D1.7 - Network planning tool prototype

Status and Version:	Network planning tool prototype v.2	
Date of issue:	16.10.2015	
Distribution:	Public	
Author(s):	Name	Partner
	Luis Velasco (Editor)	UPC
	Lluis Gifre	UPC
	Gabriel Junyent	UPC
	Jaume Comellas	UPC
	Marc Ruiz	UPC
Checked by:		

Abstract

D1.7 reports the final status of the network planning tool prototype developed within the project. The deliverable includes an executable file with the final version of the PLATON planning tool and this document, which includes both a configuration and programming guide together with a performance analysis of the planning tool.

Contents

1 Executive summary

Dynamicity at the optical layer has been kept rather limited so far as a result of the large traffic aggregation performed in the upper network layers. Hence, optical transport networks are currently statically configured and managed. In fact, long planning cycles upgrade and prepare optical transport networks for the next planning period, where spare capacity is usually installed to ensure that traffic forecast and failure scenarios can be supported. Nevertheless, forecasts predict huge yearly global growth due to the introduction of new services such as live-TV distribution, datacenter interconnection, etc.

In addition, changes in traffic will affect not only its volume but also its distribution. Periodical planning needs from as exact as possible predictions for the expected traffic volume and distribution, which, although feasible for static traffic scenarios, is unreal when dynamic traffic is considered. Hence, efficient planning methods are needed to increment the capacity of optical networks while reducing overprovisioning costs.

Aiming at reducing network expenditures, a pay-as-you-grow approach can be implemented to add capacity to the network according with traffic growth. Let us assume that a periodical planning cycle is in charge of the design of the network; as a result, new network nodes can be installed and a reduced number of spare equipment (e.g. line-cards and transponders) can be purchased and made available in some warehouses distributed over the geography or even placed on-site. In addition, just-in-time (JIT) techniques can be used to keep enough spare cards always available. Spare equipment availability is often stored in an inventory database, together with information about optical cables, optical amplifiers, fiber usage, etc. When the capacity of the optical backbone network becomes exhausted in some parts, new capacity needs to be installed.

Dealing with traffic dynamicity requires automating connection provisioning, which explains the development of centralized architectures based on the software-defined networking (SDN) concept, such as the application-based network operations (ABNO) one [RFC 7491]. Operating the network dynamically might bring cost savings but it also might cause non-optimal network resource utilization. To solve that, network resources can be made available by applying *in-operation network planning*.

It is clear from the above that the classical network life-cycle has to be extended to support both on-demand incremental capacity planning and in-operation planning. Figure 1 presents the proposed augmented network life-cycle. Once the network is in operation, its performance is monitored so either incremental capacity planning or in-operation planning can be triggered when a threshold has been exceeded.

To add a new link, the planning algorithm needs to know the current state of the network including the state of the resources and established connections. Furthermore, it needs information about physical resources, even those not yet installed. A planning tool can decide the equipment and connectivity to be installed at the minimum cost; the planning tool needs access both, the inventory database and the current state of the network stored in operation databases, i.e., the TED and the LSP-DB. Note that those databases are also needed for in-operation planning to compute re-optimizations.

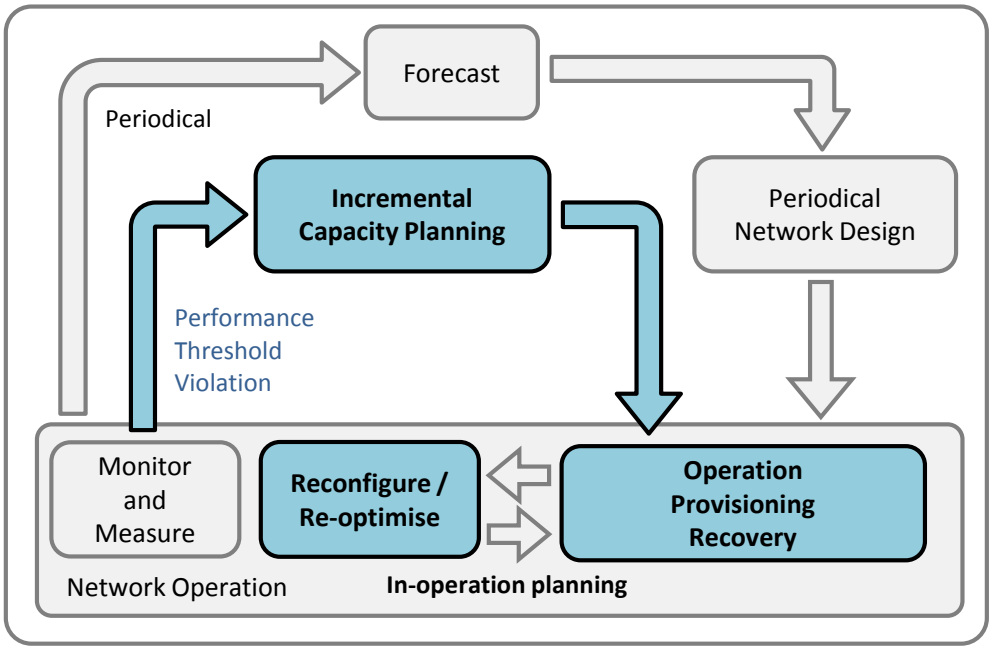

Figure 1: Extended network live-cycle.

Figure 2 presents an architecture to support both on-demand incremental capacity planning and in-operation planning. The architecture might support any planning algorithm that need to access both, operational and inventory databases. A centralized management element, e.g. ABNO, has global view of the resources and network topology as well as of the established connections. The central element is our PLATON planning tool that receives planning requests from the NMS and the PCE inside ABNO. To access data stored in the operational databases, the planning tool can be implemented as a back-end PCE and use BGP-LS and PCEP protocols.

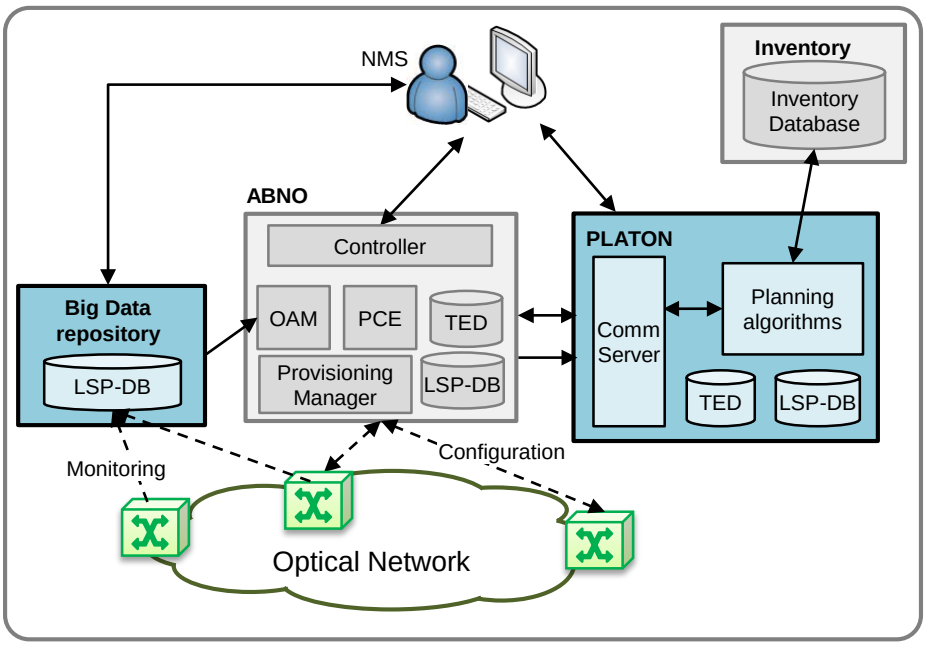

Figure 2: Management architecture.

The following table shows PLATON's development plan along the proyect.

Task	%Done
PLATON	100%
Architecture	100%
REST/API	100%
PCEP	100%
BGP-LS	100%
Manager	100%
Optimization Framework	100%
Algorithms	100%
After Failure Repair Optimization	100%
Spectrum Defragmentation	100%
Multicast Provisioning	100%

This document presents the final architecture of the PLATON network planning tool, as the configuration and programming guides, and the evaluation of the performance for the implemented algorithms.

Together with this document, a Linux version of PLATON, a set of configuration and include files, and an example of algorithm developed in C++ are delivered.

2 Introduction

2.1 Purpose and Scope

This is part of the seventh deliverable from Work Package 1 of Idealist project that includes PLATON software. The motivation and an overview of this deliverable have been provided in the Executive Summary. The main document is organized into four main sections to cover the architecture, configuration, programmation, and performance evaluation of PLATON planning tool.

2.2 Reference Material

2.2.1 Reference Documents

[1] IDEALIST Deliverable D1.2 “Network Planning Tool: Architecture and Software Design,” 2013.
[2] IDEALIST Deliverable D1.4 “Design and Tests of the On-Line Optimisation Framework,” 2014.
[3] D. King, A. Farrel, “A PCE-Based Architecture for Application-Based Netwowk Operations,” IETF RFC 7491, 2015.
[4] J. Ash, J.L. Le Roux, “Path Computation Element (PCE) Communication Protocol - Generic Requirements,” IETF RFC 4657, 2006.
[5] H. Gredler, J. Medved, S. Previdi, A. Farrel, S. Ray, “North-Bound Distribution of Link-State and TE Information using BGP,” IETF work in progress, 2014.
[6] IDEALIST Deliverable D3.2 “Design and Evaluation of the Adaptive Network Manager and Functional Protocol Extensions,” 2014.
[7] E. Crabbe, J. Medved, I. Minei, R. Varga, "PCEP Extensions for Stateful PCE," IETF work in progress, 2014.

2.2.2 Acronyms

ADT	Abstract Data Type
API	Application Programming Interface
BGP-LS	Border Gateway Protocol – Link State
bPCE	Back-end PCE
BVT	Bandwidth Variable Transponders
CAPEX	Capital Expenditures
CLI	Command-Line Interface
DB	Database
EC2	Elastic Compute Cloud
fPCE	Front-end PCE
GMPLS	Generalized Multi-Protocol Label Switching
GUI	Graphical user Interface
ILP	Integer Linear Programming
LSP	Label Switched Path

NMS	Network Management System
NP	Nondeterministic Polynomial time
OF	Objective Function
OPEX	Operational Expenditres
OSS	Operations Support System
PCC	Path Computation Client
PCE	Path Computation Element
PCEP	PCE communications Protocol
PCRep	Path Computation Reply
PCReq	Path Computation Request
PCRpt	Path Computation Report
PLATON	Planning Tool for Optical Networks
PLIs	Physical Layer Impairments
QoT	Quality of Transmission
RMLSA	Routing, Modulation Level and Spectrum Allocation
RSA	Routing and Spectrum Allocation
RWA	Routing and Wavelength Assignment
SaaS	Software as a Service
SBVT	Sliceable Bandwidth Variable Transponders
SEC	Spectrum Expansion/Contraction
SSH	Secure SHell
SVEC	Synchronization VECtor
TED	Traffic Engineering Database
UML	Unified Modelling Language
WDM	Wavelength Division Multiplexing
WP	Work Package

2.3 Document History

Version	Date	Authors	Comment
Draft	1.09.15	Luis Velasco	Contains placeholders for all contributions
Version 1	25.09.15	Lluis Gifre, Luis Velasco	First Integrated version. Contains a first version of PLATON architecture, configuration,

			programming guides and performance evaluation.
Version 2	16.10.15	Lluis Gifre, Luis Velasco	Second integrated version after minor revisions.
Version 3			

3 Architecture of the PLATON Planning Tool

In this chapter, the UPC's in-operation PLAnning Tool for Optical Networks (PLATON) architecture is described. The chapter is organized in two sections. Firstly, the architecture of PLATON is updated from the version presented in [1], [2] detailing its architectural components: the communication interfaces, the manager, the network databases, the algorithms framework, the workflow engine and workflow lifecycle. Next, the generic operations running inside PLATON are described including: the topology database update, the lightpath database update, and the computation request.

3.1 Architecture of PLATON

PLATON is implemented as a back-end Path Computation Element (bPCE) extending the classical PCE in the ABNO architecture [3]. The bPCE offloads the complex computations from the front-end PCE (fPCE) so as the latter only needs to deal with network handling and path provisioning.

The PLATON architecture, depicted in Figure 3, consists of 5 components: the communication interfaces, the manager, the network databases, the algorithms framework, and the workflow engine. Computation algorithms are defined as dynamically loadable workflows that use the offered public APIs enabling third parties to develop and plug their own algorithms.

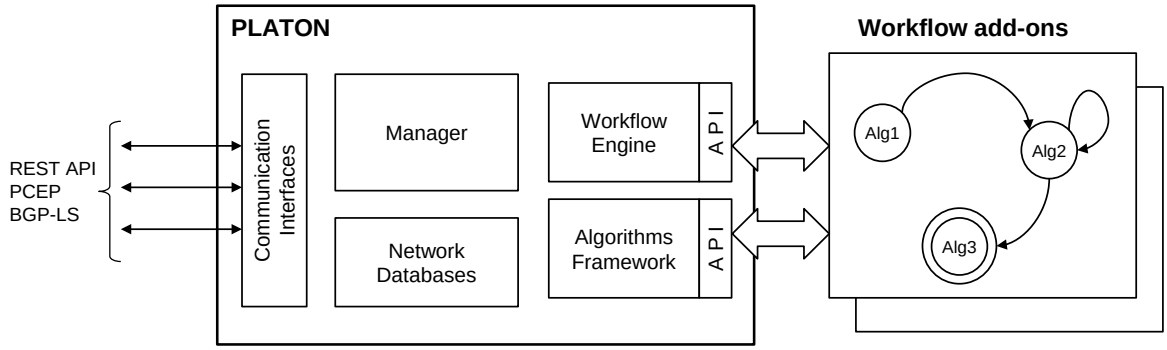

Figure 3.Architecture of PLATON

3.1.1 Communication Interfaces

The communication interfaces are the gateway for all incoming and outgoing messages encoded using standard protocols; these messages are used by PLATON to synchronize its internal databases and to handle computation requests. The supported protocols include REST API, PCE Protocol (PCEP) [4], and Border Gateway Protocol – Link State (BGP-LS) [5] protocols.

For the BGP-LS interface, PLATON operates as a BGP listener and attends for BGPUpdate messages. Each of these messages specifies the reachability or unreachability of a single node or link. To maintain the TED updated, all the BGPUpdate messages must be processed. PLATON's implementation of BGP-LS supports the advertisement of Sliceable Bandwidth Variable Transponders (SBVTs) using the Port Label Restriction extensions proposed in [6].

In contrast, the PCEP interface has been implemented supporting stateful extensions [7] to allow reception of Path Computation Report (PCRpt) messages, in addition to the standard PCReq and PCRep ones. Each received PCRpt message contains the attributes of sets of

lightpaths in the network, including the operation to be performed, i.e. creation, update, or deletion. To maintain the LSP-DB updated, all the PCRpt messages must be processed. Likewise, PLATON's implementation of PCEP supports the control subtransponders in SBVTs using the Explicit Transponder Control (ETC) sub-object proposed in [6].

The PCReq and PCRep messages are used to encode incoming computation requests and path computation replies, respectively. An OF object embedded in the PCReq message can be used to select the desired workflow to be instantiated by PLATON.

Finally, REST API implemented in PLATON supports messages encoded in XML and JSON. The RESTReq and RESTRep messages are used to encode incoming computation requests and path computation replies, respectively. The workflows can be bind to paths in the REST API server so that the desired workflow to be instantiated by PLATONcan be selected based on the REST API resource path specified by the user in the RESTReq message

Configuration details for this component are provided in section 4.2.

3.1.2 Manager

The Manager component is responsible for configuring and coordinating the rest of components. In addition, each message received by the communication interfaces is routed to the manager who creates a job, schedules it by kind and issues it to the corresponding component in the proper order, jobs belonging to the same type are processed in arrival order, to update databases, to trigger the instantiation of a new workflow, or to provide pending messages to running workflows.

The TED update jobs are executed with the highest priority, the LSP-DB updates with medium priority and the computation requests with lower priority. This prioritization ensures that the topology will be always up to date when a new lightpath needs to be updated and, similarly, the computation requests will always be executed with an updated copy of the TED and LSP-DB databases. Details on each kind of job are provided in section 3.2.

Configuration details for this component are provided in section 4.3.

3.1.3 Network Databases

The network databases component provides in-memory storage to PLATON. Currently implemented databases are the Traffic Engineering Database (TED) and the Label Switched Path Database (LSP-DB). Databases can be preloaded with specific topologies and paths. Convenient methods and functions to manage database contents, e.g. create node, create link, connect node to link, create lightpath, allocate lightpath, etc. are provided.

Messages from the communication interfaces component are used to update databases. In addition, the network databases can be configured to synchronize their changes to other external modules. For instance, BGPUpdate messages can be used to flood internal TED changes to other external modules, while PCRpt messages can be used for the LSP-DB.

Configuration details for this component are provided in section 4.4.

3.1.4 Algorithms Framework

The algorithms framework provides a set of methods and functions common to every workflow for agile algorithm development including routing algorithms, optical spectrum handling functions, metaheuristic frameworks, etc. Besides, protocol helper functions for rapid message handling are provided. All those functions are available through an API. Additional details of this API can be found in section 5.

3.1.5 Workflow Engine

Finally, workflows are implemented as finite state machines (FSM) in PLATON. Two elements are needed to add a new workflow to PLATON: a XML file defining the workflow and a dynamically linkable file containing a number of algorithms. Configuration details for this component and workflow definition are provided in section 4.5 and programming details for the dynamic linkable file are described in section 5.

Since workflows are defined as FSMs, a number of states and transitions between states are defined in the XML file; a different algorithm is executed in each state of the workflow. Workflow state algorithms use the algorithms API previously described.

The workflow engine, see Figure 4, is responsible for initiating new workflow instances, and coordinating their execution subject to incoming messages. The engine contains a message table storing identifiers of messages pending to be received and the workflow instance awaiting each message.

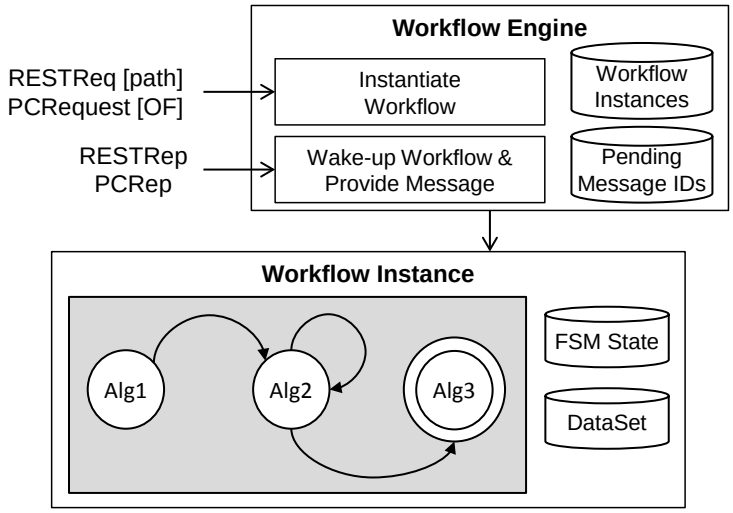

Figure 4. PLATON Workflow LifeCycle

Each workflow has: *i*) a triggering message to initiate the workflow, *ii*) the set of states, *iii*) the set of transitions between states based on incoming message types, *iv*) the initial state of the FSM, *v*) the algorithms to be run on each state of the FSM, and *vi*) an internal DataSet storing the workflow internal data, which can be defined by the workflow programmer. Workflows use the algorithms framework and workflow engine APIs to interact with PLATON.

Incoming RESTReq and PCReq messages trigger workflow instances creation. When RESTReq messages are received, the REST API resource path that user specified to issue the request is used to select the desired workflow. Besides, an Objective Function (OF) object can be included in PCReq messages to specify the desired workflow to be run; a default workflow must be defined for those PCReq messages without any OF object.

When a workflow is started, a workflow instance is created by the workflow engine containing the current state of its FSM and internal workflow data, thus enabling workflow concurrently.

To select the appropriate function in the dynamic library implementing a given workflow state algorithm, its name must be the same as the name of the state. Workflow state algorithms can execute any generic algorithm and send messages to other external modules using functions in the APIs. At the end of the state algorithm execution, the list of pending messages to be received is reported to the workflow engine, which updates the messages

table with the expected identifiers in RP and SRP objects. If no pending messages are reported, the workflow has been finished.

3.2 Operation Descriptions

In this section the internal operations performed by PLATON are described by means of sequence diagrams. Due to each kind of message need to perform different operations, one sequence diagram is provided for each kind of message.

3.2.1 Topology Update

Figure 5 depicts the sequence diagram for a topology update operation. In the event of receiving a BGP-LS Update message (labelled as 1 in Figure 5), the communication interface component requests a topology update to the Manager component (2). The Manager creates a new topology update job and schedules it for future execution (3). When the job is selected by the scheduler (4), it is executed (5) firstly checking the databases consistency (6) and then performing the TED update (7). When the update is completed, the job returns control to the scheduler (8) to select the next job to be executed.

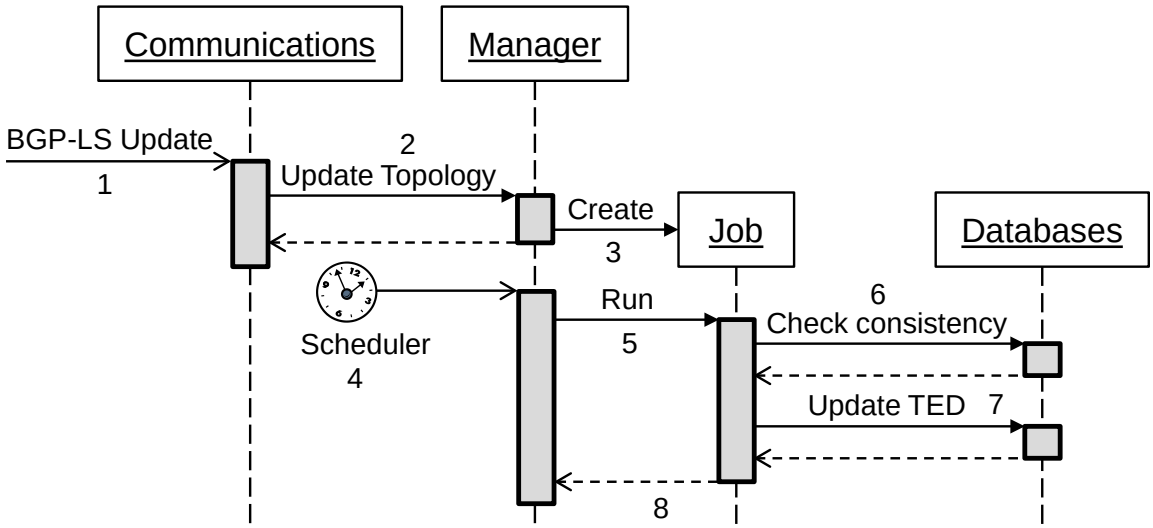

Figure 5. Sequence diagram for Topology Update

3.2.2 Lightpaths Update

Figure 6 depicts the sequence diagram for a lightpaths update job. In the event of receiving a PCEP PCReport message (labelled as 1 in Figure 6), the communication interfaces component requests a topology update to the Manager component (2). The Manager creates a new lightpaths update job and schedules it for future execution (3). When the job is selected by the scheduler (4), it is executed (5) firstly checking the databases consistency (6) and then performing the LSP-DB update (7). When the update is completed, the job returns control to the scheduler (8) to select the next job to be executed.

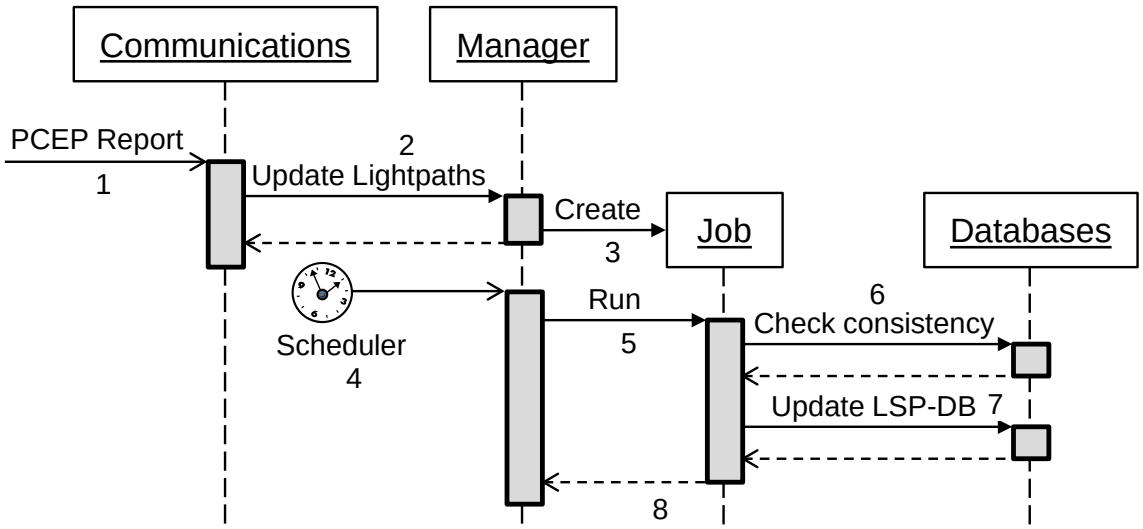

Figure 6. Sequence diagram for Lightpaths Update

3.2.3 Computation Request

Figure 7 depicts the sequence diagram for a compute algorithm job. In the event of receiving a PCEP Request (REST Request) message (labelled as 1 in Figure 7), the communication interfaces component requests an algorithm computation to the Manager component (2). The Manager creates a new compute algorithm job and schedules it for future execution (3).

When the job is selected by the scheduler (4), it is executed (5). Job execution requests to the workflow engine to process the message related to the job (6). The workflow engine detects that it is a computation request, so it instantiates a new workflow (7) selecting its type based on OF object contained in the request (the REST API resource path) and issuing the message to that new workflow instance (8). The workflow instance selects the proper state algorithm (9) as function of the current state's name and, when the algorithm is ready, it is executed passing the received message as parameter (10).

The algorithm is organized in three steps: first the input message is transcoded into an internal workflow instance's dataset (11), next the computation algorithm itself is executed over the dataset (12) and then, the solution in the dataset is transcoded (13) and replied by means of a PCEP Reply (REST Reply) message (14, 15). When the state algorithm finishes (16), the workflow instance notifies the identifiers of pending messages to be received (if any) (17). PLATON considers a workflow instance as finished when no pending messages are reported by the workflow instance. Finally, the workflow engine returns execution control to the job (18) and the latter to the Manager (19), who can select the next job to be executed.

Note that this workflow enables sending intermediate requests to other external modules such as an inventory database, and leaving the workflow instance in standby until the external module replies. In such cases, the received reply will enter through the communication interfaces (1) and will request a computation job to the Manager (2). The Manager will create and schedule a job (3). When the job is selected, the message will be passed to the workflow engine (6), who will locate the existing workflow instance that is waiting for the received message on its internal message routing table. Then, the received message will be issued to that workflow instance (8) forcing a state transition on its internal FSM, thus selecting a new state algorithm. The rest of the sequence is as described in the workflow instance creation.

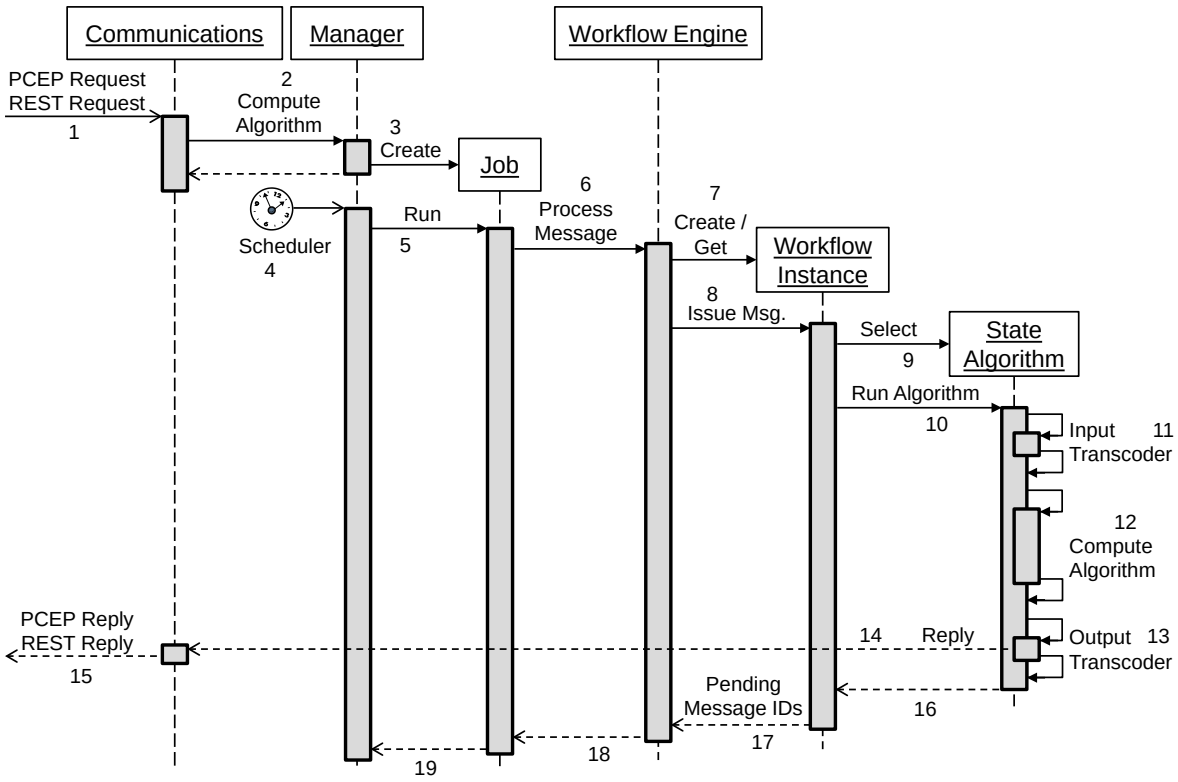

Figure 7. Sequence diagram for Compute Algorithm

4 Configuration Guide

The overall configuration of PLATON is done by means of XML files having well-defined sections for each of the architectural components.

In this chapter, the configuration details of PLATON planning tool are described. The chapter is organized by architectural components and configuration examples are provided for each of them.

4.1 PLATON main configuration file

PLATON main configuration file, see Table 1, is organized in 5 main sections: *i*) the `System` and `Log` sections for the module configuration, *ii*) the `Interfaces` section for the communication interfaces component, *iii*) the `StartupSequence` section for the Manager component, *iv*) the `Network` section for the network databases component, and v) the `WorkflowEngine` section for the workflow engine component.

Table 1. Config file– Sections

```
<iONEConfig>
  <System .../>
  <Log .../>
  <Interfaces> ... </Interfaces>
  <StartupSequence> ... </StartupSequence>
  <Network ...> ... </Network>
  <WorkflowEngine> ... </WorkflowEngine>
</iONEConfig>
```

The `System` and `Log` sections, detailed in Table 2, define, respectively, the module name, and the path of the log file to be used an the verbosity log level which can be set, in decreasing order of verbosity, to one of: `DEBUG`, `INFO`, `NOTICE`, `WARNING`, `ERROR`, `CRITICAL`, `ALERT`, `EMERGENCY`.

Table 2. Config file– Main Module configuration

```
<iONEConfig>
  <System module="PLATON"/>
  <Log filePath="logs/platon.log" level="INFO"/>
                                        ...
</iONEConfig>
```

4.2 Configuration Interfaces

The `Interfaces` section defines the set of interfaces to be set-up in PLATON. An arbitrary number of interfaces for each protocol can be configured using a XML file; Table 3 shows a fragment of such a file where two interfaces are defined. Interfaces can be defined as servers or client; when it is defined as server a list of client IPs is provided grouped as `TrustedPeers`, whereas when is defined as client, only one peer must be configured, the one configured as server. The XML file defines a number of protocol-specific parameters such as keep alive and hold timers.

Table 3. Config file– Communication Interfaces configuration

```
<iONEConfig>
                                        ...
  <Interfaces>
    <Interface name="bgp" proto="bgp" role="server" ip="10.0.0.5" port="179">
      <BGPConfig>
        <KeepAlive max="50" min="0" default="40"/>
        <HoldTime max="150" min="0" default="120"/>
        <AS id="102"/>
        <BGPLS id="52"/>
      </BGPConfig>
      <TrustedPeers>
        <Peer ip="10.0.0.3"/>
        <Peer ip="10.0.0.4"/>
      </TrustedPeers>
    </Interface>
    <Interface name="pcep" proto="pcep" role="server" ip="10.0.0.5" port="4189">
      <PCEPConfig>
        <KeepAlive max="50" min="0" default="30"/>
        <DeadTimer max="200" min="0" default="120"/>
        <Capabilities stateful="true"
                      instantiation="true"
                      update="true"
                      synchro="true"/>
      </PCEPConfig>
      <TrustedPeers>
        <Peer ip="10.0.0.3"/>
      </TrustedPeers>
    </Interface>
  </Interfaces>
                                        ...
</iONEConfig>
```

4.3 Manager

The `StartupSequence` section configures the sequence of startup actions that the Manager has to execute at startup time; these actions include: setting up a protocol session on a specific interface, waiting for an incoming connection through a specific interface, introducing a specific delay in the startup sequence, etc. For instance, in Table 4 a BGP client interface is first started, connected to a server and a BGP session is established 1 second before the PCEP interface is started.

Table 4. Config file– Startup Sequence configuration

```
<iONEConfig>
                                        ...
  <StartupSequence>
    <Step action="startInterface" interface="bgp"/>
    <Step action="waitForSession" interface="bgp"/>
    <Step action="delay" delayMS="1000"/>
    <Step action="startInterface" interface="pcep"/>
    <Step action="waitForSession" interface="pcep"/>
  </StartupSequence>
                                        ...
</iONEConfig>
```

4.4 Network Databases

The `Network` section configures the network databases component. Currently implemented databases include the TED and the LSP-DB. The databases can be pre-loaded by means of XML files and configured to synchronize their content through specific communication interfaces, as shown in Table 5. For instance, a BGP-LS interface can be configured to handle BGPUpd messages to synchronize the TED, while a PCEP interface can synchronize the state of the LSP-DB using asynchronous PCRpt messages.

Table 5. Config file – Network Databases configuration

```
<iONEConfig>
                                        ...
  <Network spectrumSizeGHz="4000" granularityGHz="6.25" minSpAllocGHz="37.5">
    <Topology filePath="data/topology.xml"/>
    <LSPDB filePath="data/lspdb.xml"/>
    <NotifyUpdatesTED   enabled="true" interface="bgp"/>
    <NotifyUpdatesLSPDB enabled="true" interface="pcep"/>
  </Network>
                                        ...
</iONEConfig>
```

4.5 Workflow Engine and Workflow Definition

The `WorkflowEngine` section, shown in Table 6, configures the workflow engine component and defines the set of workflows to be loaded by PLATON specifying the path to the XML file containing the definition of the workflow. A default OF code for those PCReq messages not containing an OF object, can be defined using the tag named as `DefaultPCReqOFCode`.

Table 6. Config file – Workflow Engine configuration

```
<iONEConfig>
                                        ...
  <WorkflowEngine messageRetryDelayMS="100" messageRetryMaxRetries="5">
    <Workflow filePath="workflows/platon_WF1_definition.xml"/>
    <Workflow filePath="workflows/platon_WF2_definition.xml"/>
    <Workflow filePath="workflows/platon_WF3_definition.xml"/>
```

```
    <DefaultPCReqOFCode ofCode="1"/>
  </WorkflowEngine>
                                    ...
</iONEConfig>
```

Each workflow is defined as an FSM, which is configured by means of an XML file as explained in section 3.1.5. The workflow definition file, example listed in Table 7, contains: *i*) the triggering message to initiate the workflow, *ii*) the set of states, *iii*) the set of transitions between states based on incoming message types, *iv*) initial state of the FSM, *v*) the path to a dynamic linked library implementing the algorithms to be run on each state of the FSM, and *vi*) a workflow state algorithms configuration file to define parameters and constants.

Table 7. PLATON Workflow Definition file

```
<Workflow name="WF3" libraryFile="platon_WF3.so"
                     configFile="platon_WF3_config.xml">
  <TriggeringMessage message="PCReq" ofCode="39"/>

  <State name="Alg1"/><State name="Alg2"/><State name="Alg3"/><State name="Err"/>

  <Transition fromState="Alg1" inputMessage="PCRep" toState="Alg2"/>
  <Transition fromState="Alg1" inputMessage="PCErr" toState="Err"/>
  <Transition fromState="Alg2" inputMessage="PCRep" toState="Alg2"/>
  <Transition fromState="Alg2" inputMessage="PCRpt" toState="Alg3"/>
  <Transition fromState="Alg2" inputMessage="PCErr" toState="Err"/>

  <InitialState name="Alg1"/>
  <ErrorState name="Err"/>
</Workflow>
```

5 Programming Guide

PLATON provides an API for implementing new workflows. The API has two parts: the first one defines the class signatures that each workflow must implement, while the second defines the set of functions and methods exported by PLATON to be used by the algorithms.

The API is designed to be used in a C++ for Linux environment. To implement algorithms using a different programming language or platform, a wrapper needs to be implemented by the user. All the C++ classes composing an algorithm must be compiled and linked together into a standard Linux shared object library.

In this chapter, the programming details to implement a new workflow for PLATON planning tool are described. The chapter is organized in three sections: firstly, the workflow interface is described, next the algorithms framework API is listed and finally, an example is provided to illustrate the implementation of a workflow.

5.1 Workflow Interface

Each workflow in PLATON, inherits from the generic `IWorkflowImpl` workflow listed in Table 8. New workflows must re-implement the set of class members of the generic `IWorkflowImpl` workflow detailed in Table 9. These definitions are the minimum set of functions to be implemented to construct a new workflow.

The `IWorkflowImpl` algorithm throws an exception in case of executing any of its functions, so leaving a workflow member not re-implemented will cause an exception to be thrown.

Table 8. PLATON `IWorkflowImpl` workflow

```
class IWorkflowImpl
{
  public:
    CREATE_EXCEPTION_KIND

    explicit IWorkflowImpl() { }
    virtual ~IWorkflowImpl() { }

    virtual void configure(const std::string& configFilePath)
    { THROW_EXCEPTION("Workflow does not implement method configure()"); }

    virtual void setNetwork(LN::Network* network)
    { THROW_EXCEPTION("Workflow does not implement method setNetwork()"); }

    virtual void setInterfaces(LI::Interfaces* interfaces)
    { THROW_EXCEPTION("Workflow does not implement method setInterfaces()"); }

    virtual void clear()
    { THROW_EXCEPTION("Workflow does not implement method clear()"); }

    virtual LWE::MessageSet runState(const std::string& stateName, LWE::Message* message)
    {
      THROW_EXCEPTION("Workflow does not implement method runState()");
      LWE::MessageSet emptyPendingMessageSet;
      return(emptyPendingMessageSet);
    }

    virtual void gracefulAbort()
    { THROW_EXCEPTION("Workflow does not implement method gracefulAbort()"); }
};
```

Table 9. Class methods to be implemented in a new workflow

(constructor) The constructor member must be used to allocate dynamic memory for the workflow internal data. When dynamic memory is not required by the workflow, this member should be re-implemented with an empty body.
(destructor) The destructor member must be used to deallocate the dynamic memory reserved by the workflow's constructor. When dynamic memory is not required by the workflow, this member should be re-implemented with an empty body.
`void setNetwork(LN::Network* network)` The `setNetwork` member must be used to acquire the pointer to PLATON's network databases to be used for the workflow's computations. If the workflow does not require access to the network databases, this member should be re-implemented with an empty body.
`void setInterfaces(LI::Interfaces* interfaces)` The `setInterfaces` member must be used to acquire the pointer to PLATON's communication interfaces to be used to issue requests to external modules during the workflow execution. If the workflow does not require issuing requests to external modules, this member should be re-implemented with an empty body.
`void configure(const std::string& configFilePath)` The `configure` member must be used to load the set of configuration parameters for the algorithm. When necessary, the `configFilePath` parameter can be used to provide the

algorithm's configuration path. If the algorithm does not require any parameter to be configured, this member should be re-implemented with an empty body.
`void clear()` The `clear` member must be used to initialize internal data structures after loading the workflow's configuration parameters. If the algorithm does not require initializing any data structure, this member should be re-implemented with an empty body.
`LWE::MessageSet runState(const std::string& stateName, LWE::Message* message)` The `runState` member must be used to execute the state algorithm corresponding to the current state in the workflow. The `stateName` parameter contains the name of the current FSM state, while the `message` parameter points to the incoming message to be passed to that state algorithm for further processing. This function is the core of the workflow execution, so it must be always re-implemented. Further details on implementing this function are given below.
`void gracefulAbort()` The `gracefulAbort` member must be used to gracefully terminate, e.g. by sending an error message to issuer of the request, the workflow in case of abnormal termination of PLATON execution. If the algorithm does not requires gracefull termination actions, this member should be re-implemented with an empty body.

The core of the workflow, the `runState` member, needs to execute the proper internal state algorithm member function to be executed for the current state. Finally, it returns the set of pending message types and identifiers that the workflow instance is waiting for. Table 10 lists an example of a `runState` method.

Table 10. `runState` member example

```
LWE::MessageSet MyWorkflow::runState(const std::string& stateName, LWE::Message* message)
{
       if(!stateName.compare("alg1")) alg1(message);
  else if(!stateName.compare("alg2")) alg2(message);
  else                                THROW_EXCEPTION("Unsupported state " << stateName);
  return(dataset.pendingMessages);
}
```

Then, each of the state algorithms, named as `alg1` and `alg2` in the previous example, needs to do three main steps: translate the incoming message into easily usable input dataset, run the computation algorithm itself, and translate the solution returned by the algorithm into a reply message. Implement the full algorithm functionality in a single member is usually tricky and hard to debug, so we split each `algX` member in three members: `algX_transcodeInput`, `algX_computeAlgorithm`, and `algX_transcodeOutput`. Table 11 lists an example of how to split the `run` member.

Table 11. `alg1` member example

```
void MyWorkflow::alg1(LWE::Message* message)
{
  try
  { alg1_transcodeInput(message); }
  catch (std::exception& e)
  { THROW_EXCEPTION("Unable to transcode input message. Exception: " << e.what()); }

  try
  { alg1_computeAlgorithm(); }
  catch (std::exception& e)
  { THROW_EXCEPTION("Unable to execute algorithm. Exception: " << e.what()); }
```

```
  try
  { alg1_transcodeOutput(); }
  catch (std::exception& e)
  { THROW_EXCEPTION("Unable to transcode output. Exception: " << e.what()); }
}
```

The `algX_transcodeInput` member translates the received message, e.g. a Path Computation Request (PCReq), into programmer-defined data structures stored in workflow instance's internal dataset and performs error checks on input data. The `algX_computeAlgorithm` member uses that data structures and the network databases, and computes the algorithm itself storing the solution back into the dataset. Finally, the `algX_transcodeOutput` member translates the solution into an output message (or a set of messages), e.g. a Path Computation Reply (PCRep), sends it (them) and stores in the dataset the pending message identifiers to be received, if any.

The `workflowInstanceMaker` function listed in Table 12 must be included in the shared object library because it is used by PLATON to create new instances of the workflow on which the function is contained.

Table 12. `workflowInstanceMaker` function

```
extern "C"
{
  IWorkflowImpl* workflowInstanceMaker() { return(new MyWorkflow ()); }
}
```

5.2 Algorithms API

The PCEP API was defined in IDEALIST Deliverable D1.2 [1]. In this section, the meaningful functions of PLATON's API are described.

PLATON's API is organized in namespaces, each of them containing a subset of classes. Namespace have alias names to make workflows easier to be written. The namespaces are defined in Table 13.

Table 13. API Namespaces

Name	Alias	Description
`Lib::Interface`	`LI`	Contains communication interfaces-related classes.
`Lib::Network`	`LN`	Contains network databases-related classes.
`Lib::RoutingAlgorithms`	`LRA`	Contains routing algorithm-related classes.
`Lib::ProtocolHelpers`	`LPH`	Contains protocol-related helper classes.
`Lib::WorkflowEngine`	`LWE`	Contains workflow engine-related classes.

5.2.1 Namespace `LI` definition

The list of classes in namespace `LI` is defined in Table 14.

Table 14. API Classes in Namespace `LI`

Class	Description	Definition
`Interface`	Contains the generic definition of an Interface.	Table 15
`InterfaceBGP`	Contains the specialization of `Interface` for BGP.	Table 16

`InterfacePCEP`	Contains the specialization of `Interface` for PCEP.	Table 17
`InterfaceREST`	Contains the specialization of `Interface` for REST.	Table 18

Table 15. Class `Interface` API

`const std::string& getName() const` Get interface name.
`Protocol getProtocol() const` Get protocol configured in that interface.
`Role getRole() const` Get the role of that interface.
`const Lib::DottedBytes4& getIP() const` Get the local IP address on which this interface is bind to.
`uint32_t getPort() const` Get the local port used by this interface. When role is server, is the port where the server is listening; otherwise, is the remote port where the client will try to connect to.
`const Lib::DottedBytes4Vector& getTrustedPeers() const` Get the trusted peers allowed to connect to this interface. When role is server, contains the remote peers allowed to connect to the listening server; otherwise, is the remote IP address where the client will try to connect to.

Table 16. Class `InterfaceBGP` API

`BGP* getBGP()` Get `BGP` instance bind to the interface.
`Sessions& getSessions()` Get active BGP sessions in the interface.

Table 17. Class `InterfacePCEP` API

`PCEP* getPCEP()` Get `PCEP` instance bind to the interface.
`Sessions& getSessions()` Get active PCEP sessions in the interface.

Table 18. Class `InterfaceREST` API

`Socket* createSession(AfterConnectHandler* ach)` Create a new HTTP Session and execute an `ach` `AfterConnectHandler` action. `AfterConnectHandler` action needs to be specialized by user.

5.2.2 Namespace `LN` definition

The list of classes in namespace `LN` is defined in Table 19.

Table 19. API Classes in Namespace `LN`

Class	Description	Definition
`FlexGrid`	Contains the definition of grid characteristics.	Table 20
`Network`	Contains the definition of the network model.	Table 21

Table 20. Class `FlexGrid` API

`float_t getSpectrumSizeGHz() const` Get spectrum size in GHz.
`Granularity getGranularity() const` Get grid granularity.
`float_t getGranularityGHz() const` Get grid granularity in GHz.
`float_t getMinSpectrumAllocGHz() const` Get minimum spectrum allocation in GHz.
`std::size_t getMinSpectrumAllocNumSlices() const` Get minimum spectrum allocation in number of frequency slices.
`std::size_t getNumSlices() const` Get count of frequency slices in the grid.
`Channel getChannel(int32_t n, uint32_t m) const` Convert a pair `n, m` from a label into a channel.
`Channel getChannel(NandM nAndM) const` Convert `nAndM` into a channel.
`NandM getNAndM(uint32_t firstSlice, uint32_t numSlices) const` Convert pair `firstSlice`, `numSlices` into an `NandM` pair.
`NandM getNAndM(Channel channel) const` Convert `channel` into an `NandM` pair.

Table 21. Class `Network` API

`const FlexGrid* gridGet() const` Get grid model.
`std::size_t modulationFormatCount()` Get count of modulation formats.
`ModulationFormat* modulationFormatAdd(ModulationFormatId id, const std::string& name, std::size_t spectralEff, float reachKm)` Add modulation format with `id`, `name`, `spectralEff` and `reachKm`.
`ModulationFormat* modulationFormatAdd(ModulationFormatId id, const std::string& name, std::size_t spectralEff, float reachKm, const ModulationFormat::ConversionList& conversions)`

Add modulation format with `id`, `name`, `spectralEff`, `reachKm` and `conversions`.
`bool modulationFormatExists(ModulationFormatId id)` Check existence of modulation format with `id`.
`bool modulationFormatExists(const std::string& name)` Check existence of modulation format with `name`.
`const ModulationFormat* modulationFormatGet(ModulationFormatId id) const` Get modulation format with `id`.
`const ModulationFormat* modulationFormatGet(const std::string& name) const` Get modulation format with `name`.
`const ModulationFormatVectorConstIterators modulationFormatGetIterators() const` Get iterators to navigate through modulation formats.
`void modulationFormatRemove(ModulationFormatId id)` Remove modulation format with `id`.
`void modulationFormatRemove(const std::string& name)` Remove modulation format with `name`.
`std::size_t forwardErrorCorrectionCount()` Get count of forward error corrections.
`ForwardErrorCorrection* forwardErrorCorrectionAdd(ForwardErrorCorrectionId id, const std::string& name)` Add forward error correction with `id` and `name`.
`bool forwardErrorCorrectionExists(ForwardErrorCorrectionId id)` Check existence of forward error correction with `id`.
`bool forwardErrorCorrectionExists(const std::string& name)` Check existence of forward error correction with `name`.
`const ForwardErrorCorrection* forwardErrorCorrectionGet(ForwardErrorCorrectionId id) const` Get forward error correction with `id`.
`const ForwardErrorCorrection* forwardErrorCorrectionGet(const std::string& name) const` Get forward error correction with `name`.
`const ForwardErrorCorrectionVectorConstIterators forwardErrorCorrectionGetIterators() const` Get iterators to navigate through forward error corrections.
`void forwardErrorCorrectionRemove(ForwardErrorCorrectionId id)` Remove forward error correction with `id`.
`void forwardErrorCorrectionRemove(const std::string& name)` Remove forward error correction with `name`.
`std::size_t nodeCount()`

Get count of nodes.
`Node* nodeAdd(NodeType type, const NodeId& id, const std::string& name)` Add node with `type`, `id` and `name`.
`bool nodeExists(const NodeId& id)` Check existence of node with `id`.
`Node* nodeGet(const NodeId& id)` Get node with `id`.
`NodeVectorIterators nodeGetIterators()` Get iterators to navigate through nodes.
`void nodeRemove(const NodeId& id)` Remove node with `id`.
`OperationalState nodeGetOpState(const NodeId& id)` Get operational state of node with `id`.
`void nodeSetOpState(const NodeId& id, OperationalState opState)` Set operational state `opState` to node with `id`.
`FailureState nodeGetFailState(const NodeId& id)` Get failure state of node with `id`.
`void nodeSetFailState(const NodeId& id, FailureState failState)` Set failure state `failState` to node with `id`.
`std::size_t portCount(const NodeId& nodeId)` Get count of ports in node `nodeId`.
`PortTrunk* portAddTrunk(const NodeId& nodeId, const PortId& id, Layer layer)` Add trunk port to node `nodeId` with `id` and `layer`.
`PortSBVT* portAddSBVT(const NodeId& nodeId, const PortId& id, Layer layer, float_t swCapGbps, std::size_t numSubTPTx, std::size_t numSubTPRx)` Add trunk port to node `nodeId` with `id`, `layer`, `swCapGbps`, `numSubTPTx` and `numSubTPRx`.
`bool portExists(const NodeId& nodeId, const PortId& id)` Check existence of port with `nodeId` and `id`.
`Port* portGet(const NodeId& nodeId, const PortId& id)` Get port with `nodeId` and `id`.
`PortVectorIterators portGetIterators(const NodeId& nodeId)` Get iterators to navigate through ports in node `nodeId`.
`void portRemove(const NodeId& nodeId, const PortId& id)` Remove port with `nodeId` and `id`.
`OperationalState portGetOpState(const NodeId& nodeId, const PortId& id)` Get operational state of port with `nodeId` and `id`.

`void portSetOpState(const NodeId& nodeId, const PortId& id, OperationalState opState)` Set operational state `opState` to port with `nodeId` and `id`.
`FailureState portGetFailState(const NodeId& nodeId, PortId id)` Get failure state of port with `nodeId` and `id`.
`void portSetFailState(const NodeId& nodeId, const PortId& id, FailureState failState)` Set failure state `failState` to port with `nodeId` and `id`.
`std::size_t linkCount()` Get count of links.
`std::pair<Link*, Link*> linkAdd(LinkType type, const NodeId& srcNodeId, const PortId& srcPortId, const NodeId& dstNodeId, const PortId& dstPortId, Layer layer, float_t defaultMetric, float_t distanceKm, bool bidirectional)` Add link with `type`, `srcNodeId`, `srcPortId`, `dstNodeId`, `dstPortId`, `layer`, `defaultMetric`, `distanceKm`, and directionality `bidirectional`.
`bool linkExists(const LinkId& id)` Check existence of link with `id`.
`bool linkExists(const NodeId& srcNodeId, const PortId& srcPortId, const NodeId& dstNodeId, const PortId& dstPortId)` Check existence of link with `srcNodeId`, `srcPortId`, `dstNodeId` and `dstPortId`.
`Link* linkGet(const LinkId& id)` Get link with `id`.
`Link* linkGet(const NodeId& srcNodeId, const PortId& srcPortId, const NodeId& dstNodeId, const PortId& dstPortId)` Get link with `srcNodeId`, `srcPortId`, `dstNodeId` and `dstPortId`.
`LinkVectorIterators linkGetIterators()` Get iterators to navigate through links.
`void linkRemove(const LinkId& id)` Remove link with `id`.
`void linkRemove(const NodeId& srcNodeId, const PortId& srcPortId, const NodeId& dstNodeId, const PortId& dstPortId)` Remove link with `srcNodeId`, `srcPortId`, `dstNodeId` and `dstPortId`.
`FailureState linkGetFailState(const LinkId& id)` Get failure state of link with `id`.
`FailureState linkGetFailState(const NodeId& srcNodeId, const PortId& srcPortId, const NodeId& dstNodeId, const PortId& dstPortId)` Get failure state of link with `srcNodeId`, `srcPortId`, `dstNodeId` and `dstPortId`.
`void linkSetFailState(const LinkId& id, FailureState failState)` Get failure state `failState` to link with `id`.
`void linkSetFailState(const NodeId& srcNodeId, const PortId& srcPortId, const NodeId& dstNodeId, const PortId& dstPortId, FailureState failState)`

Get failure state `failState` to link with `srcNodeId`, `srcPortId`, `dstNodeId` and `dstPortId`.
`std::size_t lightPathCount()` Get count of lightpaths.
`LightPath* lightPathAdd(const PathId& id, const NodeId& srcNodeId, const PortId& srcPortId, const NodeId& dstNodeId, const PortId& dstPortId, float_t bandWidthGbps, bool bidirectional)` Add lightpath with `id`, `srcNodeId`, `srcPortId`, `dstNodeId`, `dstPortId`, `bandWidthGbps`, and directionality `bidirectional`.
`LightPath* lightPathAdd(const NodeId& srcNodeId, const PortId& srcPortId, const NodeId& dstNodeId, const PortId& dstPortId, float_t bandWidthGbps, bool bidirectional)` Add lightpath with `srcNodeId`, `srcPortId`, `dstNodeId`, `dstPortId`, `bandWidthGbps`, and directionality `bidirectional`. Identifier is generated automatically.
`bool lightPathExists(const PathId& id)` Check existence of lightpath with `id`.
`bool lightPathExists(const EndPointId& srcEndPointId, const EndPointId& dstEndPointId, bool isBidirectional, const LinkIdVector& route, const Channel& downChannel, const Channel& upChannel)` Check existence of lightpath with `srcEndPointId`, `dstEndPointId`, directionality `isBidirectional`, `route`, `downChannel` and `upChannel`.
`LightPath* lightPathGet(const PathId& id)` Get lightpath with `id`.
`LightPath* lightPathGet(const EndPointId& srcEndPointId, const EndPointId& dstEndPointId, bool isBidirectional, const LinkIdVector& route, const Channel& downChannel, const Channel& upChannel)` Get lightpath with `srcEndPointId`, `dstEndPointId`, directionality `isBidirectional`, `route`, `downChannel` and `upChannel`.
`LightPathVectorIterators lightPathGetIterators()` Get iterators to navigate through lightpaths.
`void lightPathRemove(const PathId& id)` Remove lightpath with `id`.
`Network* clone()` Clone the network model into a new one, ensuring total independence from original network model.
`std::string dump(const char* strNewLine)` Dump network model into a string. Use `strNewLine` as new line character.

5.2.3 Namespace LPH definition

The list of classes in namespace LPH is defined in Table 22.

Table 22. API Classes in Namespace LPH

Class	Description	Definition

BGPLS	Contains the set of helper functions for BGP-LS.	Table 23
InterfaceHandler	Contains the set of helper functions for interface and session location.	Table 24
PCEP	Contains the set of helper functions for PCEP.	Table 25
REST	Contains the set of helper functions for REST.	Table 26
SendAfterConnectHandler	Contains the definition of a class to send a REST request just after confirming the establishment of a connection with a remote module.	Table 27

Table 23. Class BGPLS API

void floodNode(LI::InterfaceBGP* bgpInterface, LN::Node* node, Reachability reachability) Flood node node through BGP interface bgpInterface specifying reachability as reachability.
void floodLink(LI::InterfaceBGP* bgpInterface, LN::Link* link, Reachability reachability) Flood link link through BGP interface bgpInterface specifying reachability as reachability.

Table 24. Class InterfaceHandler API

LI::InterfaceBGP* getClientInterfaceBGP(const std::string& interfaceName) Get BGP interface pointer for an interface named as interfaceName.
LI::InterfacePCEP* getClientInterfacePCEP(const std::string& interfaceName) Get PCEP interface pointer for an interface named as interfaceName.
LI::InterfaceREST* getClientInterfaceREST(const std::string& interfaceName) Get REST interface pointer for an interface named as interfaceName.
BGPSession* getBGPSession(const std::string& interfaceName) Get BGP session pointer on interface named as interfaceName.
PCEPSession* getPCEPSession(const std::string& interfaceName) Get PCEP session pointer on interface named as interfaceName.

Table 25. Class PCEP API

bool hasTLV(RP* rp, TLVType tlvType) Check if rp object has a TLV of type tlvType.
LN::PathId getPathIdFromTLV(TLV* tlv) Get the PathId from TLV tlv.
std::string getPathNameFromTLV(TLV* tlv) Get a string representing the path name from TLV tlv.
TLV* getSymPathNameTLV(RP* rp) Get the Symbolic Path Name TLV from rp object.

`LN::PathId getPathId(RP* rp)` Get the Path Id from `rp` object.
`std::string getPathName(RP* rp)` Get the Path Name from `rp` object.
`RequestId getRequestId(RP* rp)` Get request identifier from `rp` object.
`bool getBidirectional(RP* rp)` Check bidirectional flag from `rp` object.
`RP* getRP(PCEP_Request* request)` Get `RP` object from `request`.
`EndPoints* getEndPoints(PCEP_Request* request)` Get `EndPoints` object from `request`.
`Bandwidth* getBandWidth(PCEP_Request* request)` Get `Bandwidth` object from `request`.
`TLV* createOrderTLV(const Order& order)` Create Order TLV from `order`.
`Order getOrderFromTLV(TLV* tlv)` Get `Order` from `tlv`.
`TLV* getOrderTLV(RP* rp)` Get Order TLV from `rp` object.
`Order getOrder(RP* rp)` Get `Order` from `rp` object.
`RP* createRP(bool p2mp, bool eroComp, bool loose, bool bidir, bool reopt, uint32_t reqId)` Create `RP` object from `p2mp`, `eroComp`, `loose`, `bidir`, `reopt`, and `reqId`.
`RP* createRPwithTLVs(bool p2mp, bool eroComp, bool loose, bool bidir, bool reopt, uint32_t reqId, std::vector<TLV*> tlvs)` Create `RP` object from `p2mp`, `eroComp`, `loose`, `bidir`, `reopt`, `reqId` and include TLVs in `tlvs`.
`EndPoints* createEndPointsIPV4P2P(const LN::NodeId& source, const LN::NodeId& destination)` Create `EndPoints` object from `source` and `destination`.
`std::pair<LN::NodeId, LN::NodeId> getEndPointsIPV4P2P(EndPoints* endPoints)` Get source and destination node identifiers from `endPoints`.
`Bandwidth* createReqBandwidth(float_t bandwidthGbps)` Create `Bandwidth` object from `bandwidthGbps`.
`float_t getBandwidth(Bandwidth* bandwidth)` Get bitrate from `bandwidth`.

`RO* createERO(LN::Network* network, const LN::LightPath* lightPath, bool addExplicitSubTPId)` Create `RO` object of type ERO from `network`, `lightPath`, `route` and `addExplicitSubTPId`.
`No_Path* createNoPath()` Create `No_Path` object.
`void setRP(PCEP_Response* response, RP* rp)` Set `rp` in `response`.
`void setRO(PCEP_Path* path, RO* ro)` Set `ro` in `path`.
`void setBandwidth(PCEP_Path* path, Bandwidth* bandwidth)` Set `bandwidth` in `path`.
`void setNoPath(PCEP_Response* response, No_Path* noPath)` Set `noPath` in `response`.
`void extractFromRO(RO* ro, LN::Network* network, LN::LinkIdVector& route, bool& bidirectional, LN::Channel& downSuperChannel, LN::Channel& upSuperChannel, LN::LightPath::SubTransponderPairConfigVector& downSubTranspondersConfig, LN::LightPath::SubTransponderPairConfigVector& upSubTranspondersConfig)` Extract route representation from `ro` using network model `network` and storing components in output variables `route`, `bidirectional`, `downSuperChannel`, `upSuperChannel`, `downSubTranspondersConfig` and `upSubTranspondersConfig`.
`PCEP_SVEC* getPSVEC(PCEP_BundleComputation* pathComp, std::size_t index)` Get `index`-th `PCEP_SVEC` from `pathComp`.
`OF* getOF(PCEP_SVEC* psvec, std::size_t index)` Get `index`-th `OF` from `psvec`.
`SVEC* getSVEC(PCEP_SVEC* psvec)` Get `SVEC` from `psvec`.
`void getRPIDs(SVEC* svec, RequestIdVector& requestIdVector, RequestIdSet& requestIdSet)` Get request identifiers from `svec` and store them in `requestIdVector` and `requestIdSet`.
`IRO* getIRO(PCEP_SVEC* psvec)` Get `IRO` from `psvec`.
`XRO* getXRO(PCEP_SVEC* psvec)` Get `XRO` from `psvec`.
`PCEPAction* createSinglePathComp(PCEPSession* session)` Create Single Path Computation `PCEPAction` for `session`.
`PCEPAction* createBundlePathComp(PCEPSession* session)` Create Bundle Path Computation `PCEPAction` for `session`.
`PCEPAction* createBundlePathComp(PCEPSession* session, uint32_t ofCode)` Create Bundle Path Computation `PCEPAction` for `session` including an `OF` object with `ofCode`.

`PCEP_SingleComputation* getPathCompSingle(PCEPAction* action)` Get `PCEP_SingleComputation` from `action`.
`PCEP_BundleComputation* getPathCompBundle(PCEPAction* action)` Get `PCEP_BundleComputation` from `action`.
`PCEP_Request* addPCompRequest(PCEPAction* action)` Add new `PCEP_Request` into `action` and return a pointer to the former.
`PCEP_Response* addPCompResponse(PCEPAction* action)` Add new `PCEP_Response` into `action` and return the former.
`PCEP_Request* getRequest(PCEP_SingleComputation* pathComp)` Get `PCEP_Request` from `pathComp`.
`std::size_t getNumRequests(PCEP_BundleComputation* pathComp)` Get count of requests in `pathComp`.
`PCEP_Request* getRequest(PCEP_BundleComputation* pathComp, std::size_t index)` Get `index`-th `PCEP_Request` in `pathComp`.
`PCEP_Response* getResponse(PCEP_SingleComputation* pathComp)` Get `PCEP_Response` from `pathComp`.
`std::size_t getNumResponses(PCEP_BundleComputation* pathComp)` Get count of responses in `pathComp`.
`PCEP_Response* getResponse(PCEP_BundleComputation* pathComp, std::size_t index)` Get `index`-th `PCEP_Response` in `pathComp`.
`PCEP_Path* addResponsePath(PCEP_Response* response)` Add new `PCEP_Path` into `response` and return the former.
`PCEPAction* createPCNotif(PCEPSession* session)` Create Path Computation Notification `PCEPAction` for `session`.
`PCEP_Notify* addNotify(PCEPAction* action)` Add new `PCEP_Notify` into `action` and return the former.
`PCEP_Notification* getNotification(PCEPAction* action)` Get `PCEP_Notification` from `action`.
`std::size_t getNumNotifies(PCEPAction* action)` Get count of notifies in `action`.
`PCEP_Notify* getNotify(PCEPAction* action, std::size_t index)` Get `index`-th `PCEP_Notify` in `action`.
`void sendPCError(PCEPAction* pcepAction, uint8_t errorType, uint8_t errorValue)` Send `pcepAction` as a PCError message with `errorType` and `errorValue`.
`void sendPCReply(PCEPAction* pcepAction)` Send `pcepAction` as a PCReply message.

Table 26. Class REST API

`void generateRequest(Lib::HTTPRequest* httpRequest, Lib::HTTPRequest::Method method, const std::string& url, ContentType contentType, const std::string& content)` Fill `httpRequest` with `method`, `url`, `contentType` and `content`.
`void generateReply(Lib::HTTPReply* httpReply, Lib::HTTPReply::StatusCode statusCode, ContentType contentType, const std::string& content)` Fill `httpReply` with `statusCode`, `contentType` and `content`.

Table 27. Class SendAfterConnectHandler API

(constructor) Construct an empty instance of the class.
(destructor) Destroy the class instance.
`void setMethod(Lib::HTTPRequest::Method method)` Set `method`.
`void setURL(const std::string& url)` Set `url`.
`void setContentType(LPH::REST::ContentType contentType)` Set `contentType`.
`void setContent(const std::string& content)` Set `content`.

5.2.4 Namespace LRA definition

The list of classes in namespace LRA is defined in Table 28.

Table 28. API Classes in Namespace LRA

Class	Description	Definition
P2MP_RSA	Contains the definition of the routing and spectrum allocation algorithm for P2MP connections.	Table 29
P2MP_RSA::Config	Contains the definition of configuration arguments for class P2MP_RSA.	Table 30Table 29
R	Contains the definition of the routing algorithm for P2P connections.	Table 31
RMSA	Contains the definition of the routing, modulation format and spectrum allocation algorithm for P2P connections.	Table 32
RSA	Contains the definition of the routing and spectrum allocation algorithm for P2P connections.	Table 33

Table 29. Class P2MP_RSA API

`P2MP_RSA(LN::Network* network, Config& config)`

Construct an instance of `P2MP_RSA` using `network` and `config`.
`P2MP_RSA(LN::Network* network, std::map<LN::LinkId, double_t>& linkMetrics, Config& config)` Construct an instance of `P2MP_RSA` using `network`, `linkMetrics` and `config`.
`void computeShortestTreeWithSpectrum(LN::EndPoint srcEndPoint, LN::EndPointVector dstEndPoints, uint32_t numSlicesRequired, std::list<SubTree>& solution)` Compute the shortest tree with spectrum allocation between `srcEndPoint` and `dstEndPoints` looking for `numSlicesRequired` and store the solution in `solution`.

Table 30. Class `P2MP_RSA::Config` API

`double getCostPerSlot()` Get cost per slot.
`void setCostPerSlot(double costPerSlot)` Set cost per slot to `costPerSlot`.
`double getCostPerHop()` Get cost per hop.
`void setCostPerHop(double costPerHop)` Set cost per hop to `costPerHop`.
`double getCostPerDestination()` Get cost per destination.
`void setCostPerDestination(double costPerDestination)` Set cost per destination to `costPerDestination`.
`double getMaxPathDistanceKm()` Get maximum path distance allowed in Km.
`void setMaxPathDistanceKm(double maxPathDistanceKm)` Set maximum path distance allowed in Km to `maxPathDistanceKm`.
`uint32_t getNumCandidatesPerDestination()` Get number of candidates per destination.
`void setNumCandidatesPerDestination(double numCandidatePathsPerDestination)` Set number of candidates per destination to `numCandidatePathsPerDestination`.

Table 31. Class `R` API

`R(LN::Network* network)` Construct an instance of `R` using `network`.
`R(LN::Network* network, std::map<LN::LinkId, double_t>& linkMetrics)` Construct an instance of `R` using `network`, `linkMetrics`.
`void computeSP(LN::NodeId srcNodeId, LN::NodeId dstNodeId, bool bidirectional, LN::LinkVector& route, LN::LinkIdVector& routeIds, double_t& metric)`

Compute the shortest path between `srcNodeId` and `dstNodeId` with directionality `bidirectional` and store solution in output variables `route`, `routeIds` and `metric`.
`void computeSP(LN::Node* srcNode, LN::Node* dstNode, bool bidirectional, LN::LinkVector& route, LN::LinkIdVector& routeIds, double_t& metric)` Compute the shortest path between `srcNode` and `dstNode` with directionality `bidirectional` and store solution in output variables `route`, `routeIds` and `metric`.
`void computeSP(LN::EndPoint srcEndPoint, LN::EndPoint dstEndPoint, bool bidirectional, LN::LinkVector& route, LN::LinkIdVector& routeIds, double_t& metric)` Compute the shortest path between `srcEndPoint` and `dstEndPoint` with directionality `bidirectional` and store solution in output variables `route`, `routeIds` and `metric`.
`void computeKSP(std::size_t k, LN::NodeId srcNodeId, LN::NodeId dstNodeId, bool bidirectional, std::vector<LN::LinkVector>& routes, std::vector<LN::LinkIdVector>& routesIds, std::vector<double_t>& metrics)` Compute `k` shortest paths between `srcNodeId` and `dstNodeId` with directionality `bidirectional` and store solution in output variables `routes`, `routeIds` and `metrics`.
`void computeKSP(std::size_t k, LN::Node* srcNode, LN::Node* dstNode, bool bidirectional, std::vector<LN::LinkVector>& routes, std::vector<LN::LinkIdVector>& routesIds, std::vector<double_t>& metrics)` Compute `k` shortest paths between `srcNode` and `dstNode` with directionality `bidirectional` and store solution in output variables `routes`, `routeIds` and `metrics`.
`void computeKSP(std::size_t k, LN::EndPoint srcEndPoint, LN::EndPoint dstEndPoint, bool bidirectional, std::vector<LN::LinkVector>& routes, std::vector<LN::LinkIdVector>& routesIds, std::vector<double_t>& metrics)` Compute `k` shortest paths between `srcEndPoint` and `dstEndPoint` with directionality `bidirectional` and store solution in output variables `routes`, `routeIds` and `metrics`.

Table 32. Class `RMSA` API

`RMSA(LN::Network* network)` Construct an instance of `RMSA` using `network`.
`RMSA(LN::Network* network, std::map<LN::LinkId, double_t>& linkMetrics)` Construct an instance of `RMSA` using `network`, `linkMetrics`.
`void compute(const LN::PathId& pathId, std::size_t maxSlotWidth)` Compute the route, modulation format and spectrum allocation for `pathId` constraining to allocate at maximum `maxSlotWidth` frequency slices.
`void expandSlot(const LN::PathId& pathId, float_t bitRateGbps)` Compute the route, modulation format and spectrum allocation expansion for `pathId` to serve `bitRateGbps`.

Table 33. Class `RSA` API

`RSA(LN::Network* network)` Construct an instance of `RSA` using `network`.
`RSA(LN::Network* network, std::map<LN::LinkId, double_t>& linkMetrics)` Construct an instance of `RSA` using `network`, `linkMetrics`.

`void computeSP(LN::NodeId srcNodeId, LN::NodeId dstNodeId, bool bidirectional, std::size_t numSlices, LN::LinkVector& route, double_t& metric, LN::Channel& downChannel, LN::Channel& upChannel)` Compute the shortest path between `srcNodeId` and `dstNodeId` with directionality `bidirectional`, spectrum width of `numSlices` and store solution in output variables `route`, `metric`, `downChannel` and `upChannel`.
`void computeSP(LN::Node* srcNode, LN::Node* dstNode, bool bidirectional, std::size_t numSlices, LN::LinkVector& route, double_t& metric, LN::Channel& downChannel, LN::Channel& upChannel)` Compute the shortest path between `srcNode` and `dstNode` with directionality `bidirectional`, spectrum width of `numSlices` and store solution in output variables `route`, `metric`, `downChannel` and `upChannel`.
`void computeSP(LN::EndPoint srcEndPoint, LN::EndPoint dstEndPoint, bool bidirectional, std::size_t numSlices, LN::LinkVector& route, double_t& metric, LN::Channel& downChannel, LN::Channel& upChannel)` Compute the shortest path between `srcEndPoint` and `dstEndPoint` with directionality `bidirectional`, spectrum width of `numSlices` and store solution in output variables `route`, `metric`, `downChannel` and `upChannel`.
`void computeKSP(std::size_t k, LN::NodeId srcNodeId, LN::NodeId dstNodeId, bool bidirectional, std::size_t numSlices, std::vector<LN::LinkVector>& routes, std::vector<double_t>& metrics, LN::ChannelVector& downChannels, LN::ChannelVector& upChannels)` Compute `k` shortest paths between `srcNodeId` and `dstNodeId` with directionality `bidirectional`, spectrum width of `numSlices` and store solution in output variables `routes`, `metrics`, `downChannels` and `upChannels`.
`void computeKSP(std::size_t k, LN::Node* srcNode, LN::Node* dstNode, bool bidirectional, std::size_t numSlices, std::vector<LN::LinkVector>& routes, std::vector<double_t>& metrics, LN::ChannelVector& downChannels, LN::ChannelVector& upChannels)` Compute `k` shortest paths between `srcNode` and `dstNode` with directionality `bidirectional`, spectrum width of `numSlices` and store solution in output variables `routes`, `metrics`, `downChannels` and `upChannels`.
`void computeKSP(std::size_t k, LN::EndPoint srcEndPoint, LN::EndPoint dstEndPoint, bool bidirectional, std::size_t numSlices, std::vector<LN::LinkVector>& routes, std::vector<double_t>& metrics, LN::ChannelVector& downChannels, LN::ChannelVector& upChannels)` Compute `k` shortest paths between `srcEndPoint` and `dstEndPoint` with directionality `bidirectional`, spectrum width of `numSlices` and store solution in output variables `routes`, `metrics`, `downChannels` and `upChannels`.
`void compute(const LN::PathId& pathId, std::size_t maxSlotWidth, LN::ModulationFormatId modFormatId)` Compute the route, modulation format and spectrum allocation for `pathId` constraining to allocate at maximum `maxSlotWidth` frequency slices and to use modulation format `modFormatId`.
`void expandSlot(const LN::PathId& pathId, float_t bitRateGbps, LN::ModulationFormatId modFormatId)` Compute the route, modulation format and spectrum allocation expansion for `pathId` to serve `bitRateGbps` using modulation format `modFormatId`.

5.2.5 Namespace LWE definition

The list of classes in namespace LWE is defined in Table 34.

Table 34. API Classes in Namespace LWE

Class	Description	Definition
IWorkflowImpl	Contains the interface to define a workflow.	Table 9
Message	Contains the definition of a generic message for the workflow engine.	Table 35
MessagePCEPPCError	Contains the specialization of Message for PCEP PCError.	Table 36
MessagePCEPPCNotif	Contains the specialization of Message for PCEP PCNotif.	Table 36
MessagePCEPPCReply	Contains the specialization of Message for PCEP PCReply.	Table 36
MessagePCEPPCRequest	Contains the specialization of Message for PCEP PCRequest.	Table 36
MessageRESTReply	Contains the specialization of Message for REST Reply.	Table 37
MessageRESTRequest	Contains the specialization of Message for REST Request.	Table 38
UniqueIdentifierGenerator	Contains the definition of a unique identifier generator.	Table 39

Table 35. Class Message API

Type getType() Type getType() const Get message type.

Table 36. Classes MessagePCEPPCError, MessagePCEPPCNotif, MessagePCEPPCReply and MessagePCEPPCRequest API

PCEPAction* getPCEPAction() Get PCEPAction in the message.

Table 37. Class MessageRESTReply API

Lib::HTTPReply* getHTTPReply() Get Lib::HTTPReply in the message.

Table 38. Class MessageRESTRequest API

Lib::HTTPRequest* getHTTPRequest() Get Lib:: HTTPRequest in the message.

Table 39. Class `UniqueIdentifierGenerator` API

`uint32_t get()` Get a unique identifier.

5.3 Workflow Example

To make sense on how an algorithm must be implemented, the SinglePathRSA workflow is provided. Figure 8 shows the sequence diagram for this workflow; we assume that PLATON is connected to a PCE who issues a request to PLATON to compute a single path RSA between some endpoints with a specified bandwidth. The workflow in PLATON expects to receive the PCRequest message defined in Table 40, and replies to the PCE with the PCReply message defined in Table 41.

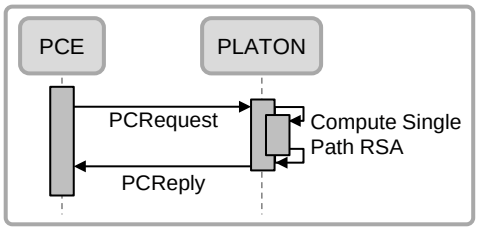

Figure 8. Single Path RSA Workflow's sequence diagram

Table 40 PCRequest message

```
<PCReq Message> ::= <Common Header>
                    <RP>
                    <END-POINTS>
                    <BANDWIDTH>
```

Table 41 PCReply message

```
<PCRep Message> ::= <Common Header>
                    <response>
where:
      <response>::=<RP>
                   <NO-PATH> | <ERO>
```

The workflow consists in four files: *i*) `SimplePathRSA.xml` file, containing the XML workflow definition; *ii*) `SimplePathRSA.h` file, containing the workflow class definition and its data set definition; *iii*) `SimplePathRSA.cc` file, containing the implementation of workflow class member functions; and *iv*) the `Makefile` file used to build the binary file from source codes in files `SimplePathRSA.h` a and `SimplePathRSA.cc` files.

5.3.1 Source files

Table 42 lists the workflow definition in XML used by PLATON to load the workflow containing:

- The name of the workflow (line 3 in Table 42)
- The path to the dynamic library (line 4) containing the workflow binary code, including the state algorithms to be executed in each state. Note that this file is generated using code in Table 43 and Table 44.
- No config file is used in this workflow for the sake of simplicity. See configuration parameters hardcoded in method `configure` listed in Table 44.
- The triggering message (line 6) is a PCReq with OF code 1. Note that in PCReq message defined in Table 40, no OF object was defined, so PLATON uses the default OF code configured, in this case 1. See section 4.5.
- The states of the FSM (line 8): a single state, named as `compute` is considered, plus the `error` state, for this simple example. Note that error state is kept unused and unimplemented for the sake of simplifying the example.

- The FSM state transitions: in this simple example no transition is required since a single request and a single reply are required.
- The initial and error states of the FSM (lines 10-11).

Table 42 `SinglePathRSA_definition.xml`

```
 1: <?xml version="1.0" encoding="UTF-8"?>
 2:
 3: <Workflow name="Single Path RSA"
 4:           libraryFile="workflows/SinglePathRSA/SinglePathRSA.so"
 5:           configFile="">
 6:   <TriggeringMessage message="PCReq" ofCode="1"/>
 7:
 8:   <State name="compute"/><State name="error"/>
 9:
10:   <InitialState name="compute"/>
11:   <ErrorState name="error"/>
12: </Workflow>
```

Table 43 lists the header file of the workflow and the instantiation function:

- Class `Config` (lines 5-9) is used to define the configuration parameters required by the workflow and has one attribute: `modFormatName` stores the name of the modulation format to be used to convert the requested bitrate into a number of frequency slices.
- Class `Demand` (lines 11-19) is used to define the data required for each demand to be computed and has two attributes: `reqId` stores the identifier for this computation request and `lightPath` points to the temporary path created to compute the RSA.
- Class `DataSet` (lines 21-31) is used to store the internal algorithm data and has four attributes: `config` references the instance of the `Config` class for this computation, `network` points to the cloned PLATON's network (see method `setNetwork` in next file), `pcepAction` points to the received PCEP Action containing the PCRequest and `demand` references the instance of the `Demand` class containing the request information.
- Class `SinglePathRSA` (lines 31-57) defines the members and attributes of the workflow. Among others, the private attribute named as `dataSet` (line 37) references the instance of `DataSet` class, and two state algorithms are defined, named as `compute` (line 39) and `error` (line 44). Likewise, the `compute` state algorithm uses three private members: `compute_transcodeInput` (line 40), `compute_algorithm` (line 41), and `compute_transcodeOutput` (line 42) as described in section 3.2.3. Besides, the common members defined in `IWorkflowImpl` interface are defined (lines 50-56).
- The instantiation function `workflowInstanceMaker` (line 61) is also provided to allow PLATON creating new instances of the workflow.

Table 43 `SinglePathRSA.h`

```
 1: #pragma once
 2:
 3: #include "Lib/WorkflowEngine/IWorkflowImpl.h"
 4:
 5: class Config
 6: {
 7:   public:
 8:     std::string modFormatName;
 9: };
```

```
10:
11: class Demand
12: {
13:   public:
14:     LPH::PCEP::RequestId reqId;
15:     LN::LightPath* lightPath;
16:
17:     explicit Demand();
18: };
19:
20: class DataSet
21: {
22:   public:
23:     Config config;
24:     LN::Network* network;
25:     PCEPAction* pcepAction;
26:     Demand demand;
27:
28:     explicit DataSet();
29: };
30:
31: class SinglePathRSA : public IWorkflowImpl
32: {
33:   public:
34:     CREATE_EXCEPTION_KIND
35:
36:   private:
37:     DataSet dataSet;
38:
39:     void compute(LWE::Message* message);
40:     void compute_transcodeInput();
41:     void compute_algorithm();
42:     void compute_transcodeOutput();
43:
44:     void error(LWE::Message* message);
45:
46:   public:
47:     explicit SinglePathRSA();
48:     virtual ~SinglePathRSA();
49:
50:     void configure(const std::string& configFilePath);
51:     void setNetwork(LN::Network* network);
52:     void setInterfaces(LI::Interfaces* interfaces);
53:     void clear();
54:     void gracefulAbort();
55:
56:     LWE::MessageSet runState(const std::string& stateName, LWE::Message* message);
57: };
58:
59: extern "C"
60: {
61:   IWorkflowImpl* workflowInstanceMaker() { return(new SinglePathRSA()); }
62: }
```

Table 44 lists the implementation file of the algorithm:

- The constructor members for classes `Demand` (line 7) and `DataSet` (line 8) are defined to initialize their attributes.
- The constructor (line 10) and destructor (line 11) members for class `SinglePathRSA` are kept empty because that class defines reference members instead of pointers, so their dynamic creation and destruction are self-managed.
- The `configure` member (lines 13-17) initializes the parameters in `config` with hardcoded values. In a complete workflow that value should be read from a configuration file.

- The `setNetwork` member (line 19) creates a clone of PLATON's network database so as to keep the original one unmodified while running the current computation.
- The `setInterfaces` member (line 20) is kept empty since that workflow does not need to issue requests to other external modules, so it does not need to deal with other interfacers.
- The `clear` member (line 21) is kept empty because the algorithm does not need to initialize any data structure.
- The `gracefulAbort` member (line 22) is kept empty because, for the sake of clarifying the example, the algorithm will not reply any error message when the workflow finishes abnormaly.
- The `runState` member (lines 24-35) executes the correct state algorithm function based on the state name provided in the parameter `stateName` passing to that function the message pointer `message` (lines 26-29). Since the algorithm does not require issuing requests to other external modules an empoty set of pending messages is returned (lines 33-34).
- The `compute` member (lines 37-56) checks that received message is a PCEP PCReq message (lines 39-40), casts that generic message into a PCEP PCReq message, extracts the PCEP Action containing the request and stores it into the data set (lines 42-43). Then, it sequentially executes the methods `compute_transcodeInput` (lines 45-47), `compute_algorithm` (lines 49-51) and `compute_transcodeOutput` (lines 53-55) as described in section 3.2.3.
- The `compute_transcodeInput` member (lines 58-73) uses the `pcepAction` attribute in the `dataSet`, obtains the request (lines 60-61), and fetches the attributes from the Request Parameters (RP) object (lines 64-66), the source and destination nodes from the EndPoints object (lines 67-68) and the bitrate from the Bandwidth object (line 69-70) from that request. Finally, it creates a temporary lightpath with that attributes for the computation (lines 71-73).
- The `compute_algorithm` member (lines 76-86) uses the `config`'s attribute `modFormatName` to select the modulation format to be used (lines 78-79), instantiates an RSA Algorithm over the cloned network (line 81), and computes the path for that temporary lightpath (lines 82-85) specifying the path id of that temporary path, the number of slices computed from the lightpath's bitrate and the selected modulation format.
- The `compute_transcodeOutput` member (lines 88-123) fetches the pointer to the temporary lightpath in the dataset (line 90), uses the `pcepAction` to construct a PCResponse `response` (line 92), creates an RP object with the request identifier obtained from the request (lines 94-95), and creates a PCEP Path with an Explicit Route Object (ERO) and Bandwidth objects encoding the solution (102-106) if it was found, or a NoPath object if no solution was found (lines 112-113). Finally, a PCReply message with the PCEP response is sent (line 122).
- The `error` member (lines 125-129) is kept unimplemented for the sake of clarity.

Table 44 `SinglePathRSA.cc`

```
1: #include "SinglePathRSA.h"
2:
3: #include "Lib/ProtocolHelpers/PCEP.h"
4: #include "Lib/RoutingAlgorithms/RSA.h"
```

```
#include "Lib/WorkflowEngine/MessagePCEPPCRequest.h"

Demand::Demand() : reqId(0), lightPath(NULL) { }
DataSet::DataSet() : network(NULL), pcepAction(NULL) { }

SinglePathRSA::SinglePathRSA() { }
SinglePathRSA::~SinglePathRSA() { }

void SinglePathRSA::configure(const std::string& configFilePath)
{
  (void)configFilePath;
  dataSet.config.modFormatName = "DP-QPSK";
}

void SinglePathRSA::setNetwork(LN::Network* network) { dataSet.network = network->clone(); }
void SinglePathRSA::setInterfaces(LI::Interfaces* interfaces) { (void)interfaces; }
void SinglePathRSA::clear() { }
void SinglePathRSA::gracefulAbort() { }

LWE::MessageSet SinglePathRSA::runState(const std::string& stateName, LWE::Message* message)
{
  if(!stateName.compare("compute"))
    compute(message);
  else if(!stateName.compare("error"))
    error(message);
  else
    THROW_EXCEPTION("Unsupported state " << stateName);

  LWE::MessageSet pendingMessages;
  return(pendingMessages);
}

void SinglePathRSA::compute(LWE::Message* message)
{
  if(message->getType() != LWE::Message::TYPE_PCEP_PCREQUEST)
    THROW_EXCEPTION("Message received is not of PCEP_REQUEST type");

  LWE::MessagePCEPPCRequest* pcReq = static_cast<LWE::MessagePCEPPCRequest*>(message);
  dataSet.pcepAction = pcReq->getPCEPAction();

  try { compute_transcodeInput(); }
  catch (std::exception& e)
  { THROW_EXCEPTION("Unable to transcode input. Exception: " << e.what()); }

  try { compute_algorithm(); }
  catch (std::exception& e)
  { THROW_EXCEPTION("Unable to execute algorithm. Exception: " << e.what()); }

  try { compute_transcodeOutput(); }
  catch (std::exception& e)
  { THROW_EXCEPTION("Unable to transcode output. Exception: " << e.what()); }
}

void SinglePathRSA::compute_transcodeInput()
{
  PCEP_SingleComputation* pathComp = LPH::PCEP::getPathCompSingle(dataSet.pcepAction);
  PCEP_Request* request = LPH::PCEP::getRequest(pathComp);

  Demand& demand = dataSet.demand;
  RP* rp = LPH::PCEP::getRP(request);
  demand.reqId = LPH::PCEP::getRequestId(rp);
  bool bidirectional = LPH::PCEP::getBidirectional(rp);
  EndPoints* endPoints = LPH::PCEP::getEndPoints(request);
  std::pair<LN::NodeId, LN::NodeId> nodes = LPH::PCEP::getEndPointsIPV4P2P(endPoints);
  Bandwidth* reqBw = LPH::PCEP::getBandWidth(request);
  float_t bitRateGbps = LPH::PCEP::getBandwidth(reqBw);
  demand.lightPath = dataSet.network->lightPathAdd( nodes.first, LN::PortId(0),
                                                    nodes.second, LN::PortId(0),
                                                    bitRateGbps, bidirectional);
```

```
 74: }
 75:
 76: void SinglePathRSA::compute_algorithm()
 77: {
 78:   const LN::ModulationFormat* modFormat =
 79:         dataSet.network->modulationFormatGet(dataSet.config.modFormatName);
 80:
 81:   LRA::RSA rsaAlgorithm(dataSet.network);
 82:   rsaAlgorithm.compute( dataSet.demand.lightPath->getId(),
 83:                         modFormat->getNumSlicesFromBitRateGbps(
 84:                             dataSet.demand.lightPath->getBandWidthGbps()),
 85:                         modFormat->getId());
 86: }
 87:
 88: void SinglePathRSA::compute_transcodeOutput()
 89: {
 90:   const LN::LightPath* lsp = dataSet.demand.lightPath;
 91:
 92:   PCEP_Response* response = LPH::PCEP::addPCompResponse(dataSet.pcepAction);
 93:
 94:   RP* rp = LPH::PCEP::createRP( false, false, false, lsp->isBidirectional(), false,
 95:                                 dataSet.demand.reqId);
 96:   LPH::PCEP::setRP(response, rp);
 97:
 98:   switch(lsp->getState())
 99:   {
100:     case LN::Path::STATE_ROUTEDANDSPECTRUMALLOCATED:
101:     {
102:       RO* ro = LPH::PCEP::createERO(dataSet.network, lsp, false);
103:       Bandwidth* bw = LPH::PCEP::createLSPBandwidth(lsp->getBandWidthGbps());
104:       PCEP_Path* path = LPH::PCEP::addResponsePath(response);
105:       LPH::PCEP::setRO(path, ro);
106:       LPH::PCEP::setBandwidth(path, bw);
107:       break;
108:     }
109:
110:     case LN::Path::STATE_DEFINED:
111:     {
112:       No_Path* noPath = LPH::PCEP::createNoPath();
113:       LPH::PCEP::setNoPath(response, noPath);
114:       break;
115:     }
116:
117:     default:
118:       THROW_EXCEPTION("Inconsistent LSP state");
119:       break;
120:   }
121:
122:   LPH::PCEP::sendPCReply(dataSet.pcepAction);
123: }
124:
125: void SinglePathRSA::error(LWE::Message* message)
126: {
127:   (void)message;
128:   THROW_EXCEPTION("NOT YET IMPLEMENTED");
129: }
```

Table 45 lists the make file used to build the SinglePathRSA workflow.

Table 45 `Makefile`

```
1: all:
2:         g++ -I"../../include" -c -fPIC -o SinglePathRSA.o SinglePathRSA.cc
3:         g++ -shared -o SinglePathRSA.so SinglePathRSA.o
4:
5: clean:
6:         rm SinglePathRSA.o SinglePathRSA.so
```

5.3.2 Environment pre-requisites

PLATON's binary code provided in this deliverable is built for 64-bit PCs running Linux operating system. All the tests executed on PLATON's binary code provided were done in a PC system with an Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz, 16 GB of RAM and 1 Tb of hard disk drive running Ubuntu Linux version 14.04.3 LTS.

The libraries that must be installed in the system to be able to run PLATON and compile the workflows are: `build-essential` and `libxerces-c-dev`. They can be installed running the Ubuntu script in Table 46.

Table 46 PLATON dependency installation script

```
1: # apt-get –y update
2: # apt-get –y install build-essential libxerces-c-dev
```

5.3.3 Workflow compilation guide

Assuming the environement configured as described in section 5.3.2, having the provided packet decompressed in folder $PLATON, where $PLATON has to be substituted by the path where reader decompressed the provided packet, the script in Table 47 have to be executed to build the shared library for the example workflow presented in section 5.3.1. A file named as `SinglePathRSA.so` should be created. If the tree structure provided is kept as is, the resulting binary file will be placed in the correct folder to allow PLATON to load it.

Table 47 Example workflow compilation script

```
1: $ cd $PLATON/workflows/SinglePathRSA
2: $ make
```

5.3.4 Execution guide

Assuming the environement configured as described in section 5.3.2, having the provided packet decompressed in folder $PLATON, where $PLATON has to be substituted by the path where reader decompressed the provided packet, the script in Table 48 have to be executed to start PLATON.

Important remarks:

- A script named as `make_interfaces.sh` is provided to configure the IP addresses in this execution scenario as 172.16.0.1 for the PCE process and 172.16.0.2 for PLATON which corresponds to the provided configuration for PLATON. To run this script super-user privileges are required.
- The provided configuration file `data/platon_config.xml` has the BGP interface disabled and a simple topology in file `data/db/topology_4n_4l.xml` is loaded using an XML file. If BGP interface is enabled, PLATON needs to be executed as super-user so as to be able to bind the privileged port 179 used by BGP protocol.

Table 48 PLATON execution script

```
1: # cd $PLATON/
2: # ./make_interfaces.sh
3: # ./platon.sh
```

5.3.5 Execution demonstration

To demonstrate how the example workflow works, some screenshors are provided in this section. The simple network topology shown in Figure 9 has been encoded in the config file `data/db/topology_4n_4l.xml` and configured in `data/platon_config.xml`. Nodes are

in IP range 10.0.0.X, where X is the identifier of the node. Each node has 3 ports, the port identified with 0 is used as a client ingress/egress port, while ports 1 and 2 are trunk ports. Each link is configured with a metric of 10, and a distance of 100 Km.

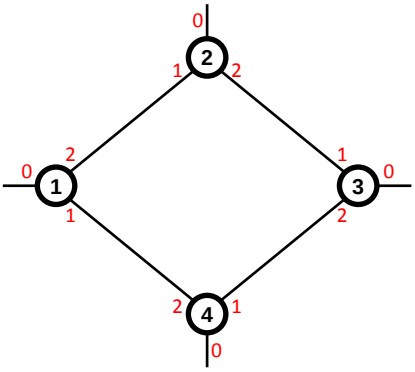

Figure 9. 4-node 4-link network topology

To simplify the demonstration an application named as `DummyClient` is provided; this application starts a PCEP session with PLATON, issues a PCRequest message requesting a path from node 10.0.0.1 to node 10.0.0.3. After starting PLATON using the script in Table 48, its log should see like in Figure 10. At that point, PLATON is listening for incoming PCEP connections at IP address 172.16.0.2 and will accept connections only from Trusted Peer 172.16.0.1.

```
2015/10/15 16:30:54 PLATON: [InterfacePCEP::initialize] PCEPTrustedPeer[0] = 172.16.0.1
2015/10/15 16:30:54 PLATON: [Manager::configure] Loading Modulation Formats...
2015/10/15 16:30:54 PLATON: [Manager::configure] Loading Forward Error Corrections...
2015/10/15 16:30:54 PLATON: [Manager::configure] Loading TED...
2015/10/15 16:30:54 PLATON: [Manager::configure] Loading LSP-DB...
2015/10/15 16:30:54 PLATON: [Manager::configure] Loading Workflows...
2015/10/15 16:30:54 PLATON: [main] VTY listening on 127.0.0.1:2601
2015/10/15 16:30:54 PLATON: [main] Starting Manager
2015/10/15 16:30:54 PLATON: [Manager::start] Running startup sequence...
2015/10/15 16:30:54 PLATON: [Callback_Run_StartupStep] Starting Interface s_pcep
2015/10/15 16:30:54 PLATON: [InterfacePCEP::start] PCEP SERVER running on 172.16.0.2:4189
```

Figure 10. PLATON log after startup

Now, executing the `DummyClient` application using the script in Table 49, the `DummyClient` will report the byte-formated list of PCEP messages exchanged with PLATON as shown in Figure 11.

Table 49 DummyClient execution script

```
1:  # $PLATON/
2:  # ./DummyClient
```

```
CONN
RECV PCEP OPEN: bytes[20] = { 0x20, 0x01, 0x00, 0x14, 0x01, 0x12, 0x00, 0x10, 0x20, 0x1e, 0x78, 0x01, 0x
00, 0x10, 0x00, 0x04, 0x00, 0x00, 0x00, 0x02 }
SEND PCEP OPEN: bytes[12] = { 0x20, 0x01, 0x00, 0x0c, 0x01, 0x12, 0x00, 0x08, 0x20, 0x1e, 0x78, 0x01 }
RECV PCEP KEAL: bytes[04] = { 0x20, 0x02, 0x00, 0x04 }
SEND PCEP KEAL: bytes[04] = { 0x20, 0x02, 0x00, 0x04 }
SEND PCEP PCREQ: bytes[36] = { 0x20, 0x03, 0x00, 0x24, 0x02, 0x12, 0x00, 0x0c, 0x00, 0x00, 0x00, 0x00, 0
x00, 0x00, 0x00, 0x01, 0x04, 0x12, 0x00, 0x0c, 0x0a, 0x00, 0x00, 0x01, 0x0a, 0x00, 0x00, 0x03, 0x05, 0x1
0, 0x00, 0x08, 0x42, 0xc8, 0x00, 0x00 }
RECV PCEP PCREP: bytes[84] = { 0x20, 0x04, 0x00, 0x54, 0x02, 0x12, 0x00, 0x0c, 0x00, 0x00, 0x00, 0x00, 0
x00, 0x00, 0x00, 0x01, 0x07, 0x12, 0x00, 0x3c, 0x04, 0x0c, 0x00, 0x00, 0x0a, 0x00, 0x00, 0x01, 0x00, 0x0
0, 0x00, 0x02, 0x03, 0x0c, 0x00, 0x02, 0x6a, 0x00, 0x00, 0x02, 0x00, 0x02, 0x00, 0x00, 0x04, 0x0c, 0x00,
 0x00, 0x0a, 0x00, 0x00, 0x02, 0x00, 0x00, 0x00, 0x02, 0x03, 0x0c, 0x00, 0x02, 0x6a, 0x00, 0x00, 0x02, 0
x00, 0x02, 0x00, 0x00, 0x01, 0x08, 0x0a, 0x00, 0x00, 0x03, 0x20, 0x00, 0x05, 0x22, 0x00, 0x08, 0x42, 0xc
8, 0x00, 0x00 }
SEND PCEP CLOSE: bytes[12] = { 0x20, 0x07, 0x00, 0x0c, 0x0f, 0x12, 0x00, 0x08, 0x00, 0x00, 0x00, 0x01 }
```

Figure 11. DummyClient execution log

If wireshark was enabled to capture the traffic and the display filter is set to “pcep”, the list of messages in Figure 12 should be seen. The capture file resulting from this test is provided in `capture/capture.pcapng` file contained in the binary package delivered together with this report. Note that total contribution of PLATON, including the round-trip-time for the request-reply sequence and the computation time took less than 0.6 ms.

No.	Time	Source	Destination	Protocol	Length	Info
4	0.000137	172.16.0.2	172.16.0.1	PCEP	88	Open
5	0.000145	172.16.0.1	172.16.0.2	PCEP	80	Open
8	0.000200	172.16.0.2	172.16.0.1	PCEP	72	Keepalive
10	0.000500	172.16.0.1	172.16.0.2	PCEP	72	Keepalive
12	0.038801	172.16.0.1	172.16.0.2	PCEP	104	Path Computation Request (PCReq)
14	0.039392	172.16.0.2	172.16.0.1	PCEP	152	Path Computation Reply (PCRep)
15	0.039525	172.16.0.1	172.16.0.2	PCEP	80	Close

Figure 12. Example Workflow message list

Figure 13 and Figure 14 illustrate the details of the PCReq and PCRep messages issued by the `DummyClient` and replied by PLATON, respectively. Finally, Figure 15 shows PLATON’s log after executing the example workflow.

```
Path Computation Element communication Protocol
▸ Path Computation Request (PCReq) Header
▾ RP object
    Object Class: RP OBJECT (2)
    0001 .... = Object Type: 1
  ▸ Flags
    Object Length: 12
    Reserved: 0x00
  ▸ Flags: 0x000000
    Requested ID Number: 0x00000001
▾ END-POINT object
    Object Class: END-POINT OBJECT (4)
    0001 .... = Object Type: 1
  ▸ Flags
    Object Length: 12
    Source IPv4 Address: 10.0.0.1
    Destination IPv4 Address: 10.0.0.3
▾ BANDWIDTH object
    Object Class: BANDWIDTH OBJECT (5)
    0001 .... = Object Type: 1
  ▸ Flags
    Object Length: 8
    Bandwidth: 100
```

Figure 13. Example Workflow PCReq message details

```
Path Computation Element communication Protocol
▸ Path Computation Reply (PCRep) Header
▾ RP object
    Object Class: RP OBJECT (2)
    0001 .... = Object Type: 1
  ▸ Flags
    Object Length: 12
    Reserved: 0x00
  ▸ Flags: 0x000000
    Requested ID Number: 0x00000001
▾ EXPLICIT ROUTE object (ERO)
    Object Class: EXPLICIT ROUTE OBJECT (ERO) (7)
    0001 .... = Object Type: 1
  ▸ Flags
    Object Length: 60
  ▸ SUBOBJECT: Unnumbered Interface ID: 10.0.0.1:2
  ▸ SUBOBJECT: Label Control
  ▸ SUBOBJECT: Unnumbered Interface ID: 10.0.0.2:2
  ▸ SUBOBJECT: Label Control
  ▸ SUBOBJECT: IPv4 Prefix: 10.0.0.3/32
▾ BANDWIDTH object
    Object Class: BANDWIDTH OBJECT (5)
    0010 .... = Object Type: 2
  ▸ Flags
    Object Length: 8
    Bandwidth: 100
```

Figure 14. Example Workflow PCReply message details

```
2015/10/15 16:30:54 PLATON: [InterfacePCEP::initialize] PCEPTrustedPeer[0] = 172.16.0.1
2015/10/15 16:30:54 PLATON: [Manager::configure] Loading Modulation Formats...
2015/10/15 16:30:54 PLATON: [Manager::configure] Loading Forward Error Corrections...
2015/10/15 16:30:54 PLATON: [Manager::configure] Loading TED...
2015/10/15 16:30:54 PLATON: [Manager::configure] Loading LSP-DB...
2015/10/15 16:30:54 PLATON: [Manager::configure] Loading Workflows...
2015/10/15 16:30:54 PLATON: [main] VTY listening on 127.0.0.1:2601
2015/10/15 16:30:54 PLATON: [main] Starting Manager
2015/10/15 16:30:54 PLATON: [Manager::start] Running startup sequence...
2015/10/15 16:30:54 PLATON: [Callback_Run_StartupStep] Starting Interface s_pcep
2015/10/15 16:30:54 PLATON: [InterfacePCEP::start] PCEP SERVER running on 172.16.0.2:4189
2015/10/15 16:30:59 PLATON: [Callback_PCEP_SessionUp] Session 0xac100001 is UP
2015/10/15 16:30:59 PLATON: [WorkflowEngine::processMessagePCEPPCRequest] PCEP_PCRequest SINGLE: using default OFcode=1
2015/10/15 16:30:59 PLATON: [WorkflowInstance::initiateWorkflow] Workflow completed TRUE / pending responses 0
2015/10/15 16:30:59 PLATON: [Callback_PCEP_SessionDown] Session 0xac100001 is DOWN
```

Figure 15. PLATON log after running example workflow

6 Performance evaluation

To demonstrate the feasibility of the front-end / back-end PCE architecture, we have carried out a number of tests to evaluate its performance. In the experiments, we considered two different real backbone network topologies. The 22-node and 35-link BT network topology, and the 30-node and 56-link TEL network topology. The current status of each network is synchronized using BGP-LS. A number of bulk path computations of different sizes are generated, and can be solved in the front-end PCE or delegated to the back-end PCE.

When the front-end PCE is in charge of performing the bulk computation, one single CPU thread is used; therefore, iterations of the provisioning algorithm are performed sequentially in the front-end PCE. In contrast, when the front-end PCE delegates the computation to the back-end PCE, the former opens a new PCEP session, creates a PCReq message containing every path request and a SVEC object, and forwards the message to the back-end PCE. Upon receiving a PCReq message, the back-end PCE uses the GPU device to accelerate the computation; each iteration is executed into an independent thread block in the GPU, and all the threads inside a block collaborate in solving the RSA problem in parallel for each demand in the bulk. The back-end PCE sends a PCRep message containing the route and spectrum allocation for each demand (Figure 16).

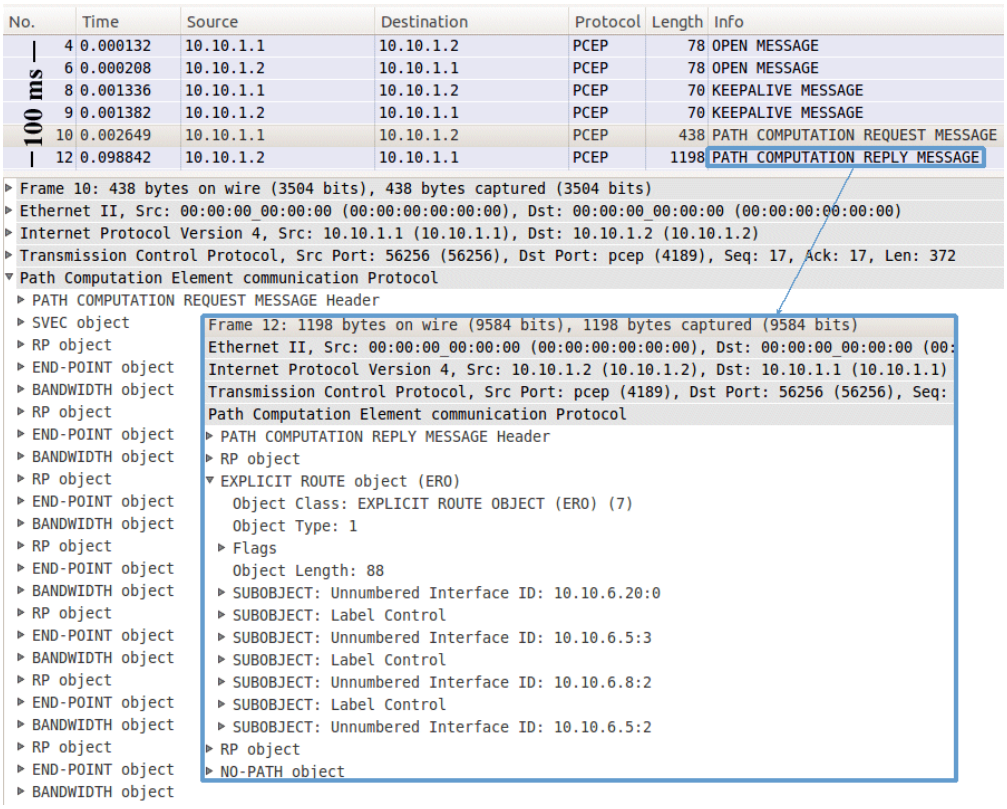

Figure 16: PCEP Messages

Table 50 details the solving times for three different bulk sizes. As observed, as soon as either the size of the bulk or the topology increases, the front-end PCE cannot perform the requested computation within 100ms. Note that bulk sizes in the order of 40-50 demands are common in restoration and re-optimization.

Table 50. Solving times

	BT network topology		TEL network topology	
Bulk Size	**Front-end PCE Time (ms)**	**Back-end PCE Time (ms)**	**Front-end PCE Time (ms)**	**Back-end PCE Time (ms)**
30	21.6	11.5	248.4	38.5
40	72.4	16.9	466.3	65.4
50	514.3	74.6	889.3	142.6

Once the front-end - back-end architecrture has proved to provide solving times reduction, let us analyse the solving times obtained from running the implemented algorithms. To that end, the BT and TEL networks topologies defined for the previous experiments, where used. Table 51 presentes the obtained results.

Table 51. Algorithms Solving times

Use Case	**BT**	**TEL**
Spectrum defragmenbtation (SPRING)	18 ms	35 ms
Re-optimization (AFRO)	21 ms.	43 ms.
Multicast Provisioning	3 ms.	12 ms.

7 Conclusions

This deliverable focused on the network planning tools to deal with on-line algorithms for next generation flexgrid optical networks, named PLATON.

Since PLATON focuses on in-operation network planning, it includes a BGP-LS interface to synchronize the state of network resources and a PCEP interface, which is used for both, synchronizing the estate of LSPs and receiving PCEP requests from a front-end PCE.

END OF DOCUMENT