Guide to the ONF Transport API (onf2016.189.04)

# Transport API Overview

Operators today are looking towards Software Defined Networks (SDN) to enable programmability of their networks for efficiency, speed of deployment and new revenue-generating network services. Widespread adoption of the programmability paradigm depends on the availability of common or standardized APIs which allow access to domain specific attributes and mechanisms without requiring the API itself to be specific to the vendor or technology. The ONF Transport API is designed to allow network operators to deploy SDN across a multi-domain, multi-vendor transport infrastructure, extending programmability across their networks end-to-end.

ONF T-API is designed to be the interface between controllers at different levels of an SDN controller hierarchy, offering control over network resources at different levels of abstraction. A typical deployment would be as the interface between the Domain Controllers for several network domains and a higher level SDN Network Orchestrator that acts as a parent controller.

The set of primitives needed to control a transport network have been explored in previous studies and prototype implementations [OIF, ONF]:

* Topology, or retrieval of network topology, resource availability and status information
* Service, allowing the client to create connectivity between service endpoints or request path computation for future connectivity services
* Notification, providing information about network events and changes to subscribed clients
* Virtualization or slicing of the network into isolated virtual partitions for specific clients or applications

This set of primitives is implemented in the ONF T-API in ways that allow for varying degrees of information and control based on operator policy. This range allows for different business relationships to be supported by the API, including clients that may be allowed very limited “black box” type views of the network to internal operator applications that may need full details of the network topology and ability to trace services across this topology.

ONF TAPI work is based on the ONF’s Common Information Model [CIM]. The ONF-CIM provides a representation of data plane resources for the purpose of management-control by the operator. The ONF TAPI derives its Information Model by pruning and refactoring the CIM Core Information Model as a purpose-specific realization of the ONF-CIM for Transport Network Control.

## T-API Documentation and Code

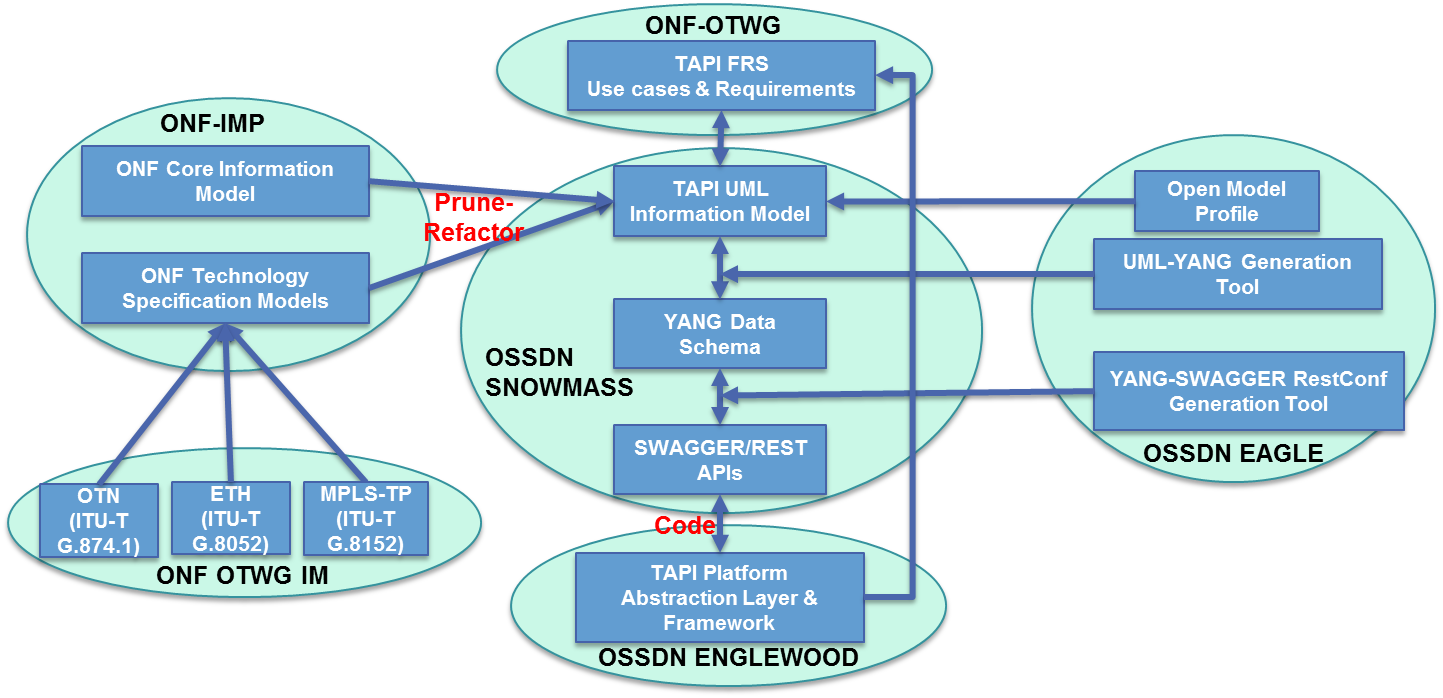


Fig. xx: TAPI components and related projects

The ONF T-API work includes several documentation and code components, including an ONF Technical Document and several modeling documents/software components that make up the T-API Software Development Kit (SDK) and which are documented through ONF’s Snowmass Open Source SDN initiative. There are also associated software tool development efforts that are supported by their own Open Source SDN projects.

### T-API Functional Requirements (ONF TR-527)

The ONF T-API Functional Requirements Document (FRD) is based on the analysis of several use cases for SDN transport network control, using the functional requirements to support these use cases to drive definition of necessary APIs. The APIs are described in terms of their basic input variables, outputs, and associated notifications. While based on use cases, the APIs are defined with the intent of not precluding additional future use cases and functions.

### T-API Software Development Kit (SDK)

The T-API SDK includes the UML Information Model, the Yang data schema and JSON Swagger specifications. Software tools described in the next section allow automated mapping from protocol-independent UML into Yang, and from Yang into JSON Swagger. Python and Java code can be generated directly from Swagger using industry tools.

* T-API UML Information Model: The UML Information Model takes the Functional Requirements and maps them into common objects and attributes applicable across transport technologies and vendors. UML allows this model to be expressed in a protocol=independent manner and mapped to different interface types as needed. The TAPI UML model can be found at the ONF OS SDN Snowmass repository.
  + The T-API UML Information Model is based on the ONF’s Common Information Model project and was derived through a pruning and refactoring of the ONF Core Information Model to fit the needs and terminology of transport networks.Ref guidelines?
  + The T-API UML Information Model was developed using the Papyrus editing tool. Directions for use of Papyrus can be found in [xx].
* T-API YANG data schema: The T-API Yang data schema is a mapping of the Information Model from UML into Yang (IETF RFC 6020) data modeling language, which results in a model expressed in text in a form that is gaining wide usage in the industry.
  + It should be noted that the UML model can be mapped into other data modeling languages as well as Yang. Efforts are underway, for example, in the Papyrus community.
* T-API Swagger specifications: The T-API Swagger specifications provide a mapping from the Yang data schema into JSON Swagger, which can then be used to generate Python and Java code for implementation of the API in RestConf.

### Associated Software Tools

* Open Source Eagle Project: The Eagle Project (<http://opensourcesdn.org/projects/project-eagle/>) is an open source project under ONF OpenSourceSDN to develop tools for automation of the development of code from abstract UML models, including:
  + Tools for automated mapping of UML to YANG
  + Tools for automated mapping of YANG to JSON Swagger
  + Tools for automated generation of code from JSON Swagger
* Open Source Englewood Project: The Englewood Project (<http://opensourcesdn.org/projects/project-englewood/>) is an ONF OpenSourceSDN project aimed at developing software modules that can be used to prototype, test, validate and facilitate the deployment of standard ONF Transport APIs over open source controller software such as ODL, ONOS and IDE.

## T-API Services and Modules

T-API has a design that allows different features to be realized as self-contained, largely independent modules that can be implemented separately to achieve a desired service, and through the modular design can evolve independently. The different service modules in the current version are described below.

## T-API modules

### Common Core

The TAPI Common Core module defines core objects that are used by all other TAPI modules, such as Context, etc. The Common Core module includes the Tapi UML file, Tapi.yang file, and Tapi.swagger file

### Topology

The Topology API module supports retrieval of Topology, Node, Link & Edge-Point details. It includes the TapiTopology UML file, TapiTopology.yang file, and TapiTopology.swagger file

### Connectivity Service

The Connectivity Service API module allows the client to retrieve information about and request new P2P, P2MP, MP2MP connectivity service across (L0/L1/L2) layers. It includes the TapiConnectivity UML file, TapiConnectivity.yang file, and TapiConnectivity.swagger file

### Notification

The Notification Service API module allows the client to subscribe to and filter autonomous notifications from the server for events such as failure or degradation. Autonomous notifications are carried via a transport mechanism as discussed under Implementation Guidelines below. The API includes the TapiNotification UML file, TapiNotification.yang file, and TapiNotification.swagger file

### Path Computation

The Path Computation API module allows the client to make a request for Computation & Optimization of paths. It includes the TapiPathComputation UML file, TapiPathComputation.yang file, and TapiPathComputation.swagger file

### Virtual Network Service

The Virtual Network Service API module allows the client to create, update, delete Virtual Network topologies. It includes the TapiVirtualNetwork UML file, TapiVirtualNetwork.yang file, and TapiVirtualNetwork.swagger file

### Technology-Specific Specs

Multiple Technology-Specific Spec models have been defined for different transport technologies such as Ethernet and OTN. The associated files include:

* TapiEth UML file, TapiEthSpec.yang file, and TapiEthSpec.swagger file
* TapiOch UML file, TapiOchSpec.yang file, and TapiOchSpec.swagger file
* TapiOdu UML file, TapiOduSpec.yang file, and TapiOduSpec.swagger file

TAPI work is being followed by other groups interested in SDN control of carrier networks, especially the OIF and MEF. OIF and ONF jointly sponsored a prototype demonstration of Transport SDN in 2014 which jump-started work on the TAPI standards at ONF, and OIF has continued to work closely with ONF on TAPI, contributing work on Virtual Network Services and planning on a joint interop demonstration in Fall 2016.

MEF has created an open source project called OpenConnectivityServices (OpenCS). “OpenCS is a MEF-facilitated ecosystem that provides reference implementations of MEF-defined connectivity services (e.g. E-Line) using combinations of Open Source software, Open Spec hardware, SDN, NFV and CE 2.0-certified devices” [MEF site]. TAPI fits into the MEF LSO “Presto” interface that allows Service Orchestration Functionality to communicate with Infrastructure Control and Management to manage the network infrastructure.

The ONF Common Information Model, which is the root of the TAPI model, has even more wide-ranging ties to other SDOs and Forums, being incorporated by ITU-T Study Group 15 into Recommendation G.7711 and being a component of multi-SDO coordination with ITU-T, MEF, TMF and ETSI NFV through the IISOMI (Informal Inter SDO Open Model Initiative) activity.

# Implementation Guidelines and templates

## RestConf RPC and/or REST flavors

The JSON Swagger specifications allow for both RPC and REST flavors of implementation. Although RPCs are useful from a documental point of view and to provide an optimization at the interface in terms of message exchange, the following considerations arise:

* RPC as well as REST operations will have both the same consequence: a modification of the underlying configuration and data tree, as defined by the T-API Yang modules. The same final configuration can be achieved in both ways.
* In a multi-supplier context, direct REST operations assure a better uniformity with respect to an RPC call, where there is a stronger dependency on the internal implementation of the specific called procedure at server side.
* Using RPC assumes a certain level of masking the complexity of the underlying data structure. However, the client application has to be aware in any case of the actual data structure it is operating on. For instance, for resynchronization and consistency check purposes, the client application shall be able to rebuild its view starting from the data information as retrieved by the server through the interface. This means that at the end, there’s no actual simplification of the view at client side since the knowledge of the detailed data tree is in any case necessary.
* Also from the server point of view, supporting both mechanisms would mean implement twice the same level of functionality with a different level of complexity without a real added value. The apparent simplification of the message exchange at the interface would be compensated by the additional processing of the specific RPC. It is therefore not granted that the overall performances at the interface, evaluated taking into account not only the message efficiency but also the underlying processing at server side, are improved using RPCs rather than a set of REST operations.

NOTE: for the 2016 OIF/ONF SDN Transport API Demo, it was agreed to use the REST paradigm rather than RPC based on a survey of participants.

## Notification: websockets vs. SSE

Two methods are possible for transport of event notifications in the Notification Service, websockets and SSE.

SSE has some advantages over Websockets:

* They are transported over simple HTTP instead of a custom protocol
* They can be poly-filled with javascript to "backport" SSE to browsers that do not support it yet
* They provide a built in support for re-connection and event-id
* They rely on a simpler protocol Additionally, based on our experience, we are confident that SSE and REST operate together seamlessly. A single connection (ip:port) with the web server can carry SSE notifications upstream and REST operations (HTTP GET/POST/PUT/PATCH) downstream.

Websockets have an advantage over SSE in terms of support of bidirectionality and real-time communications, as well as native support by the most modern browsers. In the T-API context, the notifications are inherently unidirectional (from the server to the client), so the bidirectionality aspect becomes not relevant for the choice.

As Websockets are bidirectional, it’s easy to use as well the same connection with the web server to carry both JSON downstream and events upstream, but this might require modifications in both the client and the server (possibly also on the configuration of the servlet container). It’s highly possible that it could be necessary to use two different TCP ports to support bidirectionality: one for the upstream Websocket and the other one for the downstream REST.

Ultimately both methods are possible and there is no strong argument favoring one over the other.

NOTE: for the 2016 OIF/ONF SDN Transport API Demo, it was agreed to use websockets with two different TCP ports to support Notification Service, based on a survey of participants.

## UUID

A UUID (Universally Unique Identifier) is generally used as a reference to global type objects during the interaction between client and server in a TAPI interface. The UUID is a 128-bit number with format defined in IETF RFC 4122 (ITU-T Rec. X.667 / ISO/IEC 9834-8).

## Context

Context is shared knowledge that exists prior to the establishment of a TAPI interaction, including the set of Service EndPoints known to both client and server. As such, Context cannot be created or deleted through TAPI interaction.

## Versioning

### Extension Mechanism

* The TAPI Core Spec is designed to be fully extensible with technology, SDO, operator or vendor specific *extensions* by application of following patterns
  + *Specialization* – via standard UML inheritance (sub-classing)
  + *Specification* – via UML composition decorated with *OpenModelProfile* stereotypes - <*DefinedBySpec*>, <*SpecReference*>, <*SpecTarget*>
* TAPI Core defines following base classes to model *specification* pattern
  + *Service-Spec* – Used to specify different types of TAPI *services*
  + *Resource-Spec* – Used to specify different types of TAPI *resources*
  + *Extensions-Spec* – Used to specify *extensions* to standardized TAPI *services/resources*
* All concrete TAPI *services* and *resources* are themselves *specializations* of *Service-Spec* & *Resource-Spec*
  + All concrete TAPI *global*/*local* classes (*services/resources)* can be extended via an *extensions* attribute by *specializing* the *Extensions-Spec*

# API examples – applied to a use case

## Reference network as on github site

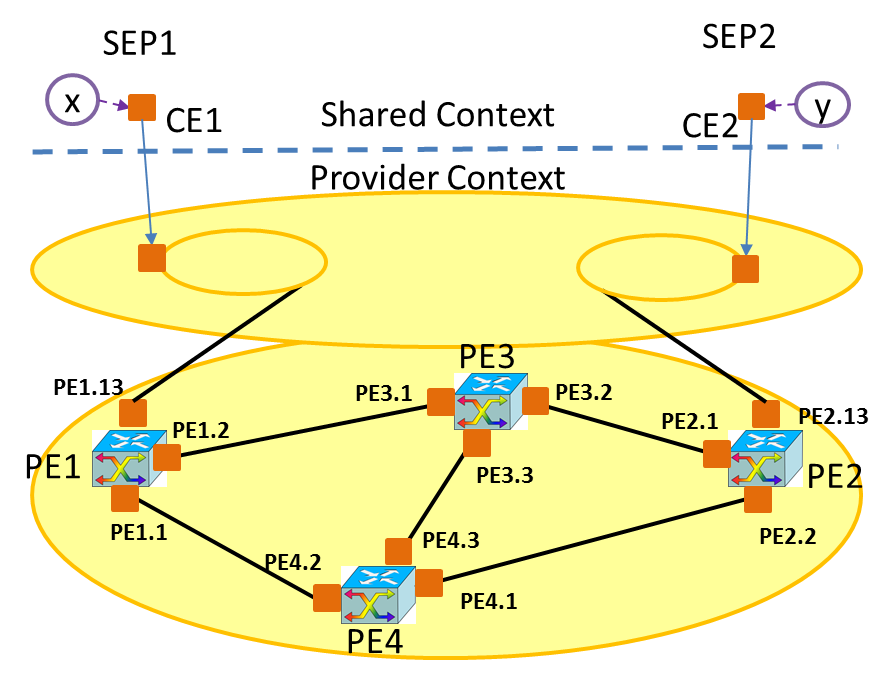


Figure 1 Proposed reference network

SCENARIO JSON:

<https://github.com/OpenNetworkingFoundation/Snowmass-ONFOpenTransport/blob/develop/DEMO/server_backend_state.json>

In order to load this scenario in T-API RI, please use the following REST call:

curl -X POST -H "Content-Type: application/json" <http://127.0.0.1:8080/backend/load_state/>

## API calls, input & output

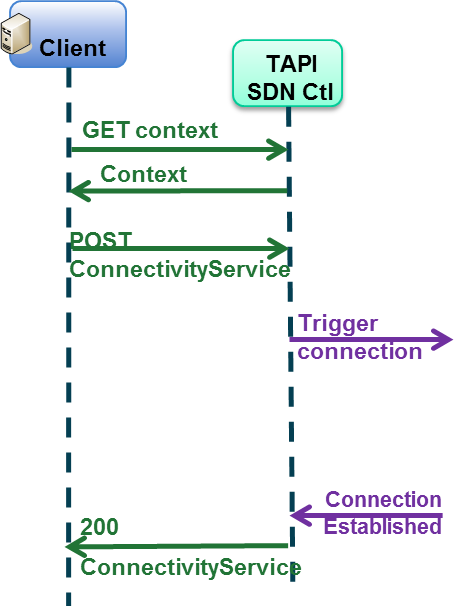


Figure 2 Connectivity Service workflow

### Get Context

curl -X GET --no-keepalive -H "Content-Type: application/json" http://127.0.0.1:8080/restconf/config/Context/

### Get topologies

curl -X GET --no-keepalive -H "Content-Type: application/json" http://127.0.0.1:8080/restconf/config/Context/\_topology/

### Get topology Top0

curl -X GET --no-keepalive -H "Content-Type: application/json" http://127.0.0.1:8080/restconf/config/Context/\_topology/top0/

### Get nodes

curl -X GET --no-keepalive -H "Content-Type: application/json" http://127.0.0.1:8080/restconf/config/Context/\_topology/top0/\_node/

### Get node 1

curl -X GET --no-keepalive -H "Content-Type: application/json" http://127.0.0.1:8080/restconf/config/Context/\_topology/top0/\_node/node1/

### Get links

curl -X GET --no-keepalive -H "Content-Type: application/json" http://127.0.0.1:8080/restconf/config/Context/\_topology/top0/\_link/

### Get link 1-3

curl -X GET --no-keepalive -H "Content-Type: application/json" http://127.0.0.1:8080/restconf/config/Context/\_topology/top0/\_link/link13/

### Get Service End Points

curl -X GET --no-keepalive -H "Content-Type: application/json" http://127.0.0.1:8080/restconf/config/Context/\_serviceEndPoint/se1/

curl -X GET --no-keepalive -H "Content-Type: application/json" <http://127.0.0.1:8080/restconf/config/Context/_serviceEndPoint/se2/>

### Create Connectivity Service

curl -X POST --no-keepalive -H "Content-Type: application/json" http://127.0.0.1:8080/restconf/config/Context/\_connectivityService/cs1/ -d'{"uuid":"cs1", "\_connConstraints":{"serviceType":"POINT\_TO\_POINT\_CONNECTIVITY", "serviceLayer":["OCH"] }, "\_servicePort":[ { "localId":"sp1", "serviceLayer":"OCH" , "direction":"BIDIRECTIONAL", "role":"SYMMETRIC", "\_serviceEndPoint":"http://127.0.0.1:8080/restconf/config/Context/\_serviceEndPoint/se1"}, { "localId":"sp2", "serviceLayer":"OCH" , "direction":"BIDIRECTIONAL", "role":"SYMMETRIC", "\_serviceEndPoint":"http://127.0.0.1:8080/restconf/config/Context/\_serviceEndPoint/se2"} ] }'

### Get Connectivity Services

curl -X GET --no-keepalive -H "Content-Type: application/json" http://127.0.0.1:8080/restconf/config/Context/\_connectivityService/

curl -X GET --no-keepalive -H "Content-Type: application/json" <http://127.0.0.1:8080/restconf/config/Context/_connectivityService/cs1/>

### Remove Connectivity Services

curl -X DELETE --no-keepalive -H "Content-Type: application/json" http://127.0.0.1:8080/restconf/config/Context/\_connectivityService/cs1/

## JSON data/captures

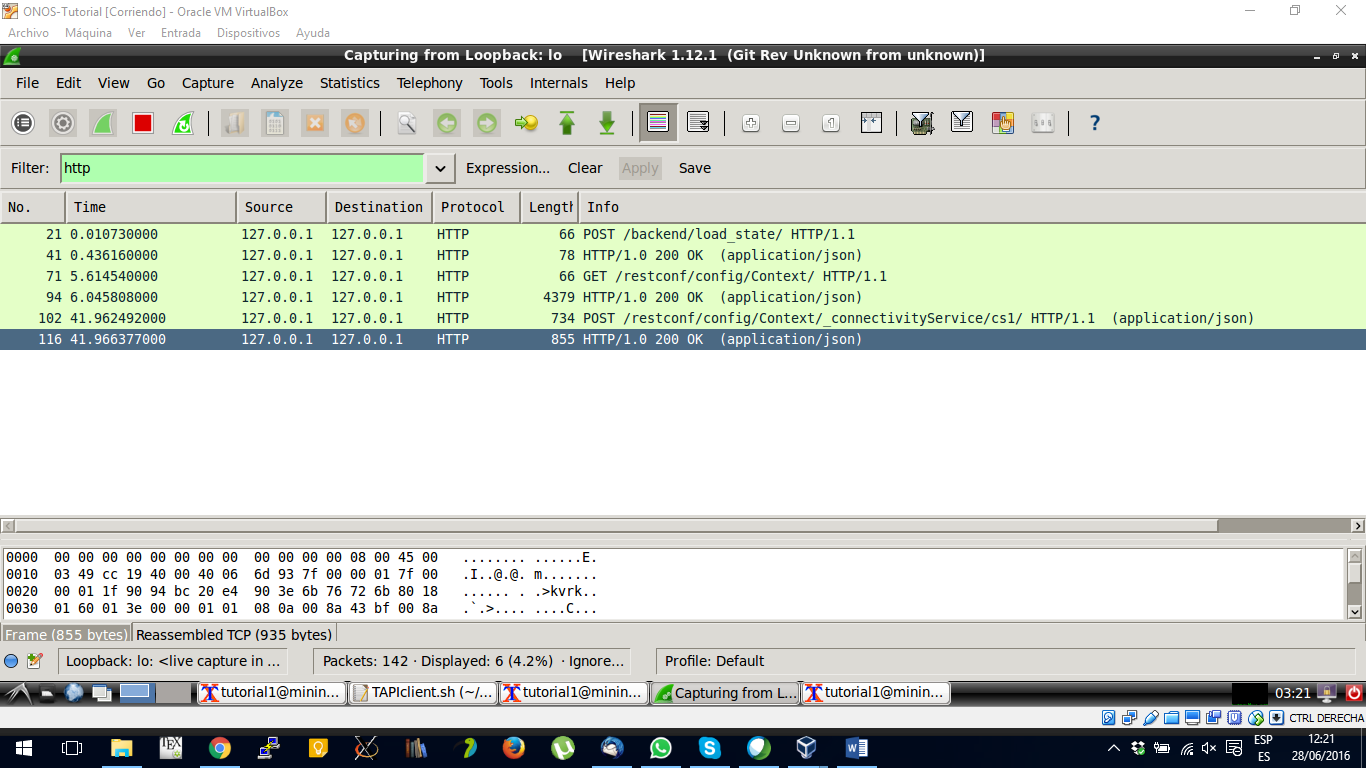


Figure 3 Wireshark capture of T-APImessage exchange

In this section, we provide a few wireshark captures (Figure 3 and Figure 4). The reference implementation in Snowmass repository allows to capture the necessary data.

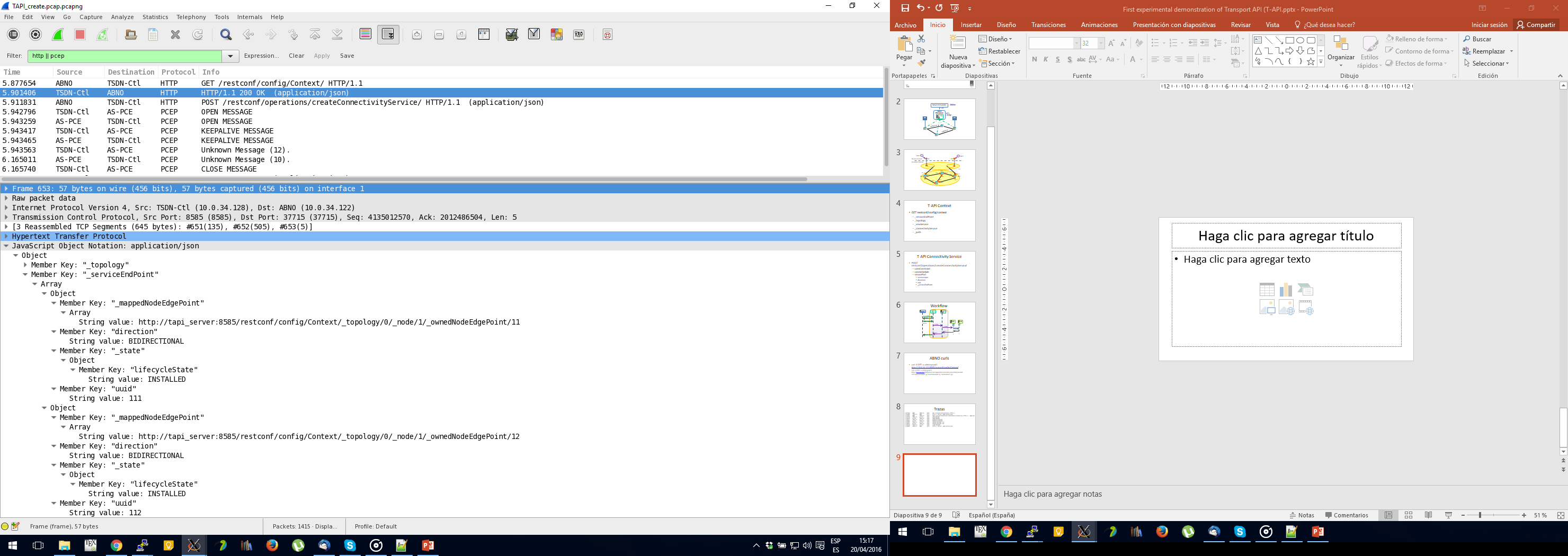


Figure 4 Context JSON

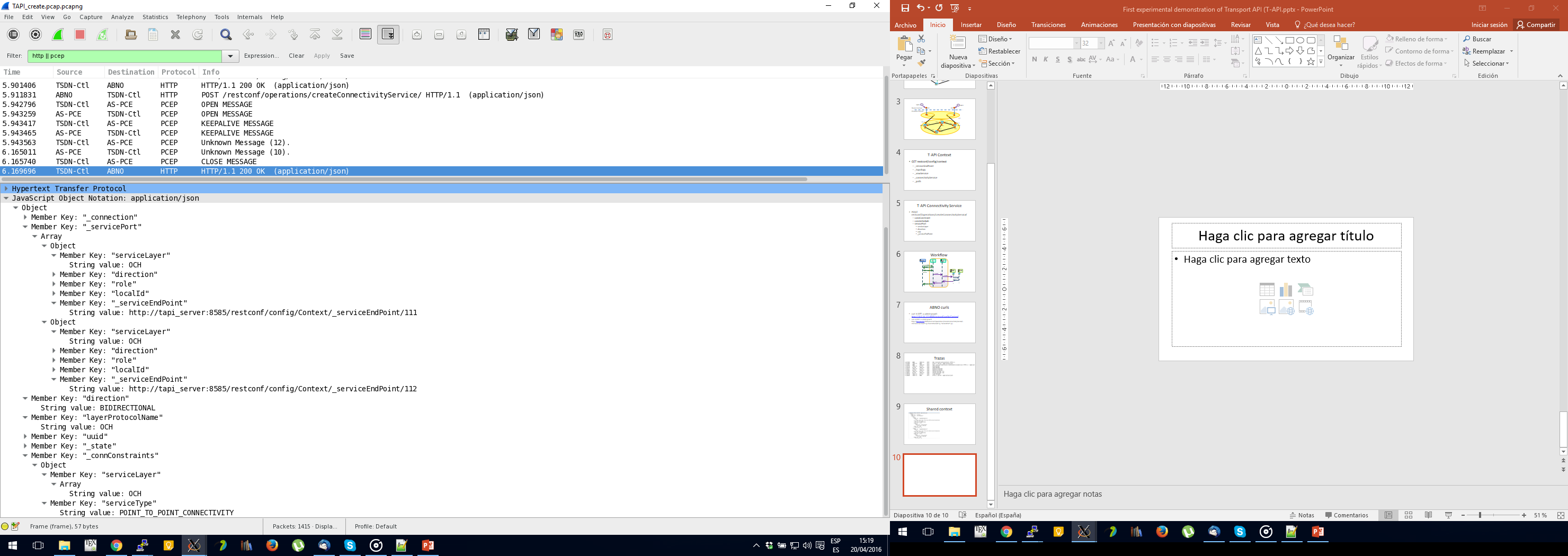


Figure 5 Connectivity Service JSON

## Notifications

Notifications will be provided with Websockets. T-API reference implementation provides an example of a client accessing the websocket, as seen in Figure 6.

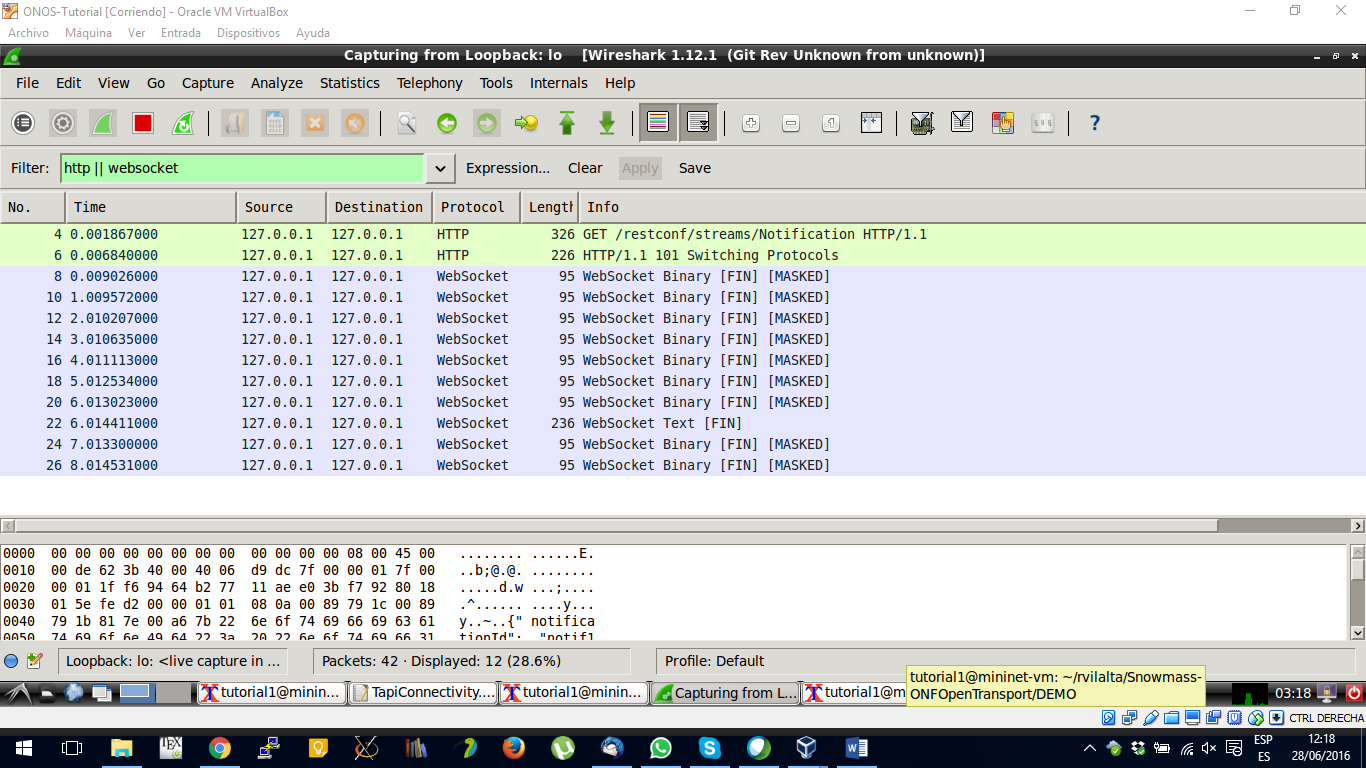


Figure 6 Wireshark of websocket for notification

# Eclipse plug-ins to view TAPI and how to install them

## Papyrus

Instructions for downloading and using Papyrus to view ONF Information Models and associated files can be found in ONF TR-515 (<https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/Papyrus_Guidelines_1.1-1.pdf>)

## XORED YANG plug-in

## JSON Tools plug-in

# Compliance/conformance

* + TBD

# What’s Next

## Known limitations/deficiencies

* Direct mapping from UML to JSON is a project under discussion in the Papyrus community.

# Contributors

Lyndon Ong (ed.)

Karthik Sethuraman

Ricard Vilalta

Paul Doolan

Germano Gasparini

Victor López