Recent Advances in Declarative Networking

Boon Thau Loo[1] Harjot Gill[1] Changbin Liu[1] Yun Mao[2]
William R. Marczak[3] Micah Sherr[4] Anduo Wang[1] Wenchao Zhou[1]

[1] University of Pennsylvania
[2] AT&T Labs Research
[3] University of California Berkeley
[4] Georgetown University

{boonloo,gillh,cliu,anduo,wenchaoz}@cis.upenn.edu maoy@research.att.com
wrm@berkeley.edu msherr@cs.georgetown.edu

Abstract. Declarative networking is a programming methodology that enables developers to concisely specify network protocols and services, and directly compile these specifications into a dataflow framework for execution. This paper describes recent advances in declarative networking, tracing its evolution from a rapid prototyping framework towards a platform that serves as an important bridge connecting formal theories for reasoning about protocol correctness and actual implementations. In particular, the paper focuses on the use of declarative networking for addressing four main challenges in the distributed systems development cycle: the generation of safe routing implementations, debugging, security and privacy, and optimizing distributed systems.

1 Introduction

Declarative networking [27–29, 31] is a programming methodology that enables developers to concisely specify network protocols and services using a distributed recursive query language, and directly compile these specifications into a dataflow framework for execution. This approach provides ease and compactness of specification, and offers additional benefits such as optimizability and the potential for safety checks. The development of declarative networking began in 2004 with an initial goal of enabling safe and extensible routers [30].

As evidence of its widespread applicability, declarative techniques have been used in several domains including fault tolerance protocols [52], cloud computing [4], sensor networks [11], overlay network compositions [33], anonymity systems [51], mobile ad-hoc networks [37, 24], wireless channel selection [23], network configuration management [10], and as a basis for course projects in a distributed systems class [14] at the University of Pennsylvania. An open-source declarative networking system called *RapidNet* [3] has also been integrated with the emerging ns-3 [40] simulator, demonstrated at SIGCOMM'09 [38], and successfully deployed on testbeds such as PlanetLab [44] and ORBIT [42].

This paper will first present a background introduction to declarative networking (Section 2). We trace its evolution from a rapid prototyping framework to a platform that serves as an important bridge connecting formal theories for reasoning about protocol correctness and actual implementations. The ability to bridge this gap is a major step forward compared to traditional approaches in which formal specifications, proof of protocol correctness and implementations are decoupled from one another; this decoupling leads to increased development time, error prone implementations, and tedious debugging.

Specifically, this paper describes recent work carried out within the NetDB@Penn [39] research group to address four significant challenges in distributed systems: generating safe routing implementations (Section 3), securing distributed systems (Section 4), debugging distributed systems (Section 5), and optimizing distributed systems (Section 6).

2 Background

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

NDlog enables a variety of routing protocols and overlay networks to be specified in a natural and concise manner. For example, traditional routing protocols such as the path vector and distance-vector protocols can be expressed in a few lines of code [31], and the Chord distributed hash table in 47 lines of code [29]. When compiled and executed, these declarative protocols perform efficiently relative to imperative implementations.

In addition to ease of implementation, another advantage of the declarative networking approach is its amenability to formal and structured forms of correctness checks. These include the use of theorem proving [53], algebraic techniques for constructing safe routing protocols [54], and runtime verification [61]. These formal analysis techniques are strengthened by recent work on formally proving correct operational semantics of *NDlog* [41]. Finally, the dataflow framework used in declarative networking naturally captures information flow as distributed queries, hence providing a natural way to use the concept of *network provenance* [60] to analyze and explain the existence of any network state.

NDlog is based on Datalog [46]: a Datalog program consists of a set of declarative *rules*. Each rule has the form `p :- q1, q2, ..., qn.`, which can be read informally as "`q1` and `q2` and ... and `qn` implies `p`". Here, `p` is the *head* of the rule, and `q1`, `q2`,...,`qn` is a list of *literals* that constitutes the *body* of the rule. Literals are either *predicates* with *attributes* (which are bound to variables or constants by the query), or boolean expressions that involve function symbols (including arithmetic) applied to attributes.

Datalog rules can refer to one another in a mutually recursive fashion. The order in which the rules are presented in a program is semantically immaterial; likewise, the order predicates appear in a rule is not semantically meaningful. Commas are interpreted as logical conjunctions (*AND*). Conventionally, the names of predicates, function symbols, and constants begin with a lowercase letter, while variable names begin with an uppercase letter. Function calls are additionally prepended by `f_`. Aggregate constructs are represented as functions with attribute variables within angle brackets (`<>`). We illustrate *NDlog* using a simple two rule program that computes all pairs of reachable nodes in a network:

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

Rules r1 and r2 specify a distributed transitive closure computation, where rule r1 computes all pairs of nodes reachable within a single hop from all input links (denoted by the link predicate), and rule r2 expresses that "if there is a link from S to N, and N can reach D, then S can reach D." The output of interest is the set of all reachable(@S,D) tuples, representing reachable pairs of nodes from S to D. By modifying this simple example, we can construct more complex routing protocols, such as the distance vector and path vector routing protocols.

NDlog supports a *location specifier* in each predicate, expressed with the @ symbol followed by an attribute. This attribute is used to denote the source location of each corresponding tuple. For example, all reachable and link tuples are stored based on the @S address field.

2.1 Query Evaluation

In declarative networking, each node runs its own set of *NDlog* rules. Typically, these rules are common across all nodes (that is, all nodes run the same protocol), but may further include per-node policy customizations. *NDlog* rules are compiled and executed as *distributed dataflows* by the query processor to implement various network protocols. These dataflows share a similar execution model with the Click modular router [21], which consists of elements that are connected together to implement a variety of network and flow control components. In addition, elements include database operators (such as joins, aggregation, and selections) that are directly generated from the *NDlog* rules. Messages flow among dataflows executed at different nodes, resulting in updates to local tables, or query results that are returned to the hosts that issued the queries. The local tables store the network state of various network protocols.

To execute *NDlog* programs, we use the *pipelined semi-naïve* (PSN) model [27]. PSN extends the traditional *semi-naïve* Datalog evaluation strategy [9] to work in an asynchronous distributed setting. PSN relaxes semi-naïve evaluation to the extreme of processing each tuple as it is received. This provides opportunities for additional optimizations on a per-tuple basis. New tuples that are generated from the semi-naïve rules, as well as tuples received from other nodes, are used immediately to compute new tuples without waiting for the current (local) iteration to complete.

In practice, most network protocols execute over a long period of time and incrementally update and repair routing tables as the underlying network changes (for example, due to link failures, and node departures). To better map into practical networking scenarios, one key distinction that differentiates the execution of *NDlog* from earlier work in Datalog is our support for continuous rule execution and result materialization, where all tuples derived from *NDlog* rules are materialized and incrementally updated as the underlying network changes. As in network protocols, such incremental maintenance is required both for timely updates and for avoiding the overhead of recomputing all routing tables "from scratch" whenever there are changes to the underlying network. In the presence of insertions and deletions to base tuples, our original incremental view maintenance implementation utilizes the count algorithm [17] that ensures only tuples that are no longer derivable are deleted. This has subsequently been improved [36] via the use of a compact form of data provenance encoded using binary decision diagrams shipped with each derived tuple.

2.2 Language Extensions

In our original work [29], predicates are allowed to be declared as *soft-state* with lifetimes. In the extreme case, *event predicates* form transient tables which are used as input to rules but are not stored. To support wireless broadcast [24, 37], we have introduced a *broadcast location specifier* denoted by `@*` which causes a tuple to be broadcast to all nodes within wireless range of the node on which the rule is executed. In order to support network functionality composition and code reuse, we introduced *Composable Virtual Views* [33], which define rule groups that perform a specific functionality when executed together. These extensions offer different levels of declarativity [32] to meet various application demands.

The meaning of a *NDlog* program is defined to be the behavior and output obtained by running the program through PSN evaluation [27, 41]. The *Dedalus* [19, 5] language is similar to *NDlog*, except its behavior and output is defined in terms of a model-theoretic semantics. Dedalus also allows users to write rules that mutate state.

Dedalus takes base Datalog, and adds an integer *timestamp* field to every tuple. State update is expressed as locally-stratified recursion through negation. Message delay and re-ordering is captured by requiring all rules to derive non-local tuples at some non-deterministic future timestamp. Dedalus uses Saccà and Zaniolo's `choice` construct [49] to model this non-determinism, which manifests itself in multiple *stable models* [13] – one model for each possible choice of timestamp.

An interesting question is to what extent the behavior and output of the program is "well-behaved." The *CALM Conjecture*, posed by Hellerstein [19] states that monotonic *coordination-free* Dedalus programs are *eventually consistent*, and non-monotonic programs are eventually consistent when instrumented with appropriate coordination. Recently, Ameloot et al. explored Hellerstein's CALM conjecture using relational transducers [6]. They proved that monotonic first order queries are exactly the set of queries that can be computed in a coordination-free fashion in their transducer formalism. Their work uses some different assumptions than traditional declarative networking—for example, they assume that all messages sent by a node are multicast to a fixed set of neighbors, whereas *NDlog* permits arbitrary unicast.

3 Generating Safe Routing Implementations

Our *Formally Verifiable Routing (FVR)* project addresses a long-standing challenge in networking research: bridging the gap between formal routing theories and actual implementations. The application of declarative networking is especially useful here, serving as an intermediary layer between high-level formal specifications of the network design and low-level implementations.

3.1 Formally Safe Routing Toolkit

The *Formally Safe Routing (FSR)* toolkit [54] attempts to bridge this gap in the context of interdomain routing by unifying research in routing algebras [16] with declarative networking to produce provably correct distributed implementations. Specifically, FSR automates the process of analyzing routing configurations expressed in algebra for safety (i.e. convergence) using the Yices SMT solver [55], and automatically compiles routing algebra into declarative routing implementations.

To enable an evaluation of protocol dynamics and convergence time, FSR uses our extended routing algebra [54] to automatically generate a distributed routing-protocol implementation that matches the policy configuration — avoiding the time-consuming and error-prone task of manually creating an implementation. FSR generates a provably correct translation to a *NDlog* specification, which is then executed using the RapidNet declarative networking engine.

Our choice of *NDlog* as the basis for FSR is motivated by the following. First, the declarative features of *NDlog* allow for straightforward translation from the routing algebra to *NDlog* programs. Second, *NDlog* enables a variety of routing protocols and overlay networks to be specified in a natural and concise manner. Given that *NDlog* specifications are orders of magnitude less code than imperative implementations, this makes possible a clean and concise proof (via logical inductions) of the correctness of the generated *NDlog* programs with regard to safety. The compact specifications also make it easy to incorporate alternative routing mechanisms to the basic path-vector protocol, as we have previously demonstrated [54]. Finally, when compiled and executed, these declarative protocols perform efficiently relative to imperative routing implementations.

Our recent prototype demonstration at SIGCOMM'11 [48] shows how FSR can detect problems in an AS's iBGP configuration (using realistic topologies and policies). We have also used our system to prove sufficient conditions for BGP safety and empirically evaluate protocol dynamics and convergence time.

FSR serves two important communities. For researchers, FSR automates important parts of the design process and provides a common framework for describing, evaluating, and comparing new safety guidelines. For network operators, FSR automates the analysis of internal router (iBGP) and border gateway (eBGP) configurations for safety violations. For both communities, FSR automatically generates realistic protocol implementations to evaluate real network configurations (e.g., to study convergence time) prior to actual deployment.

3.2 Declarative Network Verification

In addition to the FSR toolkit, we have also explored the use of theorem proving for verifying declarative networking programs. We have developed the *DNV (Declarative Network Verification)* [53] toolkit that demonstrate the feasibility of automatically compiling declarative networking programs written in *NDlog* into formal specifications recognizable by a theorem prover (e.g., PVS [2]) for verification. Unlike model checkers, DNV can express properties beyond the temporal properties to which most model-checking techniques are restricted. They also avoid the state exploration problem inherent in model checking. Theorem proving techniques are also sound and complete: once a property is verified, it holds for all instances of the protocol. Moreover, modern theorem provers come with powerful proof engines that support a large portion of automated proof exploration, enabling the proof of non-trivial theorems with relatively modest human effort.

4 Securing Distributed Systems

The *Declarative Secure Distributed Systems* (DS2) platform provides high-level programming abstractions for implementing secure distributed systems, achieved by unifying declarative networking and logic-based access control specifications [12]. DS2 has a wide range of applications, including reconfigurable trust management [35], secure distributed data processing [34], and tunable anonymity [51].

DS2 is motivated in part by the observation that distributed trust management languages share similarities with both data integration languages and the distributed Datalog languages proposed for declarative networking. These languages support the notion of *context* (location) to identify *components* (nodes) in distributed systems. The commonalities between these languages indicate that ideas and methods from the database community are also applicable to processing security policies, suggesting the unification of these declarative languages to create an integrated system.

The DS2 system is currently available for download [47].

4.1 Secure Network Datalog

We developed the *Secure Network Datalog (SeNDlog)* language [59] that unifies *NDlog* and logic-based languages for access control in distributed systems. *SeNDlog* enables network routing, information systems, and security policies to be specified and implemented within a common declarative framework. We have additionally extended existing distributed recursive query processing techniques to execute *SeNDlog* programs to incorporate secure communication among untrusted nodes.

In *SeNDlog*, we bind a set of rules and the associated tuples to reside at a particular node. We do this at the top level for each rule (or set of rules), for example by specifying:

```
At N,
  r1 p :- p1,p2,...,pn.
  r2 p1 :- p2,p3,...,pn.
```

The above rules `r1` and `r2` are in the context of `N`, where `N` is either a variable or a constant representing the principal where the rules reside. If `N` is a variable, it will be instantiated with local information upon rule installation. In a trusted distributed environment, `N` simply represents the network address of a node: either a physical address (e.g., an IP address) or a logical address (e.g., an overlay identifier). In a multi-user multi-layered network environment where multiple users and overlay networks may reside on the same physical node, `N` can include the user name and an overlay network identifier. This is unlike declarative networking in which location specifiers denote physical IP address.

SeNDlog allows different principals or contexts to communicate via import and export of tuples. The communication serves two purposes: (1) maintenance messages as part of a network protocol's updates on routing tables, and (2) distributed derivation of security decisions. Imported tuples from a principal `N` are automatically quoted using "`N says`" to differentiate them from local tuples. During the evaluation of *SeNDlog* rules, we allow derived tuples to be communicated among contexts via the use of *import predicates* and *export predicates*:

- An *import predicate* is of the form "`N says p`" in a rule body, where principal `N` asserts the predicate `p`.

- An *export predicate* is of the form "`N says p@X`" in a rule head, where principal `N` exports the predicate `p` to the context of principal `X`. Here, `X` can be a constant or a variable. If `X` is a variable, in order to make bottom-up evaluation efficient, we further require that the variable `X` occur in the rule body. As a shorthand, we can omit "`N says`" if `N` is the principal where the rule resides.

By exporting tuples only to specified principals, the use of export predicates ensures confidentiality and prevents information leakage. With the above definitions, a *SeNDlog* rule is a Datalog rule where the rule body can include import predicates and the rule head can be an export predicate.

We provide a concrete example based on the declarative path vector protocol as presented in the original declarative routing [31] paper: At every node `Z`, this program takes as input `neighbor(Z,X)` tuples that contain all neighbors `X` for `Z`. The program generates `route(Z,X,P)` tuples, each of which stores the path `P` from source `Z` to destination `X`. The basic protocol specification is similar to the all-pairs reachable example presented in Section 2, with additional predicates for computing the actual path using the `f_concat` function which prepends neighbor `X` to the input path `P`.

The input `carryTraffic` and `acceptRoute` tables respectively represent the export and import policies of node `Z`. Each `carryTraffic(Z,X,Y)` tuple represents the fact that node `Z` is willing to serve all network traffic on behalf of node `X` to node `Y`, and each `acceptRoute(Z,Y,X)` tuple represents the fact that node `Z` will accept a route from node `X` to node `Y`. A more complex version of this protocol will have additional rules that derive `carryTraffic` and `acceptRoute`, avoid path cycles and also derive shortest paths with the least hop count.

The path-vector protocol is used for inter-domain routing over the Internet and is known to be vulnerable to a variety of attacks due to the lack of mechanisms for verifying the authenticity and authorization of routing control traffic. One potential solution is to authenticate every routing control message, as proposed for Secure BGP [50].

```
At Z,
 z1 route(Z,X,P) :- neighbor(Z,X), P=f_initPath(Z,X).
 z2 route(Z,Y,P) :- X says advertise(Y,P), acceptRoute(Z,X,Y).
 z3 advertise(Y,P1)@X :- neighbor(Z,X), route(Z,Y,P),
      carryTraffic(Z,X,Y), P1=f_concat(X,P).
```

In our example program, we can specify such authentication naturally via the use of "`says`" to ensure that all `advertise` tuples are verified by the recipients for authenticity. Rule `z1` takes as input `neighbor(Z,X`) tuples, and computes all the single hop `route(Z,X,P)` containing the path `[Z,X]` from node `Z` to `X`. Rules `z2` and `z3` compute routes of increasing hop counts. Upon receiving an `advertise(Y,P)` tuple from `X`, `Z` uses rule `z2` to decide whether to accept the route advertisement based on its local `acceptRoute` table. If the route is accepted, a `route` tuple is derived locally, and this results in the generation of an `advertise` tuple which is further exported by node `Z` via rule `z3` to some of its neighbors `X` as determined by the policies stored in the local `carryTraffic` table.

SeNDlog is able to compactly specify a variety of secure distributed protocols. Our earlier work [59] has demonstrated, for example, the use of *SeNDlog* for performing secure distributed joins and securing distributed hash tables [8].

4.2 Reconfigurable Security

Although one can achieve a high level of security using a "one-size-fits-all" solution with fixed constructs like `says`, an *extensible trust management* framework where users can write and reconfigure their own constructs like `says` is applicable to a much broader range of settings. For example, programmers could customize the security protocols used by their application based on the execution environment without modifying the application logic. In the *LBTrust* [35] work, we extended SeNDlog to support user-defined security constructs that can be customized and composed in a declarative fashion. To validate our ideas in a production system, we implemented our extension in the LogicBlox [26] system, an emerging commercial Datalog-based platform for enterprise software systems.

We enhanced LogicBlox to support *meta-rules* — Datalog rules that operate on the rules of the program as input, and produce new rules as output — and *meta-constraints* — Datalog constraints that restrict the allowable rules in the program. Security constructs are written using these two ingredients. For example, the `says` construct would consist of meta-rules that rewrite the program to perform signing of all exported messages, and constraints that ensure that all imported messages have valid signatures. We demonstrate that a variety of security primitives for authentication, confidentiality, integrity, speaks-for, and restricted delegation can be supported. Based on these primitives, several existing distributed trust management systems (e.g., Binder [12], SD3 [20], Delegation Logic [22], and SeNDlog) can be implemented in LBTrust.

A follow-up to LBTrust is the *SecureBlox* [34] system, which restricts the use of meta-programming to make it a fully static, compile-time operation. We added support for physical distribution to LBTrust, and looked at performance-security tradeoffs between different constructs in distributed systems. Similar to LBTrust, SecureBlox allows meta-programmability for compile-time code generation based on the security requirements and trust policies of the deployed environment.

While we specifically study security in the LBTrust and SecureBlox work, the general pattern of using meta-programming to decompose a logic program into different aspects representing cross-cutting concerns is more broadly applicable.

4.3 Application-aware Anonymity

To further illustrate the feasibility of our methods and technologies for the development of secure distributed systems, we have conceptualized and implemented the *Application-Aware Anonymity (A^3)* system [7, 51], a distributed peer-to-peer service that provides high-performance anonymity "for the masses". A^3 uses *SeNDlog* for implementing an extensible policy engine for customizing its relay selection and instantiation strategies. A^3 allows applications to construct anonymous Onion [15] paths that adhere to application specific constraints (e.g., end-to-end latency). Unlike existing anonymity systems that construct paths according to predefined criteria, A^3 enables applications to specify the requirements of their anonymous paths. For example, anonymized Voice-over-IP services can request paths with low latency and modest bandwidth requirements, while streaming video broadcasts can request high bandwidth anonymous paths without regard for latency. A^3 is open-source and available for download [7].

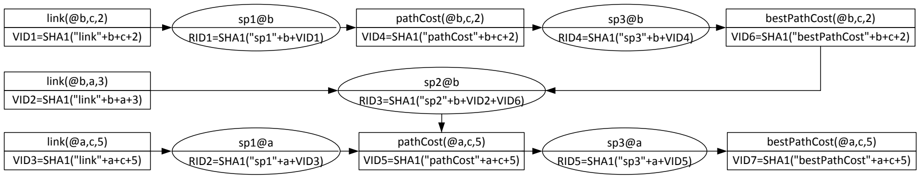


Fig. 1. The provenance graph of the tuple `bestPathCost(@a,c,5)` derived from the execution of the MinCost program.

5 Debugging Distributed Systems

In the context of distributed systems, it is very common for system administrators to perform analysis tasks that essentially amount to *network provenance* [60] queries. For example, they might ask diagnostic queries to determine the root cause of a malfunction, forensic queries to identify the source of an intrusion, or profiling queries to find the reason for suboptimal performance.

The *NetTrails* [58, 60] system is a declarative platform for incrementally maintaining, interactively navigating, and querying network provenance in a distributed system. During the system execution, *NetTrails* incrementally maintains provenance information using RapidNet as its distributed query engine. Our architecture offers a unifying framework, as both maintenance and querying functionalities are specified as *NDlog* programs.

NetTrails consists of two subcomponents: First, a *maintenance engine* takes as input either *NDlog* programs or input/output dependencies captured from legacy applications, and then incrementally computes and maintains network provenance information as distributed relational tables. Second, a *distributed query engine* executes user-customizable provenance queries that are evaluated across multiple nodes. Legacy systems are supported either by modifying the application's source code to explicitly report provenance, or by using an external specification of the application's protocol to derive provenance information by observing a node's inputs and outputs [57].

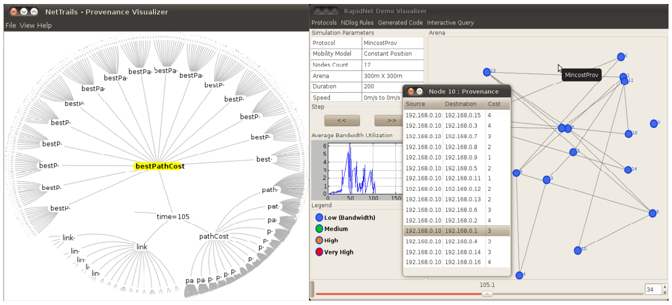
5.1 Network Provenance Model

In *NetTrails*, the provenance graph is internally maintained as relational tables which are distributed and partitioned across all nodes in the network. Network provenance is modeled as an acyclic graph $G(V, E)$. The vertex set V consists of *tuple vertices* and *rule execution vertices*. Each tuple vertex in the graph is either a base tuple or a computation result, and each rule execution vertex represents an instance of a rule execution given a set of input tuples. The edge set E represents dataflows between tuples and rule execution vertices.

To illustrate, we consider an example network consisting of three nodes `a`, `b` and `c` connected by three bi-directional links `(a,b)`, `(a,c)` and `(b,c)` with costs `3`, `5` and `2` respectively. We further consider the following three-rule MinCost program that computes the minimal path cost between each pair of nodes:

```
sp1 pathCost(@S,D,C) :- link(@S,D,C).
sp2 pathCost(@S,D,C1+C2) :- link(@Z,S,C1), bestPathCost(@Z,D,C2).
sp3 bestPathCost(@S,D,min<C>) :- pathCost(@S,D,C).
```

Figure 1 shows the provenance for a specific derived tuple `bestPathCost(@a,c,5)`, based on the dependency logic captured by the MinCost program. For instance,



Simulation Parameters

Protocol	MincostProv
Mobility Model	Constant Position
Nodes Count	17
Arena	300m X 300m
Duration	200
Speed	0m/s to 0m/s

Node 10 : Provenance

Source	Destination	Cost
192.168.0.10	192.168.0.15	4
192.168.0.10	192.168.0.3	4
192.168.0.10	192.168.0.7	3
192.168.0.10	192.168.0.8	2
192.168.0.10	192.168.0.9	1
192.168.0.10	192.168.0.5	2
192.168.0.10	192.168.0.11	1
192.168.0.10	192.168.0.12	2
192.168.0.10	192.168.0.13	2
192.168.0.10	192.168.0.6	3
192.168.0.10	192.168.0.2	4
192.168.0.10	192.168.0.1	3
192.168.0.10	192.168.0.4	3
192.168.0.10	192.168.0.14	3
192.168.0.10	192.168.0.16	4

Fig. 2. A screenshot of the *NetTrails* demonstration at SIGMOD'11.

the figure shows that `bestPathCost(@a,c,5)` is generated from rule `sp3` at node `a` taking `pathCost(@a,c,5)` as the input. To trace further, `pathCost(@a,c,5)` has two derivations: the locally derivable one-hop path $a \rightarrow c$ and the two-hop path $a \rightarrow b \rightarrow c$ that requires a distributed join at `b`.

5.2 Distributed Maintenance and Querying

Given the adoption of a declarative networking engine, data dependencies are explicitly captured in derivation rules.[5] The provenance maintenance in a dynamic system execution can be performed in a straightforward manner: an automatic rule rewrite algorithm takes as input a set of derivation rules, and outputs a modified program that contains additional rules for capturing the provenance information. These additional rules define network provenance in terms of views over base and derived tuples. As the network protocol executes and updates network state, views are incrementally recomputed.

Once generated, network provenance can be queried by issuing distributed queries. Since provenance information is distributed across nodes, query execution performs a traversal of the provenance graphs in a distributed fashion.

NetTrails allows users to customize the provenance queries. For instance, users can query for a tuple's lineage, the set of nodes that have been involved in the derivation of a given tuples, and/or the total number of alternative derivations. To reduce querying overhead, *NetTrails* adopts a set of optimization techniques [60], including caching previously queried results, leveraging alternative tree traversal orders, and performing threshold-based pruning.

An early prototype of *NetTrails* was presented at SIGMOD'11 [58]. Figure 2 shows an example execution of the current version of the demonstration that highlights the provenance of the system state (captured as tuples) for a running MinCost program. One may further issue customized provenance queries and visually inspect the progressive steps of the distributed querying.

5.3 Security and Temporal Extensions

NetTrails provides functionality required for richer provenance queries by adding (i) new provenance models and maintenance strategies for capturing the time, distribution, and causality of updates in distributed systems [56], and (ii) novel query processing and optimization techniques for efficiently and securely answering queries at scale [57].

[5] For legacy applications, the data dependencies (reported by the modified source code or inferred from the observed I/Os) can be formulated as derivation rules as well [57].

NetTrails explicitly captures *causality*: if some network state α depends on some other state β, and β is changed, the provenance of the *change* in α is attributable to the change in β. Additionally, since one of our potential use cases is forensics, *NetTrails* achieves strong *security guarantees* even in the presence of misbehaving and potentially malicious nodes. *NetTrails* utilizes *secure network provenance* [57] to provide the strong guarantee that either a returned provenance query is accurate and complete, or that a misbehaving node is identified with non-repudiable evidence against the node.

To demonstrate the capabilities of *NetTrails*'s temporal and security extensions, we describe a number of use cases of our system, as presented in [57].

Network Routing. The *Border Gateway Protocol* (BGP) used for interdomain routing over the Internet is plagued by a variety of attacks and malfunctions. We have applied *NetTrails* to the Quagga BGP daemon [45] and demonstrated how our solution enables a network administrator to determine why an entry from a routing table has disappeared. We also showed how *NetTrails* can be used to detect well-known BGP misconfigurations.

Distributed Hash Tables. We have applied *NetTrails* to a declarative implementation of the Chord [29] distributed hash table; no modifications are required to the Chord source code. We demonstrated *NetTrails*' ability to detect a well-known attack against Chord in which the attacker gains control over a large fraction of the neighbors of a correct node, and is then able to drop or reroute messages to this node and prevent correct overlay operation.

Hadoop MapReduce. Finally, we have applied *NetTrails* to Hadoop MapReduce [18]. We manually instrumented Hadoop to report provenance at the level of individual key-value pairs. We used Hadoop to encode the *WordCount* program that reports the number of occurrences of each word in a 1.2 GB Wikipedia dataset. In this scenario, we queried for the provenance of a given (unlikely) key-value pair in the output. *NetTrails* revealed that unexpected results might be attributed to a faulty or compromised map worker. More generally, *NetTrails* was able to identify the causes of suspicious MapReduce outputs.

6 Optimizing Distributed Systems

In distributed systems management, operators often configure system parameters that optimize performance objectives, given constraints in the deployment environment. In this section we present our recent work on a declarative optimization platform that enables constraint optimization problems (COP) to be declaratively specified and incrementally executed in distributed systems.

Traditional COP implementation approaches use imperative languages such as C++ or Java and often result in cumbersome and error-prone programs that are difficult to maintain and customize. Moreover, due to scalability and management constraints imposed across administrative domains, it is often necessary to execute COP in a *distributed* setting in which multiple *local* solvers must coordinate with one another. Each local solver handles a portion of the whole problem, and they together achieve a global objective.

Central to our optimization platform is the integration of a *declarative networking engine* [28] with an off-the-shelf constraint solver [1]. We highlight two use cases to which we have applied our platform:

6.1 Use Cases: PUMA and COPE

First, we have developed the *Policy-based Unified Multi-radio Architecture* (PUMA), a declarative constraint solving platform for optimizing wireless mesh networks. In PUMA, network operators can flexibly vary the choice of routing via adaptable *hybrid* routing protocols [24]. The hybrid technique combines several existing protocols (e.g., proactive, reactive, and epidemic) with specific criteria for determining when particular protocols are to be used. The hybrid compositional capabilities are particularly useful for routing in heterogeneous network settings in which application needs and network conditions keep changing over time. In addition, PUMA enables policies for *wireless channel selection* [23] to be declaratively specified and optimized; such policies may reduce network interference and maximize throughput while not violating constraints (for instance, refraining from channels owned exclusively by the primary users [43]).

Second, in our *Cloud Orchestration Policy Engine* (COPE) [25], we use our optimization framework to declaratively control the provisioning, configuration, management and decommissioning of cloud resource orchestration. COPE enables the automatic realization of customer service level agreements while simultaneously conforming to operational objectives of the cloud providers.

Beyond these two use cases, we envision that our platform has a wide-range of potential applications, including optimizing distributed systems for load balancing, robust routing, scheduling, and security.

6.2 *Colog* Language and Compilation

Our optimization platform uses the *Colog* declarative policy language. *Colog* allows operators to concisely model distributed system resources and formulate management decisions as declarative programs with specified goals and constraints. Compared to traditional imperative alternatives, *Colog* results in code that is smaller by orders of magnitude, and is easier to understand, debug and extend. Here, we present high level intuitions of *Colog*; a more comprehensive treatment of the language can be found in our earlier work [23, 25].

Language extensions. Based on *NDlog*, *Colog* extends traditional *NDlog* with constructs for expressing goals and constraints. Two reserved keywords — `goal` and `var` — respectively specify the *optimization goal* and *variables* used by the constraint solver. *Constraint* rules of the form `F1 -> F2, F3, ..., Fn` denote that whenever `F1` is true, then the rule body (`F2` and `F3` and ... and `Fn`) must also be true to satisfy the constraint. Unlike a Datalog rule which derives new values for a predicate, a constraint *restricts* a predicate's allowed values, hence representing an invariant that must be maintained at all times. These are used by the solver to limit the search space when computing the optimization goal. Using *Colog*, it is easy to customize policies simply by modifying the goals and constraints, and by adding additional derivation rules.

Distributed COP. *Colog* is extended for execution in a distributed setting. At a high level, multiple solver nodes execute a *local* COP, and then iteratively exchange COP results with neighboring nodes until a stopping condition is reached. Similar to *NDlog*, in the distributed COP program, a location specifier `@` denotes the source location of each corresponding tuple. This allows us to write rules in which the input data span multiple nodes — a convenient language construct for formulating distributed optimizations.

One of the interesting aspects of *Colog*, from a query processing standpoint, is our integration of RapidNet (an incremental bottom-up distributed Datalog evaluation engine) and Gecode (a top-down goal-oriented constraint solver). This integration allows us to implement a distributed solver that can perform incremental and distributed constraint optimizations.

To execute distributed COP rules, *Colog* uses RapidNet, which already provides a runtime environment for implementing these rules. At a high level, each distributed rule or constraint (with multiple distinct location specifiers) is rewritten using a *localization* rewrite [28] step. This transformation results in rule bodies that can be executed locally and rule heads that can be derived and sent across nodes. The beauty of this rewrite is that even if the original program expresses distributed properties and constraints, the rewrite process will realize multiple local COP operations at different nodes, and have the output of COP operations via derivations sent across nodes.

7 Acknowledgments

Our work on declarative networking has been generously funded by NSF (CNS-0721845, CNS-0831376, IIS-0812270, CCF-0820208, CNS-0845552, CNS-1040672, CNS-1065130, and CNS-1117052), AFOSR MURI grant FA9550-08-1-0352, DARPA SAFER award N66001-11-C-4020, and DARPA Air Force Research Laboratory (AFRL) Contract FA8750-07-C-0169. We would also like to thank our collaborators listed on the NetDB@Penn site [39] for their contributions to the various research efforts described in this paper.

References

1. Gecode constraint development environment. `http://www.gecode.org/`.
2. PVS Specification and Verification System. `http://pvs.csl.sri.com/`.
3. RapidNet. `http://netdb.cis.upenn.edu/rapidnet/`.
4. P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. M. Hellerstein, and R. Sears. Boom Analytics: Exploring Data-Centric, Declarative Programming for the Cloud. In *Proceedings of Eurosys*, 2010.
5. P. Alvaro, W. Marczak, N. Conway, J. M. Hellerstein, D. Maier, and R. C. Sears. Dedalus: Datalog in time and space. Technical Report UCB/EECS-2009-173, EECS Department, University of California, Berkeley, Dec 2009.
6. T. J. Ameloot, F. Neven, and J. Van den Bussche. Relational Transducers for Declarative Networking. In *PODS*, 2011.
7. Application Aware Anonymity. `http://a3.cis.upenn.edu/`.
8. H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Looking Up Data in P2P Systems. *Communications of the ACM, Vol. 46, No. 2*, 2003.
9. I. Balbin and K. Ramamohanarao. A Generalization of the Differential Approach to Recursive Query Evaluation. *Journal of Logic Prog, 4(3):259–262*, 1987.
10. X. Chen, Y. Mao, Z. M. Mao, and J. van der Merwe. Declarative Configuration Management for Complex and Dynamic Networks. In *CoNEXT*, 2010.
11. D. C. Chu, L. Popa, A. Tavakoli, J. M. Hellerstein, P. Levis, S. Shenker, and I. Stoica. The Design and Implementation of a Declarative Sensor Network System. In *5th ACM Conference on Embedded networked Sensor Systems (SenSys)*, 2007.
12. J. DeTreville. Binder: A logic-based security language. In *IEEE Symposium on Security and Privacy*, 2002.
13. M. Gelfond and V. Lifschitz. The Stable Model Semantics For Logic Programming. In *ICLP/SLP*, pages 1070–1080, 1988.

14. H. Gill, T. Saeed, Q. Fei, Z. Zhang, and B. T. Loo. An Open-source and Declarative Approach Towards Teaching Large-scale Networked Systems Programming. In *SIGCOMM Education Workshop*, 2011.
15. D. Goldschlag, M. Reed, and P. Syverson. Onion Routing. *Communications of the ACM*, 42(2):39–41, 1999.
16. T. G. Griffin and J. L. Sobrinho. Metarouting. In *ACM SIGCOMM*, 2005.
17. A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining Views Incrementally. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1993.
18. Hadoop. `http://hadoop.apache.org/`.
19. J. M. Hellerstein. Declarative imperative: Experiences and conjectures in distributed logic. 2010. SIGMOD Record 39(1).
20. T. Jim. SD3: A Trust Management System With Certified Evaluation. In *IEEE Symposium on Security and Privacy*, 2001.
21. E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3):263–297, 2000.
22. N. Li, B. N. Grosof, and J. Feigenbaum. Delegation Logic: A logic-based approach to distributed authorization. *ACM TISSEC*, 2003.
23. C. Liu, R. Correa, H. Gill, T. Gill, X. Li, S. Muthukumar, T. Saeed, B. T. Loo, and P. Basu. PUMA: Policy-based Unified Multi-radio Architecture for Agile Mesh Networking. In *4th International Conference on Communication Systems and Networks (COMSNETS)*, 2012.
24. C. Liu, R. Correa, X. Li, P. Basu, B. T. Loo, and Y. Mao. Declarative policy-based adaptive mobile ad hoc networking. *IEEE/ACM Transactions on Networking (ToN)*, 2011.
25. C. Liu, B. T. Loo, and Y. Mao. Declarative Automated Cloud Resource Orchestration. In *ACM Symposium on Cloud Computing (SOCC)*, 2011.
26. LogicBlox Inc. `http://www.logicblox.com/`.
27. 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 ACM SIGMOD International Conference on Management of Data*, 2006.
28. B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative Networking. In *Communications of the ACM (CACM)*, 2009.
29. B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing Declarative Overlays. In *Proceedings of ACM Symposium on Operating Systems Principles*, 2005.
30. B. T. Loo, J. M. Hellerstein, and I. Stoica. Customizable Routing with Declarative Queries. In *ACM SIGCOMM Hot Topics in Networks*, 2004.
31. B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative Routing: Extensible Routing with Declarative Queries. In *Proceedings of ACM SIGCOMM Conference on Data Communication*, 2005.
32. Y. Mao. On the declarativity of declarative networking. In *ACM NetDB Workshop*, 2009.
33. Y. Mao, B. T. Loo, Z. Ives, and J. M. Smith. MOSAIC: Unified Platform for Dynamic Overlay Selection and Composition. In *CoNEXT*, 2008.
34. W. R. Marczak, S. S. Huang, M. Bravenboer, M. Sherr, B. T. Loo, and M. Aref. SecureBlox: Customizable Secure Distributed Data Processing. In *SIGMOD*, 2010.
35. W. R. Marczak, D. Zook, W. Zhou, M. Aref, and B. T. Loo. Declarative Reconfigurable Trust Management. In *Proceedings of Conference on Innovative Data Systems Research (CIDR)*, 2009.
36. Mengmeng Liu and Nicholas Taylor and Wenchao Zhou and Zachary Ives and Boon Thau Loo. Recursive Computation of Regions and Connectivity in Networks. In *Proceedings of IEEE Conference on Data Engineering (ICDE)*, 2009.

37. S. C. Muthukumar, X. Li, C. Liu, J. B. Kopena, M. Oprea, R. Correa, B. T. Loo, and P. Basu. RapidMesh: declarative toolkit for rapid experimentation of wireless mesh networks. In *WINTECH*, 2009.
38. S. C. Muthukumar, X. Li, C. Liu, J. B. Kopena, M. Oprea, and B. T. Loo. Declarative toolkit for rapid network protocol simulation and experimentation. In *SIGCOMM (demo)*, 2009.
39. NetDB@Penn. `http://netdb.cis.upenn.edu/`.
40. Network Simulator 3. `http://www.nsnam.org/`.
41. V. Nigam, L. Jia, B. T. Loo, and A. Scedrov. Maintaining distributed logic programs incrementally. In *13th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP)*, 2011.
42. ORBIT - Wireless Network Testbed. `http://www.orbit-lab.org/`.
43. F. Perich. Policy-based Network Management for NeXt Generation Spectrum Access Control. In *DySPAN*, 2007.
44. PlanetLab. Global testbed. `http://www.planet-lab.org/`.
45. Quagga Routing Suite. `http://www.quagga.net/`.
46. R. Ramakrishnan and J. D. Ullman. A Survey of Research on Deductive Database Systems. *Journal of Logic Programming*, 23(2):125–149, 1993.
47. RapidNet Declarative Networking Engine. `http://netdb.cis.upenn.edu/rapidnet/`.
48. Y. Ren, W. Zhou, A. Wang, L. Jia, A. J. Gurney, B. T. Loo, and J. Rexford. FSR: Formal Analysis and Implementation Toolkit for Safe Inter-domain Routing. In *ACM SIGCOMM Conference on Data Communication (demonstration)*, 2011.
49. D. Saccà and C. Zaniolo. Stable Models and Non-Determinism in Logic Programs with Negation. In *PODS*, pages 205–217, 1990.
50. Secure BGP. `http://www.ir.bbn.com/sbgp/`.
51. M. Sherr, A. Mao, W. R. Marczak, W. Zhou, B. T. Loo, and M. Blaze. A3: An Extensible Platform for Application-Aware Anonymity. In *Network and Distributed System Security*, 2010.
52. A. Singh, T. Das, P. Maniatis, P. Druschel, and T. Roscoe. BFT Protocols Under Fire. In *USENIX Symposium on Networked Systems Design and Implementation*, 2008.
53. A. Wang, P. Basu, B. T. Loo, and O. Sokolsky. Towards declarative network verification. In *11th International Symposium on Practical Aspects of Declarative Languages (PADL)*, 2009.
54. A. Wang, L. Jia, W. Zhou, Y. Ren, B. T. Loo, J. Rexford, V. Nigam, A. Scedrov, and C. Talcott. FSR: Formal analysis and implementation toolkit for safe inter-domain routing. University of Pennsylvania CIS Technical Report No. MS-CIS-11-10, 2011, `http://repository.upenn.edu/cis_reports/954/`.
55. Yices. `http://yices.csl.sri.com/`.
56. W. Zhou, L. Ding, A. Haeberlen, Z. Ives, and B. T. Loo. Tap: Time-aware provenance for distributed systems. In *3rd USENIX Workshop on the Theory and Practice of Provenance (TaPP '11)*, 2011.
57. W. Zhou, Q. Fei, A. Narayan, A. Haeberlen, B. T. Loo, and M. Sherr. Secure network provenance. In *Proceedings of ACM Symposium on Operating Systems Principles*, 2011.
58. W. Zhou, Q. Fei, S. Sun, T. Tao, A. Haeberlen, Z. Ives, B. T. Loo, and M. Sherr. Nettrails: A declarative platform for provenance maintenance and querying in distributed systems. In *SIGMOD (demonstration)*, 2011.
59. W. Zhou, Y. Mao, B. T. Loo, and M. Abadi. Unified Declarative Platform for Secure Networked Information Systems. In *Proceedings of IEEE Conference on Data Engineering (ICDE)*, 2009.
60. W. Zhou, M. Sherr, T. Tao, X. Li, B. T. Loo, and Y. Mao. Efficient querying and maintenance of network provenance at Internet-scale. In *Proc. SIGMOD*, 2010.
61. W. Zhou, O. Sokolsky, B. T. Loo, and I. Lee. Dmac: Distributed monitoring and checking. In *9th International Workshop on Runtime Verification (RV)*, 2009.