Towards Semantic Network Models via Graph Databases for SDN Applications

Talita de Paula Cypriano de Souza and Christian Esteve Rothenberg
University of Campinas (UNICAMP)
{cypriano,chesteve}@dca.fee.unicamp.br

Mateus Augusto Silva Santos
Ericsson Research
Indaiatuba, SP, Brazil
mateus.santos@ericsson.com

Luciano Bernardes de Paula
Federal Institute of Sao Paulo (IFSP)
Braganca Paulista, SP, Brazil
lbernardes@ifsp.edu.br

***Abstract*—At the core of any network control and management system is the representation and maintenance of network topology information. Software-Defined Networking (SDN) treats topology abstractions as one of the cornerstones towards rethinking network architectures and the way they are operated. Recently, motivated by the scalability and performance needs of cloud applications, Graph Databases are being adopted as appealing alternatives to traditional relational models when data is highly interconnected and extensible schemas are called for. In addition, the use of metadata to describe how data is interconnected by means of Web Semantic standards is increasingly gaining ground. At the crossroads of these trends, this paper presents an approach to augment SDN network state with a semantic model leveraging graph database technologies. In particular, our proposal imports the Network Markup Language (NML) model into a scalable graph database (Neo4j). For validation purposes, we evaluate our proof of concept implementation against a representative set of SDN application primitives.**

I. Introduction

Software-Defined Networking (SDN) [1] has entered the networking scene as an innovative approach to build and operate networks by introducing new abstractions and programmatic interfaces to the control and forwarding planes.

So far, much of the efforts have been devoted to the packet flow abstraction and its implementation via open protocol between controllers and forwarding devices (e.g. OpenFlow [2]). Despite recent efforts at standards development organizations, network topology is another abstraction of SDN that has arguably received less attention. Regardless of the networking paradigm (e.g. traditional vs. SDN) or the control plane method (e.g. centralized vs. distributed) graphs and topologies are –and will continue to be– a main pillar of any networking approach. Graphs, in which network nodes are represented by vertices and their connectivity (logical or physical) by edges, and data structures for their implementation become fundamental aspects of control plane applications providing logical functions such as minimum spanning tree, shortest paths, recovery paths, and so on.

Recently, Semantic Web standards have met the field of networking (e.g. NML - Network Markup Language [3]) to describe multi-layer, inter-domain networks in a semantically meaningful way to benefit from a comprehensive, technology-agnostic view along the companion software tools to leverage a rich network model.

Our work aims at combining these recent networking advancements with modern graph databases in order to augment SDN with semantic network models providing rich network abstractions efficiently implemented through a scalable graph database technology (Neo4j). We first transform the semantic network model (NML) into a graph that can be imported into the Neo4j graph database, which in turn becomes the active network state to be maintained by the SDN controller and its applications. Having a semantically rich network information base (cf. NIB as per Onix [4]) in a database that natively provides graph-oriented primitives facilitates the development of SDN control applications that can also use the common semantic model to interoperate and share state through the graph database repository. Adding semantic annotation modeling to SDN opens opportunities to leverage related software engineering work, including logical reasoning and formal verification techniques [5].

As a first step towards validating the proposed technological confluence in practice, we implemented a proof of concept and carried a series of experimental evaluation works. First, we investigated from a functional perspective the ability to support SDN application primitives from the literature (NetGraph [6]) looking for any limitations in the semantic model and adequate queries/APIs in the chosen graph database. Second, to get some sense on the resulting performance, we run experiments with varying network topologies (up to 10,000 nodes) and analyzed the observed primitive execution times. Summing up, the contributions presented in this paper are:

1) Providing semantic modeling language (NML) support in a graph database (Neo4j);
2) Mapping SDN application primitives [6] to graph database queries;
3) Identifying limitations of NML for SDN applications;
4) Prototype implementation and performance evaluation – with datasets shared for code re-use and reproducibility.

This paper is organized as follows. Section 2 presents background and related work. The proposed architecture as well as the adopted primitives are presented in Section 3. In Section 4, we describe the experimental scenario and result analyses. Finally, in Section 5, we present final remarks and avenues for future work.

II. Background and Related Work

A. Network Markup Language - NML

Web Semantic standards have experienced recent advancements including OWL (Web Ontology Language) and RDF (Resource Description Language) providing appealing advantages in terms of metadata when compared to XML schema or UML. RDF/OWL metadata introduces semantically meaningful triples divided by subject, predicate and object, in which an object of a triple may be a subject of another, so that metadata can be recursively related.

One recent example of a markup language based on RDF/OWL to describe computer networks is NML (Network Markup Language) [3] – an outgrowth of NDL (Network Description Language), which was created around the year 2000. NML allows describing network elements and their interconnection, including multi-layer and multi-domain models as well as virtualized environments. NML supports modeling network topologies along their capabilities in terms of services (e.g., circuit/packet switching) and the configuration parameters.

The NML model [5], [7] start by an anchor class *Network Object* from which further abstractions inherit, including core network entities such as `Node`, `Port`, and `Link`. The configurable part of a `Network Object` can be modeled as a `Service`, e.g., the forwarding service between device ports or the adaptation service to an IP packet into a Ethernet frame.

One key feature of NML is its extensibility, allowing the model to evolve and adapt as required by the application domain. For example, CineGrid [8] defines CDL (CineGrid Description Language) [9] by inheriting NML classes and defining a set of extensions to describe their distributed digital cinema testbed infrastructure. The NOVI [10] project uses a generic ontology derived from NML referred to as INDL (Infrastructure and Network Description Language) in order to ease the federation of platforms for future Internet research (e.g., requests and monitoring services leverage different resources described from a common semantic model). Another recent EU research project extends the INDL/NML model is GEYSERS [11], which added a number of extensions to support virtualized optical infrastructures. All these examples of NML extensions and application domains confirm the versatility of the modeling language along the main characteristics and benefits that motivated our work on NML meeting SDN. To this end, our first step is modeling simple network topologies and exploring its embodiment in modern graph database technologies,

B. Graph Databases - GDB

In database systems in which graph/topological information and data connectivity are important, traditional relational databases present limitations that have lead to the development of so-called NoSQL (Not Only SQL) alternatives. Some of the well-known advantages include scalability and flexible schema [12]. In particular, graph databases (GDB) stores data as graphs composed by vertices and edges that represent their relations [13]. This graph-oriented approach allows a natural modeling of several types of scenarios like Semantic Web, recommendation engines, and computer networks, among others [14]. Robinson et al. [13] highlight two characteristics of GDB models, namely: native graph storage and native graph processing. Usually, property graphs are used as a graph model where directional edges are augmented with labels and attributes [14], and nodes have properties (key-values pairs) along oriented relations.

A recent comparison among GDB implementations [14] points to Neo4j as an attractive technology due to its native support of property graphs, ACID transactions, high availability, and high speed query processing.

Whereas traditional relational databases retrieve highly interconnected data through complex *join* operations, GDBs naturally solve this problem through native graph traversal operations at high speed. Basically, queries travel the graph through nodes and edges. There are a few available query languages for Neo4j with varying properties [15]. In our work we opt for the Cypher language, which is similar to SQL and was designed to be developer-friendly. Cypher queries can be written to retrieve GDB data matches some defined patterns, allowing different approaches to implement SDN application primitives. For example, consider a procedure to count all connections of node A:

```
1. MATCH (n:Node)-[:hasOutboundPort]->(p:Port)-
[:isSource]->(l:Link)
2. WHERE n.name="A"
3. RETURN COUNT(l) AS CountOutDegree
```

In the above-presented query, *Cypher* travels through the graph nodes and relations searching data matching the defined `node-outport-link` pattern (around node A), retrieving the amount of `Links` found.

One example of recent related work [16] used graph database with Cypher for auditing tasks in cloud computing to perform (1) risk analysis (determining affected VMs after network outages), (2) simple report (verifying VM storage systems), and (3) inventory (comparing different hierarchies equivalence), among others.

C. Graphs & SDN Control Applications

The Onix controller [4] can be considered the seminal work in applying graphs to (distributed) SDN control applications. Developed by pioneers of OpenFlow, Onix handles SDN control as a distributed system problem, leveraging well-known tools and techniques, including two types of data stores (SQL vs. DHT) to hold the logically centralized Network Information Base (NIB) according to the state consistency requirements. Onix uses graphs to aggregate low level information and share the state among multiple controllers, altogether contributing to the NIB centralized network view which simplifies and abstracts physical infrastructure details, easing the concerns of application developers.

One recent open-source SDN controller that follows design principles from Onix is ONOS (Open Network Operation System) [17]. ONOS developed a first prototype using a distributed graph database (Titan) to store the network state, and have since moved to a second prototype using a simplified model with optimized data structures and in-memory data grid for performance reasons.

Graphs applied to SDNs are the central topic considered by Pantuza *et al.* [18], where graph-oriented controller modules obtain and use network topology information. The experimental work investigates the support of dynamic network representation, e.g., maintaining minimum spanning trees in real-time over the network graph.

The aforementioned paper is similar to NetGraph [6], a library that supports dynamic updates of network state and offers data from queries that may be used by the SDN controller. NetGraph anticipates the calculation of some operations to optimize the time of queries. For example, shortest paths between nodes are (partially) calculated in advance so that they can be used by routing algorithms.

Related work so far neither explore semantic annotations to model a network graph in the context of SDN nor approaches to load rich network models into persistent GDBs. Next, we present the proposed framework to achieve these goals.

III. Semantically Augmented SDN Graphs

With the main goal of allowing SDN applications to support semantic models of SDN infrastructures we propose to embody the NML-modelled network state into a scalable and high performance graph database (GDB). The database should be easily integrated with the SDN controller of choice, providing primitives to control applications via northbound interfaces (e.g. REST) and maintaining SDN network state via southbound protocols (e.g. OpenFlow, NETCONF). The reference architecture used in this paper is presented in Figure 1. Applications external or internal to the SDN controller perform queries to the GDB after importing the NML model and the related SDN state. Applications use GDB interfaces to read and write the network state in a centralized way instead of performing individual queries to devices.

Primitives exposed by the GDB include any relations between the NML objects such as shortest path between nodes, connectivity degrees, node neighbors and their properties, among others. Since NML is based on Semantic Web standards, it is flexible and extensible. Native support of NML (and application-domain extensions) in the GDB allows defining SDN primitives by exploring the rich semantic model and combining existing ones (e.g. retrieve all multi-layer paths between virtual machines in DPDK-enabled x86 hosts with 10 G interfaces). Having a semantic network-wide view along the supporting software standard interfaces enables control/management SDN applications to take decisions based on the operational state of the network while simplifying not only the initial application logic development but also the maintenance and any technology-specific modification upfront.

As already anticipated, we opted for Neo4j as the graph database back-end due to its features and proven performance when compared to alternative GDBs [14].

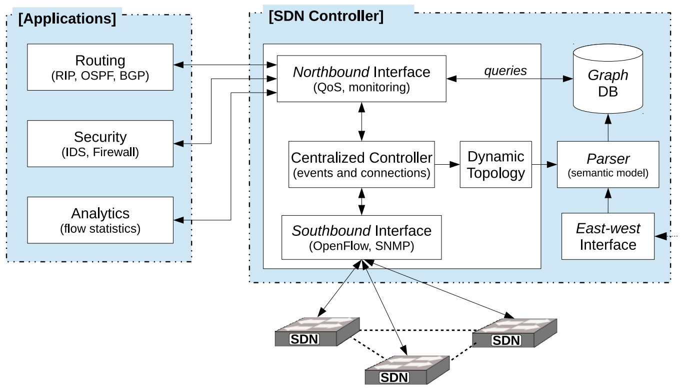

Fig. 1. Reference SDN architecture featuring a graph database supporting a semantic model based on NML (Network Markup Language).

A. Mapping SDN Primitives to Graph Queries

When adding NML model support to SDN applications, two related GDB problems need to be solved: (1) parsing NML to be imported into Neo4j property graphs, and (2) providing data access to SDN applications via adequate APIs.

Out of scope of this paper remain implementation details to feed the GDB with the required data, such as using LLDP (Link Layer Discovery Protocol) or similar methods by SDN controller implementations (OpenDaylight) to discover the topology or to define the edge weights based on observed latencies or available bandwidth/link capacities. Any southbound protocol could be used to obtain such metrics (e.g. OpenFlow, SNMP, sFlow) and additional ones to be inserted into the GDB.

We focus our work on supporting 12 SDN primitives described in the NetGraph library [6]. Basically, two main functional features are provided: (1) query the network topology including nodes and link state to keep the graph updated and (2) compute graph queries and return the results in a format allowing to be used by other modules. In NetGraph, the use case application is network virtualization support to offer Network as a Service (NaaS).

We analyze the NetGraph primitives presented in Table I regarding (*i*) the ability to be supported by the semantic model (NML) without modifications or extensions, and (ii) mapping possibilities to GDB (Neo4j) queries. We consider that a primitive is supported by the semantic model if it can be implemented using attributes and relations of the graph modeled using NML. Similarly, we consider a primitive to be supported by the GDB if it can be mapped to one or more Cypher queries. Next, we discuss each group of primitives regarding their NML and GDB implementation.

• The `setEdgeWeight` and `getEdgeWeight` primitives, assign a cost to the link between two `Nodes` and obtain such cost, respectively. These primitives are not supported by the semantic model because the NML schema lacks link attributes. Hence, the model needs to be extended for instance by adding a cost attribute to the `Link` class. Regarding the GDB, both primitives are supported because the Neo4j property graph is based on attributes for nodes and relations.

• The primitives `countInDegree`, `countOutDegree` and `countNeighbors` return the amount of input and output links of a node, and the number of neighbors, respectively. They are naturally supported through the relations by both the semantic model and GDB.

• Primitives based on shortest path calculations are naturally supported by the semantic model through its graph representation and the GDB instance. Similarly, the `doesRouteExist` primitive that verifies the existence of a route between any two nodes is supported by the semantic model and by the GDB. The `computeMST` primitive, which generates a Minimum Spanning Tree from an origin (or root), and the `computeSSSP` (Single Source Shortest Path), which returns the shortest path from a single node, use the same *shortestPath* resource from Cypher, also used to find the shortest path between all pair of nodes in the `computeAPSP` (All Pair Shortest Path) primitive. The `allShortestPath` primitive from Cypher can be used to compute `computeKSSSP` (k Single Source Shortest Path), i.e., returning k shortest paths between two any two nodes.

• The `insert` and `delete` primitives are supported by both the GDB and semantic model. Insertion of a topology `Node` in the GDB triggers the insertion of two `Ports` and their relations. In the same way, the node deletion operation removes the `Ports` and `Links` along all relations to the `Node`.

Primitive	Semantic Model	GDB	Read / Write
`setEdgeWeight`	No	Yes	W
`getEdgeWeight`	No	Yes	R
`countInDegree`	Yes	Yes	R
`countOutDegree`	Yes	Yes	R
`countNeighbors`	Yes	Yes	R
`doesRouteExist`	Yes	Yes	R
`computeMST`	Yes	Yes	R
`computeSSSP`	Yes	Yes	R
`computeKSSSP`	Yes	Yes	R
`computeAPSP`	Yes	Yes	R
`delete`	Yes	Yes	W
`insert`	Yes	Yes	W

TABLE I
ANALYSIS OF SDN PRIMITIVES FROM THE NETGRAPH LIBRARY [6].

IV. EVALUATION AND RESULTS

We now present the experimental methodology to evaluate our proof of concept implementation in terms of performance profiling and strawman comparison with a relational DB.

A. Experimental environment

All experiments were executed in a machine with an Intel i7 processor, 2.4 GHz and 8 Gigabytes of RAM memory. We used the Neo4j Community Edition 2.0.4 under the GPLv3 license. In order to create the topologies, instead of using modified versions of the Neo4j Java APIs for database manipulation methods, we opted to comply with the NML schema by means of Cypher language queries, which do not require such modifications, as explained in the primitive analyses of Section III-A.

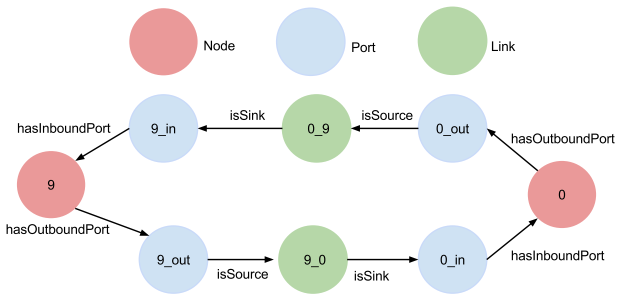

Fig. 2. The relation between nodes 9 and 0 as displayed by Neo4j.

B. Topologies

In order to evaluate the proof of concept implementation, four different topologies were created using the BRITE [19] generator using the *Flat Router-Level* and the *RouterBarabasiAlbert* models. All nodes are generated from the same seed and after topology creation the link attributes are uniformly defined by BRITE [19]. For each node, the generator defines seven attributes, however, for the scope of our experiments we only use the unique *id* attribute as *NodeId* mapped to the *name* property in the NML schema.

In order to assess the performance and scalability properties of the GDB, we investigated the following topology sizes: 10 (*tiny*), 100 (*small*), 1,000 (*medium*) and 10,000 (*large*). For each one, the nodes were imported into the GDB following the proposed model mapping in Section IV-C resulting the following semantically augmented graphs: 76 nodes (160 relations), 640 nodes (1,760 relations), 4,978 nodes (11,912 relations) and 109,932 nodes (359,728 relations). Note the increase of the resulting semantic graphs (nodes and relations), since every network topology vertex expands into a set of inter-related `Node`, `Port`, `Link` entities in the GDB.

C. Data modeling

In order to index nodes and relations generated by BRITE in GDB, following NML schema, for each `Node` a pair of `Ports` (defined by NML), one for input traffic and other for output. These ports were identified, respectively, by id_{in} e id_{out}. For each connection between both nodes, two `Links` were created (one for each direction) which relate ports with nodes. Figure 2 presents the modeling for the connectivity between nodes 9 and 0, their `Ports`, `Links` and *relations*.

D. Results

In order to measure the query execution time, a test application was developed that first inserts the NML-parsed topology into Neo4j and then executes, individually and sequentially, each SDN primitive as Cypher queries using a random test set. Every primitive was executed 1,000 times for every topology.

Table II presents the average, standard deviation and 99 percentile for each primitive for the *large* topology, which corresponds to our worst case scenario.

The primitives `countInDegree`, `countOutDegree` and `delete` have the highest latency values for all topologies.

When counting input and output connections of a node, long execution time can be explained due to the number of semantic graph hops (relations) that need to be traversed, a fact also noted in [16]. For example, to compute the number of input connections of Node A, the following pattern must be covered:

$$NodeA \leftarrow hasInboundPort \leftarrow Port \leftarrow isSink \leftarrow Link$$

The behavior applies to the `delete` primitive, since the query deletes the `Node`, including their `Ports` and associated `Links`. In this case, the number of processed relations (hops) is even larger compared to counting input and output links. We observe that the primitive latency is proportional to the degree of the deleted node.

As expected and previously identified by [14], Neo4j presents better performance in read-only operations than read-write tasks - a behavior that can be clearly observed in the `getEdgeWeight` primitives.

Figure 3 compares three routing oriented read-only primitives `computeSSSP`, `computeKSSSP` and `doesRouteExist` and one write primitive `insert` for varying topology sizes. In general, the shortest paths primitives present good individual performance and scaling properties (sub-logarithmic regarding amount of nodes/relations). Interestingly, primitives that calculate all pairs of shortest paths (`computeAPSP`) run faster when compared to the k shortest paths between two nodes (`computeKSSSP`) and the shortest paths from a specific node (`computeSSSP`). This can be explained by the fact that, in the case of queries for all pairs of shortest paths, Neo4j is capable of storing partial path calculations between intermediate nodes, an optimization that cannot be exploited when considering specific nodes (e.g. `doesRouteExist`). According to the observed results, the `computeMST` and `computeSSSP` primitives seem to share their internal implementation in Neo4j.

Primitive	Avg Value	Std Dev	99th
`setEdgeWeight`	162.33	9.46	205.01
`getEdgeWeight`	1.70	0.74	4.00
`countInDegree`	854.53	146.77	1399.05
`countOutDegree`	425.17	68.36	699.02
`countNeighbors`	4.45	2.27	10.01
`doesRouteExist`	37.51	29.09	73.06
`computeMST`	1.44	1.25	3.02
`computeSSSP`	5.47	4.98	29.00
`computeKSSSP`	26.21	37.23	81.04
`computeAPSP`	1.04	0.68	3.01
`delete`	1053.89	162.55	1637.02
`insert`	3.57	3.21	16.01

TABLE II
TIME (MS) OF PRIMITIVES EXECUTION - TOPOLOGY *large*

E. Comparison to a Relational Database Approach

In order to get a sense on the performance results with a traditional RDBM, we run a set of similar SDN primitives using a Java application with MySQL (version 5.6.17) [20] and the SQL query language. Figure 4 shows the Enhanced Entity Relationship Model diagram (EER). The model follows the data relations presented in Figure 2 representing the graph

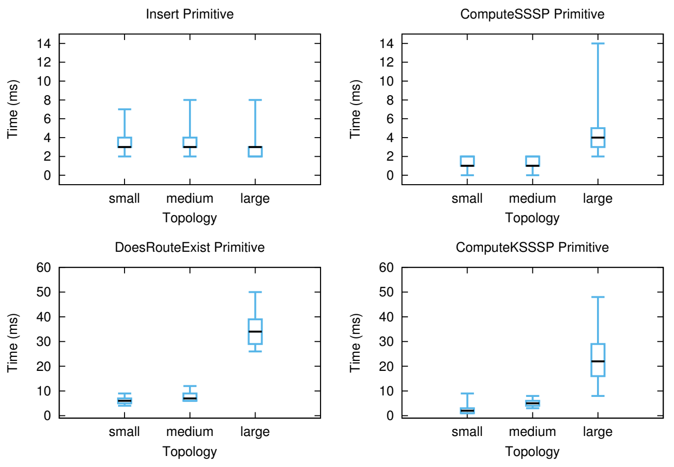

Fig. 3. Primitive response time presented as *candlesticks* with average, quartiles, and max/min values as the 95-/5-percentiles.

as tables using primary and foreign keys. Thus, there are (one-to-many) relationships between `Node` and `Port`, whereas a `Link` is a recursive (many-to-many) relationship between two Ports. During the test application design, one practical advantage of GDB became clear, namely, modeling tasks are considerably more natural compared to RDBM –as expected from such a network-centric data interconnection project.

Table III presents the measured execution times (for 1,000 runs) of a subset of primitives in the case of the *large* topology. Results for the `delete` and `countInDegree` primitives are faster with MySQL than with Neo4j, a fact that can be expected from the accumulative latency (number of hops) explained in the previous subsection. Interestingly, in the case of data insertion, the execution time of the `insert` primitive was much higher than in Neo4j, probably due to the need of inserting data in two tables (Node and Port). For the `computeSSSP` primitive, results were slightly worse than with the Neo4j. Noteworthy is the fact that in order to compute the shortest path primitives a Java implementation (based on Dijkstra [21]) was necessary due to the lack of built-in shortest path type of queries in MySQL. Firstly, our Dijkstra's algorithm implementation loads in memory all vertices and their adjacencies using one `SELECT` query. The average time of this pre-execution data fetching task is almost 2 seconds. Again, Neo4j native graph-oriented primitives can be seen as an advantage of GDB over RDBM from an application development perspective. Likewise, the `computeAPSP` primitive exhibits comparable performance but does not require any extra development effort in a GDB approach due to the native Neo4j function to compute all pairs of shortest paths.

F. Data Availability & Experiment Reproducibility

All data and source code used in this work are available in a public repository.[1] The available datasets include the NML

[1] https://github.com/intrig-unicamp/NML-Neo4j

Primitive	Avg Value	Std Dev	99th
`countInDegree`	1.39	4.57	22.02
`computeSSSP`	18.13	3.82	26.00
`computeAPSP`	2.11	1.39	7.00
`delete`	162.86	79.93	405.00
`insert`	137.36	43.80	300.00

TABLE III
EXECUTION TIME (MS) IN THE RDBM FOR THE *large* TOPOLOGY.

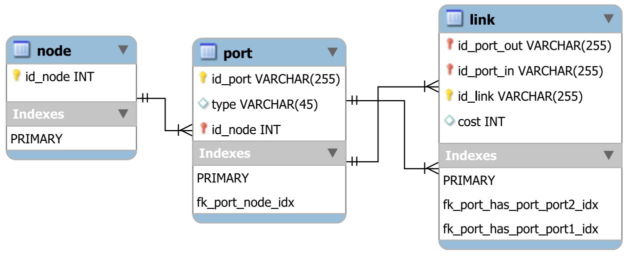

Fig. 4. Enhanced Entity Relationship Model (EER Model)

models, BRITE topologies, parsing scripts, Cypher queries for Neo4j, MySQL application codes, and the gnuplot snippets.

V. CONCLUSION AND FUTURE WORK

We presented a proposal to augment SDN applications with a semantic model (NML) that can be consumed through a graph database (Neo4j). We introduced a framework to embody the model in a GDB as an annotated relational graph, allowing SDN controllers to offer developer-friendly primitives that simplify the design of SDN control applications.

We found NML to be a compelling modeling language for the needs of SDN despite some extensions (e.g. link costs) needed for some routing primitives in addition to further SDN application domain classes to be added. The property graph used in Neo4j proved to be amenable to support semantic network models based on *entities*, *attributes* and *relations* of NML without requiring any re-work. The Cypher query language showed to be a flexible approach to implement the SDN primitives through its native means to search for graph patterns and write/read data objects. We were able to reproduce a number of SDN application primitives and the performance results obtained are promising regarding their potential to be used in real deployments. We observed appealing advantages of native graph-oriented data stores versus traditional relational database models that go beyond performance metrics.

As for future work, we are working on a number of different issues: (*i*) evaluate the performance in dynamic workloads with simultaneous R/W operations and OpenDaylight applications using REST APIs; (*ii*) developing new inter-domain SDN primitives as a means of East-West controller interfaces, allowing multiple controllers to share data about their topologies; (*iii*) explore optimizations in latency and system capabilities via pre-calculation and leveraging properties to enable/disable nodes in the graph and adding time-series capabilities; (*iv*) developing extensions to the semantic model (NML) to add support of further SDN primitives and entities in addition to Network Function Virtualization (NFV) semantic services and YANG data models, altogether contributing to the development of as-a-service consumption of SDN & NFV services.

VI. ACKNOWLEDGMENTS

This work was supported by the Innovation Center, Ericsson, Brazil. At the time this paper was written, Mateus Augusto Silva Santos was with University of Campinas.

REFERENCES

[1] D. Kreutz, F. Ramos, P. Esteves Verissimo, C. Esteve Rothenberg, S. Azodolmolky, and S. Uhlig, “Software-defined networking: A comprehensive survey,” *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, Jan 2015.
[2] N. McKeown et al., “Openflow: enabling innovation in campus networks,” *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.
[3] J. van der Ham, F. Dijkstra, R. Lapacz, and A. Brown, “The Network Markup Language (NML) A Standardized Network Topology Abstraction for Inter-domain and Cross-layer Network Applications,” *TNC2013*, 2013.
[4] T. K. et al., “Onix: A distributed control platform for large-scale production networks.” in *OSDI*, vol. 10, 2010, pp. 1–6.
[5] M. Aertsen, “Verifying functional requirements in multi-layer networks: a case for formal description of computer networks,” March 2014. [Online]. Available: http://essay.utwente.nl/64707/
[6] R. Raghavendra, J. Lobo, and K.-W. Lee, “Dynamic graph query primitives for sdn-based cloudnetwork management,” in *Proceedings of Hot Topics in Software Defined Networks*, ser. HotSDN ’12, 2012, pp. 97–102.
[7] J. van der Ham, F. Dijkstra, R. Lapacz, and A. Brown, “Network Markup Language Base Schema version 1,” *Technical Report - Open Grid Forum*, 2013.
[8] P. Grosso, L. Herr, N. Ohta, P. Hearty, and C. de Laat, “Cinegrid: Super high definition media over optical networks,” *Future Gener. Comput. Syst.*, vol. 27, no. 7, pp. 881–885, Jul. 2011.
[9] J. J. van der Ham, S. J. P. Chrysa, M. Peter, K. Yiannos, P. Grosso, and L. Lymberopoulos, “Challenges of an information model for federating virtualized infrastructures,” *5th International DMTF Academic Alliance Workshop on Systems and Virtualization Management: Standards and the Cloud*, 2011.
[10] NOVI, *Networking innovations Over Virtualized Infrastructures*, 2010.
[11] Escalona et al., “GEYSERS: A novel architecture for virtualization and co-provisioning of dynamic optical networks and IT services,” in *2011 Future Network & Mobile Summit, Warsaw, Poland, June 15-17, 2011*, 2011, pp. 1–8.
[12] NOSQL. (2015) http://nosql-database.org/.
[13] I. Robinson, J. Webber, and E. Eifrem, *Graph Databases*. O’Reilly Media, Inc., 2013.
[14] S. Jouili and V. Vansteenberghe, “An empirical comparison of graph databases,” in *Social Computing (SocialCom), 2013 International Conference on*, Sept 2013, pp. 708–715.
[15] F. Holzschuher and R. Peinl, “Performance of graph query languages: Comparison of cypher, gremlin and native access in neo4j,” in *Proceedings of the Joint EDBT-ICDT 2013 Workshops*, ser. EDBT ’13. New York, NY, USA: ACM, 2013, pp. 195–204.
[16] V. Soundararajan and S. Kakaraddi, “Applying graph databases to cloud management: An exploration,” in *Cloud Engineering (IC2E)*, March 2014, pp. 544–549.
[17] P. Berde and et al., “Onos: Towards an open, distributed sdn os,” in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN ’14. New York, NY, USA: ACM, 2014, pp. 1–6.
[18] G. Pantuza, F. Sampaio, L. F. Vieira, D. Guedes, and M. A. Vieira, “Network management through graphs in software defined networks,” in *Network and Service Management (CNSM), 2014 10th International Conference on*. IEEE, 2014, pp. 400–405.
[19] A. Medina, A. Lakhina, I. Matta, and J. Byers, “Brite: An approach to universal topology generation,” in *Proceedings of MASCOTS 01*. Washington, DC, USA: IEEE Computer Society, 2001, p. 346.
[20] MySQL. (2015) http://www.mysql.com/.
[21] Literateprograms. (2015) Dijkstra’s algorithm (java). [Online]. Available: http://en.literateprograms.org/Dijkstra’s_algorithm_(Java)/