Declarative Configuration Management for Complex and Dynamic Networks

Xu Chen[†§] Yun Mao[†] Z. Morley Mao[§] Jacobus Van der Merwe[†]
[§] *University of Michigan - Ann Arbor* [†] *AT&T Labs - Research*

Abstract— Network management and operations are complicated, tedious, and error-prone, requiring signif cant human involvement and domain knowledge. As the complexity involved inevitably grows due to larger scale networks and more complex protocol features, human operators are increasingly short-handed, despite the best effort from existing support systems to make it otherwise. This paper presents COOLAID, a system under which the domain knowledge of device vendors and service providers is formally captured by a declarative language. Through eff cient and powerful rule-based reasoning on top of a database-like abstraction over a network of devices, COOLAID enables new management primitives to perform network-wide reasoning, prevent miscon-f guration, and automate network conf guration, while requiring minimum operator effort. We describe the design and prototype implementation of COOLAID, and demonstrate its effectiveness and scalability through various realistic network management tasks.

1. INTRODUCTION

Network management and operation arguably remains a domain that continues to thwart modernization attempts by the networking community. There are a number of reasons for this state of affairs. First, network management is inherently diff cult because of the scale, the distributed nature and the increasing complexity of modern communication networks. Second, network management tools and practices fail to keep up with the ever–evolving and complex nature of the networks being managed. Third, and perhaps most importantly, current network management approaches fail to capture and utilize, in a systematic fashion, the signif cant domain expertise (from vendors, service providers and protocol designers), which in essence *is* the foundational pillar enabling the continued operation of the network.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for prof t or commercial advantage and that copies bear this notice and the full citation on the f rst page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specif c permission and/or a fee.
ACM CoNEXT 2010, November 30 – December 3 2010, Philadelphia, USA.
Copyright 2010 ACM 1-4503-0448-1/10/11 ...$10.00.

In a typical large Internet service provider setting, hundreds or thousands of network devices are distributed across vast geographic distances, and their conf gurations collectively determine the functionality provided by the network. The protocols and mechanisms that realize such network functionality often have complex dependencies that have to be satisf ed for correct operations. Such dependencies are often not precisely def ned, or at least not expressed in a systematic manner. When they are violated through miscon-f gurations, software bugs, or equipment failures, network troubleshooting becomes an extremely diff cult task.

Despite these complexities, network management operations still largely rely on fairly rudimentary technologies. With only a few notable exceptions for specialized tasks, network conf guration management is still performed via archaic, low-level command line interfaces (CLIs). Vendors describe protocol dependencies and network-wide capabilities in device manuals or other technical documents. Network engineers manually interpret these vendor documents and in turn produce service provider documentation, which describes in prose with conf guration excerpts, on how network services might be realized. Similarly, disruptive activities like planned maintenance rely on the experience of human operators and their ability to interpret and follow procedures documented by domain experts to prevent undesired side effects. In short, current network management practices depend on the knowledge base of domain experts captured in documents meant for human consumption and further attempts to derive, from this captured knowledge, systems and procedures to ensure that the correct documents are consulted and followed to perform network operations.

In cases where network operations have progressed beyond the capacity of human interpretation and manual execution of procedures, tools attempt to automate the procedures that a human operator would have performed and/or reverse engineer the protocol and network dependencies that prevail in an existing network. For instance, sophisticated network conf guration management tools [12, 9] attempt to capture the actions of human experts for subsequent automation. Existing fault and performance management practices involve, in part, reverse engineering protocol actions and dependencies [20]. Unfortunately, all such tools are highly

specialized, each focusing on a specif c management aspect.

In this paper, we present COOLAID (COnf guring cOmpLex and dynamic networks AutomatIcally and Declaratively), a network management framework that automates a variety of dominant network operations relying on conf gurations, while minimizing human involvement. The key idea is to formally capture the domain knowledge using a declarative logic-based language, then apply the knowledge on top of a database-like abstract data model that represents network-wide information. As such, COOLAID can derive high-level views for network-wide reasoning, automate network conf guration, and prevent misconf guration, allowing operators to better manage their networks without being exposed to the overwhelming details.

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. We expect that COOLAID enables 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.

This paper makes the following contributions:

- Demonstrates with real-world examples of how domain knowledge from both device vendors and service providers can be concisely captured using a declarative language;
- Builds a unif ed data model abstracting network-wide information to facilitate the automation of rule-based domain knowledge;
- Exemplif es distributed recursive query, updatable view, and distributed transaction management as useful enabling techniques for new and enhanced network management primitives;
- Implements and evaluates a prototype of the COOLAID system to automate a variety of useful network operations requiring minimal human involvement.

2. MOTIVATION

Modern networks are complexity-plagued, large-scale, and highly dynamic. This is particularly true for large ISP networks, which typically house thousands of devices interconnected across dispersed geographic regions and support diverse network services and features. Next we explain three key challenges in managing such networks.

Network functionalities are complex: Large ISPs support a variety of revenue-generating network services in addition to traditional IP transit. For example, Virtual Private Network (VPN) service, which allows multiple sites of a customer organization to be seamlessly connected together, and Voice-over-IP (VoIP) service are commonly offered. Supporting these services is very challenging, even purely from a conf guration's point of view. Setting up these services requires operators to understand voluminous conf guration manuals and apply the knowledge to specif c network setups. A particular complication stems from the fact that network services or features are commonly dependent on each other [1]. These dependencies are usually verbally described in documentations and impose a steep learning curve.

Network-wide reasoning is hard: Understanding the network functionalities given the conf gurations on the physical devices is crucial for network management. Operators commonly ask (mostly themselves) questions like, “Is that MPLS/VPN instance conf gured correctly for the customer?” and “What services might be impacted if I shut down this loopback interface?” Mistakes in answering these questions might result in network-wide outages. Yet answering these questions is diff cult for two reasons: (i) Reasoning about each network feature requires understanding a chain of dependent features; (ii) More importantly, understanding each feature requires not only inspecting individual device conf gurations but also reasoning about the distributed protocol execution logic, *e.g.,* how the routes propagate through OSPF within a network. The complexity quickly increases with the number of network elements and network features, and therefore human errors in reasoning about complicated protocols on thousands of routers become unavoidable, especially when operators are constrained with short maintenance windows to rush management tasks. Even worse, when multiple operators are working on the same network, their concurrent actions, each might be f ne to perform individually, can cause signif cant network disruptions if not properly synchronized. This calls for an automation support to accurately reason about network services and protocols at scale. To the best of our knowledge, no existing tool is capable of providing such generic support.

Networks are dynamic: Last but not the least, modern networks are quickly evolving. Besides providing new features, much of the network evolution is purely driven by growth demand, *e.g.,* adding core or edge routers to handle more customers with higher throughput. These device introduction, upgrade and re-purpose activities are commonly performed in today's large networks, and they need to be handled correctly and eff ciently to ensure the continuous service delivery. In general, removing devices must not interfere with existing network functionalities, while new devices must be conf gured with proper functionalities to become an integral part of the network, *e.g.,* to participate in OSPF routing or to establish iBGP sessions to other routers. Figuring out what to conf gure on the new devices is non-trivial and usually requires reverse-engineering the existing network setups. Re-

[1]For example, as we later show in Figure 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 conf gured IGP.

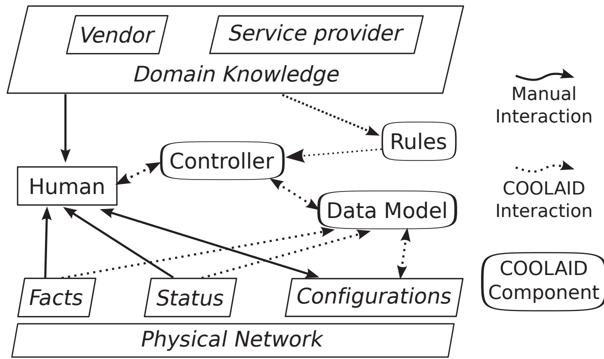

Figure 1: COOLAID vs. manual management

dundant or missing conf gurations result in redundant features or, more problematically, non-functional networks. It quickly becomes cumbersome for operators when network changes are frequent and network features are diverse. The emerging trend of network virtualization further raises the bar [27], as the network inventory can be dynamically allocated and the topology can be easily modif ed. An ideal support to address this challenge is to allow the specif cation of network-wide properties, which are enforced no matter how the network changes. Unfortunately, there is no systematic support to address this need.

3. 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 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 conf gured for a customer) or realize additional features by changing conf gurations, 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 satisf es these requirements. We f rst 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, and our system implementation in §5.

3.1 COOLAID building blocks

Conceptually, COOLAID models a network of interconnected devices as a distributed but centrally managed database, exposing an intuitive database-style interface for operators to manage the entire network. Figure 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 conf gurations, checking network status and provisioning data. The abstraction is designed to work with commodity devices, and interoperable with existing management tools. We def ne three types of base tables[2]: (i) *Regular tables* store fact-related data that naturally f t into a conventional database (*e.g.,* MySQL). (ii) *Config tables* store the *live* network conf gurations, in particular, router states that are persistent across reboots, *e.g.,* IP addresses and protocol-specif c 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.

The key benef t of having these tables is that COOLAID abstracts away the details of individual management interfaces and instead works on a unif ed abstraction, which fundamentally enables the systematic expression and integration of domain knowledge from different role players as we describe next. We want to emphasize that the traditional usage of databases in network management is predominantly for archiving snapshots of network-related information to facilitate subsequent analysis. In contrast, COOLAID uses the database notion as an *abstraction* layer that sits on top of the actual network for enabling new management primitives. To enable a new distributed transaction support, COOLAID chooses not to store data from conf g tables or status tables in a conventional SQL database but rather accesses them through software artifacts exposing database interfaces. We describe how to enable this abstraction on commodity network devices in §5.2.

Regular tables are only modif ed when necessary to ref ect network changes, *e.g.,* new devices or new customer agreements. Conf g tables are always in-sync with the network devices, and modifying these tables causes actual conf guration changes. Status tables are read-only, and table entries are generated on-demand from traditional management interface, such as CLI and SNMP[3].

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

[3]Status tables contain important information for various management operations (*e.g.,* fault diagnosis). However, because this pa-

Rules: COOLAID represents network management domain knowledge, in particular protocol dependencies and network-wide requirements, as rules in 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 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 (§3.2, §3.4) and (ii) service providers define rules on how these vendor capabilities should be utilized to reliably deliver customer services (§3.3, §3.5), 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. By applying the rule-based domain knowledge onto the network states stored in the data model, the controller significantly relieves the burden on operators. 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 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);
```

3.2 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 presented in §3.1. We demonstrate how the knowledge regard-

per primarily focuses on the configuration management, we leave exploiting status tables as future work.

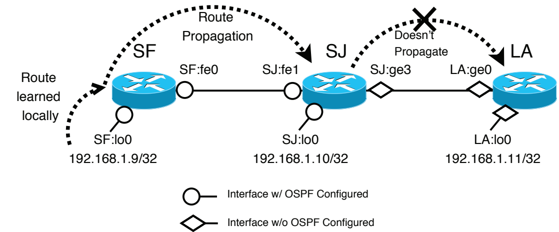

Figure 2: Example network with OSPF configuration

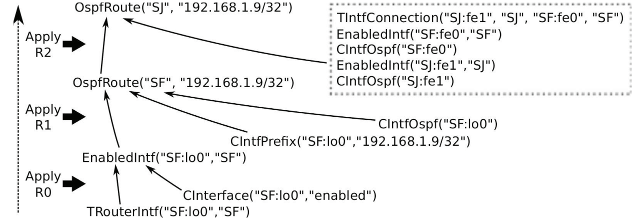

Figure 3: Bottom-up view evaluation

ing OSPF route learning can be written as three rules in Listing 1. The rules are written in a declarative language based on Datalog [29][4], 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 query touches multiple devices and the rule head is part of the rule body.

Figure 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 3 illustrates how the entries in the view tables are gener-

[4]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.

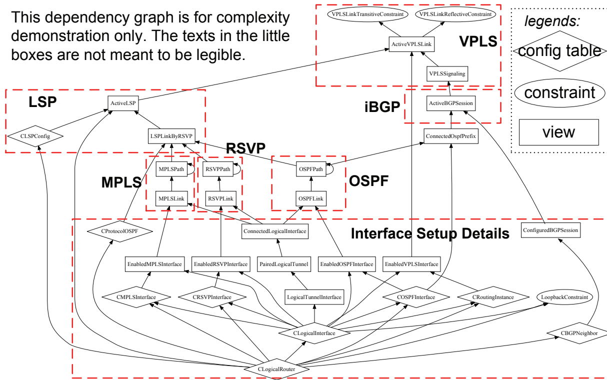

Figure 4: VPLS related view dependency graph

ated in a bottom-up fashion based on `R0-R2`, and eventually the entry `OspfRoute("SJ","192.168.1.9/32")` shows that "pref x `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 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*.

3.3 Misconfiguration prevention

COOLAID uses *constraints* to detect and prevent misconf guration. 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 who underestimate 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 traff c 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 reasoning about multiple devices and different network service layers.

Specif cally, in COOLAID, a constraint is def ned the same way as views by a set of rules. A constraint is satisf ed 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 misconf gurations when combined with our new transaction primitive (described in §3.6.) 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.

3.4 Configuration automation

COOLAID supports a new primitive of automating network conf guration by allowing *writes* to view tables. Specif cally, COOLAID allows the operators to specify intended network changes as insert/delete/update to view tables, then automatically identif es a set of low-level changes to conf g 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 conf guration management is that operators (i) change the conf gurations 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 fulf ll 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 f ddling with all the dependent network functionalities.

3.5 Network property enforcement

COOLAID allows the operators to specify certain properties to enforce on the network. For example, a network may be required to conf gure 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 conf guration tables, say modifying `CIntfPrefix` table to conf gure 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.

3.6 Atomic network operations

Device failures during network operations are not uncommon, especially in large-scale networks. If not handled

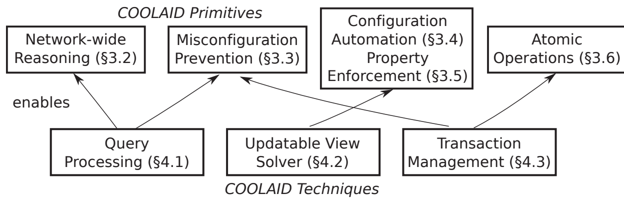

Figure 5: COOLAID primitives and techniques

properly, they often put the network in inconsistent states. A network operation involving conf guring several routers might be abandoned half way because of unforeseen circumstances, such as an unexpected transient network failure, or overloaded routers. Current operational procedures would require a manual rollback, which may be incomplete, leaving "orphaned" conf guration excerpts and leading 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), 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 conf guration 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.

3.7 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, conf g, 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, misconf guration prevention, conf guration automation, network property enforcement, and atomic network operations, all in the same cohesive framework.

4. TECHNIQUES

In this section, we explain key techniques that COOLAID utilizes to enable the network management primitives described in §3. Figure 5 shows their relationships.

4.1 Query processing

Query processing is essential for network-wide reasoning (§3.2) and misconf guration prevention (§3.3). We highlight a few design choices in building the query processor eff ciently, 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 [29]. Recent work has also examined recursive queries in a distributed environment with a relaxed eventual consistency model [23].

Second, COOLAID manages a much smaller amount but distributed data. The largest portion of the data comes from conf gurations. If we assume that a conf guration f le 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 conf guration exists on different routers might require hundreds of milliseconds of round-trip time to access, for a typical ISP with national footprints. Therefore, we always f rst aggregate all data to the main memory of a centralized master node (§5.1) before query evaluation. Centralized processing is also preferred in order to enforce a strong consistency model as opposed to the eventual consistency model [23]. Once all data are available, we apply the semi-naïve evaluation algorithm [29], which is both eff cient 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 time. 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.

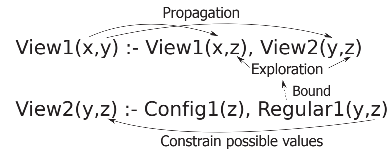

Figure 6: Solving updatable view operations

4.2 Updatable view solver

Updatable view operations, like view insertions or deletions, enable conf guration automation (§3.4) and network property enforcement (§3.5). Underneath the simple APIs called by operators, COOLAID controller f nds the conf g 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 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 f elds in views. For example, `View2` is def ned by joining a conf g 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 benef t 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 conf gure 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 benef t 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 conf gure OSPF on all non-customer interfaces in the core network, comparing to the generic solver that may give a partial conf guration. The f ip 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 unif ed database abstraction. Also, this requires one-time effort by vendors or network experts, while the operators can stay unaware of such details.

4.3 Transaction management

Misconf guration prevention (§3.3) and atomic network operations (§3.6) 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 f nd that ACID properties naturally f t conf g tables as follows:

Atomicity: The conf guration 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 atomicity also applies in a distributed transaction where conf g changes involve multiple devices. The atomic feature greatly simplif es 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 def nition in COOLAID is that all constraints must be satisf ed. 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 conf guration bugs. COOLAID provides transaction isolation guarantees to prevent such issues.

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

To implement the ACID transactional semantics in COOLAID, we use the Two-Phase Commit protocol for atomicity due to its simplicity and eff ciency; we use Write-Ahead-Logs for crash recovery; and we use Strict Two-Phase Locking for concurrency control [28]. 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 conf icting conf guration changes, it is very expensive to roll back and retry one of them. We choose to prevent conf icts from happening, even at the cost of limiting parallelism. We discuss the detailed implementations of transaction management in §5.1.

5. IMPLEMENTATION

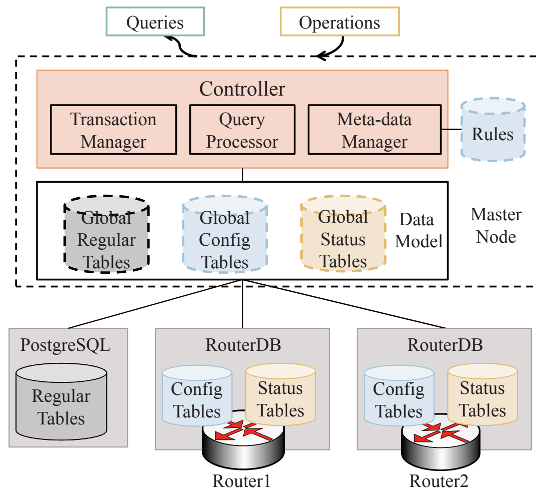

Figure 7: COOLAID **system architecture**

The overall system architecture of COOLAID is depicted in Figure 7. We have implemented a prototype system in roughly 13k lines of Python code with two major software pieces described next.

5.1 Master node

The master node unif es 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 (§5.2) instance, which exports the corresponding conf g tables and status tables. The conf g tables on RouterDBs are aggressively combined and cached on the master node for performance improvement. When an entry in a conf g table is modif ed, the appropriate RouterDB instance will be identif ed and notif ed (known as horizontal partitioning in data management) based on the primary key of the entry, which has the physical router ID encoded.

The controller on the master node has three components:

Query processor: The query processor f rst 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 [2], to express and evaluate those relational expressions. The library is capable of integrating queries from Python objects, tables in PostgreSQL, and XML data. We implemented the algorithm described in §4.1 for query evaluation and view maintenance and an updatable view solver described in §4.2.

Meta-data manager: Meta-data, such as the def nitions 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 (§4.1) for caching and incremental updates, and updatable view operations (§4.2).

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) [28] 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 f ushes 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 satisf ed (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 f ner-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 §7.

5.2 RouterDB

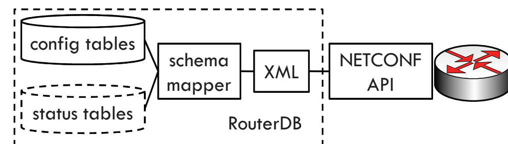

Figure 8: RouterDB implementation

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 Conf guration Protocol (NETCONF) [3] to install, manipulate, and delete the conf guration 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 conf guration in XML format. Then a schema mapper is used to convert conf gurations from the tree-structured XML format into relational conf g 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 operations, such as `insert`, `update`, `delete` to the config tables, must use the ID to indicate their transaction affiliations. 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 candidate target, which is separate from 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: We chose to host a RouterDB close to the corresponding router, *e.g.,* on the same LAN, reliably connecting to the dedicated management interface. 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 [21]. 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 [28], 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 recommits 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.

6. EVALUATION

We evaluated several key aspects of COOLAID to show that it effectively reduces human workload and prevents misconfigurations in realistic management tasks, at the same time scales to large networks. 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.

6.1 Automating configuration

We created the network topology of Internet-2 core networkwith 10 routers and 13 links on top of the ShadowNet platform [10] 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 allocating interface IPs, configuring OSPF or IS-IS routing, iBGP sessions, building a MPLS network with RSVP signaling, establishing LSPs and finally the 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.

6.2 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 spec-

ified 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.

6.3 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 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 [33]. 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 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 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 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.

6.4 Transaction overhead

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

Table 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 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.

7. DISCUSSION

Feasibility: Using the database abstraction and the declarative rules represents a drastic but reasonable shift. First, network databases are commonly practiced in modern ISPs [7]. The emerging trend of XML-based configuration files further reduces the effort, since XML files can be directly queried. Second, according to our experience, the time-consuming part of writing the rules is to derive the correct dependency by reading documentations and performing field tests. In reality, we found the amount of work manageable for a single graduate student to decipher VPLS, despite the

complex dependency involved. Furthermore, as we have suggested, we envision an environment where, in addition to providing the text documents, vendors can also provide libraries of rules. Such an approach greatly simplifies the service creation by service providers.

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 [32] 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 [16], 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: (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 reacquired in the case of rollback. The problem could be addressed through a locking mechanism to hold the resources until the transaction finishes. (iii) We assume the set of constraints are complete to prevent inconsistent states; however, this is difficult because new constraints can be introduced and discovered over time. One potential solution is to rollback previous operations to a point where no constraints, including the new ones, are violated, and then replay the transactions, such as updateable view operations. (iv) COOLAID currently does not address the issues of protocol optimization, *e.g.,* tweaking the OSPF link weights for traffic engineering [14]; however, existing techniques can be invoked in the customized view solvers to integrate their results with our data model.

8. RELATED WORK

Network management: A clean-slate approach to configuration management is advocated in CONMan [4], in which protocols are abstracted as modules and network configuration is done through piping the modules. Template-driven approaches [15, 12] are commonly used in production environments. A template program extracts parameters from provisioning databases and generates configuration snippets, optionally with some validation [35]. Unfortunately, the dependencies among templates and between the generated snippets and the existing configurations, still need to be resolved manually. Sung *et al*. built a query engine for evaluating Class of Service (CoS) configuration [34]. In contrast, COOLAID advocates using declarative rules as a concise 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. Relating to the 4D project [17], COOLAID fulfills the functionalities of the decision and dissemination planes. KarDo [22] automates generic operations on PCs, and the enabled automation does not apply to complex network management tasks.

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 [31], Sophia [36] and Rhizoma [37] 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. PoDIM [11] is a languange designed to express cross-machine constraints in an enterprise environment. COOLAID captures more general and complex dependencies and constraints in wide-area networks.

In the enterprise network management space, Ethane [8] and NOX [18] focus on network flow access control management. Along the same line, Flow-based Management Language [19] is based on the Datalog syntax to express policies of flow control. These resemble most of the policy-based network management work [1]. In contrast, the language proposed in COOLAID effectively captures domain knowledge in protocol behaviors and dependencies.

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 [25, 30]. The declarative approach has been explored by numerous projects, *e.g.,* to implement overlays [24], data and control plane composition [26], and specify distributed storage policies [5]. 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.

Databases: Database technologies are routinely utilized as part of network management and operations. One class of existing work, represented by NetDB [7], uses a relational database to store router configuration snapshots, where one can write queries to audit and perform static analysis of existing configurations in an offline fashion, *e.g.,* for BGP [13]. From a network operator's perspective, the database is read-only and is not necessarily consistent with live configurations. In contrast, COOLAID provides a unifying database abstraction that integrates router configurations, live network status and provisioning data, provides transactional write operations to change network configurations, and enforces constraints to detect and prevent policy violations during operation, as opposed to a postmortem support tool.

To realize the database abstraction of COOLAID, we take advantage of many existing techniques and concepts in the database literature, including recursive query optimization [29], distributed transaction processing [28], updatable materialized views [6], *etc*. However, we note that while some of these features are becoming available in commercial database products, no existing database systems support all of these features, or work with commodity routers as back-end storage. To our knowledge, COOLAID is the first system that integrates these features with unique optimizations customized for network management and operations.

9. CONCLUSION

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.

10. REFERENCES

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

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

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

[4] H. Ballani and P. Francis. CONMan: A Step Towards Network Manageability. In *Proceedings of SIGCOMM*, 2007.

[5] 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.

[6] A. Bohannon, J. A. Vaughan, and B. C. Pierce. Relational Lenses: A Language for Updateable Views. In *Proceedings of of PODS*, 2006.

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

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

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

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

[11] T. Delaet and W. Joosen. PoDIM: A language for high-level configuration management. In *Proceedings of LISA*, 2007.

[12] 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 USENIX ATC*, 2007.

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

[14] 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.

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

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

[17] 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 *Proceedings of SIGCOMM CCR*, 2005.

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

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

[20] C. R. Kalmanek, et al. Darkstar: Using Exploratory Data Mining to Raise the Bar on Network Reliability and Performance. In *Proceedings of DRCN*, 2009.

[21] J. Kelly, W. Araujo, and K. Banerjee. Rapid service creation using the junos sdk. In *Proceedings of SIGCOMM CCR*, 2010.

[22] N. Kushman and D. Katabi. Enabling Configuration-Independent Automation by Non-Expert Users. In *Proceedings of OSDI*, 2010.

[23] 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 *Proceedings of SIGMOD*, 2006.

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

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

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

[27] L. Peterson, S. Shenker, and J. Turner. Overcoming the Internet Impasse Through Virtualization. In *Proceedings of HotNets Workshop*, 2004.

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

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

[30] T. Roscoe, S. Hand, R. Isaacs, R. Mortier, and P. Jardetzky. Predicate routing: enabling controlled networking. In *Proceedings of SIGCOMM CCR*, 2003.

[31] 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.

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

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

[34] 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 *Proceedings of SIGCOMM*, 2009.

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

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

[37] Q. Yin, A. Schuepbach, J. Cappos, A. Baumann, and T. Roscoe. Rhizoma: a runtime for self-deploying, self-managing overlays. In *Proceedings of Middleware*, 2009.