Extension of the OpenFlow framework to foster the Software Defined Network paradigm in the Future Internet

Marco CASTRUCCI^1, Andrea SIMEONI^2
^{1,2}*University of Rome "La Sapienza"*
Dipartimento di Ingegneria Informatica, Automatica e Gestionale
Via Ariosto 25, Rome, 00185, Italy
Email: castrucci@dis.uniroma1.it, andreasimeoni84@gmail.com
^{1}*Tel: +39 348 7709997*

Abstract: Software Defined Network (SDN) is one of the most promising paradigm for the Future Internet (FI), as it is a network architectural approach that enables the network virtualization and its centralized control with open interfaces for network applications and services. OpenFlow constitutes one of the pillar technologies for the development of the SDN, defining a protocol to control open switches and routers. Based on OpenFlow, several other research results have been achieved so that now the OpenFlow framework can be used to virtualize and program open networks. The aim of this work is to contribute to the development of the OpenFlow framework, by means of a proposal for the extension of the APIs made available to program and control the network. In particular, a novel network controller component, called 'Traffic Statistics', is presented in this work, able to collect network traffic information, to be used by cognitive resource management algorithms aiming at exploiting the network in a efficient way and at the same time at providing an adequate level of QoS to network flows. The proposed 'Traffic Statistics' system design is presented here, together with the description of its preliminary prototypal implementation.

1. Introduction

Already in 2005, there was the feeling that the architecture and protocols of the Internet needed to be rethought to avoid Internet collapse [1]. However, the research on Future Internet became a priority only in the last few years, when the exponential growth of small and/or mobile devices and sensors, of services and of security requirements began to show that current Internet is becoming itself a bottleneck. Two main approaches have been suggested and investigated: the radical approach [2], aimed at completely re-design the Internet architecture, and the evolutionary approach [3], trying to smoothly add new functionalities to the current Internet towards. Right now, the technology evolution managed to cover the lacks of current Internet architecture, but, probably, the growth in Internet-aware devices and the always more demanding requirements of new services and applications will require radical architecture enhancements very soon. This statement is proved by the number of financed projects both in the USA and in Europe. In Europe, Future Internet research has been included as one of the key topics in FP6 and FP7. Moreover, in March 2010, the European Commission launched the Future Internet Public-Private Partnership (FI-PPP) [4] with the aim to foster the European research on Future Internet topics. With a budget of 600 MEuro, the FI-PPP is funding research projects related to FI. In particular, in April 2011, eight FI use case projects have been started with the aim to identify the requirements for the FI, while in May 2011 the FIWARE project [5] has been started with the aim to design and develop the FI Core Platform, namely a platform providing all the necessary technologies for the FI.

One of the key technological topic addressed by the FIWARE project is the so called Software Defined Network (SDN) paradigm [6]. SDN is a new network architecture approach that uses: (1) separation of data and control planes with a well defined vendor agnostic API/protocol between the two; (2) a logically centralized control plane with an open API for network applications and services and (3) network slicing and virtualization to support experimentation at scale on a production network. At the moment, OpenFlow [7] is considered the best candidate protocol to realize SDN. Open Networking Foundation [8], launched in March 2011 and involving major manufacturers and Internet related companies (e.g., Google, Facebook, Deutsche Telekom, Microsoft, Cisco, Ericsson, etc.), is a no-profit organization dedicated to promoting the SDN approach and the OpenFlow protocol.

In parallel, cognitive networking is considered the natural evolution for the management and the control of FI networks [9]. Cognitive networks will be managed and controlled in a context-aware way. This means that management and control algorithms and mechanisms will take advantage of a set of context information, including, for example, network status and availability information, user profile information, service/application requirements and characteristics, etc. On the basis of these context information, it will be possible to take consistent control and management decisions concerning the best way to exploit and configure the available network resources in order to efficiently and flexibly satisfy service/application requirements and, consequently, the user's needs.

The objective of this work is to close the gap between SDN and cognitive networking and to realize one of the basic pillars for the realization of the FI, extending the OpenFlow framework and contributing to its development by the definition and the realization of a new system, called 'Traffic Statistics'. This new system is able to monitor the networks and to collect traffic information to be used by cognitive resource management algorithms aiming at exploiting the network in a efficient way and, at the same time, at providing an adequate level of QoS to network flows, as well as by any other network application that will be built on top of the network operating system, following the SDN approach.

2. The OpenFlow framework

A key revolution for the Future Internet architecture is the centralization of the control-plane logic, not still spread over network nodes but implemented in a single logical entity. This approach considers network nodes as dumb forwarding elements, whose data-path is open and controlled by the central control entity. Control-plain centralization allows to define the concept of Network Operating System, that provides a flexible mean to develop (exposing high level APIs) and orchestrate different concurrent control processes, keeping the network in a safe and consistent state. With network devices opening their platforms, it is possible to define multiple networks existing at the same time, over the same physical infrastructure, allowing innovative solutions being conceived, deployed, tested and instantiated on-demand. This will bring innovation not only in little worlds such as campus networks and enterprises but also in broader business scenarios, enabling new actors such as Virtual Network Providers and Virtual Network Operators entering the business.

OpenFlow was developed at the Standford University as a mean for allowing researchers to run their own experiments exploiting the campus network infrastructure. Therefore one of its main design goals was to enable virtualization of the network, for achieving coexistence of experimental and normal production traffic over the same physical nodes and links. From this starting point, the OpenFlow protocol has been extended and an OpenFlow network framework has been developed enabling the SDN. Figure 1 depicts a typical OpenFlow network setting.

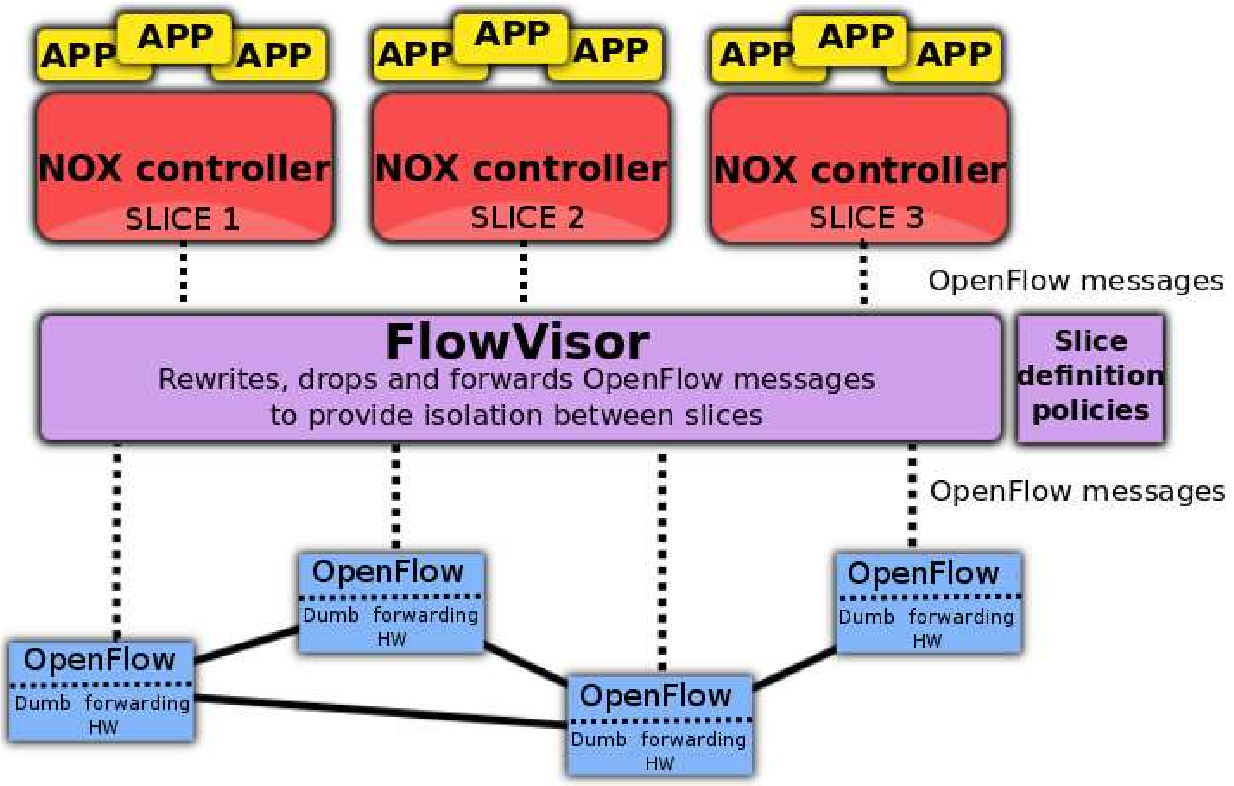

Figure 1 – OpenFlow network framework

The OpenFlow protocol defines the message exchange between the controller and the OpenFlow nodes of the network. A full operative OpenFlow network is composed by three main components:

1. A set of managed *OpenFlow switches and routers* (OpenFlow Dumb forwarding HW)*;* each OpenFlow switch/router provides the flow-table abstraction, a simple but powerful data structure representing the node's data-plane and remotely configured by the controller by means of the OpenFlow protocol (based on a SSL channel). Each flow-table entry has three different fields: 1) a matching flow header, 2) actions performed on match, 3) counters for statistics. Actions performed on header matches are a basic set of functions common to most devices' platforms; such functions are 1) Forward this packet to a given port 2) Encapsulate and forward this packet to the controller (for new incoming flows) 3) Drop packets of this flow.
2. One or more *controllers*, each one running over a network operating system like NOX [10]. From the network side NOX interacts with open network devices through OpenFlow instructions. To the application side NOX exposes an high level API for exchanging OpenFlow messages with the network, allowing to easily read and write the network's data-plane state. A controller running over the NOX platform is made up by different logically linked components running at the same time. Basically NOX provides a standard set of components aimed at building an abstract wide view of the network (i.e. shortest paths, topology, link load, etc.) and at providing basic control-plane functionalities (i.e. routing and host tracking). Such wide view is built by such components using the API provided by NOX for exchanging OpenFlow messages with the network. NOX exposes to the running components an event driven environment, in which they can register for event handling

and react upon event notifications. Events may be generated by the network (i.e., flow-in event) or by other components upon updates of their internal states.

3. An instance of the virtualization layer *FlowVisor* [11], just between controllers and the network. FlowVisor allows the network being controlled by many controllers at the same time. This layer appears invisible both to controllers (that believe to have total control over the network), and to OpenFlow switches (believing to be controlled by a single controller). In this way, we can define multiple networks slices sharing the same physical resources (topology, nodes' CPU, flow tables, traffic, link bandwidth, etc.) in a perfectly isolated environment.

As a Network Operative System, we consider NOX, which includes a set of already implemented components that can be selected and used to perform basic control-plane tasks. The following are the most important components offered by NOX:

1. *Network state*: set of components that store network state information, like node features supplied during registration phases (L1/L2 features, fragmentation support, buffering capabilities, etc.); real-time residual capacity (bps) on network links; switch-controller RTT, node statistics (packet loss due to overrun collisions, etc.) and similar.
2. *Host state*: set of components providing a full set of functionalities for host tracking and mobility management; allowing context and position aware services being supported. It comes useful for implementing accounting services and applying special security policies for different connected users [12].
3. *Routing*: key component that estimates shortest paths between each source-destination pair in the network. It exposes an API for installing flows on estimated paths. Typically, when detecting a new flow entering the network, we can retrieve the shortest path from the routing component, and use the provided API for installing the flow in each nodes' flow table along the path. Shortest paths are computed on the basis of host-tracking and topology information.
4. *Topology*: dynamically updates proper data structures to maintain a real-time snapshot of the network topology. Topology updates are advertised both by OpenFlow switches (when a node registers to the controller) or by other components (i.e., "Discovery" is a special purpose component whose assignment is to discover when network links are up or down).

3. 'Traffic Statistics' component

The 'Traffic Statistics' module that we present here is intended to be a new component to be implemented in a Network Operative System in order to collect information and statistics about the traffic crossing the network and giving the possibility to other network applications to use these information. In other words, 'Traffic Statistics' can be seen as an interface between the considered open network and the user (i.e., a network application) that provides to the user information and statistics about the traffic present in the network. This kind of information may be of interest for the user in order to monitor the traffic flows in the network and to evaluate the capability of the network to provide a pre-defined level of performances and of Quality of Service. In addition, the information provided by this interface can be used as input for the mechanisms and the algorithms used to control the traffic into the network. In particular, this interface is a key element for cognitive networks, which need a real-time monitoring of the on-going traffic.

The 'Traffic Statistics' component is defined and designed here in a open and technology independent way, but its real implementation is subject to the specific environment considered. As for example, we also provide a description of the reference implementation of this component, developed as an extension of the NOX OpenFlow controller.

3.1 'Traffic Statistics' design

In the rest of the document, the following definitions and notations are used:

- ***N*** is the set of nodes of the network.
- ***K*** is the set of Service Classes supported by the network.
- **Flow** (*f*): it is a triple <*n,m,k*> referring to the packets entering the network at a given ingress node $n \in N$, and flowing out of the network at a given egress node $m \in N$, relevant to in progress connections belonging to a given Service Class $k \in K$.
- **Traffic Packet Loss** (TPL(*f*,*t*)): fraction of packets relative to a flow *f* lost in the network (packets lost/packet transmitted) during a predefined time slot ending at time *t.*
- **Traffic Delay** (TD(*f*,*t*)): average packet delay, in a predefined time slot ending at time *t*, experienced by packets belonging to the flow *f*.
- **Traffic Jitter** (TJ(*f*,*t*)): average packet delay variation, in a predefined time slot ending at time *t*, experienced by packets belonging to the flow *f*.
- **Traffic Load** (TL(*f*,*t*)): average admitted bit-rate (bps) relative to a given traffic flow *f* in a predefined time slot ending at time *t.*
- **Network Availability** relevant to flow *f* (NA(*f*, *t*)): percentage of time, computed during a suitable time interval ending at time *t*, during which the network is available for the flow *f*. The network is considered available whenever the performance in terms of transfer traffic delay, traffic load (admitted bit rate) and traffic packet loss is satisfactory, that is the following inequalities are simultaneously met:
    - $\mathrm{TD}(f, t) \le \mathrm{TD}_{max}(k)$
    - $\mathrm{TPL}(f, t) \le \mathrm{TPL}_{max}(k)$
    - $\mathrm{TL}(f, t) \ge \mathrm{TL}_{min}(k)$

The following is a taxonomy of the functions provided by the 'Traffic Statistics' component.

- **Traffic-Delay**
    - Traffic Delay given a flow
    - Flow with maximum delay given a flow source and destination node
    - Flow with minimum delay given a flow source and destination node
    - Flows with delay higher than a given threshold, given a flow source and destination node
    - Flows with delay lower than a given threshold, given a flow source and destination node
- **Traffic-Jitter**
    - Flow Jitter given a flow
    - Flows with maximum jitter given a flow source and destination node
    - Flows with minimum jitter given a flow source and destination node
    - Flows with jitter higher than a given threshold, given a source and destination node
    - Flows with jitter lower than a given threshold, given a source and destination node
- **Traffic-LossRate**
    - Flow Packet Loss given a flow
    - Flows with maximum Packet Loss given a flow source and destination node
    - Flows with minimum Packet Loss given a flow source and destination node
    - Flows with Packet Loss higher than a given threshold, given flow a source and destination node

    - Flows with Packet Loss lower than a given threshold, given a flow source and destination node
- **Traffic-Load**
    - Traffic Load given a flow
    - Flows with maximum Traffic Load given a flow source and destination node
    - Flows with minimum Traffic Load given a flow source and destination node
    - Flows with Traffic Load higher than a given threshold, given a flow source and destination node
    - Flows with Traffic Load lower than a given threshold, given a flow source and destination node
- **Network availability**
    - Network Availability given a flow
    - Flows with maximum Network Availability given a flow source and destination node
    - Flows with minimum Network Availability given a flow source and destination node
    - Flows with Network Availability higher than a given threshold, given a flow source and destination node
    - Flows with Network Availability lower than a given threshold, given a flow source and destination node

The UML use-case diagram depicted in Figure 2 shows functional requirements of the 'Traffic Statistics' component, as part of a wider Network Information and Control (NetIC) interface between the open network and the network application.

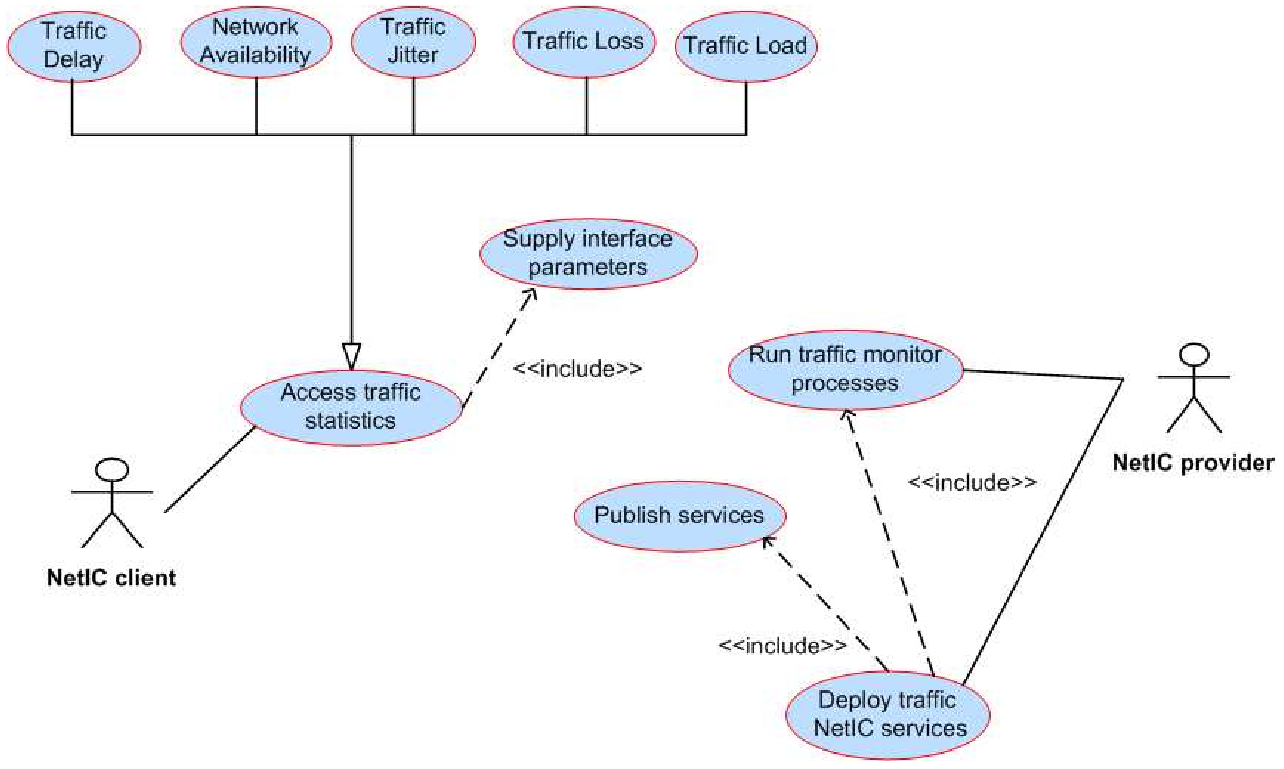

Figure 2 – 'Traffic Statistics' use case diagram

Two type of actors interact with the 'Traffic Statistics' component:

- *NetIC client:* It is the user of the interface. It accesses services in order to retrieve information about traffic statistics. Such information could be directly consumed, or composed to provide aggregated services to third users. In such a case the NetIC user can be identified as a service aggregator.

- *NetIC provider*: It materially provides traffic NetIC services to interface users. It is responsible for the implementation and the exposure of 'Traffic Statistics' functions, based on a certain set of metrics. NetIC provider also uses a service publishing platform, useful for deploying new services and to make their APIs available to external users.

Figure 3 shows the UML class diagram that describes the structure of the 'Traffic Statistics' component.

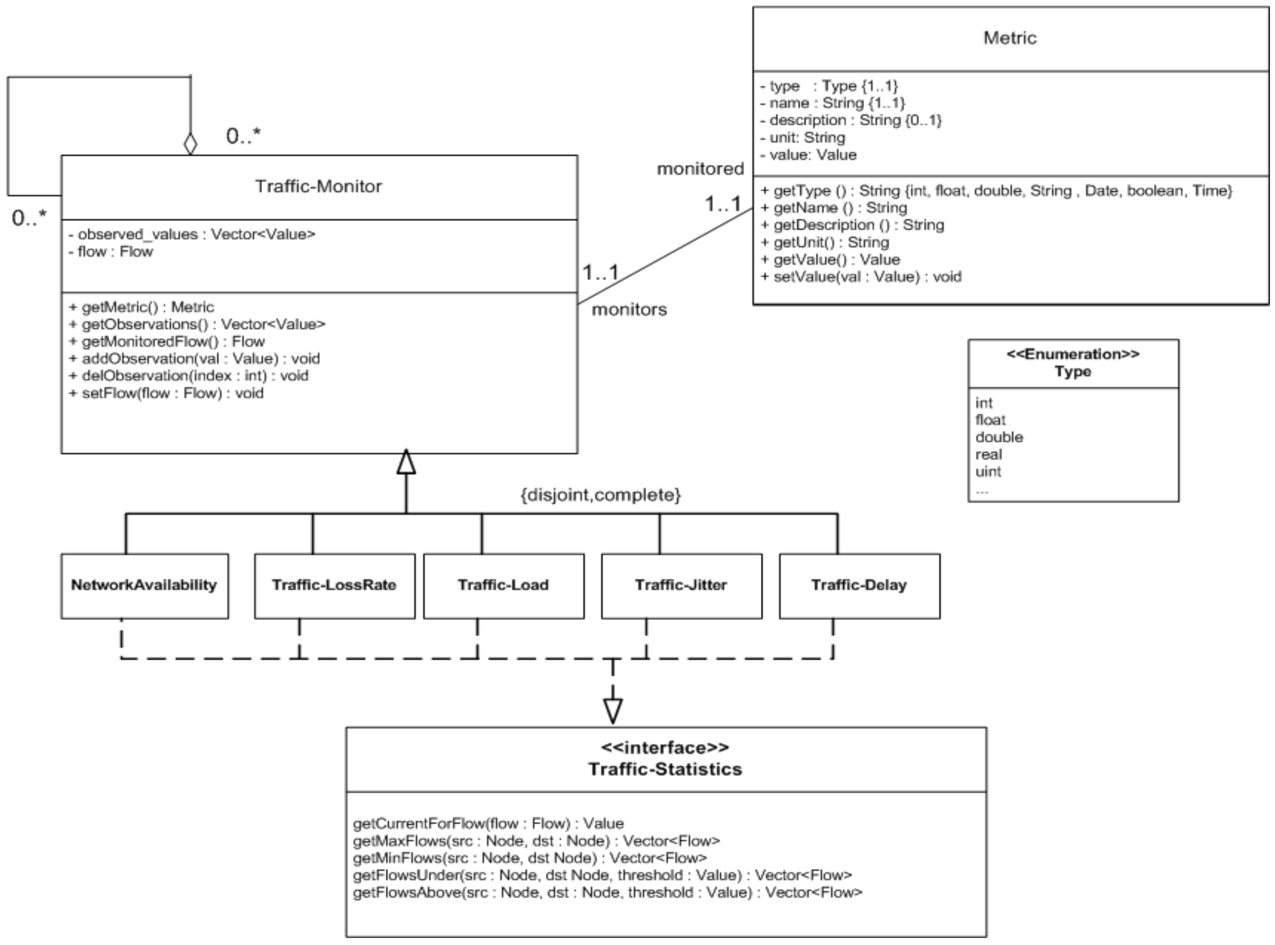
Metric
- type : Type {1..1}
- name : String {1..1}
- description : String {0..1}
- unit: String
- value: Value
+ getType () : String {int, float, double, String , Date, boolean, Time}
+ getName () : String
+ getDescription () : String
+ getUnit() : String
+ getValue() : Value
+ setValue(val : Value) : void
monitored
1..1
1..1
monitors
0..*
0..*
Traffic-Monitor
- observed_values : Vector<Value>
- flow : Flow
+ getMetric() : Metric
+ getObservations() : Vector<Value>
+ getMonitoredFlow() : Flow
+ addObservation(val : Value) : void
+ delObservation(index : int) : void
+ setFlow(flow : Flow) : void
<<Enumeration>>
Type
int
float
double
real
uint
...
{disjoint,complete}
NetworkAvailability
Traffic-LossRate
Traffic-Load
Traffic-Jitter
Traffic-Delay
<<interface>>
Traffic-Statistics
getCurrentForFlow(flow : Flow) : Value
getMaxFlows(src : Node, dst : Node) : Vector<Flow>
getMinFlows(src : Node, dst Node) : Vector<Flow>
getFlowsUnder(src : Node, dst Node, threshold : Value) : Vector<Flow>
getFlowsAbove(src : Node, dst : Node, threshold : Value) : Vector<Flow>

Figure 3 – 'Traffic Statistics' class diagram

Hereafter we provide a description of the classes and the interfaces involved in the class diagram.

Traffic-Statistics Interface: Each object that is able to measure traffic related information from the network should implement the 'Traffic-Statistics' interface. As depicted in the class diagram, there are five classes implementing that interface: Traffic-Delay, Traffic-Jitter, Traffic-Load, Traffic-LossRate and NetworkAvailability. Each of these classes implements Traffic Statistics for its own purposes. Note that such classes are the core part of the Traffic Statistics service provisioning, as they materially collect and calculate traffic measurements on the network. Ait is worth noting also that the definition of an interface allows future extensions and new 'Traffic Statistics' metrics can be introduced in a simple modular way; therefore if we would like to extend the 'Traffic Statistics' service with new traffic metrics, what we only need is to implement the Traffic-Statistics interface to estimate the newly defined measure.

The following list describes the functions provided by the Traffic-Statistics interface.

- *getCurrentForFlow(flow : Flow):* it returns the current value for the metric we are interested in, given a flow to be monitored. The value returned is a Value type. For instance, if the metric of interest is the Jitter, such function should return the Jitter experimented by the flow in a specific time interval (TJ(f,t)).
- *getMaxFlows(src : Node, dst : Node):* given a source and a destination node, it returns flows whose measured metric assumes the maximum value.
- *getMinFlows(src : Node, dst : Node):* given a source and a destination node, it returns the flows whose measured metric assumes the minimum value.
- *getFlowsUnder(src : Node, dst : Node, threshold : Value):* it returns all the flows whose relative traffic metric is under a certain specified threshold. Helpful for improving network resource control, this function comes very useful for understanding if a given traffic class is experiencing a proper network behaviours, allowing instant recovery actions for accurate service level agreement compliancy.
- *getFlowsAbove(src : Node, dst : Node, threshold : Value):* it returns all the flows whose relative traffic metric is above a certain specified threshold.

Traffic-Monitor: A Traffic-Monitor instance is an object that is able to estimate the value assumed by a certain metric for a given flow. Given the {disjoint, complete} ISA constraint, we have that each Traffic-Monitor instance is forced to be one of its subclasses.

- *Attributes:*
  - *observed_values:* it is a collection of different values assumed during time by the related metric;
  - *flow:* it is the current monitored flow.
- *Associations:*
  - *monitors:* each Traffic-Monitor object monitors the evolution in time of its related Metric. For instance, a Traffic-Delay monitor is relative to a delay metric, with its own type, current value, unit and so on;
  - *self-aggregation:* models the fact that a Traffic-Monitor instance may rely on other monitors to carry out its own measurements for a given flow. That happens when defining aggregated traffic measures. For instance, consider the Network-Availability monitor, that relies on measurements carried on by Traffic-Delay, Traffic-Loss, and Traffic-Load monitors. We used an UML aggregation (instead of an UML composition) since different monitors (containers) may share subsets of monitors (parts) to accomplish their monitoring functions. Notice that the aggregation itself doesn't allow cycles between related objects; that is exactly what is needed to avoid deadlocks and dangerous dependencies between traffic monitoring processes.
- *Functions:*
  - *getMetric():* it returns the related metric;
  - *getObservations():* it returns the set of monitored values;
  - *getMonitoredFlow():* it returns the current monitored flow;
  - *addObservation(val : Value):* it adds an observed value to the collection;
  - *delObservation(index : int):* it removes the observed value in position 'index' from the collection;
  - *setFlow(flow : Flow):* it sets the current value of the flow attribute.

Metric: The Metric class represents a measurable quantity that can be collected monitoring a traffic flow.

- *Attributes*:

    - *type*: it represents the type of the metric (int, double, float, etc.);
    - *name:* it is the metric name (i.e., “Delay”, “Loss”, etc.);
    - *description:* it’s a textual description of the metric; it could be expressed in human or machine readable formats (i.e., xml or semantic enriched metadata for provided metrics description);
    - *unit:* it describes the measure unit of the metric (i.e., delay is expressed in seconds, bandwidth in bps, etc.);
    - *value:* it is the current value assumed by the metric.
- *Associations:*
    - *monitored:* each metric is monitored by a Traffic-Monitor instance.
- *Functions:*
    - *getType()*: it returns the metric’s type (i.e., int, double, etc.);
    - *getName():* it returns the metric’s name;
    - *getDescription():* it returns the metric’s description;
    - *getUnit():* it returns the metric’s unit;
    - *getValue():* it returns the metric’s value;
    - *setValue(val : Value):* it sets the metric’s value to val.

3.2 ‘Traffic Statistics’ prototypal implementation

The ‘Traffic Statistics’ system is implemented as a NOX component, thus it has to be integrated with the NOX network operative system, and it has to be able to cooperate with the other components of NOX. In particular, the value of the measurements and statistics collected by ‘Traffic Statistics’ component have to be made available to other NOX components for control plane internal usage, or provided to external users or network applications by mean of the well defined interfaces. Since NOX is based on a set of cooperating components, we have to implement a different component for each of the above described metrics (Traffic Delay, Traffic Jitter, etc.). At this stage of the work, we have only implemented the Traffic-Delay component and we have integrated it with the NOX controller provided by the Standford University, as described in the next sub-section. In the next months we intend to complete the implementation of the whole ‘Traffic Statistics’ module and its full integration with the NOX controller.

3.2.1 Traffic Delay component implementation

The role of the Traffic-Delay component is to measure the average end-to-end delay of a set of input flows that require to be monitored. The monitoring service starts when a flow to be monitored is specified through the service interface. In the preliminary implementation done so far, for the sake of simplicity, we assume that the flow is not yet instantiated in the network when the service request arrives. Moreover, we assume that the network maintains the correct order in packet delivering, so that, given two packets of the same flow, the first packet entering the network is also the first packet that reach the destination. In the next weeks we are going to complete the Traffic-Delay component implementation deleting the mentioned hypothesis.

Traffic-Delay relies on a certain set of basic NOX components to accomplish its monitoring functions. Such components are:

- *Routing*, that provides the shortest path connecting a source node with a destination node;
- *HostTracker*, that maintain the list of host attached to each switch of the network;
- *SwitchRTT*, that estimates the Round Trip Time (RTT) between the network controller and each switch of the network.

Traffic-Delay is programmed for responding to packet-in events generated by OpenFlow switches; such events are generated when the first packet of a flow is received by a switch. Upon event notification, Traffic-Delay extracts the flow header and checks for pending monitoring requests for the new incoming flow. When the packets of a flow to be monitored enter in the network, the Traffic-Delay component reacts in the following way:

- It extracts flow's source and destination hosts.
- It asks to the HostTracker component for retrieving the last known location (switch ID, port) where the destination host is attached. This is not necessary for the source host, since the event is generated by the switch it is attached to, and the event payload contains the in-port information.
- It asks the Routing component for a path between the specified source and destination (if any).
- Made exception for the first and last nodes, it installs the flow in each node of the retrieved path, with a forwarding action according to the route specified by the routing component.
- In the first and in the last node of the path, it installs a pair of actions. The first action is "send packets of this flow to the controller", and the second action is "send this packet on the output port specified by the routing module" (in such a case the port where the destination host is attached).
- When a packet of the installed flow enters the network, the first hop of the path explicitly notifies the event to the controller, sending the packet to the controller. Upon the receipt of the packet, the controller can take an in-going timestamp.
- When a packet exits the network, the last switch explicitly notifies the event to the controller, sending the packet to the controller. Upon the receipt of the packet, the controller can take an out-going timestamp and it can calculate the end-to-end delay for that packet. For the delay calculation, we have also to take into account the time needed for the packets to be sent from the in-going and the out-going nodes to the controller, namely $RTT_{in}/2$ and $RTT_{out}/2$ (where RTT_{in} is the RTT between the in-going node and the controller, while RTT_{out} is the RTT between the out-going node and the controller). Such RTT estimations are provided by the SwitchRTT component, that periodically probes all registered switches with OpenFlow echo requests and estimates RTT. Please note that the RTT between switches and controller plays a key role for the measure quality. In fact, if $RTT_{in}/2 >$ Traffic Delay $+ RTT_{out}/2$, the packet is first notified by the out-going node and then by the in-going node, forcing us to be careful in order to assign the proper out-going timestamp to the proper in-going timestamp. In order to solve this problem, we are evaluating the possibility to associate each timestamp with a checksum of the related packet, so that a univocal association between time stamp and packet is possible.
- Once we have collected the delay measurements for a sufficient number of packets (specifiable through the interface), we can provide an estimation of the Traffic-Delay be means of the calculation of the average delay of the monitored packets.

The described technique gives an estimation of the average delay for the packets belonging to a certain flow, but the obtained measurement must be associated with a level of accuracy that gives an indication of the measurement error. In fact, there are several factors that may influence the monitoring service accuracy. One of the most important is the possibility to have packets lost in the network (in congested situation, some packets may be discarded from switch and router buffers). In that case, there is the possibility to make the mistake to associate the out-going timestamp of a packet with the in-going timestamp of a lost packet. Again this problem can be solved associating each timestamp with a checksum of the related packet, so that a univocal association between time stamp and packet is possible. Another important aspect to be considered is the trade-off between the measurement accuracy and the scalability of the measurement system. Indeed, scalability is maybe the most relevant drawback of the SDN approach; actually there is a lot of research [13] for adapting the SDN approach to carrier grade networks with ultra dense traffic patterns. With respect to the Traffic-Delay component, we should notice that, in conditions of high network load, monitoring each packet of each flow may overwhelm the system resources. For this reason, the Traffic-Delay component can be instructed to estimate only a sample of the monitored traffic (i.e., one packet each n, or a contiguous sequence of n packets, etc.). Obviously, as the number of observed packets increase, also the estimation accuracy increases. An important point of our research is to evaluate the estimation accuracy with different sampling patterns and network load conditions (number of flows, number of monitored flows, loss rate, etc.), but relevant results are still not available at this stage of the work.

4. Conclusions

In this paper we have presented 'Traffic Statistics', a new system able to collect information and statistics related to the traffic in a open network. The design of 'Traffic Statistics' system is presented using UML use case and class diagrams. Also, a preliminary implementation of part of it is proposed here. The 'Traffic Statistics' system has been conceived as a part of the NOX controller, and extends it providing a fundamental element for the development of cognitive networks. In this sense, our work constitute a key element for the creation and development of a Future Internet based on two main concepts: Software Defined Networking and Cognitive networking, as indicated by the Future Internet guidelines developed both at academic and industry level. It is worth noting that the work presented in this document will be integrated in the FI-WARE Core Platform, under development within the FI PPP supported and financed by the European Commission, as the authors of this work are members of the FI-WARE consortium. This means that the work presented in this document has been already preliminarily approved by European companies leading the Internet business and involved in the FI-WARE project (i.e., Telefonica, Orange, Telecom Italia, Nokia Siemens Networks, Ericsson, IBM, Alcatel-Lucent, etc.), as they see this work as a fundamental element for the Future Internet.

As application example of the proposed system, we may consider the use of 'Traffic Statistics' by a Content Distribution Network Service Provider, who can select the best route to send contents to its customers on the basis of real-time information about the traffic in the network. Another application could be related to the cognitive resource management of a network, where traffic information are the key inputs for routing, handover and in general for any other cognitive resource management mechanism. Many other application scenarios for 'Traffic Statistics' can be considered and illustrated, but they are out of the scope of this document.

At the time of the preparation of this document, the work presented here is incomplete, as several 'Traffic Statistics' components still need to be implemented and integrated with the NOX controller, and at the same time several open issues like scalability have to be properly addressed. Anyway, the 'Traffic Statistics' design is complete and it is important to underline that it has been realized in a technology independent way. Thanks to this, the proposed design can be considered as a reference design for other possible implementation for different Network Operating Systems and/or different open networks not based on the OpenFlow protocol. In the next months, the authors will be continuing working on the development of the whole 'Traffic Statistics' system, and the individuated open issues will be subject of investigation.

References

[1]: Talbot, D.: The Internet is broken, Technology Review, December 2005-January 2006 (2006), http://www.technologyreview.com/article/16356/

[2] FISS: Introduction to content centric networking, Bremen, Germany, 22 June (2009), http://www.comnets.uni-bremen.de/typo3site/uploads/media/vjCCNFISS09.pdf

[3] Miller., R.: Vint Cerf on the Future of the Internet. The Internet Today, The Singularity University (2009), http://www.datacenterknowledge.com/archives/2009/10/12/vint-cerf-on-the-future-of-the-internet/

[4] Future Internet Public-Private Partnership web site: http://www.fi-ppp.eu/

[5] EU FP7 FI-WARE project web site: http://www.fi-ware.eu/

[6] Goth, "Software-Defined Networking could shake up more than packets", IEEE Internet Computing, Vol.5, Issue 4, July-August 2011, pp. 6-9

[7] McKeown et al., "OpenFlow: enabling innovation in campus networks", March 2008

[8] Open Networking Foundation web site: http://www.opennetworkingfoundation.org/

[9] Castrucci et al., "A Cognitive Future Internet Architecture", Future Internet Assembly, LNSC 6656, pp.91-102, 2011

[10] Gude et al., "NOX: towards an operating system for networks"

[11] Sherwood et al., "FlowVisor: a network virtualization layer", October 2009

[12] M. Casado et al., "Ethane: Taking Control of the Enterprise", SIGCOMM'07, August 27–31, 2007, Kyoto, Japan

[13] EU FP7 SPARC project web site: http://www.fp7-sparc.eu/