Network Management through Graphs in Software Defined Networks

Gustavo Pantuza, Frederico Sampaio, Luiz F. M. Vieira, Dorgival Guedes, Marcos A. M. Vieira
Departament of Computer Science
Universidade Federal de Minas Gerais
Belo Horizonte, MG – Brazil
Email: {pantuza,fredmbs,lfvieira,dorgival,mmvieira}@dcc.ufmg.br

Abstract— **Software Defined Networks (SDN) is an emergent architecture that is dynamic, flexible, manageable, low cost, consistent with the dynamics of the modern applications. This paper shows a network representation model using a graph as the control plane of an SDN controller. The graph approach provides a globally consistent view of the network in real time. Our experiments show that graphs are a reliable representation of the real network, simplifying management in Software Defined Networking.**

Key Words: SDN, Graph, Network Management, OpenFlow.

I. INTRODUCTION

Software environments designed to provide SDN applications are named SDN controllers. They have also been called network operating systems because they provide a layer that isolates and control the access to the physical network elements, providing a standardized interface to them. That interface may take different forms, like events in NOX [1], an internal database in Onix [2], a predicate-rule based reasoning system [3], or a query language in Frenetic [4], for example.

No matter the API, almost all applications of Software Defined Networks (SDN) need a topological view of the network; in fact, that global view is one of the key aspects of that paradigm [5]. Graphs model the network topology in a direct, natural and precise way, so describing the network using a graph has become a common practice in many works, protocols and network software, including those on SDN [6]. In this sense, a graph should be a basic resource of an SDN controller, providing the network representation, the access and the control of the network elements in a single structure. Thus, a graph module can be of multiple uses in this kind of software.

A graph model would be useful for both the internal modules of the SDN controller as for the applications using the controller. They need topological information or just access to the network data, and the search for and access to the network entities can be provided by the graph module. It may have the capacity to notify state changes, be it through events, callbacks or other notification mechanisms.

Moreover, in practice, many SDN applications use graph algorithms to obtain informations that affect the control of the underlying network, such as Shortest Path, Minimum Spanning Tree, Graph Coloring, etc. The graph module store this kind of representation directly and execute any of those algorithms internally, making the results available for other software modules, with no need for them to repeat the computation, and for the application developer to re-implement complex data structures and algorithms. A graph module can be implemented for that goal, avoiding new dependencies among different modules.

The interactions with the graph module can be encapsulated and defined as a semantically stable and standardized interface. Its implementation can be modified to adjust to different systems, using different resources like local memory, remote databases, centralized or distributed processing, concurrency control, parallelism, persistence, performance and other relevant characteristics.

This paper presents a description of a new kind of abstraction for network management, integrating elements like automated fault detection and provision for dynamic graphs, a real implementation on a system using OpenFlow [7], and its experimental validation.

The remainder of this paper provides, first, a description of the POX controller and the design, properties and project decisions that lead to the graph model implementation. After that we show some experiments and results obtained in a network simulation environment, in this case, Mininet [8]. Finally, we discuss some paths for future work and a brief conclusion.

II. OUR PROPOSED SOLUTION

We adopted POX, a Python-based controller, as the basic controller for our work [9]. It exports an event-oriented interface, where the major events are the discovery of switches and the reception of packets send by them to the controller. A module built on top of POX is a producer/consumer of events. It can register any event it creates with the *core*, as well as subscribe to events registered by other modules. To be able to handle different SDN protocols, POX encapsulates all the details of the OpenFlow infrastructure inside an OpenFlow class hierarchy. On top of these basic events, POX constructs a set of infrastructure modules that interact a publish/subscribe interface controlled by a *core* element.

A. POX original modules

To implement the graph abstraction, we extended some of the modules already available in POX to get the functionality we wanted. We first describe the modules already available and their functionalities as provided by POX. In the following sections we discuss how they were integrated and how we created the graph abstraction, with its associated interface.

The major module for our purposes is `Topology`. In the original POX implementation, this module is responsible for maintaining the Network Object Model (NOM), a dictionary of objects representing each *entity* (switch, router, host or other devices) in the network. It exports two methods, *addEntity* and *removeEntity*, which can be called by other modules to register or delete entities that they are responsible for, and publishes two events, *entityJoin* and *entityLeave*, that may be used by other modules that want to be notified of changes in the topology of the network.

As mentioned previously, POX encapsulates all interaction with OpenFlow elements in the OpenFlow class hierarchy (`of`). The two major elements for our discussion are `of.discovery` and `of.topology`. The first one implements the Link Layer Discovery Protocol (LLDP), used to identify devices in the network [10]. OpenFlow switches should be able to identify other switches connected to them using this protocol. The second module is responsible for interfacing with Topology to store the identified entities (switches) in it.

Just as *of.discovery* identifies switches, the *host_tracker* module does the same for hosts. It does so by using a multitude of techniques to identify and keep track of end hosts. The major element in its operation is an ARP interceptor: all switches are programmed to send *ARPQuery* packets to the controller. If the query refers to a host already known by the module, an *ARPReply* message is built and sent back to the host that initiated the query. At the same time, the location of that host is recorded for future use. Periodically, *host_tracker* issues fake *ARPQuery* messages to every known host to verify if they are still up. It may also access every switches counters to identify activity on each link and infer host presence without issuing ARP messages.

Recently, POX received a DCHP module (*misc.dhcpd*) that can be used to configure hosts in the network. As available in the distribution, it has no interaction with the topology system. However, the fact that it can be used to configure end hosts provided an interesting opportunity in this work.

For our work, the modules just described, already present in POX, had to be extended to build the complete graph abstraction to represent a network. Those extensions are described next.

B. Changes needed for the new architecture

Based on the functionalities of the existing modules, we extended POX to build the graph abstraction. For that, the modules were extended to interact more closely with each other. Figure 1 shows the final architecture.

The *entity* class was extended through inheritance to hold all the information about each network element, associated with the unique identifier already maintained by that module. The modules that discover and track network entities,

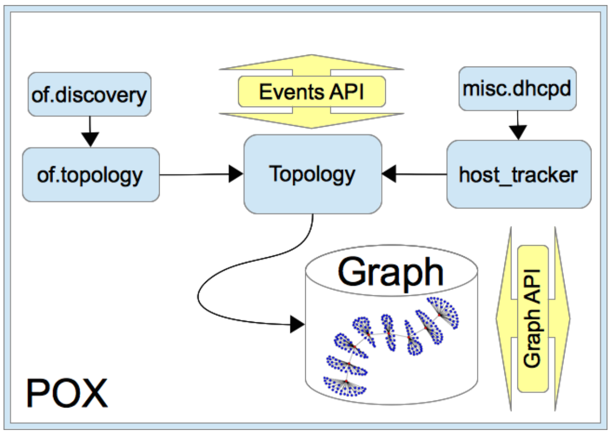

Figure 1. Module integration

host_tracker, *of.discovery/of.topology*, and *misc.dhcpd*, create entities through *topology*, which trigger events. Those events are subscribed by the *graph* module that creates, updates, deletes, executes algorithms and retrieves data from the network graph.

Topology was extended to keep informations about links as entities, and to associate events with them (like link up/down); *of.discovery* keeps track of links identified using LLDP, and *of.topology* was altered to register the links themselves with *Topology*.

The new DCHP module (*misc.dhcpd*) triggers *DHCPLease* events every time an IP address is associated with a host. *Host_tracker* was extended to subscribe to that event. When notified, it updates its active host database with that information, which helps in the process of answering ARP queries (avoiding broadcasts)

We altered *host_tracker* trigger events whenever a host is added or removed. *Topology* listens to those events and updates its object model accordingly, to keep track of all entities. The event handler in this later module creates an instance of the *Host* class, which is added to the topology. The same behavior is already used by *of.topology* to add an instance of the *Switch* class to *topology* when element of that kind is identified in the network.

Similarly, when a switch stops responding to LLDP, a *SwitchLeave* event is triggered by *of.topology*; when a host becomes inactive, does not answer to ARP probes sent by *host_tracker*, or generates no observable traffic, a similar event is triggered. *Topology* listens to those events and updates its database.

C. The graph abstraction and its module

The proposed graph is represented by $G = (V, E)$, in which V and E are finite sets of vertices and edges, respectively. Each vertex $v \in V$ represents one *host* or *switch* in the network. They are objects of class *Entity* — to be precise, of its sub-classes *Host* or *Switch*.

Each edge $u \to v \in E$ represents a link between two vertices. Edges between hosts and switches are derived indirectly through the events of host addition or removal, assuming there is a link to the switch where the host is first detected. It is essential, then, that *host_tracker* notifies topology about the identity of the new host, as well as the identifier for the switch and the port used. The system must guarantee that such events are only generated by the switch to which the host connects directly. That is achieved in our case because no other packet can traverse the network from a host before *host_tracker* identifies it (and its original switch) and adds forwarding rules at that switch. Edges between switches are identified by *of.discovery* in the current implementation. It publishes a *LinkEvent* that notifies *of.topology* whether a link turns out to be *up* or *down*. The later module updates the status of the associated switches. Originally, that was not informed directly to *topology*, though. In our solution, a new kind of entity, *Link*, was added, as well as the events *LinkJoin* and *LinkLeave*, to fix that.

The edges weight $g(u, v)$ describes the traffic in bytes received and transmitted through the edge/*link*. The edge weight is obtained by periodically reading the counters from the OpenFlow switches [11].

On its turn, every time *topology* identifies a change in the network, it triggers a related event, *topologyUpdate*, so that other modules that may want to keep track of that can do so. The Graph module will do just that, and react to any changes triggered by any publisher in one of the channels observed by *topology*.

D. Programming Interface (API)

The graph module has an application programming interface (API) so that other modules of the controller can retrieve information about the network topology from it.

The major graph methods are:

- *get_vertex(id)*: Returns an entity of the graph (*host/switch*).
- *get_adjacents(id)*: Returns the adjacency list of a vertex.
- *snapshot()*: Copy the graph in the form of two collections (vertices and edges)
- *to_dot(harq, layout = "dot")*: Create a copy of the graph to be used by Graphviz [12].

Besides that, the module is fully extensible, and our goal is that different graph algorithms may be implemented by developers or the end users, and always retrofitted to the module, so they can become available to others. By doing so, code reuse is maximized; anyone implementing a different version of a routing algorithm could benefit from a previously implemented shortest path algorithm, for example. Right now, just a minimum spanning tree algorithm has been implemented, being available through the method *get_mst()*.

III. Experiments

The operation of the *graph* module was tested using Mininet, a simulation system created exactly to simulate SDN scenarios [8]. We had to extend Mininet with methods to control the addition and removal of network elements, so we could observe the system behavior as the topology changes dynamically.

Besides the code interfaces, which affected different classes, the standard command interface *prompt* was extended with new commands to dynamically add/remove controllers, switches, hosts, links and network interfaces. All changes were made with compatibility in mind, so the original Mininet data structures and interfaces were preserved as much as possible.

The goal of the experiments that follow was to validate and evaluate the performance of the proposed system. Consider a network with a simple topology as shown in Figure 2, that has two switches, each with three hosts; there is an edge (link) between the switches, and one instance of the POX controller controls both switches.

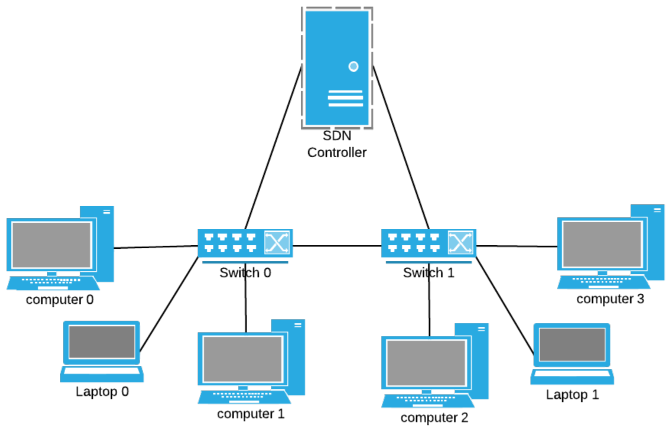

Figure 2. Simple topology

A. Entity Detection

When the system starts, the graph is empty. The switches are the first entities to be identified, because the controller is directly connected to the switches through the OpenFlow interface and it sees nothing before those connections are established. A *ConnectionUp* event is triggered at the controller core. Figure 3 shows the events logged during the start of the execution.

```
INFO:topology.graph:SwitchJoin id: 2
INFO:topology.graph:SwitchJoin id: 1
INFO:topology.graph:1, 2
DEBUG:openflow.discovery:Dropping LLDP packet ↩
    275
INFO:topology.graph:LinkEvent fired
INFO:host_tracker:Learned 1 1 7e:e6:9b:89:39:2e↩
     got IP 10.0.0.1
INFO:topology.graph:HostJoin id: 7e:e6:9b↩
    :89:39:2e
INFO:host_tracker:Learned 2 1 62:77:44:24:13:49↩
     got IP 10.0.0.2
INFO:topology.graph:HostJoin id: ↩
    62:77:44:24:13:49
```

Figure 3. Entity Detection

In the first two lines of the log shown in Figure 3, the graph module was notified about the discovery of two switches in the network. Therefore, two vertexes of the switch entity type were referenced in the graph. Line 4 indicates one event related to LLDP, when switches start to look for others, and *of.discovery* is activated. Right after that, in line 5, we notice the discovery of a link (between switches). Lines 7 and 9 show two hosts discovered by *topology*, as a result of them being discovered by the *host_tracker* module, possibly during their fist effort to connect to the DHCP server. Any new package that passes through a switch and has no installed flow in the flow table is forwarded to the controller, that triggers a *PacketIn* event and identifies the entities (hosts and switches) involved in the communication. By continuing acting like this, the network entities are identified and represented in the graph.

B. Removing Entities

Two experiments were executed to observe how the graph was updated when an entity became unavailable. For the event of host removal, we turned off one host's network interface using the Mininet command prompt. The *host_tracker* module, in the absence of traffic from a certain host, triggers a timer event and an ARPRequest message is sent to the missing host. The module uses a two-out-of-three policy to decide if a host is down. By doing so, after thirty seconds the host was marked as unavailable and a *HostLeave* event was triggered, updating the graph.

For the second experiment, a switch was turned off using Mininet. Since the controller had a direct connection to the switch (for the OpenFlow protocol), once the POX *core* detected the loss of the connection to the switch, the graph was updated, removing the switch and hosts connected to it.

C. Real-time visualization

Figure 4 shows the graph of a network with 8 switches, each one with 30 hosts connected, totalizing 248 network entities. This graph was updated in real time by events that occurred inside the controller, like entities joining/leaving, observation of traffic volume in the network, and others.

Events were programmed to occur during a predefined interval and the final graph was obtained within seconds of the end of the series of add/remove commands. As observed previously with the individual events, host removals were detected withing 30 seconds, host additions were detected in milliseconds after their first transmission, and switch events were detected promptly as the OpenFlow connections went up/down.

D. Network Traffic Identification

We used the Iperf tool as a server on host 'Host 0a' to compute the (TCP) traffic on the network shown in Figure 5. The host 'Host 1e' connected as a client. As can be seen in Figure 5, the traffic in bytes on the edges that connect those hosts is larger than that of the other hosts. In the moment that the port counters were read and the edges weights computed, the traffic was 55894 bytes through the path between the two hosts.

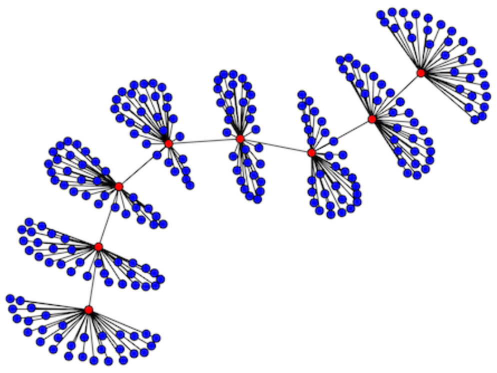
Figure 4. 248 nodes network represented by a graph

The values observed for the other links (41 bytes) are from the ARP messages sent by those hosts and by *host_tracker*. The experiment with Iperf shows a forwarding rate (throughput) between hosts of 300 Megabits per second, confirming there is no bottleneck due to the graph abstraction.

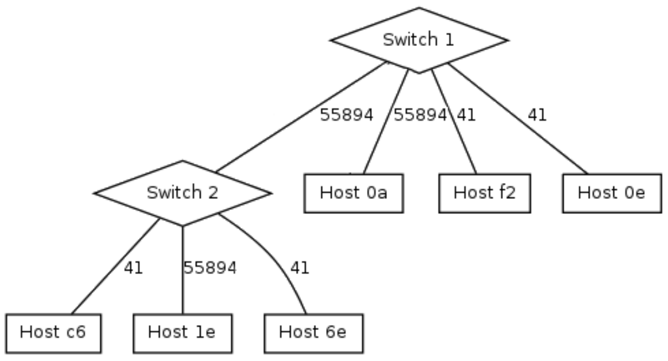

Figure 5. TCP Traffic (number of bytes) between the hosts 'Host 1e' and 'Host 0a'

E. Minimum Spanning Tree

Minimum spanning trees are essential in many tasks in network management. Tasks like trigger 'alarms' when the tree is disconnected or loops are detected, as presented are essential [13]. One can think of a distributed system that uses such a tree to execute a message propagation algorithm using flooding to limit the number of retransmissions [14]. A network with multiple paths can implement dynamic load balancing by computing multiple spanning trees in real time [15].

The minimum spanning tree algorithm implemented in the *graph* module was tested by using it to maintain a dynamic minimum spanning tree as the network was updated. Whenever the graph was altered by the addition/removal of an edge/vertex in response to a change in the underlying topology, the minimum spanning tree was re-computed as expected.

IV. Related work

The idea of representing the network as a graph was mentioned by Casado *et al.* in one of the first SDN papers [5]. However, there were no details about their implementation. In a later work, SDN was used to implement different network topologies in a datacenter scenario, but that was not done by implementing a graph abstraction inside the controller [16].

Raghavendra *et al.* presented a graph module with dynamic update capabilities and a public API that could be extended to include different graph algorithms [6]. Although the work was aimed at SDN/cloud scenarios, there was no actual integration with any SDN controller, which was the major focus or the present work.

The Onix controller [2] was designed around the concept of a network information base (NIB). That base keeps a global view of the network in a form similar to SNMP's MIB. However, the representation of the graph is achieved by indexing an element's entry in relation to its neighbors, not directly.

DSLs (domain specific languages) are presented by Frenetic [4] and Pyretic [14] as good solutions for the network data retrieval problem. Neither of them export the network graph as a first order element, but it can be built externally based on the information available.

V. Conclusion

This paper showed the use of graphs in the Software Defined Networking context. A real time graph of the network meets one of the mainly advantages expected from SDN by decoupling the control plane from data plane, which is to achieve a global view of the network. It is noted that the system keeps a reliable, consistent and dynamic representation of the real network, facilitating the management tasks in a Software Defined Network.

As a future work we plan to build an on-line graph visualizer that interacts with the network administrator and shows, in an easier way, the entire network operation. Various popular graph algorithms should be provided by the graph module. Those should be implemented in the future releases of the system.

One element of concern is reliability: just like POX by itself, if the controller process is terminated, then the entire graph is lost. For that, a distributed graph database can be used to store the network graph in a persistent, reliable and fault tolerant structure.

Finally, the graph abstraction has been identified in other network scenarios, like cloud computing. OpenStack, one of the most popular systems for cloud management/virtualization orchestration, includes a network topology view in one of its modules, Neutron [17]. That abstraction has already been combined with SDN to implement isolated multi-tenant networks [18], it might be interesting to consider how they could be further combined to build a unified, shared view.

Acknowledgements

This work was partially sponsored by UOL, Fapemig, CNPq, and the National Institute of Science and Technology of the Web, InWeb (MCT/CNPq 573871/2008-6).

Code availability

The implemented system is available at https://github.com/pantuza/pox [19].

References

[1] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: towards an operating system for networks," *SIGCOMM Computer Communication Review*, vol. 38, pp. 105–110, July 2008. [Online]. Available: http://doi.acm.org/10.1145/1384609.1384625

[2] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker, "Onix: a distributed control platform for large-scale production networks," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–6. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924943.1924968

[3] T. L. Hinrichs, N. S. Gude, M. Casado, J. C. Mitchell, and S. Shenker, "Practical declarative network management," in *Proceedings of the 1st ACM workshop on Research on enterprise networking*, ser. WREN '09. New York, NY, USA: ACM, 2009, pp. 1–10. [Online]. Available: http://doi.acm.org/10.1145/1592681.1592683

[4] R. Foster, M. Freedman, and J. Rexford, "Frenetic: a network programming language," *SIGPLAN Notes*, vol. 46, no. 9, pp. 279–291, Sep. 2011. [Online]. Available: http://doi.acm.org/10.1145/2034574.2034812

[5] M. Casado, T. Koponen, R. Ramanathan, and S. Shenker, "Virtualizing the network forwarding plane," in *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow*, ser. PRESTO '10. New York, NY, USA: ACM, 2010, pp. 8:1–8:6. [Online]. Available: http://doi.acm.org/10.1145/1921151.1921162

[6] R. Raghavendra, J. Lobo, and K.-W. Lee, "Dynamic graph query primitives for SDN-based cloud network management," in *Proceedings of the Workshop on Hot Topics on Software Defined Networking, HotSDN'12*, 2012.

[7] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, pp. 69–74, March 2008.

[8] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, ser. Hotnets-IX.

New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1868447.1868466

[9] "The POX platform," http://www.noxrepo.org/pox/about-pox/, 2012, accessed on November, 2012.

[10] "Ieee standard for local and metropolitan area networks - station and media access control connectivity discovery," IEEE Computer Society, New York, NY, USA, IEEE Standard 801.1AB, September 2009.

[11] "Openflow switch specification," https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf, 2013, [Online; Accessed on July 2014].

[12] J. Ellson and E. Koutsofios, "Graphviz and dynagraph – static and dynamic graph drawing tools," in *GRAPH DRAWING SOFTWARE*. Springer-Verlag, 2003, pp. 127–148.

[13] S. Schmid and J. Suomela, "Exploiting locality in distributed SDN control," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '13. New York, NY, USA: ACM, 2013, pp. 121–126. [Online]. Available: http://doi.acm.org/10.1145/2491185.2491198

[14] C. Monsanto, J. Reich, and J. Rexford, "Composing software-defined networks," in *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, ser. nsdi'13. Berkeley, CA, USA: USENIX Association, 2013.

[15] J. Mudigonda, P. Yalagandula, M. Al-Fares, and J. C. Mogul, "Spain: Cots data-center ethernet for multipathing over arbitrary topologies," in *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, ser. NSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–16. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855711.1855729

[16] B. Heller *et al.*, "Ripcord: a modular platform for data center networking," *SIGCOMM Comput. Commun. Rev.*, vol. 40, pp. 457–458, 2010.

[17] "Openstack:open source software for building private and public clouds," http://openstack.org/, March 2012.

[18] R. V. Nunes, R. L. Pontes, and D. Guedes, "Virtualized network isolation using software defined networks," in *Proceedings of the 38th IEEE Conference on Local Computer Networks (LCN)*. IEEE, 2013, pp. 700–703.

[19] G. Pantuza and F. Sampaio, "Pox: graph module," https://github.com/pantuza/pox, 2013, Online; accessed on September 30, 2014.