

A network's computational model for Software Defined Networking

Andrés Laurito

Departamento de Computación - FCEN -UBA
andy.laurito@gmail.com

Abstract. In this paper we define a computational model for network's programming languages that use Software Defined Network (SDN) paradigm. For this purpose, network's will be model as graphs, and a graph grammar semantic will be defined. Production's in the grammar will model actions that can take place in the network, and programs will be just a sequence of instruction's modifying a starting graph.

After graph grammar's definition, we will extend this grammar to a type graph grammar, and we will explain how this new grammar can be applied to create a type programming language for networks. Finally we will show how this computational model is well defined, by using it to create a compiler for a new network programming language recently emerged called Nemo.

1 Introduction

Nowadays network's have became a crucial resource of any software application. Technologies for improving code crafting are always emerging. By contrast, network's have several issues to be solved. Some of this issues may be tagged as debugging problem's (it is rather difficult to debug problems in network's using known tools such as ping, traceroute, netstat, between others), changeability problem's (changing traffic rule's, or security rule's in network may imply modifying hundred, or even thousands code line's distributed in several devices around the network), deploys problem's (most changes are made manually in several devices, making human error's easier), between others. Is it possible to do better ?, Can software be of help in this task?, Is it possible to know when network's changes will break something before applying them?.

We believe that all this question's have possitive answers, and the best way of attacking this problem nowadays is using network's programming languages which are build over the Software Defined Networking (SDN) paradigm.

Multiple SDN's programming languages have emerged over the past year's (some examples of these are Pyretic, Frenetic, Nemo, etc.), having each of them different's approach to solve some of the problems described before.

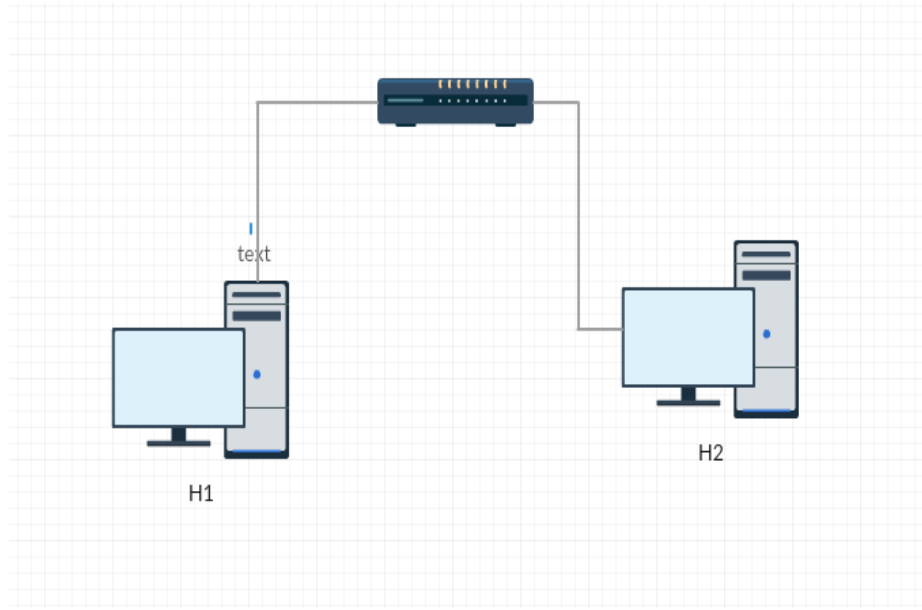
Let us remember that a program may be defined as a sequence of instruction's that executed perform a task in a computer or multiple computer's. Every program can be model as a Turing's machine or a finite lambda calculus computation, and despite of the model chosen, the operations are performed over a structure (usually this structure is a memory). In consequence, every network program is model as a finite sequence of operations over a structure.

Our first proposal is to define this structure as an oriented graph, since network's can be easily mapped to graphs, and also is perfect reasonably to think program's operation as graph's rewriting. After this reasoning, the second propose is to define a graph grammar in order to model all the network program's that may be written in any network programming language.

In order to define the grammar production's, we will narrow to model program's that solve the following problems:

First Problem:

Suppose that we have the following network:



where H1 want's to establish an ssh connection with H2, and let's assume that there are not forwarding rules in order to establish such connection's, and also H2 has not an ssh server client to fulfilled this connection.

Defining a graph grammar allows us to check easily some of the problems described before, particularly static properties, such as examples 1 to 4 (as we will shall show in following sections). However, dynamic properties (such as applying some load balancing policy when a queue start's to drop more than 50% of packets) are not easily checked by grammar production's. Some problems required to execute our program in a traffic network, however our approach is identifying problems in program's before they actually happen in the real network. In order to identify this class of problems, we propose an execution of some of the grammar production's in a simulator which will be inserted inside the language compiler. The program will then be executed in several simulation's, in order to check if the production applied to the actual graph is possible. For doing this, we will mapped the dynamic grammar production's to devs (a formal language specification for defining simulator's), and we will run a simulation using this

formal specification.

In section 2 we will define the graph grammar production's, which will model a primitive set of operations that can be performed over the network. In section 3 we will use this grammar to model sequence of operations to show how the problems introduced before are solved. In section 4 we extend the graph grammar of section 2 to a type graph grammar. In section 5 we show how this grammar can be apply to model Nemo program's. AND MORE

2 Graph grammar

We start by defining the most basic grammar production's. These productions are the one's that allows us to modify the network topology:

- Node creation

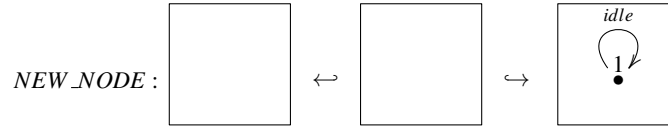


Fig. 1. Node creation

This production creates a new logical node (this means that the node could be either physic (which mean's it's a physical host with an associated address) or logic (this mean's a group of physical nodes, meaning that the address could be a CIDR).

It is important to remark that the created node is in an idle state, meaning that is not interacting with any other node in the network.

- Service creation

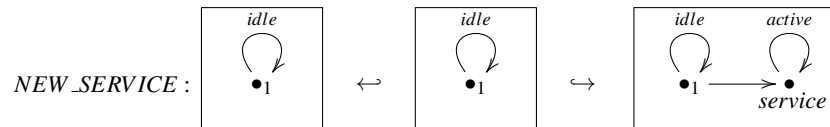


Fig. 2. Service creation

This is the production which defines new services in my node. Services are defined by the user, and they can be whatever service make sense in a node (also is im-

portant to notice that services in logical nodes, will not be the same as services in physical nodes). A service could be an Enum Type or something like that. Examples of a service could be ssh,http,https,sql,etc. (Note that physical services can be mapped to an active port).

Another thing to notice is that services will start in an active state, expecting to be used by the network. This state is going to be important in a future, because this state will define whenever is possible to use the service.

What would it be a service when we are talking in the logical node case?.

– Link creation

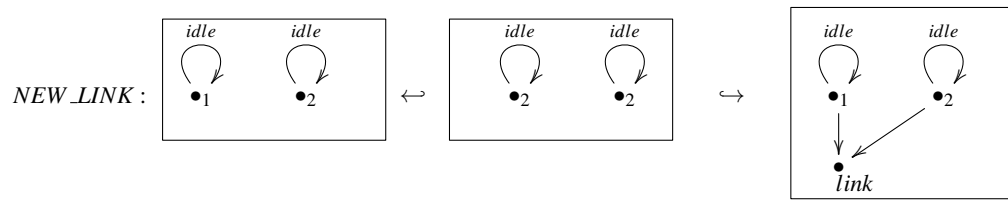


Fig. 3. Link creation

Here we define a new link between nodes 1 and 2. The only condition to apply this production is that both nodes must be in an idle state.

Note: Being in an idle state does not mean that they are doing nothing!. An idle state behaves like a wildcard, meaning that from this state, multiply actions can be applied to the node.

– Link's property creation

NEW LINK_PROPERTY :

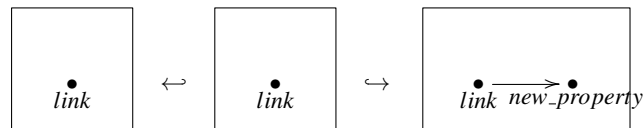


Fig. 4. Service creation

In this production we are allowing definition of link's property. This properties are going to be perhaps physical properties in physical links (for example, we can define here bandwidth, average latency, perhaps if it's either ethernet or wifi), and

some other properties in logical nodes.

QUESTIONS:

- 1) What properties could be assigned to a logical link?
- 2) Can I always define properties in a node? (It really doesn't matter what is going on with that link in the network?. Perhaps it happens something similar as nodes, that we need some 'idle' state, for example, an empty state).

So far we have defined productions that which are intended to be use between the interactions with the elements in the network. With the following productions, we will start using the productions listed before, in order to crate interaction between the elements in the network.

– Flow creation

NEW_FLOW :

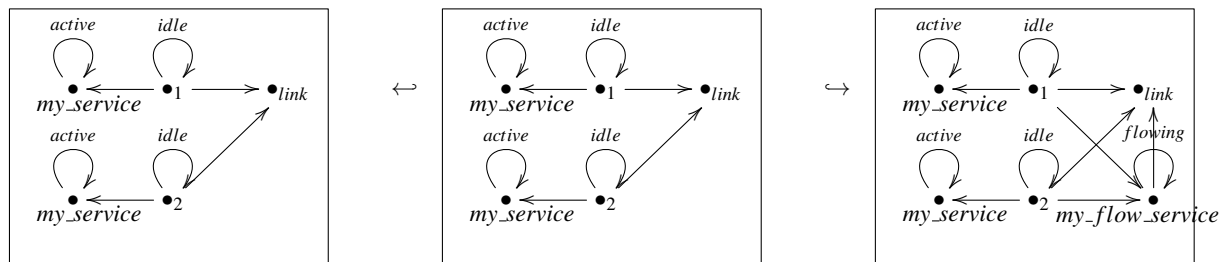


Fig. 5. Flow creation

In order to create a flow, I first need two things:

- I need to have a link between the nodes
- I need to have in both nodes the service which will be the content type of the flow.

Lets go through these rules one more time. The first rule is just telling us that, in order to be able to create a node, I need a connection between this node's (remember that this connection can be either logical or physical).

The second rule is telling us that, in order to create a flow of type X (lets say for example, that I want to create a ssh flow between this nodes), I need to guarantee that both nodes support ssh connections. This is represented as both nodes having a link to the service.

Finally, it's important to notice that a flow is associated between nodes and a link (this last one is going to be the place where it's flowing).

THINS THAT HAD TO BE THINK A BIT MORE:

- 1) Some rules may not be valid in this context ... How do I deny multiple ssh connections between same nodes? (Also, this sounds to me like a type condition, since another services may allow this behaviour, for example sql.)
 - 2)
- Flow's properties creation

NEW_FLOW :

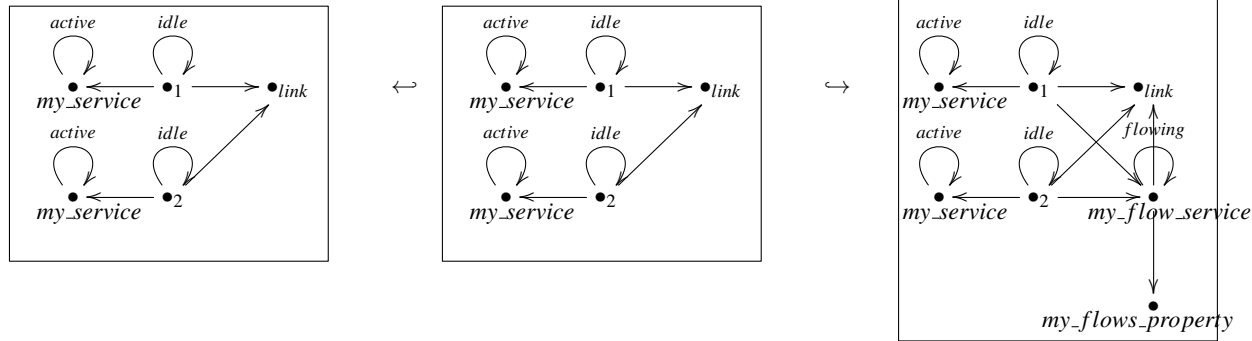


Fig. 6. Flow's property creation

This production allow's the creation of flow's properties.

QUESTIONS:

- 1) Can this properties be the actions associated in the nemo language ? Perhaps properties can be understood as premises that have to be guaranteed while the flow exist.
- 2) Some of this properties could be dynamic (this has sense, since flow is the representation of dynamism in the network).

3 Graph instantiation

In this section we will focus in giving several examples of the usage of the grammar recently defined. The examples used in this section will be those defined in a previous document (REMEMBER TO REWRITE EXAMPLES HERE)

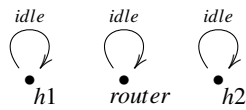
Lets first defined the first example, a network with two host and a router, which want to start a ssh connection.

There were two problems here:

- Neither of the host had ssh active
- Neither of the host were connected to the router.

Let's model our network, and see how the grammar defined before help us to detect problems in what we are trying to do.
This would be the network represented in our model:

FIRSTEXAMPLE



Having model our network, what we can check is that `NEW_FLOW` production can't be applied to my network (since there is no morphism between the left side of the production and this model). In this case, if I write a program for the situation described, we will have a compiler error, since there's going to be an instruction that cannot be done, and this is because the production associated with it cannot be executed.