KnowNet: Towards a Knowledge Plane for Enterprise Network Management

Ren Quinn, Josh Kunz, Aisha Syed, Joe Breen, Sneha Kasera, Rob Ricci, Jacobus Van der Merwe
renquinn@cs.utah.edu, josh.kunz@utah.edu, aisha.syed@utah.edu,
joe.breen@utah.edu, kasera@cs.utah.edu, ricci@cs.utah.edu, kobus@cs.utah.edu
University of Utah

***Abstract*—Network management tasks remain tedious and error-prone, and often require complex reasoning on the part of the network administrator. With KnowNet we address the challenge of reasoning about network management by approaching it as a set of cooperating applications executing over a knowledge graph which captures data and information about the network and the applications that manage and reason over it. We apply our approach to enterprise network management by developing a suite of cooperating applications that deals with security and application performance management in an enterprise network.**

I. Introduction

Despite longstanding recognition by the networking research community of the importance of an improved approach to network management and the need for automated solutions [1], the state of the art continues to resist these efforts. In practice network management remains manual, tedious and error-prone, and often requires complex reasoning on the part of the network administrator. Automating the reasoning done by domain experts is the key challenge in realizing a more systematic approach to network management. This is difficult, largely in part because this reasoning must take into account large and diverse quantities of data. This data is collected from a number of different sources, and in order to be meaningfully interpreted, it must be correlated with the operational state of the network at the time it was collected. Complex relationships between hardware, protocols, services, and their operational state must be untangled in order to fully understand faults and effect change in the network.

We argue that a network management system that can support such functionality requires the means to capture network data and knowledge of the network in a holistic manner, allows for network management applications to easily interact with and reason about this knowledge, and to effect change in the network itself. In our work we take a pragmatic approach towards realizing such a system. Our key insight is that network administrators already deploy a multitude of systems to monitor and manage their networks, and that *they* (the administrators) are the reasoning and logic that tie these systems together. Rather than imposing a new network management system on network administrators that might require them to replace the tools and systems they already use, we propose to provide them with a framework that can turn the systems they are already using into a cooperative whole and allow them to add to that whole by easily automating the reasoning tasks they are currently performing manually.

978-1-5090-0223-8/16/$31.00

Towards this end, we present KnowNet, a knowledge-centric network management system that facilitates knowledge sharing and discovery, thus enabling operators to more effectively reason about network state, determine appropriate actions, and 'write' to the network to effect needed change. Applications use the *knowledge graph* at the core of KnowNet to represent data collected about the network and knowledge derived from that data. Knowledge graphs have gained popularity in a number of domains [2], [3], [4], [5], due to their ability to express and discover interesting relationships over large datasets, enabling complex reasoning about interrelated data [2], [6], [7]. In our work on KnowNet, we explore the use of a knowledge graph to capture information and knowledge in a network management setting. As such, our work is a modest step towards a practical (single domain) knowledge plane [1].

The knowledge graph in KnowNet is used to capture more than just data about the network—specifically, network management applications in KnowNet interact with each other and the network itself via the knowledge graph. This allows network management actions to also be captured as part of the holistic network view in KnowNet. To realize such a collaborative network management system, we have developed a knowledge graph, called *Kilo*[1], in order to explore features that are particularly useful in the context of network management, and may not exist in such a combination in available knowledge graph implementations. Specifically, Kilo includes the typical set of knowledge graph primitives: `insert`, `query`, and `delete`. In addition to these, Kilo also supports a `subscribe` primitive to learn about network related events and deals with `time` natively because of the importance of capturing network state over time.

To explore the practical utility of our architecture, we developed a suite of network management applications specifically aimed at *enterprise network management*. We envision an enterprise network where both basic network functionality as well as sophisticated enterprise specific network management functions are governed by KnowNet-enabled applications. Motivated by a recent report on the topmost enterprise network challenges [8], we developed applications to deal with performance and security management. We show that existing tools can be incorporated into KnowNet, allowing them to interact with new network management functions to realize our objective of holistic network management.

While borrowing from earlier network management ef-

[1]The term Knowledge Graph shares an abbreviation, KG, with Kilogram. Thus, Kilo.

forts [9], [10], [1], [11], [12], [13], [14], our work on KnowNet is unique along several dimensions. First, we believe our work is among the first to explore the use of a knowledge graph abstraction to capture all of the data, knowledge and actions associated with network management. Second, our approach of attacking the network management problem with a collection of cooperating applications affords a unique tradeoff between flexibility and simplicity. In KnowNet we do not impose structure in our knowledge graph allowing applications to define structure according to the data and management functions they deal with. On the flip side, this requires more sophistication from applications wanting to cooperate. Finally, in KnowNet we take a pragmatic approach to network management (and indeed illustrate the aforementioned flexibility of our approach) by incorporating both existing as well as new network management applications in a holistic framework.

This paper makes the following contributions:

• We present the design and implementation of KnowNet, a network management framework that divides the responsibility of reasoning about the network among cooperating applications (§ III).
• We present the design and implementation of Kilo, a knowledge graph that is specialized to the domain of network management (§ III-A).
• To illustrate the utility of KnowNet, we develop a suite of cooperating *security*- and *performance*-related network management applications for enterprise networks (§ IV). Some applications in this suite exhibit complex reasoning, while other applications incorporate existing network management tools (sFlow-RT and Snort)—we show how KnowNet enables these existing applications to be brought into a holistic network management system.
• We perform an extensive evaluation of our approach in an emulated enterprise network environment (§ V). This evaluation demonstrates the ability of a KnowNet-enabled network to maintain network health under challenging conditions, and to scale to enterprise-sized networks.

II. Background and Motivation

Beyond the knowledge plane vision [1], our work is specifically influenced by NetSearch [15], a search and information retrieval tool inspired by web search engines, but designed for networks. The NetSearch goal is to organize network data in order to expose relationships that would otherwise be difficult to infer. In KnowNet we go beyond information retrieval to a more sophisticated knowledge store, providing operators with a systematic reasoning platform, and in so doing, also allowing the platform to serve as the basis for automating closed-loop network management actions. Our key insight is that relationship mining can be improved using a knowledge graph whose core representation is built on such relationships, and that capturing these relationships is critical to enabling automated network management functions.

A *knowledge graph* (KG) [5] is a data structure used to represent a collection of *facts*, which collectively form *knowledge*. Facts are composed of *node* entities and the relations (*edges*) between them. These are represented as *triples*: for example, `switchA has-interface if0`. A *query* against this graph may take the form of a subgraph to match (for example, *"all subgraphs such that* `A has-type switch` *and* `A has-interface B` *and* `C uses-interface B`*"* for searching for paths between two nodes). The query is then fulfilled by returning all subgraphs with entities that can satisfy the query and returning them in place of the variables (denoted by a string where the first character is a capital letter. e.g., A, B and C in the above examples).

Using a knowledge graph abstraction has become a popular model [2] for analyzing data. By expressing data as a network of relationships, it is possible to explore connections and infer new facts from the connection [3]. Knowledge graphs have been shown to scale to large datasets and complex queries [2], [6]. As such, our current efforts focus on developing the correct primitives and abstractions for a network management knowledge graph, and using it in a modest sized network setting. Therefore, scalability is not a focus of our current work.

The following sections justify our decision to use a knowledge graph as the core of the KnowNet approach.

A. Network Data Representation

We have analyzed a number of approaches to network data representation [16], [9], [17], [18], [10] and combined them with our own experience to identify a number of key requirements that we feel must be met in any new approach to network data representation in order to enable the complete range of network management and operations tasks. We argue that these requirements are best met with a knowledge-graph based approach.

Enable knowledge composition. Most of network management data is key-value oriented. Sflow data, MIB tables, IDS alerts, interface configuration, OpenFlow rules, and even packets themselves are all simply collections of key-value pairs. While this data representation is clean and concise, there is no way to systematically glean meaning or understanding directly from the raw pairs; this function usually occurs in the minds of the network operators. Therefore, as we represent this data, we need an additional feature associated with the keys and the values to give them semantics. The KnowNet knowledge-graph based approach allows these relationships and connections between the basic key-value data representation to be captured.

Data source agnostic. Data representation should be designed independent from the tools that provide the data. Many proposed data constructs reflect the source of the data (e.g., events from Lithium [17], packets from NetSight [18]). Exposing the semantics of the data source in the data representation limits the application of the data to devices and tools that are already constructed using it. By converting lower level data representations into a knowledge-graph, KnowNet allows us to integrate with any data sources and to combine them into a holistic network data representation.

B. Network Management with Knowledge Graphs

Beyond data representation, a knowledge-graph based approach has properties that make its use particularly attractive in the network management domain.

Flexible Structure With Hierarchy: The data model imposed by a knowledge graph is simple, and makes it easy to build flexible, complex structures on top. These properties mean that KnowNet does not need to impose any particular schema on the applications written for it; this makes it simple to extend it and to add new types of network management applications. Of course, there are natural classes of objects, as well as layering hierarchies, in a network, and the set of base applications in KnowNet builds a basic set of facts using a basic hierarchical model. Other applications can interact with the model at whichever level (service, forwarding path, physical topology) is appropriate for the application. It is also possible to draw inferences between layers of abstractions.

Data in a knowledge graph is organized by *all* of its relationships, whether that be the kind of data (e.g., performance measurements, security alerts) or the network elements they are explaining (e.g., nodeA, switchB). (In contrast to a traditional database setting where such data will be in separate tables.) This means instead of querying for all performance measurements and all security alerts and then filtering by whichever element we are interested in, we can now simply query for the node we are interested in and get all knowledge associated with it. This not only simplifies large queries, it also more easily enables knowledge discovery by allowing applications to query for relations previously unknown to them as they were inserted by other applications.

While the flexibility of a dynamic schema is useful, it also implies a number of challenges. Particularly difficult is understanding the semantics of different entity and relation names, especially when facts are inserted by different sources. We save this problem for future work. At this point, loose semantics allow us to better explore relationships between network data. Our current goal is primarily to explore the use of a knowledge graph with embedded network primitives.

Network-like Structure: The clearest natural fit between a knowledge graph and a network is that the topology of a network itself is naturally represented as a graph. Many common network algorithms, such as the minimum spanning tree algorithm and various routing algorithms, already work on graph structures. The KnowNet apps that provide basic network functionality, e.g., routing, make use of these properties.

Dependency Chains: A less obvious, but even more important, fit between knowledge graphs and the network management domain is their strength in representing *dependencies* [13]. For example, a *service* (such as a webserver, fileserver, or public IP connectivity) relies upon *paths* through the network to deliver that service's packets. These paths are, in turn, dependent on the protocols that establish them and the hardware that implements them. These dependencies, are not, however, strictly hierarchical: for example, the performance of a given path (and therefore the service that uses it) can depend on the other services using the path or hops along the path, and their dynamic state, such as the current offered load, level of congestion, or failures (both hard and soft). Representing the network using a knowledge graph enables us to explore the causes and effects of network behavior.

III. KnowNet Architecture

The KnowNet architecture is shown in Figure 1. It is divided into two parts: a centralized knowledge store, realized as the Kilo knowledge graph, and a set of cooperating applications that use the data, information and knowledge in the knowledge store to perform their network management functions.

The KnowNet cooperating applications can be divided into a set of *network abstraction* apps that interface with the network proper, as well as a set of *network management* apps that realize the various network management functions. The KnowNet architecture presents a generic network management framework that can be applied to any network management environment. The functionality of the applications, however, is specific to the particular network environment in which the system is deployed. For example, in Section IV we describe the suite of applications we have developed targeted to enterprise network management.

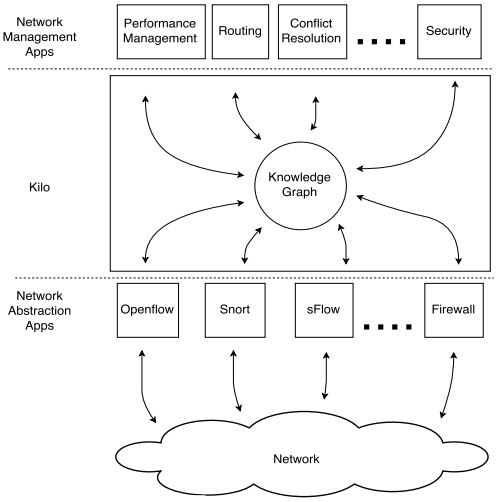


Fig. 1: KnowNet architecture

We note that in an abstract sense, the KnowNet architecture "simply" mimics the network management actions taking place today: Network administrators deploy various disjoint "read" or "write" systems in their networks, and interpret and reason about the output from some "read" system to decide what actions to induce in another "write" system. Our goal with KnowNet is to create a holistic network management system that allows these disparate systems to cooperate and to further facilitate network operators in complex, timely reasoning over the network state.

A. A Network Management Knowledge Graph

Conventional knowledge graph systems, such as Freebase [2], Naga/Yago [3], [6], and DeepDive [4], are primarily designed for harnessing relatively static knowledge.

The knowledge associated with network management, on the contrary, is much more dynamic. We identify three specific network management related requirements that existing knowledge graph systems are not supporting, or are supporting poorly.

Reactive Queries. Network management functions are often executed in a reactive manner, e.g., if event A happens, execute action B. This suggests the need for a knowledge graph primitive that would inform applications should certain knowledge be added to the graph, i.e., a *subscription* primitive.

Time-Varying Data. In a network management scenario, knowledge changes over time, and tracking knowledge as a function of time becomes important. This implies the need for a knowledge graph *time tracking* primitive to support network management.

Data Curation. Because knowledge changes over time in a network management scenario, the utility of facts become dependent on the time they were inserted. Facts might become irrelevant or even misleading as time progresses. Further, network management functions tend to generate significant volumes of data. This implies the need for *data curation* primitives in support of network management.

B. KnowNet's Knowledge Graph: Kilo

To experiment with features necessary or useful for network-management, we implemented Kilo, KnowNet's knowledge graph, from scratch. This approach allows us to more flexibly explore knowledge graph primitives that facilitate network management. Kilo is inspired by Naga [3], a knowledge graph built for harnessing large amounts of text-based knowledge. Kilo borrows Naga's core algorithms and functionality, namely, inserts and queries. We also implemented an HTTP interface enabling multiple apps to asynchronously work with the knowledge graph.

Kilo provides typical knowledge graph primitives such as *insert*, *query* and *delete*. In addition we realized network management specific primitives in Kilo, namely *subscriptions* and dealing explicitly with *time*. We consider relevant implementation details of the Kilo primitives below.

Subscriptions. A key part of network management is reacting to specific events. Kilo enables this dynamic by allowing applications to receive new results for a query as the knowledge graph is updated through the subscription primitive. One motivation for including this primitive in the graph is to ensure applications can react as quickly as possible to network state changes. External mechanisms require techniques such as polling which might miss transient states.

Delete. Every knowledge store needs the ability to forget a piece of knowledge. Knowledge can become stale, irrelevant, or cumbersome to manage. In Kilo, we take a reference-counting approach to fact deletion. Each participating application is required to increment, and later decrement, a counter which signifies whether or not an application still needs access to that data. When the counter reaches zero, Kilo understands that the data is no longer necessary and is eligible for removal from the active knowledge graph. We have implemented an `archiving` app which uses the properties implemented by the delete primitive to perform data curation based on the policies and needs of an enterprise network.

Time. We natively support time as a primitive by attaching a timestamp to every fact. This allows for queries that can include optional minimum and maximum timestamps. As the query executes, the facts are filtered based on these time parameters, effectively pruning the graph and allowing us to see a snapshot of the graph as it existed at any point in time.

IV. Enterprise Management Apps

Applications written for KnowNet are straightforward. The common workflow involves: (1) *Querying for or subscribing to certain facts:* for example, an application working to route around outages might look for facts indicating that interfaces are down, or an application that performs active measurements might look for facts stating that another application has requested a measurement of a certain path. Note that this means more than simple data queries. The ecosystem of multiple apps running on top of the knowledge graph provides for the composition of higher-level *knowledge* as a result of understanding lower-level data points. (2) *Interpreting the facts:* this is where the bulk of the "work" of the application is done: computing routes, performing measurements, etc. This is where the use of a knowledge graph is most desirable. It allows applications to understand network behavior as an operator would, by examining the network at a low level in order to produce insights at a higher level. The knowledge graph enables applications to systematically capture the results of this human-like reasoning in a relationship-based manner. This step is essentially the application of domain knowledge to the state of the network, the result of which will later be inserted into the graph for other applications to learn from. (3) *Inserting any resulting facts into the knowledge graph:* these facts may be simple inferences from existing data, may represent the results of measurements or probes of the network, may comprise requests for other applications to take actions, etc. This new knowledge (whether inferred or observed) exists by way of the domain knowledge encoded in the individual apps. Because of the open nature of our knowledge graph, most apps can focus on a single, relatively simple task, relying on the cooperation of other simple apps to enable rich knowledge composition. We believe this streamlined approach greatly simplifies the process of capturing and applying domain knowledge to the network.

To explore KnowNet's ability to enable network management, we have developed a suite of simple, yet sophisticated, example applications to enable knowledge-driven network management in an enterprise network. While these applications are simple, they focus on continuing challenges in enterprise network management [8], namely, basic network functionality and bootstrapping, performance monitoring and management, and security management. We believe these tasks remain difficult (if not impossible) to perform in existing frameworks (e.g., taking active measurements on paths that are currently not in use).

A. Basic Network Functionality

Our first set of applications provides a base level of network functionality: establishing the IP connectivity that is expected of any enterprise network.

Bootstrapping: Two applications in KnowNet work together to bootstrap basic connectivity: the `topology` app, which configures all participating nodes and switches, inventories switches, and maintains basic local properties such as interface state; and the `openflow` app, which wraps an OpenFlow controller and provides an interface for configuring the controller and its flow rules. Both of these apps are heavily used by other apps in the ecosystem, and as such provide a *de facto* set of abstractions.

IP connectivity: Once basic connectivity has been enabled, the `routing` app begins handling global routing. It uses information from the `topology` app to calculate routes, and sends commands to the `openflow` app to update routing tables. If QoS was enabled on the network, the `routing` app also takes that into account in its path installation.

Basic service management: Our `services` app essentially serves as a QoS manager for services by dynamically configuring rate limiters and priority queues as informed by operator-defined SLAs. It builds on the basic topology information and flow rule commands exported by the bootstrapping apps.

Conflict resolution: We implemented a proof-of-concept `resolver` that detects and resolves conflicting knowledge by using an extensible network dependency model to detect conflicts between commands that may be acting at the same or different levels. For example, it is capable of determining, by traversing the knowledge graph, that a change at the interface level might conflict with another higher-level change at the path level. Note that our architecture does not depend on performing conflict resolution in this manner, and can work with other conflict resolution approaches [14], [19].

B. Performance Monitoring and Management

Our next set of applications look at more than just maintaining connectivity, they maintain the performance of the network by: monitoring network activity, re-routing flows to meet performance goals, and re-configuring the network if necessary to solve problems, all by combining knowledge about the network from different sources within the knowledge graph.

Measurement: We provide wrapper apps for commonly-used sources of network performance data such as `sFlow-RT` and `ping`. These apps insert one-time or periodic measurements of the network and insert them into the knowledge graph. The `sflow` app is passive—it collects flow statistics from switches, and is capable of providing analysis of specific flows on-demand when triggered by commands from other apps. The `ping` app takes active measurements of path latency when requested. The `ping` app is capable of measuring arbitrary paths: it interacts with the `routing` app to arrange for its packets to be sent on specific paths to enable it to measure paths that are not currently in use by any service.

Network status: The `iface` and `link` apps are responsible for tracking the state of interfaces and links, respectively. They watch for both interfaces that have failed completely (hard errors) and those that have started to see errors such as bad CRCs (soft errors). In both cases, the apps use their ability to explore the knowledge graph to discover paths and services using the affected interfaces or links, and to insert facts indicating that affected services need to be re-routed; actual re-routing is left to the apps monitoring those services.

Performance management: The Performance Tracker (`PT`) app is responsible for keeping track of the performance of all service flows in the network, checking to see if any SLAs are violated, then making any necessary changes, such as rerouting affected flows to correct any violations. Fulfilling these responsibilities requires various types of dynamic information. First, the app needs to constantly receive fresh connectivity and latency measurements for all network paths on which any service flows are running. In addition, it also needs periodic measurement information for the *non-service paths* so that it knows if better performing paths are available. Second, `PT` needs information about current forwarding tables so that if any paths currently in use appear to be falling below the acceptable performance threshold, the app can discover which service flows are using the affected path. Finally, `PT` needs to know what SLAs it is trying to meet.

All of this information needed by `PT`, is performance app which is simply stored as facts "connected" with the appropriate nodes in the knowledge graph.

The `flow` app is responsible for rate-limiting individual flows that are exceeding configured traffic thresholds as defined by the SLA. The app works in concert with the `sflow` app, from which it receives per-flow statistics, and the `openflow` app—when `flow` detects excessive bandwidth in the `sflow` data, it issues a command to the `openflow` app to install rate-limiting rules.

C. Security Management

Our final set of apps manage the security of the network; they leverage standard components such as Intrusion Detection Systems (IDS). However, because they are able to examine the entire state of the network—including looking at past behavior—they are able to go much farther in terms of dealing with the practical concerns faced by enterprise networks.

Intrusion detection and prevention: We wrote a `snort` app for KnowNet that runs as an output module for the Snort IDS, inserting the alerts that it produces as facts in the KnowNet knowledge base.

With Snort alerts in KnowNet, other applications are now capable of taking this information and combining it with knowledge about current alerts, topology, services, and past behavior. For instance, a security reasoning or correlation app can combine knowledge from other apps to more effectively determine how to handle the alert, or decide to ignore the alert if it can be determined that the alert poses no real threat. We built such an app to interpret the Snort alerts and combine them with knowledge of the topology as well as

the network operator policy. We then use this information to write more expressive (and more dynamic) host-based firewall rules. In our implementation, these rules are managed by the `firewall` app.

Vulnerability management and event discovery: To extend the knowledge capabilities of our security apps, we can apply event discovery techniques [20] to infer high-level events from stored facts. Event discovery like this is especially useful in an enterprise network for situations like tracking exploits of an unknown vulnerability. This type of event discovery is relatively easy to implement using our knowledge graph. It works by looking for relations which share a single fact in the knowledge store (e.g., an IP address). This collection of facts could then be filtered by common attributes such as time (e.g., facts occurring on the same day or within 10 minutes of each other) or space (e.g., routers in the same building or a specific subnet). After filtering, the remaining facts precisely define the high-level event as a certain shape in the graph. The shape is simply represented by a graph query. Operationally, the administrator (or security professional) can be presented with these facts and given the opportunity to define the event. This means, at a minimum, naming the event and describing it. Any future set of facts that match this shape or query, will be explained or summarized using more concise, event facts as defined by the network operator. In this way KnowNet allows *knowledge growth* as the new knowledge becomes part of the stored/known knowledge.

We explored the knowledge growth abilities of KnowNet by developing a basic `event discovery` app. This app takes as input the class of facts that are of interest, e.g., for host-based event discovery we tell the app to match on IP address facts. The app dynamically subscribes to measurements and events associated with this class of facts. Once the subscription triggers, the app performs other knowledge graph queries for facts which are associated with the triggering fact within a specified time window. The results are then presented to an administrator as a possible event. We describe the use of this app in a simple intrusion detection scenario in Section V-B.

V. Evaluation

To develop and test apps within KnowNet, we emulate an enterprise network in Emulab. Our emulated network consists of a typical two-tiered hierarchical hub-and-spoke topology. Switching is performed in software by Open vSwitch, which provides a rich configuration interface, including support for OpenFlow.

The network is bootstrapped by the `topology` and `openflow` apps. We enable three QoS classes, and set services to default to the lowest priority queue. This is enough to begin running the `routing` app to establish basic connectivity. All experiments also initialize our measurement apps on the access switches; they are configured to monitor both service and non-service paths. Flow statistics are collected by the `sflow` app.

Primitive	Performance (ms) Mean	Std. Dev.
Query	43	37
Subscription	52	38
Insert	0.1	0.2

TABLE I: Primitive performance times.

A. Performance Evaluation

With the exception of a few queries and subscriptions, performance remains acceptable as long as we responsibly manage our data, keeping the fact count to a manageable level. Table I reveals that most of the time, the primitive operations execute within tens of milliseconds. We noticed that, especially for queries, the worst case performance occurs with larger queries which have more variables to match on. However, the majority of the queries in our apps use no more than 4 or 5 variables.

These benchmarks show that our domain-specific primitives in Kilo can be performant in the context of realistic networks and applications. The prototype of Kilo is a from-scratch implementation of a knowledge graph—these results give us confidence that introducing our primitives into a mature knowledge graph implementation would yield acceptable performance.

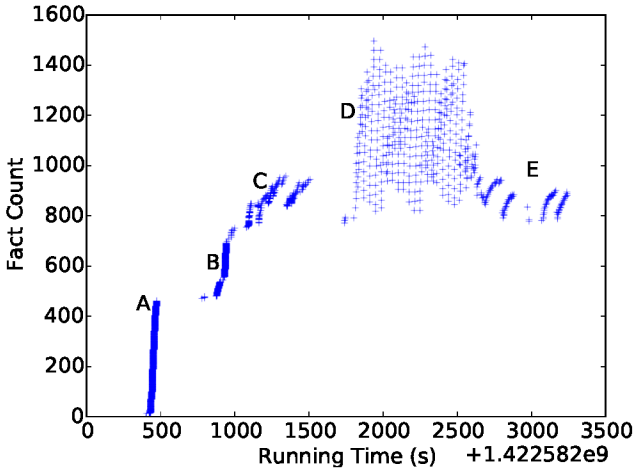


Fig. 2: Facts in the system as a function of run time.

Figure 2 shows the number of facts in the system over time, illustrating the behavior of the system as a whole. At point A we bootstrap the network, configuring interfaces and storing the interface information. At B, we continue setting up the network, this time installing and configuring OpenFlow. The gap between the fact insertions of A and B is due to the time taken to install and start Open vSwitch. At time C, we install and run our `snort` application, as well as our generic `ping` measurement app. Notice that at this time we start seeing delete operations—the `archive` app is instructed to delete old `snort` alerts rapidly in this configuration. At D, we launch the `sflow` app, which inserts significantly more facts, but the `archive` app is removing `sflow` facts at a higher rate to help keep the fact count under control. We also

have a few snort alert insertions throughout this period at D. Finally, at E, we disable the sflow app and turn up the ping measurement frequency. We see that the system stabilizes and returns to similar behavior as was exhibited at C. This shows the system reacts well to fluctuating network behaviors.

B. Functional Evaluation

We now consider case studies showing how applications in KnowNet are able to preserve the healthy functioning of the network under various conditions.

Link Tracking and Selective Rerouting: We evaluate the ability of our apps to selectively reroute traffic flows based on network policy in the face of soft errors. In this case, we consider elevated bit error rates on a link, an indication of possible impending failure. This is a particularly challenging case, because at the service level, the dropped packets can easily be mistaken for congestion: the correlation of information at multiple levels is required to understand the root cause. Here, our link app notices (emulated) bit errors on a link that it is monitoring, and connects that with the set of services using that link. One of the services is high priority, so the link app proactively requests a re-route of the service around the failing link. The graph in Figure 3a shows this selective re-routing in action—the high priority link is re-routed to preserve its performance and to avoid unavailability if the physical link does fail.

Historical Analysis for Handling a New Vulnerability: Figure 3b shows the evaluation for our vulnerability app. We insert a fact in the knowledge graph at the time marked by the dotted line, indicating a vulnerability in a particular service. There are two flows connected at this time, and we immediately block both. The vulnerability app then looks at historical sflow data in the knowledge graph for a common access pattern for the protected node—the assumption is that access patterns that have been common in the past are not likely to be attacks. The vulnerability app creates a query describing the pattern and subscribes to it, so that whenever traffic attempting to access the protected service meets this pattern, the vulnerability app can selectively un-block just these flows. In this case, only flow 1 matches historical access patterns, so it becomes unblocked once the analysis completes.

DOS Protection: In this example, we consider an ICMP flood originating inside the enterprise network due to a compromised host. snort is able to detect such attacks, and we can block the traffic based on its alert. However, two problems still exist: (1) the traffic still travels through the local network to get to the firewall, causing internal performance degradation, and (2) simply seeing the snort alert does not provide a high-level explanation for the attack. Using the combination of security-oriented apps we have written for KnowNet, we can overcome these problems, identifying a ping flood attack with more accuracy. These apps not only stop the attack at the source, but also stop the attack with fine granularity (only blocking ping traffic, allowing normal user traffic to continue) and explain the attack at a high level (a thousand snort alert facts will become a small handful of attack facts).

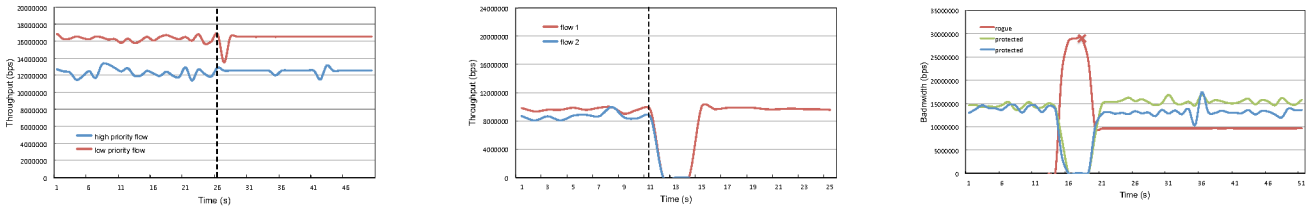
Fine-grained Flow Tracking: We run the flow tracking app in our network to monitor per-flow statistics and rate-limit rogue flows that are exceeding configured traffic thresholds. The flow app makes use of sflow analysis results periodically inserted in the knowledge graph to identify any heavy hitting flows belonging to unknown services. These flows are then put into appropriate QoS classes so that known service flows are protected. Figure 3c shows the apps coordinating to protect two flows of a high priority service. The flow starting in the middle of the graph immediately starts starving the other flows—it is given a few seconds to quiet down before the flow tracking app marks it for rate-limiting and requests the openflow app to install rate-limiters.

Intrusion Detection Event Discovery: To demonstrate the usefulness of our event discovery app, we designed a scenario in which a network administrator is concerned about a yet-to-be-discovered bug in server software running in the network which might enable an intruder to gain undetected access to hosts on the network. While it might be easy to detect a remote connection from an unrecognized source, confirming whether it was an intruder or not is difficult since authorized users are becoming increasingly mobile. It would therefore be useful to combine other suspicious activity dealing with the vulnerable host to a degree where it is clear that an authorized user is actually not the initiator of this suspicious session but rather a malicious one. The intruder in this case, already knowing about the vulnerability and how to exploit it, might use tools such as nmap to determine if any hosts in our network would be vulnerable to such an attack. We evaluated our event discovery app assuming such a scenario, i.e., an nmap scan followed by access from the same address signifies potential malicious activity. Our *generic* event discovery app was able to discover this relationship in our testbed by simply monitoring for facts inserted on all IP addresses in the network, recognizing a collection of facts describing the associated malicious activity as an event.

VI. Related Work

To our knowledge, the first use of a knowledge graph for network management was to store relationships between events, and tie those in with the network topology [21]. However the knowledge graph itself was only used to store relationships *after* they had been inferred; it was not used in the actual inference process.

Our work also relates to recent efforts in data center network management [14] and network policy [19]. Statesman [14] exposes a particular set of variables to its configuration apps; this makes it possible to maintain strong invariants as part of the framework itself, and supports loosely-coupled apps that may not have even been aware of each others' presence. In contrast, KnowNet offers a more flexible, extensible representation in its knowledge graph. This enables us to easily incorporate existing network management tools in KnowNet. PGA [19] uses a graph representation of network policies to allow



(a) Flow rerouting (b) Vulnerability control (c) Rate limiting

Fig. 3: (a) Selective rerouting of flows in case of soft failure. (b) Access to a protected node is blocked; legitimate access pattern restored. (c) Apps working together to rate limit unknown heavy hitters, protecting high priority service flows.

the reconciliation of conflicting policies. KnowNet follows a complimentary approach where we use a knowledge graph to capture network information and provide a framework through which applications can cooperate to manage the network.

Flowlog [16] is a language/run-time hybrid for network management that is based on Prolog. It allows its users to manipulate and create tables that store information about the network. The lowest-level tables are compiled directly into OpenFlow flow rules and installed on switches. These systems deal with the specific sub-task of network configuration management, similar to KnowNet's suite of basic network functionality apps. tIn contrast, KnowNet allows all network management tasks to be addressed.

Sophia [9] is a distributed "Information Plane" that collects information from a number of sensors (*read*), and allows the user to invoke a number of actuators (*write*). This closed-loop control realized in Sophia is similar to the same functionality in KnowNet. Unlike KnowNet's flexible approach which imposes very few restrictions on applications and how they interact, Sophia is based on a Prolog-like logic language where both sensors and actuators are expressed with logical primitives.

Similar to the query capabilities provided by KnowNet's knowledge graph presentation of data and information, network search [22] is inspired by the field of information retrieval and brings a search-based paradigm to network management. The network search work limits itself to a distinct "search plane" between the network and the operator.

Existing technologies such as RDF [23], OWL [24], or SWRL [25], were not considered at this stage in our work. We chose to explore the relationship based model at its simplest form without any extra cruft. As we better understand network knowledge, we will further explore integrating other descriptive and reasoning approaches.

VII. Conclusion

KnowNet provides a knowledge graph based framework for writing apps that cooperate to manage a network. The custom knowledge graph in KnowNet, Kilo, has primitives specifically tailored to the network management domain, including subscriptions, time-oriented data, and reference-counted deletion. These primitives proved useful for simplifying the job of a network operator to reason about network state. We have demonstrated that this platform provides a good base for developing a variety of rich network management applications. Our future work will include the application of other knowledge graph learning techniques to the network management domain.

References

[1] D. Clark *et al.*, "A knowledge plane for the internet," in *SIGCOMM '03*.

[2] K. Bollacker *et al.*, "Freebase: A collaboratively created graph database for structuring human knowledge," in *SIGMOD '08*, 2008.

[3] G. Kasneci *et al.*, "Naga: Searching and ranking knowledge," in *ICDE*. IEEE, 2008.

[4] F. Niu *et al.*, "Deepdive: Web-scale knowledge-base construction using statistical learning and inference." 2012.

[5] M. Chein and M.-L. Mugnier, *Graph-based Knowledge Representation: Computational Foundations of Conceptual Graphs*, 1st ed. Springer Publishing Company, Incorporated, 2010.

[6] J. Hoffart *et al.*, "Yago2: A spatially and temporally enhanced knowledge base from wikipedia," *Artif. Intell.*

[7] M. Gardner *et al.*, "Improving learning and inference in a large knowledge-base using latent syntactic cues," 2013.

[8] Freedom Dynamics Ltd, "Controlling Application Access - A network security and QoS checkpoint," http://www.freeformdynamics.com/fullarticle.asp?aid=1738, January 2014.

[9] M. Wawrzoniak *et al.*, "Sophia: An information plane for networked systems," *SIGCOMM Comput. Commun. Rev.*

[10] C. R. Kalmanek *et al.*, "Darkstar: Using exploratory data mining to raise the bar on network reliability and performance," in *DRCN*. IEEE, 2009.

[11] S. Kandula *et al.*, "Detailed diagnosis in enterprise networks," *SIGCOMM Comput. Commun. Rev.*

[12] P. Kanuparthy *et al.*, "Pythia: detection, localization, and diagnosis of performance problems," *Communications Magazine, IEEE*, vol. 51, no. 11, pp. 55–62, November 2013.

[13] H. Yan *et al.*, "G-rca: A generic root cause analysis platform for service quality management in large ip networks," in *Co-NEXT '10*, 2010.

[14] P. Sun *et al.*, "A network-state management service," in *SIGCOMM '14*.

[15] T. Qiu *et al.*, "Netsearch: Googling large-scale network management data," in *Networking Conference, 2014 IFIP*, June 2014, pp. 1–9.

[16] T. Nelson *et al.*, "Tierless programming and reasoning for software-defined networks," in *NSDI'14*, 2014.

[17] H. Kim *et al.*, "Lithium: Event-driven network control," 2012.

[18] N. Handigol *et al.*, "I know what your packet did last hop: Using packet histories to troubleshoot networks," in *NSDI'14*, 2014.

[19] C. Prakash *et al.*, "Pga: Using graphs to express and automatically reconcile network policies," in *SIGCOMM '15*, 2015.

[20] A. Das Sarma *et al.*, "Dynamic relationship and event discovery," in *WSDM*, 2011.

[21] A. Turner *et al.*, "Automatic discovery of relationships across multiple network layers," in *INM '07*, 2007.

[22] M. Uddin *et al.*, "Management by network search," in *NOMS 2012*.

[23] D. Brickley and R. Guha, "Rdf vocabulary description language 1.0: Rdf schema," W3C, Tech. Rep., feb 2004, http://www.w3.org/TR/2004/REC-rdf-schema-20040210/.

[24] S. Bechhofer *et al.*, "Owl web ontology language reference," W3C, Tech. Rep., feb 2004, http://www.w3.org/TR/2004/REC-owl-ref-20040210/.

[25] I. Horrocks *et al.*, "Swrl: A semantic web rule language combining owl and ruleml," *W3C Member submission*, vol. 21, p. 79, 2004.