons [37]. IGP migration is a challenging task as IGP is deep down the dependency stack — many other network services and protocols depend on it. Let us consider the task of migrating a network from running OSPF to IS-IS (actually performed by two large ISPs previously [19, 61].)

The migration process first enables IS-IS (with a lower preference) in the network and then disables OSPF. One of the challenges is to prevent transient forwarding loops. Consider a simplified ISP topology in Figure 3.13. After IS-IS is enabled and running together with OSPF, it is possible that link $CR1 \rightarrow CR2$ has a high weight in OSPF and $CR2 \rightarrow CR3$ has a high weight in IS-IS. The traffic from $BR1$ to $BR2$ goes from $BR1 - CR1 - CR3 - CR2 - BR2$, as OSPF is still the preferred IGP. If OSPF is disabled first on $CR3$, $CR1$ still forwards traffic to $CR3$ because $CR1$ still runs and prefers OSPF, and the shutdown of $CR3$'s OSPF will not be detected after a timeout. $CR3$, on the other hand, switches to IS-IS immediately, thus starts to forward traffic via the path $CR3 - CR1 - CR2 - BR2$. As a result, packets would bounce between $CR1$ and $CR3$, until OSPF re-converges. A simple solution to prevent this in common ISP setups is to disable OSPF on all edge routers first and then on all core routers [19]. This enforcement, however, is unreliable and requires much manual effort in existing approach.

PACMAN automates this process using a composed execution task, with two major stages, as shown in Figure 3.14:

Stage 1: for each router, i) configure `iso` layer on all interfaces; ii) verify `iso` is enabled; iii) configure IS-IS protocol to run with a lower preference than OSPF; iv) verify the IS-IS protocol has learned all the routes as OSPF does.

Stage 2: for each router, i) deactivate OSPF; ii) verify no loss of routes; iii) remove OSPF config, adjust IS-IS preference.

Both stages are also composed tasks, executed in sequential order. For stage1, all sub-tasks are executed in a simple parallel fashion, because they do not interfere with each other. For stage2, all sub-tasks are executed in parallel, with additional policy enforcement (ordering constraint) to avoid forwarding loops. We will illustrate in the §3.6.1 the effectiveness and correctness of this composition.

3.5 Implementation

In this section, we briefly describe our implementation of the PACMAN framework. Two major components are the AD creation framework and the execution engine. All implementations were performed in Java, and we mostly focus on Juniper routers due to availability in our test environment, but our methodology extends to other network devices.

3.5.1 Active Document Creator

Our implementation of AD creator contains several pieces, some of which leverage existing software packages. We customize `screen` and `script` Linux commands such that SSH sessions can be made simultaneously to the same or different devices while each session interaction being recorded with timing information. SNMP messages and device log messages are constantly being monitored and later retrieved to correlate with console commands based on timing. The annotation is done in a Java-based GUI. For each action or condition node, a pop-out window allows the operator to specify the parameters. For condition nodes, a chain of tests is specified to represent an if-then-else decision making. Each test need to specify: an information source, which could be the result of a status-checking command, SNMP or device logs, or previously saved information; a predicate as test body, which can be as simple as string matching, or as complicated as an XML query — Juniper routers support XML-based interaction for retrieving device status; a test result, which can be an arrow to enable or calling an API.

3.5.2 Execution Tasks and Execution Engine

A simple execution task is created from an AD and a parameter assignment. A quick sanitization process is performed to make sure that enough parameters are specified and the values conform to the parameter types. When composing execution tasks together, the node names and parameter names used in ADs of different sub-tasks are renamed to avoid confusion. For example, node `M` is renamed as `N_M` where `N_` is a prefix added for all the nodes of the sub-task. (This renaming effect can be seen in Figure 3.5.)

Figure 3.15 shows the high-level architecture for our execution engine. Each running execution task is associated with a list of enabled arrows. The execution engine

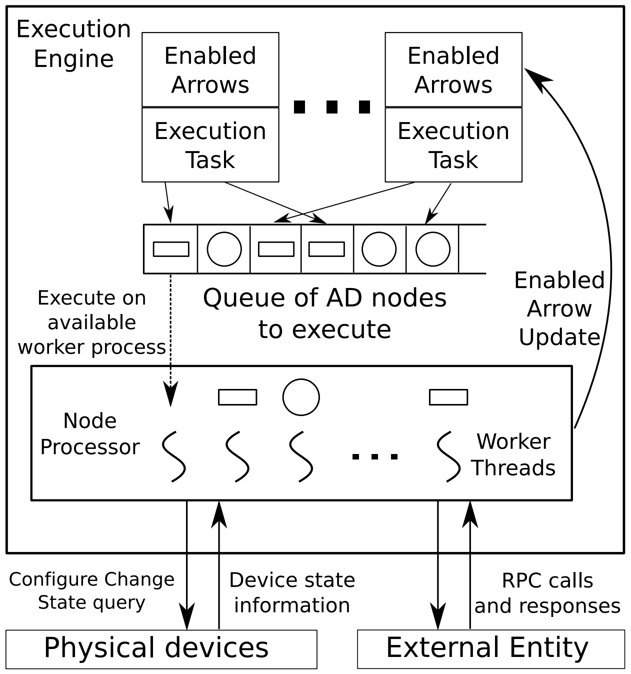


Figure 3.15: Execution engine architecture

scans all execution tasks periodically. Based on the enabled arrows, the nodes that are ready to execute in each executed task is added into a queue waiting for execution.

A *node processor* is responsible for actual execution of the nodes. Multiple worker threads are spawned to handle concurrency. If a worker thread is available, a node is fetched from the waiting queue. Rather than picking nodes from the head of the queue, a node is randomly selected from the queue, to ensure fairness and avoid potential live lock. To execute a node, parameter values are copied from the execution task to replace the parameter placeholders in the node.

To handle `CommitConfigDelta()` and `QueryDeviceStatus()` in a node, the worker thread contacts physical devices specified via either CLI or NetConf interface. Connections to recently contacted physical devices are cached and reused to reduce connection establishment overhead. Configuration changes made to the same device are serialized to avoid potential conflicts. `QueryEntity()` and `NotifyEntity()` are simple wrappers to external scripts. For example, executing *"NotifyEntity('mail', 'a@b.com', 'done')"* invokes a shell command *"./mail.sh a@b.com done"*.

3.5.3 Programming Policy Nodes

Algorithm 1 Implementation of the prioritization policy node

```
Require: AdSpec $AS$, ExecutionState $ES$, NetworkState $NS$, DemandMatrix $DM$
1: $W \leftarrow GenWaitingNodeList(AS, ES)$
2: for action $n$ in $W$ do
3:     $NewState \leftarrow NS$ applies action of $n$
4:     calculate connectivity matrix and traffic on each link based on $NewState$ and $DM$
5:     if in $NewState$ network is not partitioned and no overloaded link then
6:         return $n$
7:     end if
8: end for
9: return $null$
```

Policy nodes can be much more complicated than the regular condition nodes created via the AD creator. In fact, we allow policy nodes to be written in Java and handled using the same execution engine. When executed, a policy node first identifies a set of action nodes that have all other pre-conditions satisfied and are waiting for its permission to proceed. Among these nodes, the policy node can choose one from them to allow its execution by enabling the final arrow. It is possible for the policy node to decide that none of those actions should proceed at the moment. On the other hand, a policy node sometimes need to consider the action nodes that might be executed in the future, because it might be a better choice to execute them rather than currently ready nodes. Determining these two sets of nodes can be done via flow analysis based on the graph structure of the composed AD and execution state. We provide generic helper functions to ease the development process.

Here we describe the sketch of a policy node, which specializes in avoiding network partitioning and traffic overloading caused by arbitrary simultaneous network tasks, shown in Algorithm 1. The first line uses a provided function to generate a list of action nodes that are waiting for permission. Line 2-8 iterate through all such nodes. For each action node being considered, the resulting network state $NewState$, including reachability and routing table, is calculated based on the current network state and the configuration change embedded in the action node. If a partitioned network is detected, the action node will not be permitted. Combining the traffic demand matrix and new routing table, an action is permitted if it does not cause

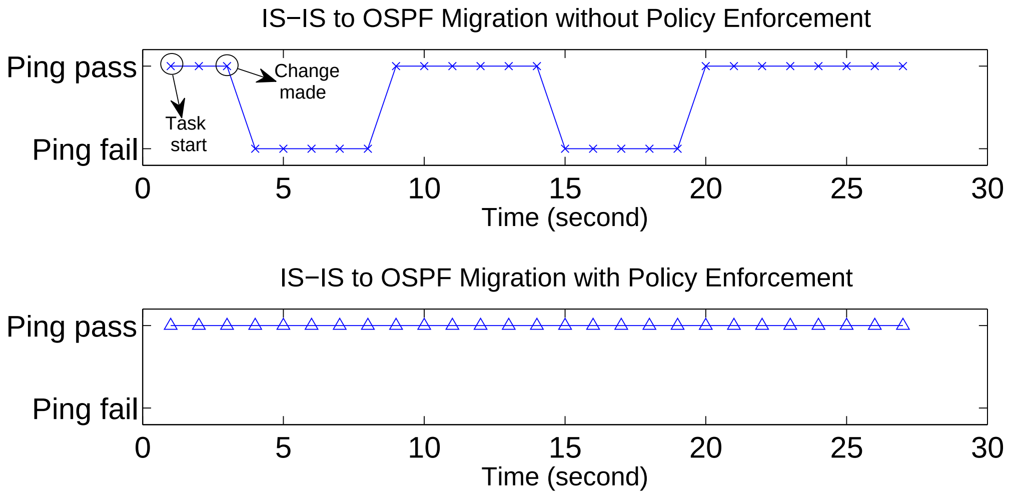


Figure 3.16: Effectiveness of policy enforcement

other links to overload or exceed some pre-defined threshold, *e.g.*, 90% utilization ratio.

3.6 Evaluation

We evaluated our prototype implementation to demonstrate its effectiveness in preventing operational errors and ensuring efficient configuration management, which scales well with network size.

3.6.1 Network-awareness Support

To exemplify the effectiveness of policy enforcement, we perform stage 2 of the IGP migration task (disabling OSPF on all routers) in two different ways: i) Through multiple individual tasks, each of which disables OSPF on one router, processed by the execution engine concurrently, mimicking the effect of several operators working on different part of the network simultaneously, yet unaware of the potential problem. Note that such concurrent and collaborative management is very common in large ISPs, as we studied in Chapter II. ii) Through one composed task, using the prioritization policy to ensure edge routers are first updated before changing the configuration of any core routers.

We used six Juniper routers on our evaluation platform described in Chapter V, connected as shown in Figure 3.13. The experiments were performed on the network state after stage 1 of IGP migration had finished (IS-IS was configured as the less-preferred IGP, while OSPF was still running as the preferred IGP). One machine connected to $BR1$ was sending `ping` to another machine connected to $BR3$ during the migration period. Link weights of OSPF and IS-IS were intentionally tweaked to create the situation discussed in §3.4.3.

Figure 3.16 shows the result. When individual tasks were executed in parallel, repeating this experiment multiple times showed that when the task working on $CR3$ is executes first a forwarding loop was indeed created as shown in the top figure: the connectivity was temporarily lost for a few seconds (the amount of time to commit configuration changes on Juniper routers) after the task started. The connectivity was resumed and lost again before it eventually stabilized, mostly due to the complex interaction of the two IGP protocols. In contrast, the composed task using prioritization policy did not experience any problems, as shown in the bottom figure.

3.6.2 Automating Network Operations

We again use the IGP migration task in an ISP depicted in Figure 3.14 to estimate the time saving by using PACMAN to automate network operations. For comparison purposes, one of the authors who is proficient in network management and router configuration performed the migration task. That author performed the task several times beforehand for training purposes and then reported the lower-bound estimate of how long a sub-task would take when executed manually (we expect the actual performance numbers from real operators to be quite similar).

The amount of time to manually perform configuration change on the routers takes less than 2 minutes each, thus ideally 12 minutes to finish six routers. Interestingly, the total amount of time to finish the migration task for all routers takes no less than 25 minutes, due to additional network status verification and inter-device synchronization. In contrast, PACMAN finishes the whole migration task within 2 minutes - 90% of time on effecting configuration change and acquiring status via NetConf and the rest on internal processing. If we extrapolate to a network of 100 routers, the manual operation time is over 400 minutes, exceeding an entire maintenance window, which typically lasts for at most three to four hours. Even worse, when the operated network is considerably larger, the manual operation time is unlikely to scale linearly, despite the potential use of automated scripts, due to more complicated network status verification and additional synchronization between involved human operators. In fact, the IGP migration processes documented online [19] took several maintenance windows across 3-5 days to finish. Automated scripts can offer limited help, as they cannot be easily coordinated. For PACMAN, since all the verification process are accurately modeled and automatically carried out, it can easily scale with the network size.

3.6.3 System Constraints

The execution engine directly interacts with physical devices. Out-of-band access, which is standard in ISP environment, provides a more reliable connectivity channel, but the bandwidth is limited, ranging from 9.6kbps serial console, 56Kbps modem line to 1.5Mbps T1 connection. Router configuration in XML format is usually tens or hundreds of kilobytes. Assuming a T1 connection, it may take around hundreds of milliseconds to transfer a complete configuration file. Fortunately, most management activities can be performed in-band where bandwidth is not an issue.

We performed some micro-benchmarks to investigate resource constraints the server that runs the execution engine. On a server with 2.5G Intel core 2 duo CPU, it takes about $360\mu s$ to load a 2KB XML file with 86 lines, describing 10 routes in the routing table. It takes about $950\mu s$ to perform an XPath query to count the number of routes described in the XML file. The processing time should be on the order of hundreds of milliseconds to handle 10000s routes. The processing power may become a bottleneck when the reasoning activity becomes significantly more complicated. This can either be mitigated by using multiple execution engines for load-balancing, or offloading some reasoning logic to programmable routers.

3.7 Summary

In this chapter, we proposed the PACMAN platform, aiming to automate existing network management operations and enabling the adoption of new network-wide operational practices. The key intuition behind our work is to use the right level of abstraction which is both close enough to current management approach, thus enable quick adoption, general enough to capture the complexity of existing approaches, and powerful enough to automate and augment them.

Towards the goal of building automated network management system, PACMAN uses the Active Document abstraction to systematically capture the dynamics of network management tasks. This abstraction allows the composition and execution at task level, thus raising the level of abstraction. The ability to integrate network-wide policies distinguishes PACMAN from device-centric support from vendors and task-oriented nature of MOPs.

We described the design and implementation of the PACMAN framework, and used realistic usage scenarios to show its effectiveness. As future work, we plan to corroborate with network operators for feedback and comments in order to further improve the usability and practicality of PACMAN. In particular, we aim at allowing more flexible creation and more programmable composition of active documents by improving the interaction between human operators and our system.

CHAPTER IV

A Declarative System for Automated Network Management

In the previous two chapters, we start from the low-level management practices, in terms of the actual commands executed in a network and the method of procedure documents that guide those executions, and build abstractions (DFA and Petri-Net) on top of them with the goal to automate and augment network management operations. In this chapter, we take a top-down approach to satisfy the same goal. Starting with the well-defined database model, we explore how the data model and database operations fit network management purpose.

The traditional usage of databases in network management is predominantly for archiving snapshots of network-related information to facilitate subsequent analysis, such as data mining for misconfigurations [84] or traffic patterns [83]. In contrast, we use the database notion as an abstraction layer that sits on top of the actual network. We build a system, called COOLAID (COnfiguring cOmpLex and dynamic networks AutomatIcally and Declaratively), which interacts with the underlying physical network and provides a database-like interface to facilitate network management operations.

While the database model provides a clean and unified abstraction to capture network-wide information and control network devices, the true power of our approach comes from a declarative logic-based language that we use to capture the domain knowledge in network management. In particular, the configuration requirements for distributed protocols, the dependencies among network functionalities, and the constraints for network-wide properties, can all be expressed accurately and succinctly in such a language. The captured domain knowledge can then be used in a seamless and automated fashion to manage the data in our database abstraction, which translates to the management of the physical network, while exposing simple database interfaces to network operators. We show that new network management primitives can be enabled, including network-wide reasoning, network configuration automation, and misconfiguration prevention, allowing operators to better manage their networks without being exposed to the overwhelming details.

Unlike PACMAN that captures and in some sense replays operational procedures, COOLAID represents an approach that captures domain knowledge in abstract, while leveraging an underlying system to reason and operate on a network following the abstracted knowledge. We expect COOLAID to enable a move towards higher formalism in representing domain knowledge from different stakeholders and role players (*e.g.,* device vendors, service providers, network management tool developers), so that such knowledge can be captured within the same framework and combined systematically to automate network operations by systems like COOLAID, fundamentally relieving the excessive burden on human operators.

We describe the design and implementation of COOLAID, and demonstrate the effectiveness and scalability of COOLAID in a realistic distributed network testbed and on other simulated large-scale topologies.

4.1 Motivation for Using a Database Abstraction

In this section, we explain in detail why database model is a good abstraction for network management.

First, the database model allows a unified abstraction to view and manipulate the network management related data. As we describe in Section 4.2.1, the database operations in COOLAID capture configuration manipulation and status checking, which are currently performed through a variety of interfaces. Operators have to deal with the troublesome details of CLI, NetConf [15], SNMP [43], device logs, *etc.* Each of these traditional management interface requires significant domain expertise to manipulate, while in COOLAID everything is done through database table queries and updates, which are straightforward and convenient.

Second, this abstraction allows a single access point for *all* the network management related data. Traditional management practice requires the operators to act as an aggregation unit for accessing, interpreting and reasoning across multiple data sources, which are usually in different formats and accessed with significantly different methods. Putting everything together not only simplifies network access, but also, and more importantly, allows management tools to combine and reason about multiple data sources without the need to explicitly dealing with the details. Note that single access point does not mean single-point failure. We explain in Section 4.6 how such an abstraction can be realized in a robust fashion.

Third, recent advances in declarative systems [86] make database abstraction more attractive. In particular, declarative languages are shown to be accurate and succinct in expressing network policies [68] and distributed protocols [88]. Rather than using declarative systems to replace how current network devices work [88], we use declarative language to build a knowledge plane that sits on top of the existing network infrastructure, and use the knowledge derived from declarative reasoning to guide management practice.

Finally, modern databases follow the famous ACID properties [66], *i.e.,* atomicity, consistency, isolation, and durability. We argue that these properties are actually very desirable in network management as well. In COOLAID, consistency translates to the network-wide properties that should always hold if the network is functioning correctly, such as all backbone nodes should be connected to each other. We show that by enforcing consistency, we can prevent misconfigurations (Section 4.2.3) and handle network dynamic events (Section 4.2.5). We provide detailed discussion regarding all four properties in Section 4.3.3.

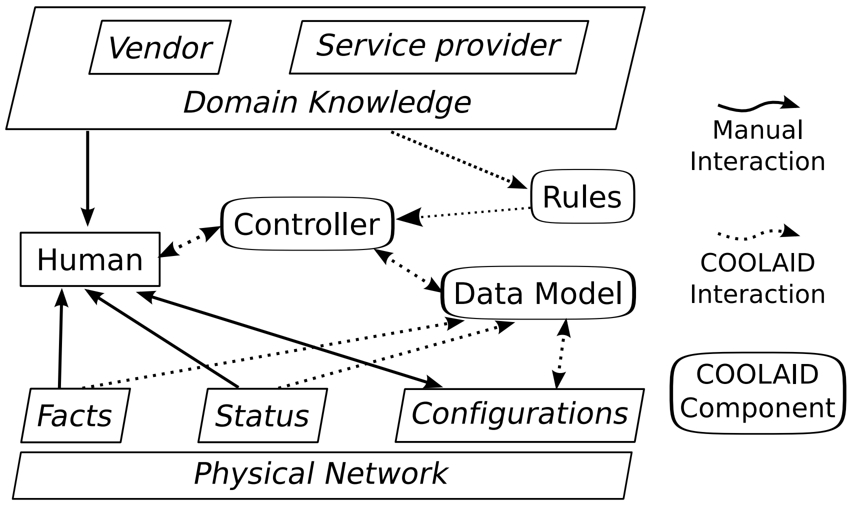


Figure 4.1: COOLAID vs. manual management

4.2 Managing Networks in COOLAID

Our key observation about the current network management practice is that the required domain expertise is unfortunately not captured in a systematic manner for the purpose of re-use and automation. Therefore, the current practice is inherently limited by human capability. As illustrated in Figure 4.1 with solid lines, human operators play a central role to absorb a tremendous amount of domain knowledge and directly interact with the underlying physical networks via a variety of rudimentary interfaces, *e.g.,* router CLIs. In particular, operators need to interpret network *facts* (*e.g.,* the list of routers, how they are connected, and customer service agreements), the current *configurations* and up-to-date network *status* (*e.g.,* if a BGP session is indeed established), based on which to reason about existing network-level functionalities (*e.g.,* if a VPN service is correctly configured for a customer) or realize additional features by changing configurations, and enforce network-wide constraints (*e.g.,* there must be a BGP full-mesh on core routers).

To minimize human involvement, a management system must satisfy three requirements: (i) it must systematically and formally capture the domain knowledge, in particular protocol dependencies and network-wide properties; (ii) the resulting representations should allow the system to expose high-level management primitives for automating management tasks; (iii) the system can re-use the knowledge base to assist operators and other network management tools in a network-wide (cross-device) manner.

In this section, we describe COOLAID, a network management system that satisfies these requirements. We first overview three key building blocks of the system, and then unfold the new network management primitives enabled by COOLAID. The enabling techniques for these primitives are described in §4.3, and our system implementation in §4.4.

4.2.1 COOLAID building blocks

Conceptually, COOLAID models a network of inter-connected devices as a distributed but centrally managed database, exposing an intuitive database-style interface for operators to manage the entire network. Figure 4.1 depicts the COOLAID building blocks with rounded-boxes, and their interaction with operators and the network using dotted lines.

Data model: The data model creates a logically centralized database abstraction and access point to cover all the traditional network management interfaces, which largely include reading and modifying router configurations, checking network status and provisioning data. The abstraction is designed to work with commodity devices, and interoperable with existing management tools. We define three types of base tables[1]: (i) *Regular tables* store fact-related data that naturally fit into a conventional database (*e.g.,* MySQL). (ii) *Config tables* store the *live* network configurations, in particular, router states that are persistent across reboots, *e.g.,* IP addresses and protocol-specific parameters. (iii) *Status tables* represent the volatile aspect of device/network state, such as protocol running status, routing table entries, or other dynamic status relevant to network health, for example, `ping` results between routers. We describe how to enable this abstraction on commodity network devices in §4.4.2.

Regular tables are only modified when necessary to reflect network changes, *e.g.,*

[1]We use the following naming convention: names of regular, config and status tables begin with T, C, S respectively.

when new devices are introduced and new customer agreements are reached. Config tables are always in-sync with the network devices, and modifying these tables causes actual configuration changes. Status tables are read-only, and table entries are generated on-demand from traditional management interface, such as CLI and SNMP[2].

Rules: COOLAID represents network management domain knowledge, in particular protocol dependencies and network-wide requirements, as rules in the form of a declarative query language. Each rule defines a database view table (or view in short), which is derived from a query over other base tables or views. Intuitively, a view derives higher-level network properties (*e.g.,* if a feature is enabled) based on lower-level information (*e.g.,* the availability of the required configurations and other dependent services). Formalizing domain knowledge as declarative rules has two benefits. First, view querying is a well-defined procedure that hides intermediate steps and presents meaningful end-results to the operators. Comparing to a manual reasoning process, which is inherently limited by human operators, COOLAID can handle an expanding knowledge base and network size with ease. Second, the rules can be re-used, as they can be queried many times even on different networks. Note that operators do not need to write any such rules. Specifically, we envision an environment where (i) device vendors provide rules to capture both device-specific capabilities and network-wide protocol configuration and dependency details (§4.2.2, §4.2.4) and (ii) service providers define rules on how these vendor capabilities should be utilized to reliably deliver customer services (§4.2.3, §4.2.5), and more importantly these rules operate within the same framework.

Controller: As the "brain" of COOLAID, the controller acts as a database engine to support straightforward database operations, like table query, insertion and deletion. We will explain in the following sections about how these operations correspond to a set of new management primitives. Fundamentally, the controller relieves majority of the workload from the operators, by applying the rule-based domain knowledge

[2]Status tables contain important information for various management operations (*e.g.,* fault diagnosis). However, because this chapter primarily focuses on the configuration management, we leave exploiting status tables as future work.

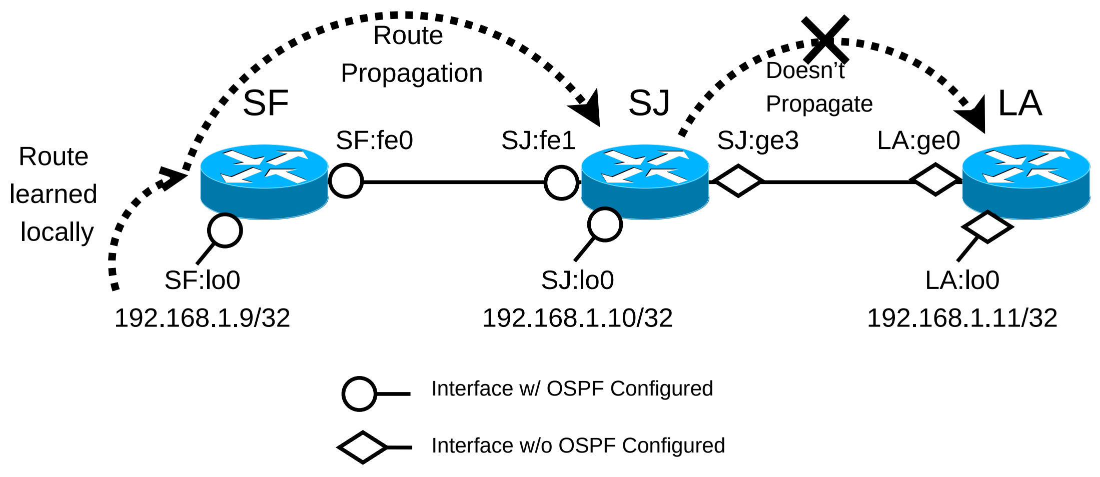


Figure 4.2: Example network with OSPF configuration

onto the network state stored in the data model. The operators can stay at a high-level of interaction, without touching the low-level details of the network. From the database perspective, the controller supports recursive query processing, global constraint enforcement, updatable views, and distributed transaction management.

Listing IV.1: Rules for OSPF Route Learning

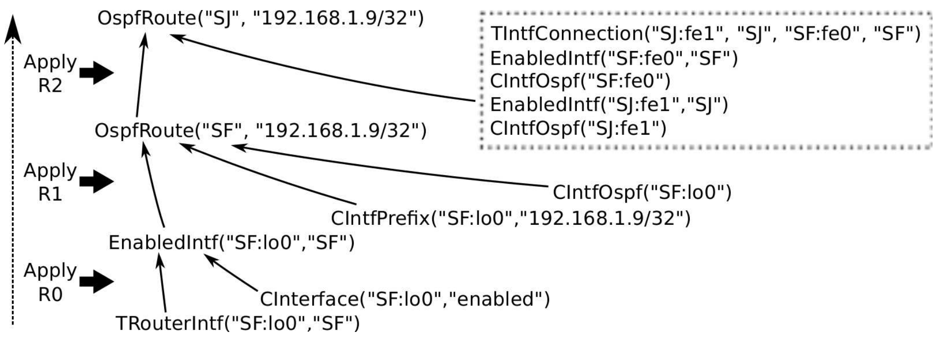
```
R0 EnabledIntf(ifId, rId) :- TRouterIntf(ifId, rId),
    CInterface(ifId, "enabled");

R1 OspfRoute(rId,prefix) :- EnabledIntf(ifId,rId),
    CIntfPrefix(ifId,prefix),
    CIntfOspf(ifId);

R2 OspfRoute(rId1,prefix) :- OspfRoute(rId2,prefix),
    TIntfConnection(ifId1,rId1,ifId2,rId2),
    EnabledIntf(ifId1,rId1), CIntfOspf(ifId1),
    EnabledIntf(ifId2,rId2), CIntfOspf(ifId2);
```

4.2.2 Network-wide reasoning

COOLAID achieves the primitive of automated network-wide reasoning through materializing the views by distributed recursive queries on top of the data model



OspfRoute("SJ", "192.168.1.9/32")
TIntfConnection("SJ:fe1", "SJ", "SF:fe0", "SF")
EnabledIntf("SF:fe0","SF")
CIntfOspf("SF:fe0")
EnabledIntf("SJ:fe1","SJ")
CIntfOspf("SJ:fe1")
Apply R2
OspfRoute("SF", "192.168.1.9/32")
Apply R1
CIntfOspf("SF:lo0")
CIntfPrefix("SF:lo0","192.168.1.9/32")
EnabledIntf("SF:lo0","SF")
Apply R0
CInterface("SF:lo0","enabled")
TRouterIntf("SF:lo0","SF")

Figure 4.3: Bottom-up view evaluation

presented in §4.2.1. We use a simple example to demonstrate how the knowledge regarding OSPF route learning can be written as three rules in Listing IV.1. The rules are written in a declarative language based on Datalog [100][3], where each rule is defined as

`rule_name rule_head :- rule_body;`

The rule head contains exactly one predicate as the view to be defined, and the rule body contains predicates and Boolean expressions that derive the view. A rule is intuitively read as “if everything in the body is true, then the head is true.”

Rule `R0` defines a view `EnabledIntf` for identifying the list of enabled interfaces in the network. It first joins a regular table `TRouterIntf` that contains the router interface inventory and a config table `CInterface` with interface setups, and then selects the interfaces that are configured as `"Enabled"`. Rule `R1` captures how a router imports local OSPF routes, by stating that if an interface on a router is enabled (as in `EnabledIntf`) and configured to run OSPF (as in `CIntfOspf`), then the prefix of its IP address should be in the OSPF routing table of the router (`OspfRoute`). We are ignoring some details, such as OSPF areas, for brevity. Finally, rule `R2` expresses how routes are propagated across routers, by stating that any `OspfRoute` on router `rId2` can propagate to router `rId1` if they have directly connected interfaces and both are enabled and OSPF-speaking. Note that `R2` is both distributed and recursive, as the

[3]We choose Datalog with stratified negation as our query language for its conciseness in representing recursive queries and negations and its tight connection to logic programming. Other query languages, such as SQL and XQuery, if augmented with necessary features, such as recursion, are also suitable to our framework.

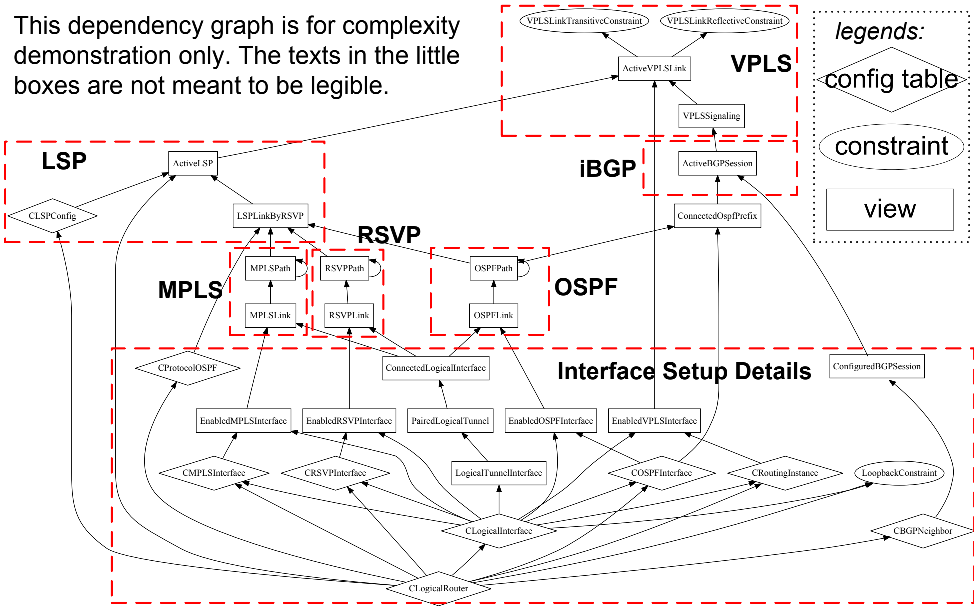


Figure 4.4: VPLS related view dependency graph

query touches multiple devices and the rule head is part of the rule body.

Figure 4.2 shows a small network with three routers. The interfaces connecting routers `SF` and `SJ`, as well as their loopback interfaces, are OSPF-speaking and enabled, so that the loopback IP `"192.168.1.9/32"` configured on router `SF` should propagate to router `SJ`, according to how OSPF works. Figure 4.3 illustrates how the entries in the view tables are generated in a bottom-up fashion based on `R0-R2`, and eventually the entry `OspfRoute("SJ","192.168.1.9/32")` shows that "prefix `192.168.1.9/32` in the OSPF route table of router `SJ`." On the other hand, there is no `("LA","192.168.1.9/32")` entry, because the dependencies are not met.

Effectively, a simple query over `OspfRoute` can reveal the OSPF routes on all routers to the operators without requiring them to understand how the route propagation works across distributed network devices. Figure 4.4 shows that the knowledge regarding complicated services like VPLS can be modeled with a stack of dependent views. Operators only need to query the top view `ActiveVPLSLink` to acquire a list of enabled VPLS connections, without understanding the details of all the dependent protocols, such as MPLS, RSVP, *etc.*

4.2.3 Misconfiguration prevention

COOLAID uses *constraints* to detect and prevent misconfiguration. The constraints dictate what data should *not* appear if the database is in a good state. That is, COOLAID rejects an operation (*e.g.,* made by operators that may not fully estimate the network-wide impact) if the outcome would violate the given constraints, *before* the changes take effect on the routers. As a result, COOLAID can help prevent undesired properties, such as network partitioning, service disruption, or large traffic shift. Constraints exist in traditional relational database management systems (RDBMS), but are usually limited to uniqueness of primary keys and referential integrity of foreign keys. In contrast, COOLAID allows more expressive constraints capable of checking and reasoning about multiple devices at different layers across the network in an efficient fashion.

Specifically, in COOLAID, a constraint is defined the same way as views by a set of rules. A constraint is satisfied if and only if the associated view is evaluated to an empty list. Conceptually, each entry in a non-empty constraint view corresponds to a violation to a desired network property.

Constraints help prevent the misconfigurations that are captured using constraints, when combined with our new transaction primitive (described in §4.2.6.) In essence, a group of network intended changes are declared as a transaction and executed in an all-or-nothing fashion. The changes are effective only if the transaction commits. Before committing a transaction, COOLAID checks if any constraints are violated by the changes, and if so aborts the transaction. For example, an access router has two interfaces connecting to the core network, and one of them is shut down for maintenance. If an operator mistakenly attempts to shut down the other link, such an operation (on `CInterface` table) would not be committed, because it violates the constraint that an access router must be connected to the core. Such support automates a network-wide "what-if" analysis, avoiding erroneous network operations due to operators' lack of understanding of complex network functions or their inability to reason at a large scale.

Note that COOLAID clearly is unable to prevent the misconfigurations that are not captured in any constraint rules. This is similarly true for the current management practice, as operators try to prevent a misconfiguration only after it occurs and is understood. COOLAID provides a more systematic way of assisting the formal specification of misconfigurations and automated prevention in the future.

4.2.4 Configuration automation

COOLAID supports a new primitive of automating network configuration by allowing *writes* to view tables. Specifically, COOLAID allows the operators to specify intended network changes as insert/delete/update to view tables, then automatically identifies a set of low-level changes to config tables that can satisfy the given intention. For example, an operator can express goals, like establish a VPLS connection between two interfaces, by a simple view insert statement:

```
ActiveVPLSConnection.insert("intA","intB")
```

The traditional mindset for configuration management is that operators (i) change the configurations on one or more devices and (ii) check if a network feature change is effected. These two steps are repeated until the check succeeds. For a failed network check, the operators reason about the symptom and fulfill the missing dependencies based on domain knowledge. In COOLAID, to the contrary, operators can stay unaware of how to realize certain network functions, instead they specify at a high-level what functions they need. In the previous example, the operator only needs to deal with `ActiveVPLSConnection` view, rather than fiddling with all the dependent network functionalities.

4.2.5 Network property enforcement

COOLAID allows the operators to specify certain properties to enforce on the network. For example, a network may be required to configure loopback IP address on every router, and establish full-mesh iBGP sessions. We model a desired network property also using constraint views, while an empty constraint means that the associated property is valid on the network. When the underlying network changes, *e.g.,* with a new router introduced, constraint violations may occur, meaning that certain network-wide properties no longer hold. COOLAID takes advantage of deletion operations on a view to automatically resolve the constraint violations. For example, by calling `LoopbackAddressConstraint.remove_all()`, COOLAID automatically changes related configuration tables, say modifying `CIntfPrefix` table to configure the loopback interfaces in question, so that the constraint view becomes empty. This means that the operator only needs to specify the desired properties, and COOLAID can automatically enforce them in the face of dynamic network changes.

4.2.6 Atomic network operations

Device failures during network operations are not uncommon, especially in large-scale networks. If not handled properly, they often put the network in an inconsistent state. For example, a network operation involving configuring several routers might be abandoned midway because of unforeseen circumstances, such as an unexpected transient network failure, or overloaded routers. Current operational procedures would attempt a manual rollback; however, that may be incomplete, leaving some "orphaned" configuration excerpts, which might lead to security holes or unintended network behavior.

The problem in the above example is due to the lack of "all-or-nothing", or atomicity, in network management primitives. In fact, we argue that the ACID properties of transactional semantics (§4.3.3), namely atomicity, consistency, isolation, and durability, are all highly desirable as primitives to compose network operations. They are provided naturally in COOLAID by the database abstraction.

We note that modern routers already allow atomic configuration changes on a per-device basis. In contrast, COOLAID not only extends such semantics to a *network-wide* fashion, but also supports additional assertions on network-wide states, by checking constraint views, to validate transactions.

In COOLAID, a network operation is defined as a procedure of a series of database read and write commands, from and to the tables and views. We introduce a transaction primitive for various reasons. First, as described previously, certain constraints may be violated by the operations. These operations should be voided to prevent misconfigurations. Second, certain operations are no longer desired when the underlying physical network changes. For example, during an operation, a physical link may suddenly fail, invalidating some previous view query results. We abort on-going transactions if any of those conditions occur.

Unfortunately, the ACID properties are difficult to achieve without a global exclusive control over a network. As a result, COOLAID assumes a single control entity. However this level of control is difficult to acquire, especially in early experimental stage. A more likely scenario is that our system is given an increasing portion of the devices in a network, following an incremental deployment. In this case, COOLAID may have incomplete control over a network, thus unable to guarantee ACID. Nevertheless the support of network reasoning still stands, as long as COOLAID can "read" from all the network elements. At the same time, misconfiguration prevention and configuration automation are valid for the network portion under control. We provide more discussions in Section 4.6.

4.2.7 Summary

In this section, we have presented an overview of the COOLAID framework. COOLAID builds on a database abstraction that captures all aspects of the network and its operations in a data model, consisting of regular, config, and status tables. COOLAID allows vendors and providers to collaboratively capture domain knowledge in the form of rules, in a declarative query language. By leveraging such knowledge, COOLAID provides new network management primitives to network operators, including network-wide reasoning, misconfiguration prevention, configuration automation, network property enforcement, and atomic network operations, all in the same cohesive framework.

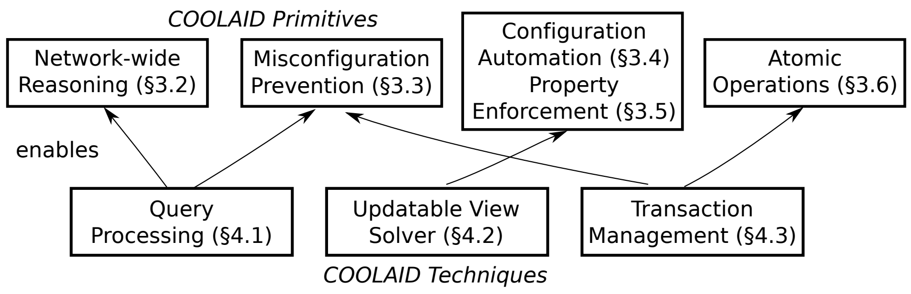

COOLAID Primitives
Network-wide Reasoning (§3.2)
Misconfiguration Prevention (§3.3)
Configuration Automation (§3.4) Property Enforcement (§3.5)
Atomic Operations (§3.6)
enables
Query Processing (§4.1)
Updatable View Solver (§4.2)
Transaction Management (§4.3)
COOLAID Techniques

Figure 4.5: COOLAID primitives and techniques

4.3 Techniques

In this section, we explain key techniques that COOLAID utilizes to enable the network management primitives described in §4.2. Figure 4.5 shows their relationships.

4.3.1 Query processing

Query processing is essential for network-wide reasoning (§4.2.2) and misconfiguration prevention (§4.2.3). We highlight a few design choices in building the query processor efficiently, despite the differences between COOLAID and conventional RDBMS.

First, besides traditional database-style queries, COOLAID heavily relies on recursive queries due to the distributed nature of network protocols. Recursive query evaluation and optimization is a well-studied area in databases [100]. Recent work has also examined recursive queries in a distributed environment with a relaxed eventual consistency model [86].

Second, COOLAID manages a much smaller amount of data, but these data are distributed. The largest portion come from configurations. If we assume that a configuration file is 100KB on average, and there are a thousand routers in a network, then we need roughly a hundred megabytes of space to store the raw data. On the other hand, the configuration data on different routers might require hundreds of milliseconds of round-trip time to access for a typical ISP with national footprints. Therefore, we always first aggregate all data to the main memory of a centralized master node (§4.4.1) before query evaluation. Centralized processing is also preferred

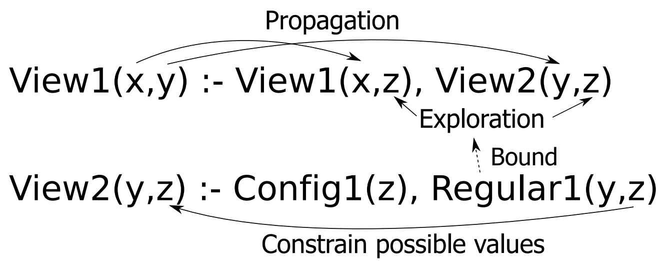

Propagation
View1(x,y) :- View1(x,z), View2(y,z)
Exploration
Bound
View2(y,z) :- Config1(z), Regular1(y,z)
Constrain possible values

Figure 4.6: Solving updatable view operations

in order to enforce a strong consistency model as opposed to the eventual consistency model [86]. Once all data are available, we apply the semi-naïve evaluation algorithm [100], which is both efficient and generic, to evaluate recursive queries.

We further apply the technique of *materialized view maintenance* to speed up query performance. The entire contents of all views are cached in memory once computed, rather than generated on-demand at each query evaluation. Each view has the meta data that describe which base tables it depends on. Once the base tables of a view are updated, the view is *incrementally* updated by only computing the differences to reduce overhead.

4.3.2 Updatable view solver

Updatable view operations, like view insertions or deletions, enable configuration automation (§4.2.4) and network property enforcement (§4.2.5). Underneath the simple APIs called by operators, COOLAID controller finds the config table entries to change to realize the intended view changes.

We explain two techniques to update views with different trade-offs. In practice, we use a combination of both to achieve the best performance and usability. First, we designed an automatic updatable view solver, using standard techniques from Prolog, such as exploration and propagation. As illustrated in Figure 4.6, to insert an entry `(x,y)` into `View1`, we need to recursively guarantee tuples `(x,z)` and `(z,y)` are in `View1` and `View2`. If there are no such combination, a recursive view insertion is attempted. For the value of `x` and `y`, we can directly propagate from the left-hand side to the right-hand side. But we have to enumerate the possible values for `z` and try them one-by-one: some guessed values may not be possible to insert into `View2`, for example. For non-recursive rules, the recursion in this solving process is bounded by the level of dependencies. For recursive rules, this solving process might be expensive: for example, to insert tuple `(x,y)` into `View1`, we need to further insert `(x,z)` into `View1`, and this may go on many times. There are two key factors that keep this process feasible. (i) We do not change regular tables, because the values are treated as facts of the network. As a result, regular tables bound the domain for many fields in views. For example, `View2` is defined by joining a config table and a regular table, so the tuples in `View2` can only possibly come from `Regular1`. In this case, COOLAID can bound the exploration for literal `z`, when inserting to `View1`. (ii) Network functionalities are almost always cumulative, so that negations rarely occur in the rules. This greatly reduces the search space.

Note that COOLAID prunes the solutions that violate constraints. The key benefit of this approach is that COOLAID only needs a single solver to handle all protocols and features. The main downside, however, is that the results provided by the solver may not always be preferred. The is because many solutions can be found to satisfy the same updatable view operation. For example, if we want to establish IGP connectivity on a set of ISP core routers, we can use OSPF, IS-IS, or simply static routes. With OSPF, we can configure a subset of the interfaces to establish a spanning tree touching all routers, still enabling all-pair connectivity, although this is clearly undesired for reliability concerns. In practice, we assign customizable preference values to different rules, so that the solver prioritizes accordingly.

Second, an alternative solution is to allow the rule composers to customize resolution routines for view insertion and deletion. For example, when an insertion is called on a view, the corresponding resolution routine is executed based on the value of the inserted tuple. The key benefit is that rule composers have better control over the resulting changes to the network. Such resolution routines can explicitly encode the best practice. For example, to enable OSPF connectivity, we can customize the routine to configure OSPF on all non-customer interfaces in the core network, comparing to the generic solver that may give a partial configuration. The flip side is the extra work on rule composers to develop these routines, comparing to using a generic solver to automatically handle the intended changes. Based on our experience, however, such resolution functions are very simple to develop, largely thanks to the unified database abstraction. Also, this requires one-time effort by vendors or network experts, while the operators can stay unaware of such details.

4.3.3 Transaction management

Misconfiguration prevention (§4.2.3) and atomic network operations (§4.2.6) both rely on the transaction processing capability in COOLAID. We describe the transactional semantics and our design choices.

In the context of databases, a single logical operation on the data is called a transaction. Atomicity, consistency, isolation, and durability (ACID) are the key properties that guarantee that database transactions are processed reliably. In COOLAID, a network operational task is naturally expressed as a distributed database transaction that may span across multiple physical devices. In our data model, the regular tables inherit the exact ACID properties from a traditional RDBMS. Interestingly, we find that ACID properties naturally fit config tables as follows:

Atomicity: The configuration changes in an atomic operation must follow an "all-or-nothing" rule: either all of the changes in a transaction are performed or none are. COOLAID aborts a transaction if failure is detected, and rolls back to the state before the transaction started. Note that we can only guarantee the configuration states (tables), because status tables can change dynamically and take time to converge to stationary values. This holds true for the rest of the three properties. Note that atomicity also applies in a distributed transaction where config changes involve multiple devices. The atomic feature greatly simplifies the management logic in handling device and other unexpected failures.

Consistency: The database remains in a consistent state before the start of the transaction and after the transaction terminates regardless of its outcome. The consistency definition in COOLAID is that all constraints must be satisfied. Before each commit in a transaction, COOLAID checks all the constraints. In case of constraint violations, an operator can simply instruct COOLAID to roll-back thus abort the transaction, or resolve all violations and still proceed to commit. The database ends up in a consistent state in both cases.

Isolation: Two concurrent network operations should not interfere with each other in any way, *i.e.,* as if both transactions had executed serially, one after the other. This is equivalent to the serializable isolation level in a traditional RDBMS. For example, an operation in an enterprise network might be to allocate an unused VLAN in the network. Two of such concurrent operations without isolation might choose the same VLAN ID because they share the same allocation algorithm. Such a result is problematic and can lead to security breach or subtle configuration bugs. COOLAID provides transaction isolation guarantees to prevent such issues.

Durability: Once the user has been notified of the success of a transaction commit, the configurations are already effective in the routers. Most commodity routers already provide this property.

Note that COOLAID assumes to be the only management entity in order to deliver ACID — any configuration changes that do not go through our system undermine all guarantees. To implement the ACID transactional semantics in COOLAID, we use the Two-Phase Commit protocol for atomicity due to its simplicity and efficiency; we use Write-Ahead-Logs for crash recovery; and we use Strict Two-Phase Locking for concurrency control [99]. These design decisions are customized for network management purposes. For example, we favor conservative, pessimistic lock-based concurrency control because concurrent network management operations occur much less frequently than typical online transaction processing (OLTP) workload, such as online banking and ticket booking websites. Once two concurrent network operations have made conflicting configuration changes, it is very expensive to roll back and retry one of them. We choose to prevent conflicts from happening, even at the cost of limiting parallelism. We discuss the detailed implementations of transaction management in §4.4.1.

4.4 Implementation

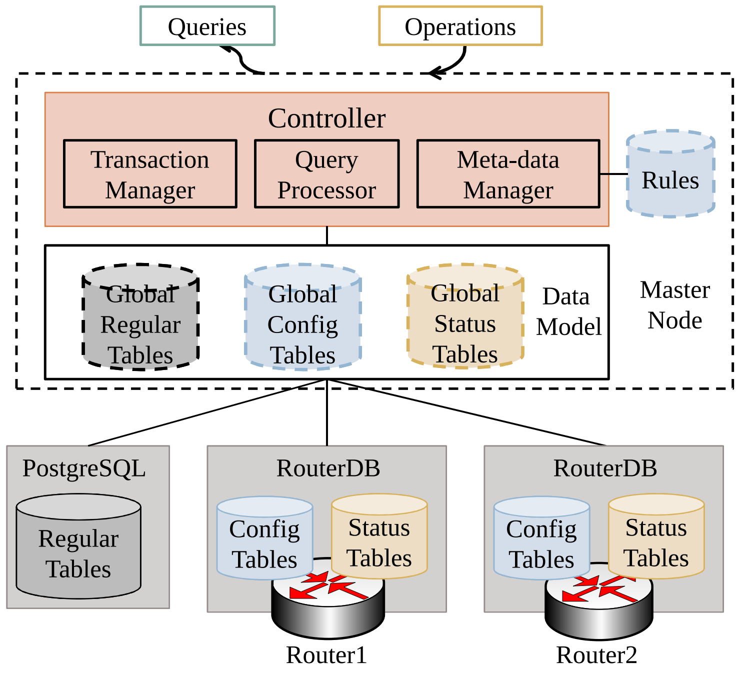


Figure 4.7: COOLAID system architecture

The overall system architecture of COOLAID is depicted in Figure 4.7. We have implemented a prototype system in roughly 13k lines of Python code with two major software pieces described next.

4.4.1 Master node

The master node unifies all data sources and manages them as a centralized database. We use PostgreSQL as the backend to manage regular tables. Each physical router is managed by a RouterDB (§4.4.2) instance, which exports the corresponding config tables and status tables. The config tables on RouterDBs are aggressively combined and cached on the master node for performance improvement. When an entry in a config table is modified, the appropriate RouterDB instance will be identified and notified based on the primary key of the entry, which has the physical router ID encoded. This technique is known as horizontal partitioning in data management.

The controller on the master node has three components:

Query processor: The query processor first parses the declarative rules and rewrites them in expressions of relational algebra (set-based operations and relational operators such as join, selection and projection). We implemented a library in Python, with a usage pattern similar to Language INtegrated Query (LINQ) in the Microsoft .NET framework [12], to express and evaluate those relational expressions. The library is capable of integrating queries from Python objects, tables in PostgreSQL (by an object-relational mapper [23]), and XML data. We implemented the algorithm described in §4.3.1 for query evaluation and view maintenance and an updatable view solver described in §4.3.2.

Meta-data manager: Meta-data, such as the definitions of all tables, views and constraints, are managed in the format of tables as well. In particular, the controller manages the meta-data by keeping track of the dependencies between the views, which is used by the view maintenance algorithm (§4.3.1) for caching and incremental updates, and updatable view operations (§4.3.2).

Transaction manager: The master node serves as a distributed transaction coordinator, and passes data records to and from the underlying local database engines. It does not handle any data storage directly, and achieves the transactional ACID properties as follows:

Atomicity and durability are achieved by realizing the two-phase commit protocol (2PC) [99] among the underlying database participants (*i.e.,* PostgreSQL and RouterDB instances): In phase 1, the master node asks all of the participants to prepare to commit. The transaction aborts if any participant responds negatively or fails to reply in time. Otherwise, in phase 2, the master node flushes the commit decision to a log on disk, then asks all nodes to commit.

Consistency is enforced by checking all constraints after the commit request is received. Unless all constraints are satisfied (directly or through violation resolution), the 2PC protocol starts to complete the transaction.

Isolation is enforced by a global lock among transactions in the current prototype. Effectively, this only allows a single transaction at a time—the most conservative scheme. While it clearly limits the parallelism in the system, serializing them is acceptable as backlog is unlikely even in large networks. Using finer-grained locks for higher parallelism could introduce distributed deadlocks, which could be costly to resolve. We leave exploring this trade-off as future work.

To recover from a crash of the master node, the transaction manager examines the log recorded by the 2PC protocol. It will inform the participants to abort pending transactions without commit marks, and recommit the rest. If the master node cannot be restarted, it is still possible for network operators to directly interact with individual RouterDBs. This allows raw access and control over the network for emergency and manual recovery. We talk about removing master node as a single point of failure in §4.6.

4.4.2 RouterDB

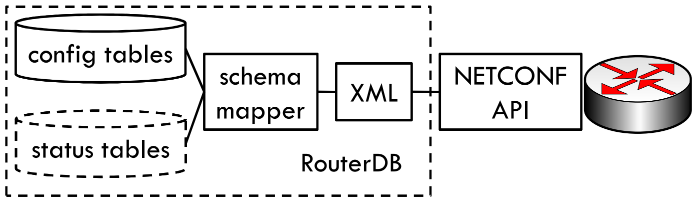


Figure 4.8: RouterDB implementation

RouterDB provides a 2PC-compliant transactional database management interface for a single router device. Our current prototype works for Juniper routers, but can be easily extended to other router vendors. RouterDB utilizes the programmable APIs standardized by the Network Configuration Protocol (NETCONF) [15] to install, manipulate, and delete the configuration of network devices over XML.

When a RouterDB instance starts, it uses a given credential to initiate a NETCONF session over `ssh` with the corresponding router, and fetches the currently running configuration in XML format. Then a schema mapper is used to convert configurations from the tree-structured XML format into relational config tables.

Transaction APIs: To update config tables, a transaction must be started by calling the `begin_txn` RouterDB API. It saves a snapshot of the current configuration in XML, and returns a transaction context ID. Further data manipulation operation calls, such as `insert`, `update`, `delete` to the config tables must use the ID to indicate its transaction affiliation. Once a manipulation call is received, the schema mapper converts it back to an XML manipulation snippet, and uses the `edit-config` NETCONF API to change the configuration on the router. Note that this change is made to a separate target, called the candidate target, so that it does not interfere with the running configuration of the router. Then, the updated configuration in the candidate target is fetched, and the change is propagated to the config tables via the schema mapper.

To be compliant with the two-phase commit protocol used by the master node, RouterDB implements the `prepare`, `commit`, and `rollback` APIs. When executing `prepare()`, the configuration in the candidate target is validated by the router. An invalidated configuration will raise an exception so that the transaction will be aborted. During `commit()`, the configuration in the candidate target is first made effective by issuing a `commit` NETCONF call, and then the saved snapshots are freed. During `rollback()`, the candidate target is discarded on the router.

Placement: Technically, a RouterDB instance might be hosted anywhere between the master node and the router. We chose to host RouterDB close to the router and assume the node has reliable network access to the dedicated management interfaces on the managed router. The placement is advantageous over hosting RouterDB on the physical router itself because (i) Data processing on RouterDB is isolated from other tasks on the router, and it is guaranteed not to compete for router resources (*e.g.,* CPU and memory); (ii) When RouterDB is separated from the router, it is much more likely to differentiate failures between RouterDB and the physical router from the master node, and treat them differently; (iii) Only selected high-end commercial routers provide enough programmability to build RouterDB [80]. On the other hand, by placing RouterDB close to the router instead of the master node, we have the opportunity to reduce the amount of data transferred from RouterDB to the master node, by pushing some database operators, such as filters, into RouterDB.

Handling failures: Following the Write-Ahead-Log protocol [99], RouterDB records every operation in a log file on persistent storage. When recovering from a previous crash, RouterDB locates all ongoing transactions at the time of crash, rolls back the ones that are not committed, and re-commits those transactions that the master node has issued commit commands.

During the downtime of a RouterDB instance, the master node still has the configuration data in its cache so that it is readable. However, any write requests will be denied. The data in corresponding status tables become unavailable too.

Physical router failures detected by RouterDB are reported to the master node, which temporarily marks the related entries in the regular table caches as "offline" so that they do not show up in query results, until the physical router comes back online. Operators cannot change configuration or check status on the router during the offline time.

4.5 Evaluation

We evaluated several key aspects of COOLAID to show that it effectively reduces human workload and prevents misconfigurations in realistic network management tasks, at the same time scales to large networks. In all experiments, we used Juniper M7i routers running JUNOS V9.5. The Linux servers, which host master nodes and RouterDB instances, were equipped with Intel Dual Core 2.66GHz processors and 4GB RAM.

4.5.1 Automating configuration

We created the network topology of Abilene core network [21] with 10 routers and 13 links on top of the ShadowNet platform [47] for network experimentation. The actual router instances are distributed across Texas, Illinois and California. Besides the links in the topology, each router has another interface connecting a local virtual machine, simulating a customer site. We run one RouterDB for each router and a single master node in Illinois. All routers in this experiment started with minimum configurations that only describe interface-level physical connectivity.

Our goal is to configure a VPLS service connecting two customer-facing interfaces on two different routers. This is a heavily involved procedure as operators need to deal with IP allocation for interfaces, configuring OSPF or IS-IS routing, establish iBGP sessions, configure MPLS network with RSVP signaling, establish LSPs, and configure VPLS instances.

If an operator were to manually perform the task entirely, she must start with executing at least 25 lines of configuration commands on average on all routers, and 9 additional lines on the two customer-facing routers, in total 268 lines. For larger networks with more routers and links, this number should increase linearly. The lines of configuration changes is measured by `show configuration | display set` on JUNOS, which displays the current configuration with minimum number of commands. In reality, the actually executed commands are usually more. Besides, this number does not reflect the manual reasoning effort to realize this VPLS service, which commonly requires multiple iterations of trial-and-test and accessing low-level CLIs.

In COOLAID, enabling such a complicated service requires a *single* operation by the operator, calling `ActiveVPLSConnection.insert(int1_id,int2_id)`. This stays the same no matter how large the network is. Also, the operator does not have to deal with any of the dependencies.

4.5.2 Handling network dynamics

In contrast to the previous setup, we started with a well-configured 9-router subset of the Abilene network topology on ShadowNet. The intention is to study how COOLAID enforces network properties when new, barely configured routers are introduced in an existing network. When the regular tables were updated to include the 10th router and the associated links, several network properties that were specified as constraints were immediately flagged as violated. For example,
`LoopbackAddressConstraint` showed that the new router did not have an loopback interface configured with a proper IP address and `BGPFullMeshConstraint` reported that

the new router had no iBGP sessions to other routers. COOLAID checks constraints for property enforcement whenever there is a network change, and automatically tries to resolve the violations. In this case, the customized view solver was used to produce 26 lines of config changes on the new router, and 9 lines on the existing routers for iBGP sessions, such that specified network properties are enforced automatically.

4.5.3 Performance

In this section we isolate the DB processing capability from device access overhead to evaluate the performance of the view query processor and the view update solver.

Network	Abilene	3967	1755	1221	6461	3257	1239
Router #	10	79	87	108	141	161	315
Link #	13	147	161	153	374	328	972
Time (ms)	0.3	20	24	28	73	116	592

Table 4.1: Query processing time for `OSPFRoute`

Processing queries: To evaluate the query processing performance, we chose the recursive view `OspfRoute` because it is one of the most expensive queries, where the complexity grows quadratically with the network size. We use the topologies of Abilene backbone and five other ASes inferred by Rocketfuel [106]. The config tables were initialized such that all interfaces on each router are OSPF enabled, including the loopback interfaces. Then we queried `OspfRoute` to get the OSPF routes on all routers for each topology. The query time is showed in Table 4.1. It only took 0.3ms to complete the query for Abilene. For the largest topology on AS1239 with 315 routers and 972 links, it took less than 600ms. This suggests that processing queries has negligible overhead compared with device related operations, such as physically committing config to routers (on the order of tens of seconds on the Juniper routers).

Case 1: OSPF	Case 2: iBGP	Case 3: iBGP w/ OSPF
14.112s	14.287s	0.025s

Table 4.2: Time to solve view updates

Solving view updates: We tested our view update solver in three cases with the

Abilene topology. We picked a pair of routers ($r1$ and $r2$) that are farthest from each other in the topology. In Case 1, starting with the minimal configuration, we inserted two tuples into `OspfRoute`, intending to have the loopback IPs of $r1$ and $r2$ reachable to each other via OSPF. In Case 2, also starting with the minimum configuration, we inserted a single tuple in `ActiveIBgpSession`, intending to create an iBGP session between $r1$ and $r2$. In Case 3, we started with a network with OSPF configured on all routers, and performed the same operation as in Case 2. As captured by the rules, active iBGP sessions depend on IGP connectivity, so in Case 2 the solver automatically configured OSPF to connect $r1$ and $r2$ first and then configured BGP on both routers.

Table 4.2 shows the running time for each case. We observe that (i) Case 3 was much faster, because the solver was able to leverage existing configurations; (ii) Case 1 and Case 2 took about the same amount of time, because the OSPF setup dominated. The OSPF setup in Case 1 is slow because it starts with a network without configuration and requires multiple levels of recursion to solve this view insertion. While 14 seconds is not short, in practice, one only needs to configure OSPF for a network once, and most of the common tasks, including configuring a new router to run OSPF, are incremental to existing configurations, thus can be done quickly, like in Case 3.

We also evaluated the same tasks using the rules with customized resolution routines. In this case, view update operations are achieved by calling a chain of hard-coded resolution routines, thus the reasoning overhead is zero.

4.5.4 Transaction overhead

	Step 1	Step 2	Outcome
w/o COOLAID	8.4s	8.4s	Disconnected network
w/ COOLAID	8.4s	Rejected	Disruption avoided

Table 4.3: Network operations with and without COOLAID

To study the device-related performance and transaction overhead, we use the following setup. First, we assume 3 routers $r1$-$r3$ with pair-wise links, and all routers are configured with OSPF. In step 1, we shut down the link between $r1$ and $r2$ (by disabling one of its interfaces). Such operations are common for maintenance purpose and benign, because the network is still connected. In step 2, we try to shut down the link between $r1$ and $r3$ to emulate a misconfiguration that would cause a network partition.

We compare the experience of using COOLAID to perform such operations with using a script that directly calls NETCONF APIs, and then show the result in Table 4.3. Without COOLAID, the two steps took 8.4 seconds each, ending with a disconnected network. The time is mostly spent by the router internally to validate and commit the new configuration. With COOLAID, step 1 takes the same amount of time, suggesting a negligible overhead in constraint checking or any other extra overhead introduced by COOLAID. Because we specified a constraint that every router's loopback IP must be reachable to all other routers, step 2 is rejected by COOLAID before it could take effect on the actual routers.

4.6 Discussion

Feasibility: One critical question is how feasible it is to create the database abstraction and the declarative rules. First, network databases are common for modern ISPs [41]. The emerging trend of XML-based configuration files further reduces the effort, since XML files can be directly queried. The rules indeed take much effort to derive according to our experience. However, the time-consuming part is to derive the correct dependency information by reading documentations and performing field tests. Once the dependency is known, which is the case for vendors and network experts, it takes little time to express it in the declarative language.

Deployment: While COOLAID is designed to take over managing the whole network, we note that it is amenable to a variety of partial deployment scenarios. For example, COOLAID can initially work in a read-only mode to assist network reasoning. When operators are comfortable enough about using the new database primitives, they can gradually enable write permission to config tables. Note that configuring certain network features do not require touching all routers.

Availability: In the current centralized implementation, the system is not available when the master node is offline. To remove this single point of failure, we can adopt the replicated state machine approach [104] where multiple copies of the COOLAID controller are running simultaneously as primary node and backup nodes. Another alternative is to adopt a fully decentralized architecture, where all query processing and transaction management is handled in a distributed fashion by RouterDB instances. There are sophisticated algorithms and protocols, such as Paxos commit [65], that are designed for this scenario. How they compare with the centralized architecture in performance and ease of maintenance is an interesting direction for our future work.

Limitations: We note two potential problems of COOLAID by deriving the transactional semantics over the network configuration. (i) Routing protocols are not transaction-aware, as they require time to converge upon configuration changes. The order and timing of such changes are important in determining the consequences, *e.g.,* temporary routing loops and route oscillations. Therefore, transaction rollback support for handling failures in such tasks is usually inadequate without creating case-specific handlers to deal with failure exceptions. (ii) It is possible that some resources are released during the transaction execution and cannot be re-acquired in the case of rollback. The problem could be addressed through a locking mechanism to hold the resources until the transaction finishes. Finally, COOLAID currently does not address the issues of protocol optimization, say tweaking the OSPF link weights for traffic engineering [57]; however, existing techniques can be invoked in the customized view solvers to integrate their results with our data model.

4.7 Summary

We presented COOLAID as a unifying data-centric framework for network management and operations, where the domain expertise of device vendors and service providers can be systematically captured, and where protocol and network dependencies can be automatically exposed to operational tools. Built on a database abstraction, COOLAID enables new network management primitives to reason and automate network operations while maintaining transactional semantics. We described the design and implementation of the prototype system, and used case studies to show its generality and feasibility. Our future plan is to improve the design and implementation of COOLAID by adding new management primitives, increasing concurrency, and improve reliability. While COOLAID currently covers a variety of dominant network operations that rely on configuration changes, we also plan to explore COOLAID's applicability in other management areas such as fault diagnosis and performance management.

CHAPTER V

A Platform for Evaluating Network Systems and Services

Effecting network change is fundamentally difficult. This is primarily due to the fact that modern networks are inherently shared and multi-service in nature, and any change to the network has the potential to negatively impact existing users and services. Historically, production quality network equipment has also been proprietary and closed in nature, thus further raising the bar to the introduction of any new network functionality. The negative impact of this state of affairs has been widely recognized as impeding innovation and evolution [98]. Indeed at a macro-level, the status quo has led to calls for a clean slate redesign of the Internet which in turn has produced efforts such as GENI [5] and FEDERICA [4].

At a more modest micro-level, the fact that network change is inherently difficult, is a major *operational* concern for service providers. Specifically, the introduction of new services or service features typically involves long deployment cycles: configuration changes to network equipment are meticulously lab-tested before staged deployments are performed in an attempt to reduce the potential of any negative impact on existing services. This is especially true for network management tools, whose success and failure directly determine how well a network runs.

In this chapter we address these concerns through a platform called *ShadowNet*. ShadowNet is designed to be an operational trial/test network consisting of ShadowNet *nodes* distributed throughout the backbone of a tier-1 provider in the continental US. Each ShadowNet node is composed of a collection of carrier-grade equipment, namely routers, switches, and servers. Each node is connected to the Internet as well as to other ShadowNet nodes via a (virtual) backbone. ShadowNet has been utilized to test and evaluate PACMAN and COOLAID, described in the two previous chapters.

ShadowNet provides a *sharable, programmable, and composable* infrastructure to enable the rapid trial or deployment of new network services or service features, or evaluation of new network management tools in a realistic operational network environment. Specifically, via the Internet connectivity of each ShadowNet node, traffic from arbitrary end-points can reach ShadowNet. ShadowNet connects to and interacts with the provider backbone much like a customer network would. As such the testing and experimentation that take place within ShadowNet can be isolated from the "regular" provider backbone, just like how it would protect itself from any other customers. In the first instance, ShadowNet provides the means for testing services and procedures for subsequent deployment in a (separate) production network. However, in time we anticipate ShadowNet-like functionality to be provided by the production network itself to directly enable rapid but safe service deployment.

ShadowNet has much in common with other test networks [35, 97, 118]: (i) ShadowNet utilizes virtualization and/or partitioning capabilities of equipment to enable *sharing* of the platform between different concurrently running trials/experiments; (ii) equipment in ShadowNet nodes are *programmable* to enable experimentation and the introduction of new functionality; (iii) ShadowNet allows the dynamic *composition* of test/trial topologies.

What makes ShadowNet unique, however, is that this functionality is provided in an *operational network* on *carrier-grade equipment*. This is critically important for our objective to provide a rapid service deployment/evaluation platform, as technology or service trials performed in ShadowNet should mimic technology used in the provider network as closely as possible. This is made possible by recent vendor capabilities that allow the partitioning of physical routers into subsets of resources that essentially provide logically separate (smaller) versions of the physical router [74].

In this chapter, we describe the ShadowNet architecture and specifically the ShadowNet control framework. A distinctive aspect of the control framework is that it provides a clean separation between the *physical-level* equipment in the testbed and the *user-level* slice specifications that can be constructed "within" this physical platform. A *slice*, which encapsulates a service trial, is essentially a container of the service design including device connectivity and placement specification. Once instantiated, a slice also contains the allocated physical resources to the service trial. Despite this clean separation, the partitioning capabilities of the underlying hardware allows virtualized equipment to be largely indistinguishable from their physical counterparts, except that they contain fewer resources. The ShadowNet control framework provides a set of interfaces allowing users to programmatically interact with the platform to manage and manipulate their slices.

We make the following contributions in this work:

- Present a sharable, programmable, and composable network architecture which employs strong separation between user-level topologies/slices and their physical realization (§5.2).
- Present a network control framework that allows users to manipulate their slices and/or the physical resource contained therein with a simple interface (§5.3).
- Describe physical-level realizations of user-level slice specifications using carrier-grade equipment and network services/capabilities (§5.4).
- Present a prototype implementation (§5.5) and evaluation of our architecture (§5.6).

5.1 Motivation for a Realistic Testing Environment

In this section, we explain the main drivers for network changes in general and argue for the need of a more realistic testing environment.

5.1.1 The need for network changes

There are primarily three drivers for changes in modern service provider networks (also apply to any large networks):

Growth demands: Fueled by an increase in broadband subscribers and media rich content, traffic volumes on the Internet continue to show double-digit growth rates year after year. The implication of this is that service providers are required to increase link and/or equipment capacities on a regular basis, even if the network functionality essentially stays the same.

New services and technologies: Satisfying customer needs through new service offerings is essential to the survival of any network provider. “Service” here spans the range from application-level services like VoIP and IPTV, connectivity services like VPNs and IPv4/IPv6 transport, traffic management services like DDoS mitigation or content distribution networks (CDNs), or more mundane (but equally important and complicated) service features like the ability to signal routing preferences to the provider or load balancing features.

New operational tools and procedures: The increasing use of IP networks for business critical applications is leading to growing demands on operational procedures. For example, end-user applications are often very intolerant of even the smallest network disruption, leading to the deployment of methods to decrease routing convergence in the event of network failures. Similarly, availability expectations, in turn driven by higher level business needs, make regularly planned maintenance events problematic, leading to the development of sophisticated operational methods to limit their impact.

As we have alluded to already, the main concern of any network change is that it might have an impact on existing network services, because networks are inherently shared with known and potentially unknown dependencies between components. For example, a traffic engineering tool and a network maintenance tool might conflict with each other if not coordinated properly — in the worst case, traffic might be shifted to a link that is actively maintained. Another example would be the multi-protocol extensions to BGP to enable MPLS-VPNs or indeed any new protocol family. The change associated with rolling out a new extended BGP stack clearly has the potential to impact existing IPv4 BGP interactions, as bugs in new BGP software could negatively impact the BGP stack as a whole.

Note also that network services and service features are normally "cumulative" in the sense that once deployed and used, network services are very rarely "switched off". This means that over time the dependencies and the potential for negative impact only increases rather than diminishes.

A related complication associated with any network change, especially for new services and service features, is the requirement for corresponding changes to a variety of operational support systems including: (i) configuration management systems (new services need to be configured typically across many network elements), (ii) network management systems (network elements and protocols need to be monitored and maintained), (iii) service monitoring systems (for example to ensure that network-wide service level agreements, *e.g.,* loss, delay or video quality, are met), (iv) provisioning systems (*e.g.,* to ensure the timely build-out of popular services). ShadowNet does not address these concerns per se. However, as described above, new operational solutions are increasingly more sophisticated and automated, and ShadowNet provides the means for safely testing out such functionality in a realistic environment.

Our ultimate goal with the ShadowNet work is to develop mechanisms and network management primitives that would allow new services and operational tools to be safely deployed directly in production networks. However, as we describe next, in the work presented here we take the more modest first step of allowing such actions to be performed in an operational network that is separate from the production network, which is an important transitional step.

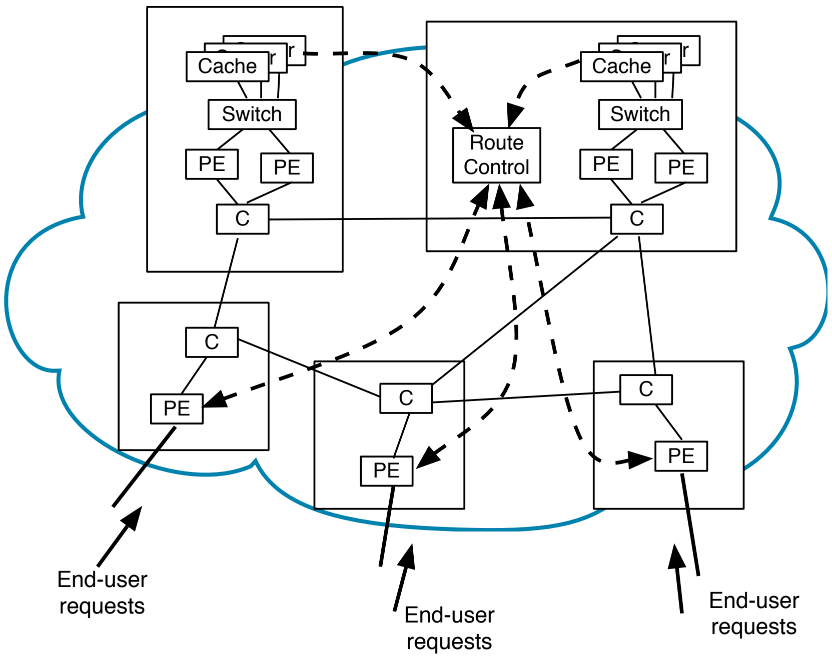


Figure 5.1: Usage scenario: load-aware anycast CDN.

5.1.2 Case study for network testing

In this section we briefly describe an example usage scenario that illustrates the type of sophisticated network services that can be tested using the ShadowNet infrastructure. The example we use is an infrastructure-assisted network service, which requires testing changes to both the network core and end-hosts and is more general. We discuss the requirements for testing these services and explain why existing platforms fall short in these scenarios.

We consider the customer trial of a *load-aware anycast content distribution network (CDN)* [32]. Figure 5.1 depicts how all the components of such a CDN can be realized on the ShadowNet platform. Specifically, a network, complete with provider edge (PE) and core (C) routers, can be dynamically instantiated to represent a small backbone network. Further, servers in a subset of the ShadowNet nodes can be allocated and configured to serve as content caches. A load-aware anycast CDN utilizes route control to inform BGP selection based on the cache load, *i.e.,* using BGP, traffic

	SN	EL	PL	VN
Production grade devices	Y	N	N	N
Realistic workloads	Y	N	Y	Y
High capacity backbone	Y	N	N	Y
Geographical coverage	Y	N	Y	Y
Dynamic reconfiguration	Y	N	N	N

Table 5.1: Capability comparison between ShadowNet (SN), EmuLab (EL), PlanetLab (PL) and VINI (VN)

can be steered away from overloaded cache servers. In ShadowNet, this BGP speaking route control entity can be instantiated on either a server or a router depending on the implementation. Appropriate configuration/implementation of BGP, flow-sampling, and server load monitoring complete the infrastructure picture. Finally, actual end-user requests can be directed to this infrastructure, *e.g.,* by resolving a content URL to the anycast address(es) associated with and advertised by the CDN contained in the ShadowNet infrastructure.

Using this example we can identify several capabilities required of the ShadowNet infrastructure to enable such realistic service evaluation (see Table 5.1): (i) to gain confidence in the equipment used in the trial it should be the same as, or similar to, equipment used in the production network (*production-grade devices*); (ii) to thoroughly test load feedback mechanisms and traffic steering algorithms, it requires participation of significant numbers of customers (*realistic workloads*); (iii) this in turn requires sufficient network capacity (*high capacity backbone*); (iv) realistic network and CDN functionality require realistic network latencies and geographic distribution (*geographic coverage*); (v) finally, the CDN control framework could dynamically adjust the resources allocated to it based on the offered load (*dynamic reconfiguration*).

While ShadowNet is designed to satisfy these requirements, other testing platforms, with different design goals and typical usage scenarios, fall short in providing such support, as we describe next.

Emulab achieves flexible network topology through emulation within a central testbed environment. There is a significant gap between emulation environments and real production networks. For example, software routers typically do not provide the same throughput as production routers with hardware support. As EmuLab is a closed environment, it is incapable of combining real Internet workload into experiments. Compared to EmuLab, the ShadowNet infrastructure is distributed, thus the resource placement in ShadowNet more closely resembles future deployment phases. In EmuLab, an experiment in a slice is allocated a fixed set of resources during its life cycle — a change of specification would require a "reboot" of the slice. ShadowNet, on the other hand, can change the specification *dynamically*. In the CDN example, machines for content caches and network links can be dynamically spawned or removed in response to increased or decreased client requests.

PlanetLab has been extremely successful in academic research, especially in distributed monitoring and P2P research. It achieves its goal of amazing geographical coverage, spanning nodes to all over the globe, obtaining great end-host visibility. The PlanetLab nodes, however, are mostly connected to educational networks without abundant upstream or downstream bandwidth. PlanetLab therefore lacks the capacity to realize a capable *backbone* between PlanetLab nodes. ShadowNet, on the other hand, is built upon a production ISP network, having its own virtual backbone with bandwidth and latency guarantees. This pushes the tested service closer to the core of the ISP network, where the actual production service would be deployed.

VINI is closely tied with PlanetLab, but utilizes Internet2 to provide a realistic backbone. Like EmuLab and PlanetLab, VINI runs software routers (XORP and Click), the forwarding capacity of which lags behind production devices. This is mostly because its focus is to use commodity hardware to evaluate new Internet architectures, which is different from the service deployment focus of ShadowNet. VINI and PlanetLab are based on the same control framework. Similar to EmuLab, it lacks the capability of changing slice configurations dynamically, *i.e.,* not closing the loop for more adaptive resource management, a functionality readily available in ShadowNet.

5.2 ShadowNet overview

We present ShadowNet which serves as a platform for rapid and safe network change. The primary goal of ShadowNet is to allow the rapid composition of distributed computing and networking resources, contained in a slice, realized in carrier-grade facilities which can be utilized to introduce and/or test new services or network management tools. The ShadowNet control framework allows the network-wide resources that make up each slice to be managed either collectively or individually.

In the first instance, ShadowNet will limit new services to the set of resources allocated for that purpose, *i.e.,* contained in a slice. This would be a sufficient solution for testing and trying out new services in a realistic environment before introducing such services into a production network. Indeed our current deployment plans espouse this approach with ShadowNet as a separate overlay facility [109] connected to a tier-1 production network. Longer term, however, we expect the base functionality provided by ShadowNet to evolve into the production network and to allow resources and functionality from different slices to be gracefully merged under the control of the ShadowNet control framework.

In the remainder of this section, we describe the ShadowNet architecture and show how it can be used to realize a sophisticated service. Several experimental network platforms are compared against it, and we show that ShadowNet is unique in terms of its ability to provide realistic network testing. Finally we describe the architecture of the primary system component, namely the ShadowNet controller.

5.2.1 ShadowNet architecture

Different viewpoints of the ShadowNet network architecture are shown in Figures 5.2(a) and (b). Figure 5.2(a) shows the topology from the viewpoint of the tier-1 provider. ShadowNet nodes connect to the provider network, but are essentially separate from it. Each ShadowNet node has connectivity to other ShadowNet nodes as well as connectivity to the Internet. As shown in Figure 5.2(b), connectivity to other ShadowNet nodes effectively creates an overlay network [109] to form a virtual

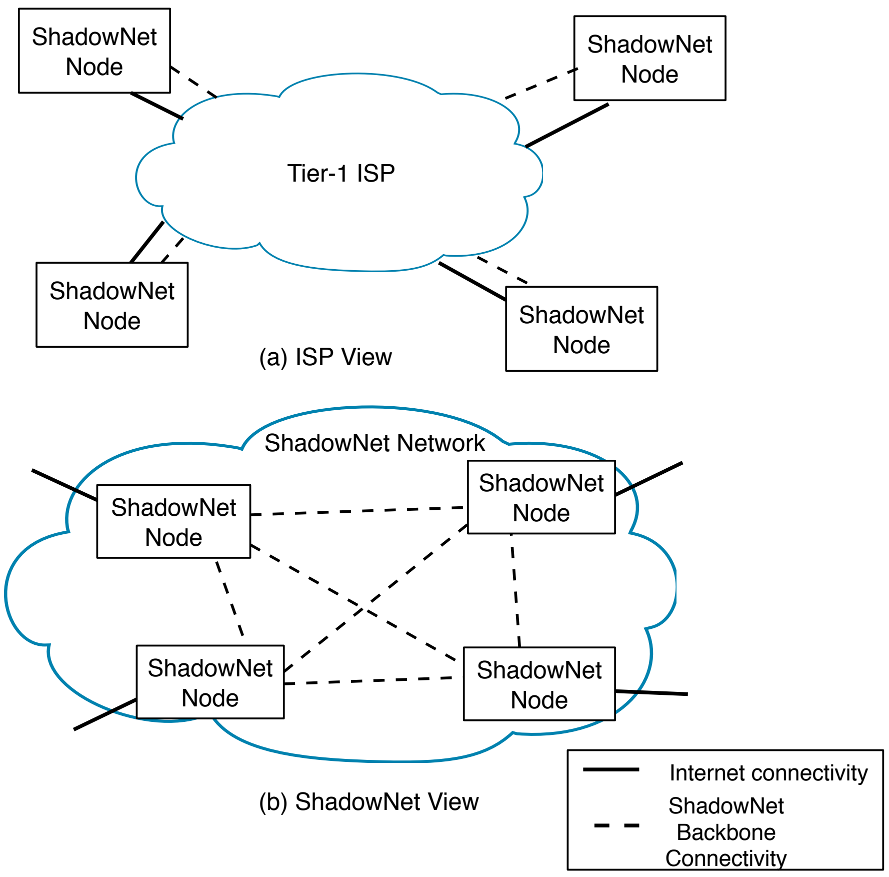


Figure 5.2: ShadowNet network viewpoints

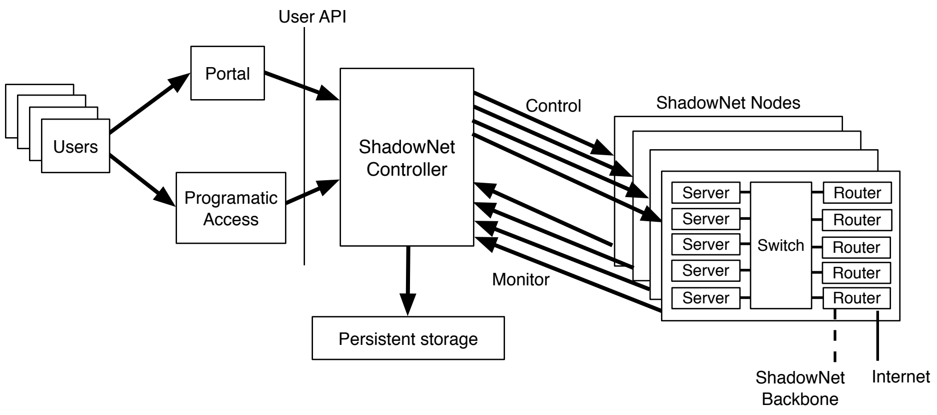


Figure 5.3: ShadowNet functional architecture

backbone among the nodes. Via the provided Internet connectivity, the ShadowNet address space is advertised (*e.g.,* using BGP) first to the provider network and then to the rest of the Internet. Thus ShadowNet effectively becomes a small provider network itself, *i.e.,* a *shadow* of the provider network.

The ShadowNet functional architecture is shown in Figure 5.3. Each ShadowNet node contains different types of computing and networking devices, such as servers, routers, and switches. Combined with the network connectivity received from the ISP, they complete the physical resource for ShadowNet. ShadowNet manages the physical resources and enables its users to share them. The devices provide virtualization/partitioning capabilities so that multiple logical devices can share the same underlying physical resource. For example, modern routers allow router resources to be partitioned so that several logical routers can be configured to run simultaneously and separately on a single physical router [74]. (Note that modern routers are also programmable in both control and data planes [76].) Logical interfaces can be multiplexed from one physical interface via configuration and then assigned to different logical routers. We also take advantage of virtual machine technology to manage server resources [27]. This technology enables multiple operating systems to run simultaneously on the same physical machine and is already heavily used in cloud computing and data-center environments. To facilitate sharing connectivity, the physical devices in each ShadowNet node are connected via a configurable switching layer, which shares the local connectivity, for example using VLANs. The carrier-supporting-carrier capabilities enabled by MPLS virtual private networks (VPNs) [50, 73] offer strong isolation and are therefore an ideal choice to create the ShadowNet backbone.

As depicted in Figure 5.3, central to ShadowNet functionality is the *ShadowNet Controller*. The controller facilitates the specification and instantiation of a service trial in the form of a slice owned by a user. It provides a programmatic application programming interface (API) to ShadowNet users, allowing them to create the topological setup of the intended service trial or deployment. Alternatively users can access ShadowNet through a Web-based portal, which in turn will interact with the ShadowNet Controller via the user-level API. The ShadowNet Controller keeps track of the physical devices that make up each ShadowNet node by constantly monitoring them, and further manages and manipulates those physical devices to realize the user-level APIs, while maintaining a clean separation between the abstracted slice specifications and the way they are realized on the physical equipment. The user-level APIs also enable users to dynamically interact with and manage the physical instantiation of their slices. Specifically, users can directly access and configure each instantiated logical device.

ShadowNet allows a user to deactivate individual devices in a slice or the slice as a whole, by releasing the allocated physical resources. ShadowNet decouples the persistent state from the instantiated physical devices, so that the state change associated with a device in the specification is maintained even if the physical instantiation is released. Subsequently, that device in the specification can be re-instantiated (assuming that sufficient resources are available), the saved state restored and thus the user perceived slice remains intact. For example, the configuration change made by the user to a logical router can be maintained and applied to a new instantiated logical router, even if the physical placement of that logical device is different.

5.2.2 The ShadowNet Controller

The ShadowNet controller consists of a user-level manager, a physical-level manager, a configuration effector and a device monitor, as shown in Figure 5.4. We describe each component below. The current ShadowNet design utilizes a centralized controller that interacts with and controls all ShadowNet nodes.

5.2.2.1 User-level manager

The user-level manager is designed to take the input of user-level API calls. Each API call corresponds to an action that the users of ShadowNet are allowed to perform. A user can create a topological specification of a service trial (§5.3.1), instantiate the specification to physical resources (§5.3.2), interact with the allocated physical resources (§5.3.3), and deactivate the slice when the test finishes (§5.3.4). The topology

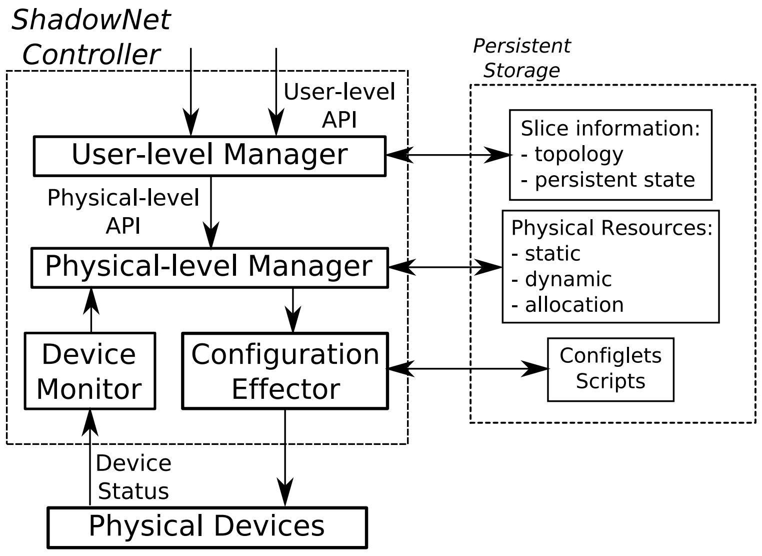


Figure 5.4: The ShadowNet controller

specification of a slice is stored by the user-level manager in persistent storage, so that it can be retrieved, revived and modified over time. The user-level manager also helps maintain and manage the saved persistent state from physical instantiations (§5.3.3). By retrieving saved states and applying them to physical instantiations, advanced features, like device duplication, can be enabled (§5.3.5).

The user-level manager is essentially a network service used to manipulate configurations of user experiments. We allow the user-level manager to be accessed from within the experiment, facilitating network control in a closed-loop fashion. In the example shown in Figure 5.1, the route control component in the experiment can dynamically add content caches when user demand is high by calling the user-level API to add more computing and networking resource via the user-level manager.

5.2.2.2 Physical-level manager

The physical-level manager fulfills requests from the user-level manager in the form of physical-level API calls by manipulating the physical resources in ShadowNet. To do this, it maintains three types of information: 1) "static" information, such as the devices in each ShadowNet node and their capabilities; 2) "dynamic" information, *e.g.,* the online status of all devices and whether any interface modules are not functioning; 3) "allocation" information, which is the up-to-date usage of the physical resources. Static information is changed when new devices are added or old devices are removed. Dynamic information is constantly updated by the device monitor. The three main functions of the physical-level manager is to configure physical devices to spawn virtualized *device sliver*s (§5.4.1) for the instantiation of user-level devices (§5.4.1.1) and user-level connectivities (§5.4.1.2), to manage their states (§5.4.4) and to delete existing instantiated slivers. A *sliver* is a share of the physical resource, *e.g.,* a virtual machine or a sliced physical link. The physical-level manager handles requests, such as creating a VM, by figuring out the physical device to configure and how to configure it. The actual management actions are performed via the configuration effector module, which we describe next.

5.2.2.3 Configuration effector

The configuration effector specializes in realizing configuration changes to physical devices. *Configlets* are parametrized configuration or script templates, saved in the persistent storage and retrieved on demand. To realize the physical-level API calls, the physical-level manager decides the appropriate configlet to use and generates parameters based on the request and the physical resource information. The configuration effector executes the configuration change on target physical devices.

5.2.2.4 Device monitor

A device monitor actively or passively determines the status of physical devices or components and propagates this "dynamic" information to the physical-level manager. Effectively, the device monitor detects any physical device failures in real time. As the physical-level manager receives the update, it can perform appropriate actions to mitigate the failure. The goal is to minimize any inconsistency of physical instantiation and user specifications. We detail the techniques in §5.4.5. Device or component recovery can be detected as well, and as such the recovered resource can again be considered usable by the physical-level manager.

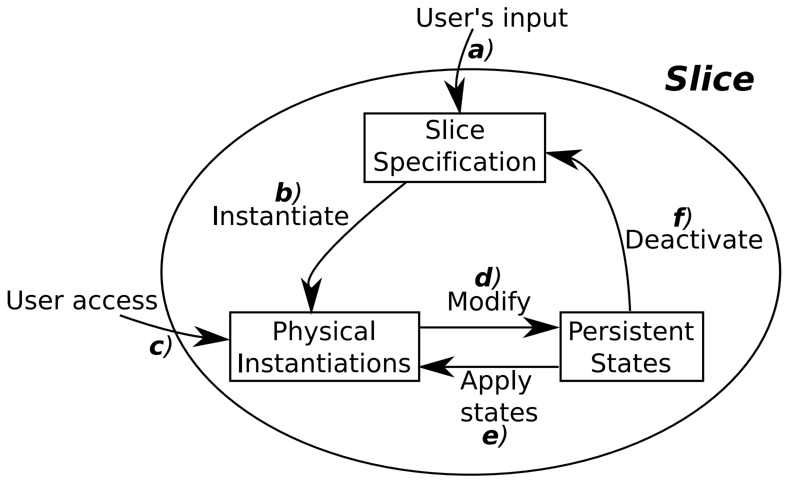


Figure 5.5: The slice life cycle

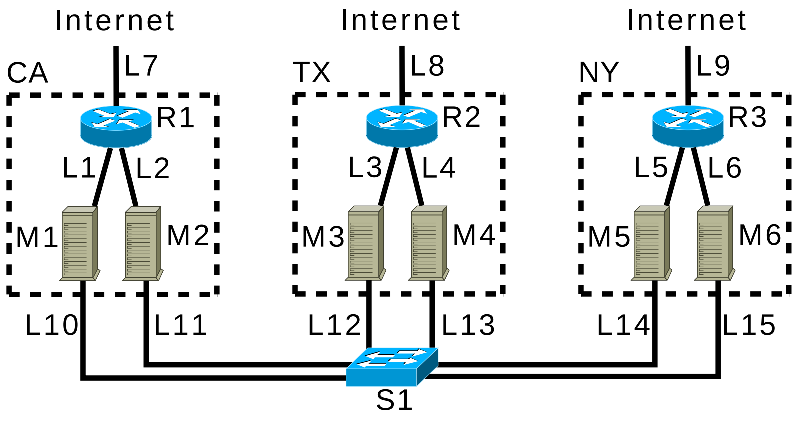
5.3 Network service in a slice

A user of ShadowNet creates a service topology in the form of a slice, which is manipulated through the user-level API calls supported by the ShadowNet controller. The three layers embedded in a slice and the interactions among them are depicted in Figure 5.5 and detailed below. In this section, we outline the main user-exposed functionalities that the APIs implement.

5.3.1 Creating user-level specification

To create a new service trial, an authorized user of ShadowNet can create a slice. As a basic support, and usually the first step to create the service, the user specifies the topological setup through the user-level API (*a* in Figure 5.5). As an example, Figure 5.6 depicts the intended topology of a hypothetical slice and the API call sequence that creates it.

The slice created acts like a placeholder for a collection of *user-level objects*, including devices and connectivities. We support three generic types of user-level devices (UsrDevice): router (UsrRouter), machine (UsrMachine), and switch (UsrSwitch). Two UsrDevices can be connected to each other via a user-level link (UsrLink). User-



```
$SL  = AddUsrSlice();
$S1  = AddUsrSwitch($SL);
$R1  = AddUsrRouter($SL,"CA");
$M1  = AddUsrMachine($SL,"CA","Debian");
$M2  = AddUsrMachine($SL,"CA","Windows");
$L1  = AddUsrLink($M1,$R1); # similar for M2
$L10 = AddUsrLink($M1,$S1); # similar for M2
$L7  = AddToInternet($R1, "141.212.111.0/24");
# similar for "TX" and "NY"
```

Figure 5.6: Example of user-level API calls

level interfaces (UsrInt) can be added to a UsrDevice explicitly by the slice owner; however, in most cases, they are created implicitly when a UsrLink is added to connect two UsrDevices.

Functionally speaking, a UsrMachine (*e.g.,* $M1$ in Figure 5.6) represents a generic computing resource, where the user can run service applications. A UsrRouter (*e.g.,* $R1$) can run routing protocols, forward and filter packets, *etc.* Further, UsrRouters are programmable, allowing for custom router functionality. A UsrLink (*e.g.,* $L1$) ensures that when the UsrDevice on one end sends a packet, the UsrDevice on the other end will receive it. A UsrSwitch (*e.g.,* $S1$) provides a single broadcast domain to the UsrDevices connecting to it. ShadowNet provides the capability and flexibility of putting geographically dispersed devices on the same broadcast domain. For example, $M1$ to $M6$, although specified in different locations, are all connected to UsrSwitch $S1$.

Besides internal connectivity among UsrDevices, ShadowNet can drive live Internet traffic to a service trial by allocating a public IP prefix for a UsrInt on a UsrDevice. For example, $L7$ is used to connect $R1$ to the Internet, allocating an IP prefix of `141.212.111.0/24`.

Besides creating devices and links, a user of ShadowNet can also associate properties with different objects, *e.g.,* the OS image of a UsrMachine and the IP addresses of the two interfaces on each side of a UsrLink. As a distributed infrastructure, ShadowNet allows users to specify location preference for each device as well, *e.g.,* California for $M1$, $M2$ and $R1$. This location information is used by the physical layer manager when instantiation is performed.

5.3.2 Instantiation

A user can instantiate some or all objects in her slice onto physical resources (b in Figure 5.5). From this point on, the slice not only contains abstracted specification, but also has associated physical resources that the instantiated objects in the specification are mapped to.

ShadowNet provides two types of instantiation strategies. First, a user can design a full specification for the slice and instantiate all the objects in the specification together. This is similar to what Emulab and VINI provide. As a second option, user-level objects in the specification can be instantiated upon request at any time. For example, they can be instantiated on-the-fly as they are added to the service specification.This is useful for users who would like to build a slice interactively and/or modify it over time, *e.g.,* extend the slice resources based on increased demand.

Unlike other platforms, such as PlanetLab and EmuLab, which intend to run as many “slices” as possible, ShadowNet limits the number of shares (slivers) a physical resource provides. This simplifies the resource allocation problem to a straightforward availability check. We leave more advanced resource allocation methods as future work.

5.3.3 Device access & persistent slice state

ShadowNet allows a user to access the physical instantiation of the UsrDevices and UsrLinks in her slice, *e.g.,* logging into a router or tapping into a link (c in Figure 5.5). This support is necessary for many reasons. First, the user needs to install software on UsrMachines or UsrRouters and/or configure UsrRouters for forwarding and filtering packets. Second, purely from an operational point of view, operators usually desire direct access to the devices (*e.g.,* a terminal window on a server, or command line access to a router).

For UsrMachines and UsrRouters, we allow users to log into the device and make any changes they want (§5.4.3). For UsrLinks and UsrSwitches, we provide packet dump feeds upon request (§5.4.3). This support is crucial for service testing, debugging and optimization, since it gives the capability and flexibility of sniffing packets at any place within the service deployment without installing additional software on end-points.

Enabling device access also grants users the ability to change the persistent state of the physical instantiations, such as files installed on disks and configuration changes on routers. In ShadowNet, we decouple the persistent states from the physical instantiation. When the physical instantiation is modified, the changed state also become part of the slice (d in Figure 5.5).

5.3.4 Deactivation

The instantiated user-level objects in the specification of a slice can be deactivated, releasing the physical instantiations of the objects from the slice by giving them back to the ShadowNet infrastructure. For example, a user can choose to deactivate an under-utilized slice as a whole, so that other users can test their slices when the physical resources are scarce. While releasing the physical resource, we make sure the persistent state is extracted and stored as part of the slice (f in Figure 5.5). As a result, when the user decides to revive a whole slice or an object in the slice, new physical resources will be acquired and the stored state associated with the object applied to it (*e* in Figure 5.5). Operationally speaking, this enables a user to deactivate a slice and reactivate it later, most likely on a different set of resources but still functioning like before.

5.3.5 Management support

Abstracting the persistent state from the physical instantiation enables other useful primitives in the context of service deployment. If we instantiate a new UsrDevice and apply the state of an existing UsrDevice to it, we effectively duplicate the existing UsrDevice. For example, a user may instantiate a new UsrMachine with only the basic OS setup, log into the machine to install necessary application code and configure the OS. With the support provided by ShadowNet, she can then spawn several new UsrMachines and apply the state of the first machine. This eases the task of creating a cluster of devices serving similar purposes. From the ShadowNet control aspect, this separation allows sophisticated techniques to hide physical device failures. For example, a physical router experiences a power failure, while it hosts many logical routers as the instantiation of UsrRouters. In this case, we only need to create new instantiations on other available devices of the same type, and then apply the states to them. During the whole process, the slice specification, which is what the user perceives, is intact. Naturally, the slice will experience some downtime as a result of the failure.

5.4 Physical layer operations

While conceptually similar to several existing systems [35, 118], engineering ShadowNet is challenging due to the strong isolation concept it rests on, the production-grade qualities it provides and the distributed nature of its realization. We describe the key methods used to realize ShadowNet.

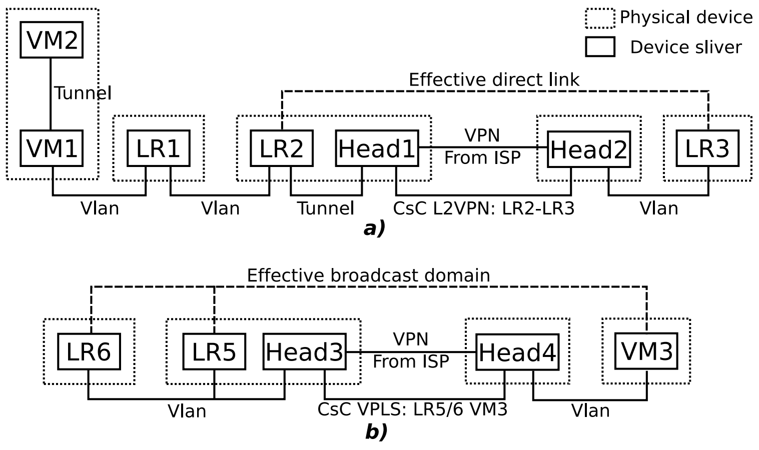


Figure 5.7: Network connectivity options.

5.4.1 Instantiating slice specifications

The slice specification instantiation is performed by the ShadowNet controller in a fully automated fashion. The methods to instantiate on two types of resource are described as follows.

5.4.1.1 User-level routers and machines

ShadowNet currently utilizes VirtualBox [27] from Sun Microsystems, and Logical Routers [74] from Juniper Networks to realize UsrMachines and UsrRouters respectively. Each VM and logical router created is considered as a device *sliver*. To instantiate a UsrRouter or a UsrMachine, a ShadowNet node is chosen based on the location property specified. Then all matching physical devices on that node are enumerated for availability checking, *e.g.,* whether a Juniper router is capable of spawning a new logical router. When there are multiple choices, we distribute the usage across devices in a round-robin fashion. Location preference may be unspecified because the user does not care about where the UsrDevice is instantiated, *e.g.,* when testing a router configuration option. In this case, we greedily choose the ShadowNet node where that type of device is the least utilized. When no available resource can be allocated, an error is returned.

5.4.1.2 User-level connectivity

The production network associated with ShadowNet provides both Internet connection and virtual backbone connectivity to each ShadowNet node. We configure a logical router, which we call the *head* router of the ShadowNet node, to terminate these two connections. With the ShadowNet backbone connectivity provided by the ISP, all head routers form a full-mesh, serving as the core routers of ShadowNet. For Internet connectivity, the head router interacts with ISP's border router, *e.g.,* announcing BGP routes.

Connecting device slivers on the same ShadowNet node can be handled by the switching layer of that node. The head routers are used when device slivers across nodes need to be connected. In ShadowNet, we make use of the carrier-supporting-carrier (CsC) capabilities provided by MPLS enabled networks. CsC utilizes the VPN service provided by the ISP, and stacks on top of it another layer of VPN services, running in parallel but isolated from each other. For example, layer-2 VPNs (so called pseudo-wire) and VPLS VPNs can be stacked on top of a layer-3 VPN service [73].

This approach has three key benefits. First, each layer-2 VPN or VPLS instance encapsulates the network traffic within the instance, thus provides strong isolation across links. Second, these are off-the-shelf production-grade services, which are much more efficient than manually configured tunnels. Third, it is more realistic for the users, because there is no additional configuration needed in the logical routers they use. The layer-2 VPN and VPLS options that we heavily use in ShadowNet provides layer-2 connectivity, *i.e.,* with router programmability, any layer-3 protocol besides IP can run on top of it.

Figure 5.7 contains various examples of enabling connectivity, which we explain in detail next.

UsrLink: To instantiate a UsrLink, the instantiations of the two UsrDevices on the two ends of the UsrLink are first identified. We handle three cases, see Figure 5.7a). (We consider the UsrLinks connected to a UsrSwitch part of that UsrSwitch, which we describe later):

1) Two slivers are on the same physical device: for example, $VM1$ and $VM2$ are on the same server; $LR2$ and $Head1$ are on the same router. In this case, we use local bridging to realize the UsrLink.

2) Two slivers are on the same ShadowNet node, but not the same device: for example, $VM1$ and $LR1$, $LR1$ and $LR2$. We use a dedicated VLAN on that node for each UsrLink of this type, *e.g.,*, $LR1$ will be configured with two interfaces, joining two different VLAN segments, one for the link to $VM1$, the other one to $LR2$.

3) Two slivers are on different nodes: for example, $LR2$ and $LR3$. In this case, we first connect each sliver to its local head router, using the two methods above. Then the head router creates a layer-2 VPN to bridge the added interfaces, effectively creating a cross-node tunnel connecting the two slivers.

In each scenario above, the types of the physical interfaces that should be used to enable the link are decided, the selected physical interfaces are configured, and the resource usage information of the interfaces is updated.

MPLS-VPN technologies achieve much higher levels of realism over software tunnels, because almost no configuration is required at the end-points that are being connected. For example, to enable the direct link between $LR2$ and $LR3$, the layer-2 VPN configuration only happens on $Head1$ and $Head2$. As a result, if the user logs into the logical router $LR2$ after its creation, she would only sees a "physical" interface setup in the configuration, even without IP configured, yet that interface leads to $LR3$ according to the layer-2 topology.

User-view switches: Unlike for UsrMachines and UsrRouters, ShadowNet does not allocate user-controllable device slivers for the instantiation of UsrSwitches, but rather provide an Ethernet broadcasting medium. (See Figure 5.7b).)

To instantiate a UsrSwitch connecting to a set of UsrDevices instantiated on the same ShadowNet node, we allocate a dedicated VLAN-ID on that node and configure those device slivers to join the VLAN (*i.e.,* $LR5$ and $LR6$). If the device slivers mapped to the UsrDevices distribute across different ShadowNet nodes, we first recursively bridge the slivers on the same node using VLANs, and then configure one VPLS-VPN instance on each head router (*i.e.,* $Head3$ and $Head4$) to bridge all those VLANs. This puts all those device slivers (*i.e.,* $VM3$, $LR5$, $LR6$) onto the same broadcast domain. Similar to layer-2 VPN, this achieves a high degree of realism, for example on $LR5$ and $LR6$, the instantiated logical router only shows one "physical" interface in its configuration.

Internet access: We assume that ShadowNet nodes can use a set of prefixes to communicate with any end-points on the Internet. The prefixes can either be announced through BGP sessions configured on the head routers to the ISP's border routers, or statically configured on the border routers.

To instantiate a UsrDevice's Internet connectivity, we first connect the UsrDevice's instantiation to the head router on the same node. Then we configure the head router so that the allocated prefix is correctly forwarded to the UsrDevice over the established link and the route for the prefix is announced via BGP to the ISP. For example, a user specifies two UsrRouters connecting to the Internet, allocating them with prefix `136.12.0.0/24` and `136.12.1.0/24`. The head router should in turn announce an aggregated prefix `136.12.0.0/23` to the ISP border router.

5.4.2 Achieving isolation and fair sharing

As a shared infrastructure for many users, ShadowNet attempts to minimize the interference among the physical instantiation of different slices. Each virtual machine is allocated with its own memory address space, disk image, and network interfaces. However, some resources, like CPU, are shared among virtual machines, so that one virtual machine could potentially drain most of the CPU cycles. Fortunately, virtual machine technology is developing better control over CPU usage of individual virtual machines [27].

A logical router on a Juniper router has its own configuration file and maintains its own routing table and forwarding table. However, control plane resources, such as CPU and memory are shared among logical routers. We evaluate this impact in §5.6.3.

The isolation of packets among different UsrLinks is guaranteed by the physical device and routing protocol properties. We leverage router support for packet filtering and shaping, to prevent IP spoofing and bandwidth abuse. The corresponding configuration is made on head routers, where end-users cannot access. For each UsrLink, we impose a default rate-limit (*e.g.,* 10Mbps), which can be upgraded by sending a request via the user-level API. We achieve rate limiting via hardware traffic policers [77] and Linux kernel support [24].

5.4.3 Enabling device access

Console or remote-desktop access: For each VM running on VirtualBox, a port is specified on the hosting server to enable Remote Desktop protocol for graphical access restricted to that VM. If the user prefers command line access, a serial port console in the VM images is enabled and mapped to a UNIX domain socket on the hosting machine's file system [27]. On a physical router, each logical router can be configured to be accessible through SSH using a given username and password pair, while confining the access to be within the logical router only.

Though the device slivers of a slice can be connected to the Internet, the management interface of the actual physical devices in ShadowNet should not be. For example, the IP address of a physical server should be contained within ShadowNet rather than accessible globally. We thus enable users to access the device slivers through one level of indirection via the ShadowNet controller.

Sniffing links: To provide packet traces from a particular UsrLink or UsrSwitch, we dynamically configure a SPAN port on the switching layer of a ShadowNet node so that a dedicated server or a pre-configured VM can sniff the VLAN segment that the UsrLink or UsrSwitch is using. The packet trace can be redirected through the controller to the user in a streaming fashion or saved as a file for future downloading. There are cases where no VLAN is used, *e.g.,* for two logical routers on the same physical router connected via logical tunnel interfaces. In this case, we deactivate the tunnel interfaces and re-instantiate the UsrLink using VLAN setup to support packet capture. This action, however, happens at the physical-level and thus is transparent to the user-level, as the slice specification remains intact.

5.4.4 Managing state

To extract the state of an instantiated UsrMachine, which essentially is a VM, we keep the hard drive image of the virtual machine. The configuration file of a logical router is considered as the persistent state of the corresponding UsrRouter. Reviving stored state for a UsrMachine can be done by attaching the saved disk image to a newly instantiated VM. On the other hand, UsrRouter state, *i.e.,* router configuration files, need additional processing. For example, a user-level interface may be instantiated as interface `fe-0/1/0.2` and thus appear in the configuration of the instantiated logical router. When the slice is deactivated and instantiated again, the UsrInt may be mapped to a different interface, say `ge-0/2/0.1`. To deal with this complication, we normalize the retrieved configuration and replace physical-dependent information with user-level object handles, and save it as the state.

5.4.5 Mitigating and creating failures

Unexpected physical device failures can occur, and as an option ShadowNet tries to mitigate failures as quickly as possible to reduce user perceived down time. One benefit of separating the states from the physical instantiation is that we can replace a new physical instantiation with the saved state applied without affecting the user perception. Once a device or a physical component is determined to be offline, ShadowNet controller identifies all instantiated user-level devices associated to it. New instantiations are created on healthy physical devices and saved states are applied if possible. Note that certain users are specifically interested in observing service behavior during failure scenarios. We allow the users to specify whether they want physical failures to pass through, which is disabling our failure mitigation functionality. On the other hand, failure can be injected by the ShadowNet user-level API, for example tearing down the physical instantiation of a link or a device in the specification to mimic a physical link-down event.

For physical routers, the device monitor performs periodic retrieval of the current configuration files, preserving the states of UsrRouters more proactively. When a whole physical router fails, the controller creates new logical routers with connectivity satisfying the topology on other healthy routers and applies the saved configuration, such as BGP setup. If an interface module fails, the other healthy interfaces on the same router are used instead. Note that the head router is managed in the same way as other logical routers, so that ShadowNet can also recover from router failures where head routers are down.

A physical machine failure is likely more catastrophic, because it is challenging to recover files from a failed machine and it is not feasible to duplicate large files like VM images to the controller. One potential solution is to deploy a distributed file system similar to the Google file system [60] among the physical machines within one ShadowNet node. We leave this type of functionality for future work.

5.5 Prototype Implementation

In this section, we briefly describe our prototype implementation of the ShadowNet infrastructure, including the hardware setup and management controller.

5.5.1 Hardware setup

Currently, a four node ShadowNet instance is deployed as an operational network with nodes in Texas, Illinois, New Jersey and California. Each node has two gigabit links to the production network, one used as regular peering link and the other used as the dedicated backbone.

In this section, we use our original local two-node deployment as evaluation. Each prototype node has two Juniper M7i routers running JUNOS version 9.0, one Cisco C2960 switch, as well as four HP DL520 servers. The M7i routers are equipped with one or two Gigabit Ethernet PICs (Physical Interface Cards), FastEthernet PIC, and tunneling capability. Each server has two gigabit Ethernet interfaces, and we install

VirtualBox in the Linux Debian operating system to host virtual machines. The switch is capable of configuring VLANs and enabling SPAN ports.

In the local deployment, two Cisco 7206 routers act as an ISP backbone. MPLS is enabled on the Cisco routers to provide layer-3 VPN service as the ShadowNet backbone. BGP sessions are established between the head router of each node and its adjacent Cisco router, enabling external traffic to flow into ShadowNet. We connect the network management interface `fxp0` of Juniper routers and one of the two Ethernet interfaces on machines to a dedicated and separate management switch. These interfaces are configured with private IP addresses, and used for physical device management only, mimicking the out-of-band access which is common in ISP network management.

5.5.2 Controller

The ShadowNet controller runs on a dedicated machine, sitting on the management switch. The controller is currently implemented in Perl. A Perl module, with all the user-level APIs, can be imported in Perl scripts to create, instantiate and access service specifications, similar to the code shown in Figure 5.6. A `mysql` database is running on the same machine as the controller, serving largely, though not entirely, as the persistent storage connecting to the controller. It saves the physical device information, user specifications, and normalized configuration files, etc. We use a different set of tables to maintain physical-level information, *e.g.,*, `phy_device_table`, and user-level information, *e.g.,*, `usr_link_table`. The Perl module retrieves information from the tables and updates the tables when fulfilling API calls.

The configuration effector of the ShadowNet controller is implemented within the Perl module as well. We make use of the NetConf XML API exposed by Juniper routers to configure and control them. Configlets in the form of parametrized XML files are stored on the controller. The controller retrieves the configuration of the physical router in XML format periodically and when UsrRouters are deactivated. We wrote a specialized XML parser to extract individual logical router configurations and normalize relative fields, such as interface related configurations. The normalized configurations are serialized in text format and stored in the mysql database associating to the specific UsrRouter.

Shell and Perl scripts, which wrap the VirtualBox management interface, are executed on the hosting servers to automatically create VMs, snapshot running VMs, stop or destroy VMs. The configuration effector logs into each hosting server and executes those scripts with the correct parameters. On the servers, we run low-priority `cron` jobs to maintain a pre-configured number of default VM images of different OS types. In this case, the request of creating a new VM can be fulfilled fairly quickly, amortizing the overhead across time. We use the following steps to direct the traffic of an interface used by a VM to a particular VLAN. First, we run `tunctl` on the hosting server to create a `tap` interface, which is configured in the VMM to be the "physical" interface of the VM. Second, we make use of 802.1Q kernel module to create VLAN interfaces on the hosting server, like `eth1.4`, which participates in VLAN4. Finally we use `brctl` to bridge the created tap interface and VLAN interface.

Instead of effecting one configuration change per action, the changes to the physical devices are batched and executed once per device, thus reducing authentication and committing overheads. All devices are manipulated in parallel. We evaluate the effectiveness of these two heuristics in §5.6.1.

The device monitor module is running as a daemon on the controller machine. SNMP trap messages are enabled on the routers and sent over the management channel to the controller machine. `Ping` messages are sent periodically to all devices. The two sources of information are processed in the background by the monitoring daemon. When failures are detected, the monitoring module calls the physical-level APIs in the Perl module, which in response populates configlets and executes on the routers to handle failures. An error message is also automatically sent to the administrators.

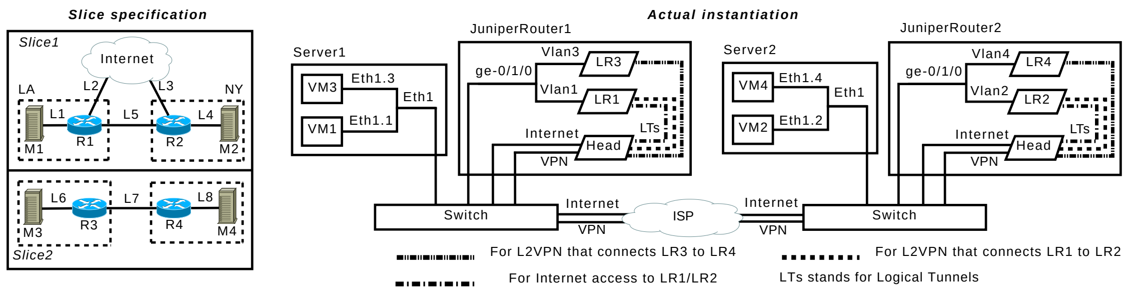


Figure 5.8: User slices for evaluation

5.6 Prototype Evaluation

In this section, we evaluate various aspects of ShadowNet based on two example slices instantiated on our prototype. The user specifications are illustrated on the left side of Figure 5.8; the physical realization of that specification is on the right. In *Slice*1, two locations are specified, namely LA and NY. On the LA side, one UsrMachine (M1) and one UsrRouter (R1) are specified. R1 is connected to M1 through a UsrLink. R1 is connected to the Internet through L2 and to R2 directly via L5. The setup is similar on NY side. We use minimum IP and OSPF configuration to enable the correct forwarding between M1 and M2. *Slice*2 has essentially the same setup, except that the two UsrRouters do not have Internet access.

The right side of Figure 5.8 shows the instantiation of *Slice*1 and *Slice*2. VM1 and LR1 are the instantiation of M1 and R1 respectively. UsrLink L1 is instantiated as a dedicated channel formed by virtualized interfaces from physical interfaces, `eth1` and `ge-0/1/0`, configured to participate in the same VLAN. To create the UsrLink L5, ShadowNet first uses logical tunnel interfaces to connect LR1 and LR2 with their head routers, which in turn bridge the logical interfaces using layer-2 VPN.

5.6.1 Slice creation time

Table 5.2 shows the creation time for *Slice*1, broken down into instantiation of machine and router, along with database access (DB in the table.) Using a naive approach, the ShadowNet controller needs to spend 82 seconds on the physical routers

	Router	Machine	DB	Total
Default (ms)	81834	11955	452	94241
Optimized (ms)	6912	5758	452	7364

Table 5.2: Slice creation time comparison

bandwidth (Kbps)	packet size	Observed bandwidth	Delta (%)
56	64	55.9	.18
	1500	55.8	.36
384	64	383.8	.05
	1500	386.0	.52
1544	64	1537.2	.44
	1500	1534.8	.60
5000	1500	4992.2	.16
NoLimit	1500	94791.2	NA

Table 5.3: Cross-node link stress test

alone by making 13 changes, resulting a 94-second execution time in total. For machine configuration, two scripts are executed for creating the virtual machines, and two for configuring the link connectivity. With the two simple optimization heuristics described in §5.5.2, the total execution time is reduced to 7.4 seconds. Note that the router and machine configurations are also parallelized, so that we have $total = DB + max(Router_i, Machine_j)$. Parallelization ensures that the total time to create a slice does *not* increase linearly with the size of the slice. We estimate creation time for most slices to be within 10 seconds.

5.6.2 Link stress test

We perform various stress tests to examine ShadowNet's capability and fidelity. We make L5 the bottleneck link, setting different link constraints using Juniper router's traffic policer, and then test the observed bandwidth M1 and M2 can achieve on the link by sending packets as fast as possible. Packets are dropped from the head of the queue. The results are shown in Table 5.3, demonstrating that ShadowNet can closely mimic different link capacities.

When no constraint is placed on L5, the throughput achieved is around 94.8Mbps, shown as "NoLimit" in the table. This is close to maximum, because the routers we used as ISP cores are equipped with FastEthernet interfaces, which have 100Mbps capacity and the VM is specified with 100Mbps virtual interface. Physical gigabit switches are usually not the bottleneck, as we verified that two physical machines on the same physical machines connected via VLAN switch can achieve approximately 1Gbps bandwidth.

As we are evaluating on a local testbed, the jitter and loss rate is almost zero, while the delay is relatively constant. We do not expect this to hold in our wide-area deployment.

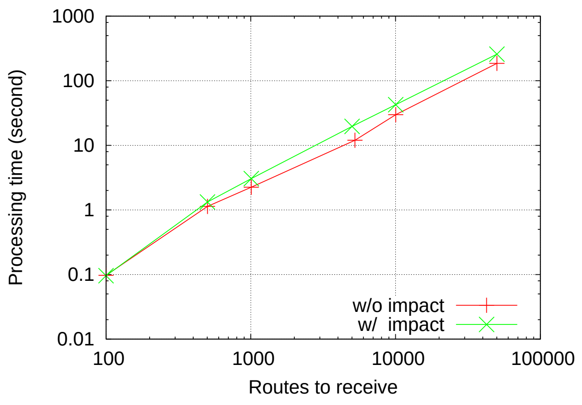
5.6.3 Slice isolation

We describe our results in evaluating the isolation assurance from the perspectives of both the control and data plane.

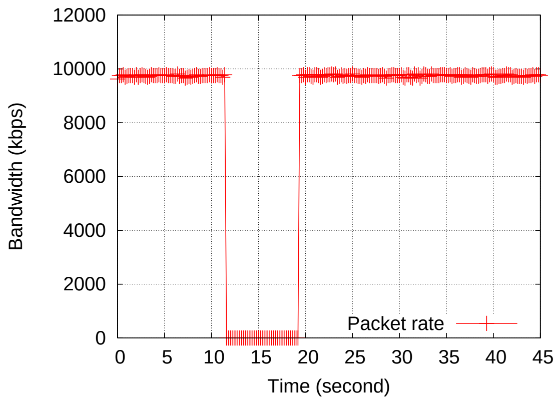
5.6.3.1 Control plane

To understand the impact of a stressed control plane on other logical routers, we run software routers, `bgpd` of `zebra`, on both M1 and M3. The two software routers are configured to peer with the BGP processes on LR1 and LR3. We load the software routers with BGP routing tables of different sizes, transferred to LR1 and LR3. The BGP event log on the physical router is analyzed by measuring the duration from the first BGP update message to the time when all received routes are processed.

In Figure V.9(a), the bottom line shows the processing time of the BGP process on LR1 to process all the routes if LR3 is BGP-inactive. The top line shows the processing time for LR1 when LR3 is also actively processing the BGP message stream. Both processing times increase linearly with the number of routes received. The two lines are almost parallel, meaning that the delay is proportional to the original processing time. The difference of receiving 10k routes is about 13 seconds, 73 seconds



(a) Impact of shared control planes



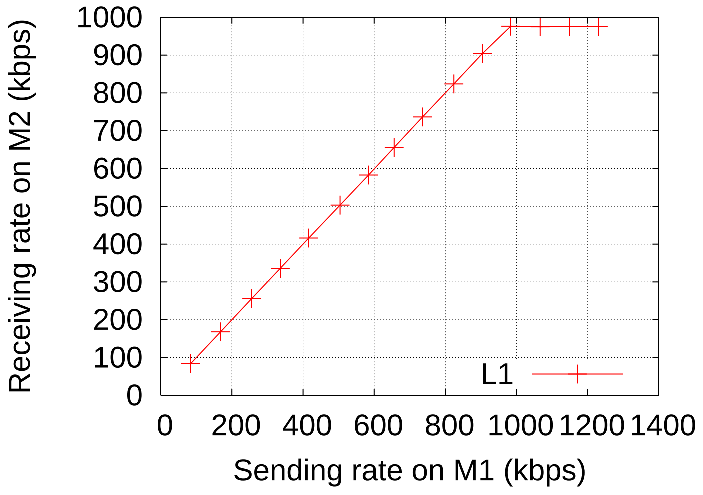
(b) Hardware failure recovery

Figure 5.9: Control plane isolation and recovery test.

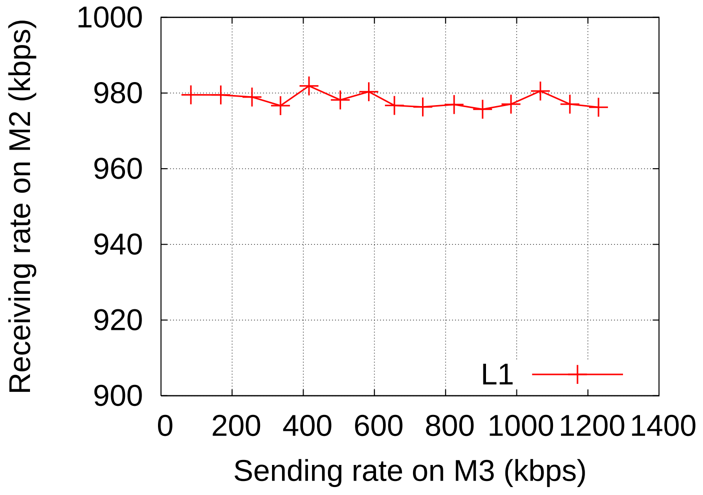
for 50k routes. We have verified that the CPU usage is 100% even if only LR1 is BGP-active. We have also used two physical machines to peer with LR1 and LR3 and confirmed that the bottleneck is due to the Juniper router control processor. If these limitations prove to be problematic in practice, solutions exist which allow a hardware separation of logical router control planes [75].

5.6.3.2 Data plane

L1 and L6 share the same physical interfaces, `eth1` on $Server1$ and `ge-0/1/0` on $JuniperRouter1$. We restrict the bandwidth usage of both L1 and L6 to be 1Mbps



(a) Variable packet rate (L6's rate is maxed)



(b) Max packet rate (L6's rate is variable)

Figure 5.10: Data plane isolation test.

by applying traffic policer on the ingress interfaces on LR1 and LR3. From the perspective of a given UsrLink, say $L1$, we evaluate two aspects: regardless of the amount of traffic sent on $L6$, (1) $L1$ can always achieve the maximum bandwidth allocated (*e.g.,* 1Mbps given a 100Mbps interface); (2) $L1$ can always obtain its fair share of the link. To facilitate this test, we apply traffic policer on the ingress interfaces (`ge-0/1/0`) on LR1 and LR3, restricting the bandwidth of L1 and L6 to 1Mbps. Simultaneous traffic is sent from M1 via L1 to M2, and from M3 via L6 to M4.

Figure 5.10(a) shows the observed receiving rate on M2 (y-axis) as the sending rate of M1 (x-axis) increases, while M3 is sending as fast as possible. The receiving rate matches closely with the sending rate, before reaching the imposed 1Mbps limit, This demonstrates that $L1$ capacity is not affected, even if $L6$ is maxed out. Figure 5.10(b) shows the max rate of $L1$ can achieve is always around 980kbps no matter how fast $M2$ is sending.

5.6.4 Device failure mitigation

We evaluate the recovery time in response to a hardware failure in ShadowNet. While $Slice1$ is running, M1 continuously sends packets to M2 via L1. We then physically yanked the Ethernet cable on the Ethernet module `ge-0/1/0`, triggering SNMP `LinkDown` trap message and the subsequent reconfiguration activity. A separate interface (not shown in the figure) is found to be usable, then automatically configured to resurrect the down links. Figure V.9(b) shows the packet rate that M2 observes. The downtime is about 7.7 seconds, mostly spent on effecting router configuration change. Failure detection is fast due to continuous SNMP messages, and similarly controller processing takes less than 100ms. This exemplifies the benefit of strong isolation in ShadowNet, as the physical instantiation is dynamically replaced using the previous IP and OSPF configuration, leaving the user perceived slice intact after a short interruption. To further reduce the recovery time, the ShadowNet controller can spread a UsrLink's instantiation onto multiple physical interfaces, each of which provides a portion of the bandwidth independently.

5.7 Summary

In this chapter, we propose an architecture called ShadowNet, designed to accelerate network change in the form of new networks services and sophisticated network operation mechanisms. Its key property is that the infrastructure is connected to, but functionally separated from a production network, thus enabling more realistic service testing. The fact that production-grade devices are used in ShadowNet greatly improves the fidelity and realism achieved. In the design and implementation of ShadowNet, we created strong separation between the user-level representations from the physical-level instantiation, enabling dynamic composition of user-specified topologies, intelligent resource management and transparent failure mitigation. ShadowNet is now a deployed infrastructure in a tier-one ISP, and indeed helped the evaluation of PACMAN and COOLAID. Though ShadowNet currently provides primitives mainly for service testing purposes, as a next step, we seek to broaden the applicability of ShadowNet, in particular, to merge the control framework into the production network for allowing service deployment.

CHAPTER VI

Related Work

6.1 Understanding current network management practice

When the network functionality is described by a distributed set of low-level configurations, it is difficult to derive the high-level services that the network enables. Thus an important line of related work is about bridging this gap by helping people to understand the network-wide implication of the configurations, thus enabling fault diagnosis, performance diagnosis, *etc.* It has been shown that managing networks through router configuration is a challenging task and a significant contributor to unsatisfactory network availability [96]. Many existing studies in this area of configuration management in IP networks focus on extracting network service features, *e.g.,* quality of service setup [108], and diagnosing network-wide misconfigurations from router configuration files [55, 84, 119]. Our DFA work is different from previous work in that we analyze data capturing continuous changes of the configuration of most routers in the network. Our approach provides more fine-grained analysis and reveals many interesting issues that are impossible to be discovered by studying static configuration snapshots. For example, incorrect ordering of management operations could lead to severe transient network disruptions, even though the eventual configuration state is correct.

The EDGE architecture [41] had the ambitious goal of moving to automated network configuration by first analyzing existing network configuration data to identify the network's policies and to then use such network intelligence to create a database to drive future automated configuration changes. This work seemed to have stopped short of actually taking the last step to create an automated configuration system and is therefore more similar to efforts that attempted to analyze the correctness of existing network configurations [55, 58].

Other systems are built to understand the potential impact of network problems, thus better prepare the network in problematic states. Reval [113] is a system to scale up the magnitude of DDoS attacks to see how tolerable a network is. Kim *et al.* [95] evaluated the impact of multiple network failures and the effectiveness of different protection scheme. Mao *et al.* [91] analyzed the negative impact of BGP route damping to network routing convergence. These analysis can be integrated into our COOLAID framework, when the analysis is expressed in the declarative language. We plan to explore building more generic reasoning support using COOLAID.

6.2 Dirty-slate solutions

A variety of tools and systems are built to assist the current management practice. Our solutions can be largely described as dirty-slate as well, because we do not require any infrastructure or hardware modification. We categorize them as follows.

6.2.1 Configuration management systems

The state-of-the-art in network configuration management is exemplified by the PRESTO configuration management system [53]. The PRESTO system uses configuration templates, created by network experts and called active templates, which are populated by instance-specific parameters to generate specific configuration specifications. The generated configurations can be optionally validated [112]. PRESTO's active templates, however, do not specify any conditional predicates that have to be satisfied before the next set of configurations are applied. Therefore, PRESTO templates lack the necessary execution logic to capture sophisticated operational tasks, or indeed the ability to intelligently create composed tasks from simpler components. Also, the dependencies among templates and between the generated snippets and the existing configurations, still need to be resolved manually Similar to PRESTO, a simple template based approach has been used to automate BGP configuration of new customer links [63]. While also limited to BGP configuration, the work by Bohm *et al.* [38] had a broader scope in that it addressed the creation of configuration files to realize network-wide inter-domain routing policies.

In contrast, COOLAID advocates using declarative rules as a concise and compact representation of domain knowledge, which can be contributed by both vendors and service providers. The reasoning support is generic to all services. COOLAID further provides constraint checking with transactional semantics, not simply emitting configuration snippets to network devices. PACMAN does not generate configurations, but can automatically perform generic configuration modification procedures without human involvement.

NCG [14] is a recently published system from the network operator community. They recognize the same urgent need to automate network configuration to ensure correctness and timeliness while meeting scalability requirements. The tool can build network configurations, but currently still requires manually pushing the changes to the network elements. COOLAID, on the other hand, seamlessly integrates the two. NCG advocates the separation of workload between architects and engineers: architects specify the models and templates, while engineers can modify the data in the model. This is conceptually similar to both PACMAN and COOLAID. In PACMAN, we build assisted environments to make active document generation easier, while in COOLAID, the architects only need to develop declarative rules based on the database abstraction, which is much easier than dealing with low-level configuration languages.

Besides NCG, there are many tools that originate from operators. For example, RANCID [18] helps push and identify (diff) configuration changes. NetworkAuthority Inventory [16] (previously known as ZipTie) and NETDISCO [13] help discover network devices and map network topology. These tools are complementary to our work. For example, RANCID does not generate configurations, but rather provides the mechanism to realize and manage configuration changes. Similarly the network discovery tools can be used to create and maintain a network inventory database for COOLAID.

6.2.2 Network operation support

Major router vendors have proposed their own network management automation frameworks [3, 10]. Those management frameworks remain device-centric and are mostly used to handle local failure response, while PACMAN and COOLAID allows a full spectrum of network management tasks with a network-wide perspective.

Alimi *et al.* suggest the use of "shadow configurations" which would allow both an old and a new network configuration to be active in the same network element [31]. This would allow an operator to test new configuration before activating it. While limiting the potential impact of configuration changes, this approach does not do anything to simplify or automate network configuration changes — the operators still have to manually configure the shadow configuration. Similarly, the VROOM work [114] attempts to avoid configuration changes altogether through the live migration of virtual routers. Again this work indirectly deals with the complexity of configuration management without per se making it any easier.

Maestro [40] features reasoning about network states to safe-guard network operations. However, it falls short of PACMAN and COOLAID, because our solutions are more general. For example, PACMAN can be composed to realize a variety of operations, and COOLAID can be used to automatically generate configuration.

A number of "autonomic" network architectures are related to PACMAN [62, 107]. Conceptually the FOCALE architecture [107] is perhaps the closest to PACMAN. Specifically, like PACMAN, the FOCALE architecture contains an explicit control loop so that network operations can be cognizant of network conditions. However, unlike PACMAN, which closely models current operational practices and ties in with existing network capabilities, the FOCALE approach requires the adoption of new paradigms and tools.

6.2.3 Rule-based management systems

There are also many existing systems that apply rule-based approaches to general system management. On the commercial side, IBM's Tivoli management framework and HP's OpenView allow event-driven rules to be expressed and automated for system management. These languages are best suited for reacting to system condition changes by triggering pre-defined procedural code, but not suitable for specifying domain knowledge of network protocol behaviors and dependencies. On the research side, InfoSpect [103], Sophia [116] and Rhizoma [28] all proposed to use logic programming to manage nodes in large-scale distributed systems such as PlanetLab or cloud environments. Providing advanced support for and meeting the distinct requirements of network management, COOLAID's main techniques differ drastically from those systems. For example, features like distributed recursive query processing, view update resolution, and transactional semantics with constraint enforcement, are all unique to COOLAID.

6.2.4 Declarative systems

Declarative programming in system and networking domains has gained considerable attention in recent years. The declarative networking project proposes a distributed recursive query language to specify and implement traditional routing protocols at the control plane [88]. The declarative approach has been explored by numerous projects, *e.g.,* to implement overlays [87], data and control plane composition [90], specify distributed storage policies [36], building sensor network applications [49], and simulate Byzantine fault tolerant protocols [105]. Compared to those studies, COOLAID focuses on re-factoring current network management and operations practices. Specifically, in COOLAID the declarative language is used for describing domain knowledge, like dependencies and restrictions among network components, as opposed to implementing protocols for execution or simulation. As a stand-alone management plane, COOLAID orchestrates network devices in a declarative fashion, while not requiring the existing routers to be modified. Our work shows that COOLAID works with off-the-shelf carrier-grade routers and is designed for production environments.

6.3 Clean-slate solutions

Several proposals exist that address network management complexity through approaches that are less tethered to current operational practices and device limitations [31, 33, 114]. These works do not cover the full range of network management tasks required in operational networks [33], or attempt to limit the potential negative impact of configuration management without directly addressing its complexity [31, 114]. One such approach is the CONMan architecture [33], in which device and network functionality is captured by abstract modules which describe either data plane or control plane functions. Certain network configuration tasks, specifically connectivity management tasks, can be readily accomplished via manipulation of these abstract modules. It is not clear, however, to what extent this approach is applicable to more generic network operational tasks.

Relating to the 4D project [67], COOLAID fulfills the functionalities of the decision and dissemination planes. Tesseract [120] is an implemented control plane under the 4D concept. It is designed to a flexible platform such that different management tools and algorithms can be plugged in easily. To the contrary, COOLAID use a unified declarative language to capture the domain knowledge, which is expressed accurately and succinctly.

In the enterprise network management space, Ethane [42] and NOX [68] focus on network flow access control management. Along the same line, Flow-based Management Language [69] is based on the Datalog syntax to express policies of flow control. These resemble most of the policy-based network management work [7]. In contrast, the language proposed in COOLAID effectively captures domain knowledge in protocol behaviors and dependencies.

6.4 Network evaluation systems

ShadowNet has much in common with other test/trial networks [35, 97, 118]. However, to our knowledge, ShadowNet is the first platform to exploit recent advances in the capabilities of networking equipment to provide a sharable, composable and programmable infrastructure using carrier-grade equipment running on a production ISP network. This enables a distinct emphasis shift from experimentation/prototyping (enabled by other test networks), to service trial/deployment (enabled by ShadowNet). The fact that ShadowNet utilizes production quality equipment frees us from having to deal with low-level virtualization/partitioning mechanisms, which typically form a significant part of other sharable environments. Please refer to Section5.1.2 for comparison between ShadowNet and other networking experimental testbeds.

From a composable testbed perspective, Shadownet is closely related to VINI [35] and Emulab [52, 102, 118]. Emulab and VINI take user specifications of a network setup and realize it through simulation or emulation on commodity PCs within a local testbed or distributed across wide-area networks. Emulab, in particular, focuses on facilitating network experiment, providing a batch-like interface, with which all events are serialized and effected using a state machine. ShadowNet achieves the similar goal on a set of heterogeneous network devices, including commercial-grade slicable routers, virtual machine, *etc.* As a result, the realization process in Shadownet is much more sophisticated, while gaining an even higher level of realism and usability. The current implementation of VINI is closely coupled with PlanetLab [97] nodes with Internet-2 [8] connectivity. Similarly, Shadownet nodes are deployed connecting to or close to ISP backbone for production-grade network access and potential traffic demand. Emulab, in particular, considers resource allocation [101] as a crucial component for managing the system. ShadowNet also provides resource management support. Beyond Emulab and VINI, Shadownet provides a set of primitives that enhance the interaction between the user and the testbed, making it an easy-to-use platform to deploy real services. ModelNet [110] is a network testing environment, where end-hosts installed with unmodified software connect to an emulated network core to test Internet-scale services. ModelNet trades off accuracy for scalability by optionally distilling the user-input topology.

A similar service deployment incentive to that espoused by ShadowNet was advocated in [97]. Their service definition is, however, narrower than ShadowNet's scope which also includes network layer services. Amazon's EC2 provides a platform for rapid and flexible edge service deployment with a low cost [1]. This platform only rents computing machines with network access, lacking the ability to control the networking aspects of service testing, or indeed network infrastructure of any kind. PLayer [72] is designed to provide a flexible and composable switching layer in data-center environment. It achieves dynamic topology change with low cost; however, it is not based on commodity hardware.

We heavily rely on hardware-based and software-based virtualization support [30] in the realization of ShadowNet, for example virtual machines [27] and Juniper's logical router [74]. The isolation between the logical functionality and the physical resource can be deployed to achieve advanced techniques, like router migration in VROOM [114] and virtual machine migration [51, 94], which can be used by ShadowNet.

CHAPTER VII

Conclusion

Network management is one of the most important areas of networking research. Successful network management practice not only guarantees satisfiable service delivery but also enables continuous network evolution. In this dissertation, we explored various abstractions and systems to understand the patterns and structures of existing management practice, automate operations that used to be performed manually and requires significant domain expertise, augment these operations to provide network-wide guarantees. We showed that using Petri-Net and database abstractions our proposed systems perform network operations in a both timely and correct fashion, reducing human involvement and at the same time preventing misconfigurations from making into the network. To truly evaluate the usefulness of the proposed network management solutions, and indeed provide a playground for safe network evolution, we proposed a new platform that enables multiple network testings that run in isolation and realistic setups. This platform was used to evaluate both PACMAN and COOLAID and is currently deployed in a large ISP to evaluate novel network service and infrastructure design. In the following sections, we summarize some of insights and limitations of our approach as well as look to future areas of interest suggested by our work.

7.1 Insights

First and foremost, this dissertation provides valuable insight into designing, implementing and evaluating alternative management abstractions. We have shown that abstractions like active documents in PACMAN and database model in COOLAID can significantly reduce human workload without compromising the ability to control the underlying network. Such abstractions allow network-wide reasoning to be performed in a more automated fashion, thus providing the means to seamlessly enforce desired network properties, for example, preventing misconfigurations. On the other hand, comparing to other management systems that require drastic changes to existing infrastructure, PACMAN and COOLAID are also compatible with existing network devices, showing that useful management systems do not necessarily require ground-up changes. This ability also translate to the fact that operators can still review and appreciate the results of our systems, because ultimately our systems interacts with existing management interfaces. This would certainly reduce the possible reluctance of adoption.

Secondly, our work has provided insights into capturing and automating domain knowledge in management practice. PACMAN is a bottom-up approach, using a graph representation to capture the workflow logic of network management operations. COOLAID is a more dramatic shift, using a declarative language to represent domain knowledge in abstract and allowing a reasoning engine to piece together the knowledge and operate on the target network. Comparing to today's common practice of using Method of Procedure (MOP) documents to guide manual operations, our approach is undoubtedly a step forward. Our work calls for a gradual shift from and eventual abandoning of text-based manuals and documentations.

Third, a key insight from our approach is that human operators cannot single-handedly scale with large, complex and dynamic networks. As fully automated network management is not within close reach, human operators are still in the loop of most management operations. Both PACMAN and COOLAID have attempted to give operators the flexibility to decide the operations to perform, but at the same time perform automated reasoning about network-wide implications and inform or even intervene when undesired events are likely to happen. Indeed many of the misconfigurations or network failures stem from the fact that operators cannot fully reason about the negative implications of their actions or natural causes. We have shown that combining such support seamlessly with management tools, rather than building a separate system for manual consultant, allows more effective management operations.

Finally, we have shown that virtualization can help management practice by isolating the impact of network changes. In ShadowNet, different management tool trials and service implementations can be tested in parallel while not impacting each other and the underlying backbone, even with current off-the-shelf network devices. Indeed, a virtualized environment allows innovative network experiments to co-exist with traditional services, utilizing the same infrastructure thus providing real insights on system and service development. We envision network virtualization would have a lasting impact on the evolution of network management practice.

7.2 Limitations

The works presented in this dissertation all base on real management practices and are designed to facilitate daily operations. Our key design philosophy is practicality, such that our tools and systems can be immediately applied to current network infrastructure, which would immediately benefit from our support. Yet, our proposed work still need to overcome the following limitations.

First, we usually assume full exclusive access to all the devices under the same administrative domain. While full access would certainly enable more complete support, such as network-wide reasoning and control, this level of access is unlikely to be granted for newly-proposed management systems. Although extensive tests and trials can be performed in lab environments or realistic testbeds like ShadowNet, applying a new management tool to a real network is likely to be incremental, for example, starting with a single router or a POP of routers. While our work can be tweaked slightly to circumvent this difficulty, for example setting read-only access to config table entries that are related unmanaged routers, we have not fully studied the issue of incremental deployment.

Second, we confine our work mostly in a single-ISP setup. In reality, a lot of operations happen upon inter-ISP communications. For example, operators from different ISPs can negotiate to perform an cross-ISP traffic engineering. These operations are currently not directly supported by our systems. A key issue is that the automation of such activities to a certain extent requires cooperating ISPs to run the same or at least compatible management tools. Nevertheless, PACMAN tries to abstract external synchronization as part of the active document primitives, thus has the potential of realizing such support.

Finally, we do not specifically deal with the compatibility issues of knowledge base. For example, in COOLAID, we envision that the declarative rules can be contributed from device vendors, service providers, and management tool developers. While the benefit of such an approach is unquestionable, it is difficult to achieve a global standard. For example, different router vendors can expose totally different database schemas and declarative rules. This problem is evident based on how differently Cisco and Juniper implement the same NetConf protocol. However, the industry is not totally against open standards. In fact, many device vendors are building devices to support the OpenFlow standard [22].

7.3 Future work

One avenue of work unexplored in this thesis is the evaluation methodologies of network management tools and systems. For most other research avenues, the evaluation metrics are fairly obvious: throughput achieved for a transport layer protocol or a multi-hop wireless routing mechanism, computation time reduction for a datacenter scheduling algorithm, *etc.* For network management, unfortunately, there is no standard evaluation suite, like SPEC [20] or TPC [25], to quantitatively measure the "correctness" of management practices. Indeed network management operations often take compromises among multiple dimensions, for example, trading-off delay with loss-rate in some traffic engineering operations. To complicate matters further, the abstractions exposed by management tools and systems often differ from traditional interfaces, such as CLI. For operators that are already used to think at the lowest level of individual commands, it is hard to force them to switch to new management systems over night. Also, different people clearly have different preference over the interfaces exposed by management tools. Although these questions are hard to answer by nature, solving them, even partially, would certainly steer the evolution of network management.

The focus of this dissertation is mostly on service provider networks, where scale, timeliness and correctness are of the biggest concerns. It remains to be studied if the abstractions proposed are applicable to other networking environments, for example, home and data center networking. Diversity in home networking environments imposes the biggest challenge: an automation system needs to deal with a variety of service providers, Internet access technology, home equipments or even levels of interference from the vicinity. In data centers, on the other hand, network services are closely coupled with the network infrastructure, which must enable and assist sophisticated service management, such as scaling up and down. As a result, an automation system must incorporate service-level awareness when performing network operations.

We have shown with the instance of ShadowNet that virtualization can significantly reduce the negative impact of network changes on existing services and features. Currently, ShadowNet is designed to be a testing facility that is connected to but isolated from a production network. A key question is that whether we can apply the management techniques we developed in ShadowNet controller to control a real production network itself. As such, virtualized network slices can run concurrently with significantly different service features and even support systems. This would certainly unleash the potential of innovative networking research.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Amazon Elastic Compute Cloud. http://aws.amazon.com/ec2/.

[2] AS 7007 Incident. http://en.wikipedia.org/wiki/AS_7007_incident.

[3] Cisco Active Network Abstraction. http://www.cisco.com.

[4] FEDERICA: Federated E-infrastructure Dedicated to European Researchers Innovating in Computing network Architectures. http://www.fp7-federica.eu/.

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

[6] How a System Error in Pakistan Shut YouTube. http://online.wsj.com.

[7] IETF Policy Framework Charter. http://ietf.org.

[8] Internet 2 Network. http://www.abilene.iu.edu/.

[9] Juniper for Network Service Providers. http://puck.nether.net/mailman/listinfo/juniper-nsp.

[10] Juniper Networks, Configuration and Diagnostic Automation Guide. http://www.juniper.net.

[11] Juniper Networks: Troubleshooting Layer 3 VPNs. http://www.juniper.net/.

[12] LINQ. http://msdn.microsoft.com/netframework/future/linq/.

[13] NETDISCO - Network Management Tool. http://www.netdisco.org/.

[14] Netomata Config Generator (NCG). http://www.netomata.com/products/ncg.

[15] Network configuration (netconf). http://www.ietf.org/html.charters/netconf-charter.html.

[16] NetworkAuthority Inventory. http://inventory.alterpoint.com/.

[17] Providing IPv4 VPN Services Across Multiple Autonomous Systems. Juniper BGP and MPLS Configuration Guide.

[18] RANCID - Really Awesome New Cisco confIg Differ. http://www.shrubbery.net/rancid/.

[19] Results of the GEANT OSPF to ISIS Migration. http://www.geant.net/eumedconnect/upload/pdf/GEANT-OSPF-to-ISIS-Migration.pdf.

[20] SPEC: Standard Performance Evaluation Corporation. http://www.spec.org/.

[21] The Internet2. http://www.internet2.edu/network/.

[22] The OpenFlow Switch Consortium. http://www.openflowswitch.org/.

[23] The SQLAlchemy Project. http://www.sqlalchemy.org.

[24] Traffic Control HOWTO. http://linux-ip.net/articles/Traffic-Control-HOWTO/.

[25] Transaction Processing Performance Council. http://www.tpc.org/.

[26] Tutorial: Automating Network Configuration. NANOG'49, San Francisco, CA, June 2010.

[27] VirtualBox. http://www.virtualbox.org.

[28] Rhizoma: a runtime for self-deploying, self-managing overlays. In *Proceedings of ACM Middleware*, 2009.

[29] V. D. Aalst. The application of petri nets to workflow management, 1998.

[30] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, 2006.

[31] R. Alimi, Y. Wang, and Y. R. Yang. Shadow configuration as a network management primitive. In *Proc. ACM SIGCOMM*, 2008.

[32] H. Alzoubi, S. Lee, M. Rabinovich, O. Spatscheck, and J. Van der Merwe. Anycast CDNs Revisited. 17th International World Wide Web Conference, April 2008.

[33] H. Ballani and P. Francis. CONMan: A Step towards Network Manageability. In *Proc. ACM SIGCOMM*, 2007.

[34] H. Ballani and P. Francis. Conman: a step towards network manageability. *SIGCOMM Comput. Commun. Rev.*, 37(4):205–216, 2007.

[35] A. Bavier, N. Feamster, M. Huang, L. Peterson, and J. Rexford. In VINI veritas: realistic and controlled network experimentation. *SIGCOMM Comput. Commun. Rev.*, 36(4):3–14, 2006.

[36] N. Belaramani, J. Zheng, A. Nayte, M. Dahlin, and R. Grimm. PADS: A Policy Architecture for building Distributed Storage systems. In *Proc. of NSDI*, 2009.

[37] M. Bhatia, V. Manral, and Y. Ohara. IS-IS and OSPF Difference Discussions. `http://www.join.uni-muenster.de/Dokumente/drafts/draft-bhatia-manral-diff-isis-ospf-01.txt`.

[38] H. Bohm, A. Feldmann, O. Maennel, C. Reiser, and R. Volk. Network-wide inter-domain routing policies: Design and realization. Presentation at the NANOG34 Meeting.

[39] A. B. Brown and D. A. Patterson. Embracing failure: A case for recovery-oriented computing (roc), 2001.

[40] Z. Cai, F. Dinu, J. Zheng, A. L. Cox, and T. S. E. Ng. The Preliminary Design and Implementation of the Maestro Network Control Platform. Rice University Technical Report TR08-13.

[41] D. Caldwell, A. Gilbert, J. Gottlieb, A. Greenberg, G. Hjalmtysson, and J. Rexford. The cutting EDGE of IP router configuration. In *Proceedings of ACM SIGCOMM HotNets Workshop*, November 2003.

[42] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: taking control of the enterprise. In *Proc. ACM SIGCOMM*, 2007.

[43] J. D. Case, M. Fedor, M. L. Schoffstall, and J. Davin. Simple network management protocol (snmp), 1990.

[44] X. Chen, Y. Mao, Z. M. Mao, and J. Van der Merwe. DECOR: DEClarative network management and OpeRation. In *PRESTO Workshop*, 2009.

[45] X. Chen, Z. M. Mao, and J. Van der Merwe. Towards Automated Network Management: Network Operations using Dynamic Views. In *Proceedings of ACM SIGCOMM Workshop on Internet Network Management (INM)*, 2007.

[46] X. Chen, Z. M. Mao, and J. Van der Merwe. PACMAN: a Platform for Automated and Controlled network operations and configuration MANagement. In *Proc. CoNext*, 2009.

[47] X. Chen, Z. M. Mao, and J. Van der Merwe. ShadowNet: A Platform for Rapid and Safe Network Evolution. In *Proc. USENIX ATC*, 2009.

[48] J. M. Christensen and J. M. Howard. Field Experience in Maintenance. In *In NATO Symposium on Human Detection and Diagnosis of System Failures*, 1981.

[49] D. Chu, L. Popa, A. Tavakoli, J. Hellerstein, P. Levis, S. Shenker, and I. Stoica. The design and implementation of a declarative sensor network system. In *Proc. of SenSys*, Sydney, Australia, November 2007.

[50] Cisco Systems. MPLS VPN Carrier Supporting Carrier. http://www.cisco.com/en/US/docs/ios/12_0st/12_0st14/feature/guide/csc.html.

[51] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, 2005.

[52] E. Eide, L. Stoller, and J. Lepreau. An Experimentation Workbench for Replayable Networking Research. In *NSDI*, 2007.

[53] W. Enck, P. McDaniel, S. Sen, P. Sebos, S. Spoerel, A. Greenberg, S. Rao, and W. Aiello. Configuration management at massive scale: system design and experience. In *Proceedings of the USENIX'07*.

[54] N. Feamster and H. Balakrishnan. Detecting BGP Configuration Faults with Static Analysis. In *NSDI*, May 2005.

[55] N. Feamster and H. Balakrishnan. Detecting BGP Configuration Faults with Static Analysis. In *2nd Symp. on Networked Systems Design and Implementation (NSDI)*, Boston, MA, May 2005.

[56] A. Feldmann. Netdb: IP Network Configuration Debugger/Database. Technical report, AT&T Research, July 1999.

[57] A. Feldmann, A. Greenberg, C. Lund, N. Reingold, and J. Rexford. NetScope: Traffic engineering for IP networks. IEEE Network Magazine, March/April 2000, pp. 11-19.

[58] A. Feldmann and J. Rexford. IP network configuration for intradomain traffic engineering. *IEEE Network Magazine*, pages 46–57, September/October 2001.

[59] P. Ferguson and D. Senie. Network ingress filtering: Defeating denial of service attacks which employ ip source address spoofing (rfc 2827), 2000.

[60] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.

[61] V. Gill and J. Mitchell. OSPF to IS-IS. `http://www.nanog.org/mtg-0310/pdf/gill.pdf`.

[62] H. Gogineni, A. Greenberg, D. A. Maltz, T. S. E. Ng, H. Yan, and H. Zhang. MMS: An Autonomic Network-Layer Foundation for Network Management. Rice University Technical Report TR08-11, December 2008.

[63] J. Gottlieb, A. Greenberg, J. Rexford, and J. Wang. Automated Provisioning of BGP Customers. *IEEE Network*, 17, 2003.

[64] J. Gray. Why do computers stop and what can be done about it?, 1985.

[65] J. Gray and L. Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, 2006.

[66] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

[67] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang. A Clean Slate 4D Approach to Network Control and Management . In *SIGCOMM CCR*, 2005.

[68] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. Nox: towards an operating system for networks. In *CCR*, 2008.

[69] T. L. Hinrichs, N. S. Gude, M. Casado, J. C. Mitchell, and S. Shenker. Practical declarative network management. In *WREN Workshop*, 2009.

[70] L. E. Holloway, B. H. Krogh, and A. Giua. A survey of petri net methods for controlled discrete eventsystems. *Discrete Event Dynamic Systems*, 7(2):151–190, 1997.

[71] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* July 2006.

[72] D. A. Joseph, A. Tavakoli, and I. Stoica. A policy-aware switching layer for data centers. *SIGCOMM Comput. Commun. Rev.*, 38(4), 2008.

[73] Juniper Networks. Configuring Interprovider and Carrier-of-Carriers VPNs. `http://www.juniper.net/`.

[74] Juniper Networks. Juniper Logical Routers. `http://www.juniper.net/techpubs/software/junos/junos85/feature-guide-85/id-11139212.html`.

[75] Juniper Networks. Juniper Networks JCS 1200 Control System Chassis. `http://www.juniper.net/products/tseries/100218.pdf`.

[76] Juniper Networks. Juniper Partner Solution Development Platform. `http://www.juniper.net/partners/osdp.html`.

[77] Juniper Networks. JUNOS 9.2 Policy Framework Configuration Guide. `http://www.juniper.net/techpubs/software/junos/junos92/swconfig-policy/frameset.html`.

[78] B. H. Kantowitz and R. D. Sorkin. Human Factors: Understanding People-System Relationships. In *Wiley*, 1983.

[79] K. M. Kavi, A. Moshtaghi, and D.-J. Chen. Modeling multithreaded applications using petri nets. *Int. J. Parallel Program.*, 30(5):353–371, 2002.

[80] J. Kelly, W. Araujo, and K. Banerjee. Rapid service creation using the junos sdk. *SIGCOMM Comput. Commun. Rev.*, 40(1):56–60, 2010.

[81] Z. Kerravala. Configuration Management Delivers Business Resiliency. The Yankee Group, November 2002.

[82] M. Lad, D. Massey, D. Pei, Y. Wu, B. Zhang, and L. Zhang. Phas: A prefix hijack alert system, 2006.

[83] A. Lakhina, M. Crovella, and C. Diot. Characterization of network-wide anomalies in traffic flows. In *IMC '04: Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 201–206, New York, NY, USA, 2004. ACM.

[84] F. Le, S. Lee, T. Wong, H. S. Kim, and D. Newcomb. Minerals: Using Data Mining to Detect Router. In *ACM Sigcomm Workshop on Mining Network Data (MineNet)*, September 2006.

[85] F. Le, S. Lee, T. Wong, H. S. Kim, and D. Newcomb. Minerals: using data mining to detect router misconfigurations. In *Proceedings of the 2006 SIGCOMM workshop on Mining network data*, 2006.

[86] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative Networking: Language, Execution and Optimization. In *Proc. of SIGMOD*, June 2006.

[87] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing Declarative Overlays. In *Proc. of SOSP*, 2005.

[88] B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative Routing: Extensible Routing with Declarative Queries. In *Proc. of SIGCOMM*, Philadelphia, PA, 2005.

[89] A. A. Mahimkar, Z. Ge, A. Shaikh, J. Wang, J. Yates, Y. Zhang, and Q. Zhao. Towards automated performance diagnosis in a large iptv network. In *SIGCOMM '09: Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, pages 231–242, New York, NY, USA, 2009. ACM.

[90] Y. Mao, B. T. Loo, Z. G. Ives, and J. M. Smith. MOSAIC: Unified Declarative Platform for Dynamic Overlay Composition. In *CoNEXT*, 2008.

[91] Z. M. Mao, R. Govindan, G. Varghese, and R. H. Katz. Route flap damping exacerbates internet routing convergence. *SIGCOMM Comput. Commun. Rev.*, 32(4):221–233, 2002.

[92] A. B. Mnaouer, T. Sekiguchi, Y. Fujii, T. Ito, and H. Tanaka. Colored petri nets based modelling and simulation of the static and dynamic allocation policies of the asynchronous bandwidth in the fieldbus protocol. In *Application of Petri Nets to Communication Networks*, pages 93–130, 1999.

[93] T. Murata. Petri Nets: Properties, Analysis and Applications. Proceedings of the IEEE 77, 4 (1989).

[94] M. Nelson, B.-H. Lim, and G. Hutchins. Fast transparent migration for virtual machines. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 25–25, Berkeley, CA, USA, 2005. USENIX Association.

[95] I. O. Networks, S. il Kim, and S. S. Lumetta. Evaluation of protection reconfiguration for multiple failures. In *In Proc. of Optical Fiber Communication Conference and Exhibit (OFC*, pages 210–211, 2003.

[96] D. Oppenheimer. The Importance of Understanding Distributed System Configuration. In *Proceedings of CHI*, April 2003.

[97] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A Blueprint for Introducing Disruptive Technology Into the Internet. In *Proc. of ACM HotNets*, 2002.

[98] L. Peterson, S. Shenker, and J. Turner. Overcoming the Internet Impasse through Virtualization. Proc. of ACM HotNets, 2004.

[99] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, third edition, 2002.

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

[101] R. Ricci, C. Alfeld, , and J. Lepreau. A Solver for the Network Testbed Mapping Problem. In *CCR*, 2003.

[102] R. Ricci, J. Duerig, P. Sanaga, D. Gebhardt, M. Hibler, K. Atkinson, J. Zhang, S. Kasera, and J. Lepreau. The Flexlab Approach to Realistic Evaluation of Networked Systems. In *NSDI*, 2007.

[103] T. Roscoe, R. Mortier, P. Jardetzky, and S. Hand. InfoSpect: Using a Logic Language for System Health Monitoring in Distributed Systems. In *Proceedings of the SIGOPS European Workshop*, 2002.

[104] F. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4), 1990.

[105] A. Singh, T. Das, P. Maniatis, P. Druschel, and T. Roscoe. BFT Protocols Under Fire. In *Proc. of NSDI*, Apr 2008.

[106] N. Spring, R. Mahajan, D. Wetherall, and T. Anderson. Measuring ISP topologies with rocketfuel. *IEEE/ACM Trans. Netw.*, 12(1):2–16, 2004.

[107] J. C. Strassner, N. Agoulmine, and E. Lehtihet. FOCALE A Novel Autonomic Networking Architecture. Latin American Autonomic Computing Symposium (LAACS), 2006.

[108] Y.-W. E. Sung, C. Lund, M. Lyn, S. G. Rao, and S. Sen. Modeling and understanding end-to-end class of service policies in operational networks. In *SIGCOMM*, 2009.

[109] J. Turner and N. McKeown. Can overlay hosting services make ip ossification irrelevant? PRESTO: Workshop on Programmable Routers for the Extensible Services of TOmorrow, May 2007.

[110] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker. Scalability and accuracy in a large-scale network emulator. In *OSDI '02: Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, 2002.

[111] W. van der Aalst. The application of petri nets to workflow management, 1998.

[112] L. Vanbever, G. Pardoen, and O. Bonaventure. Towards Validated Network Configurations with NCGuard. In *INM Workshop*, 2008.

[113] R. Vasudevan and Z. M. Mao. Reval: A tool for real-time evaluation of ddos mitigation strategies. In *In Proceedings of USENIX Annual Technical Conference*, 2006.

[114] Y. Wang, E. Keller, B. Biskeborn, J. Van der Merwe, and J. Rexford. Virtual Routers on the Move: Live Router Migration as a Network-Management Primitive. In *Proc. ACM SIGCOMM*, 2008.

[115] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. Mahlke. Gadara: Dynamic Deadlock Avoidance for Multithreaded Programs. In *Proceedings of OSDI*, 2008.

[116] M. Wawrzoniak, L. Peterson, and T. Roscoe. Sophia: an Information Plane for networked systems. In *Proceedings of SIGCOMM CCR*, 2004.

[117] G. R. Wheeler. The modelling and analysis of ieee 802.6's configuration control protocol with coloured petri nets. In *Application of Petri Nets to Communication Networks*, pages 69–92, 1999.

[118] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, 2002.

[119] G. Xie, X. Jibin, Z. David, A. Maltz, H. Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford. On static reachability analysis of IP networks. In *Proc. IEEE INFOCOM*, 2005.

[120] H. Yan, D. A. Maltz, T. S. E. Ng, H. Gogineni, H. Zhang, and Z. Cai. Tesseract: A 4d network control plane. In *in Proc. Networked Systems Design and Implementation*, 2007.