D1.2 - Network Planning Tool: Architecture and Software Design

Status and Version:	Final version	
Date of issue:	04.11.2013	
Distribution:	Public Report	
Author(s):	Name	Partner
	Luis Velasco (Editor)	UPC
	Lluís Gifre	UPC
	Gabriel Junyent	UPC
	Jaume Comellas	UPC
	Emmanouel (Manos) Varvarigos	UPAT
	Aristotelis Kretsis	UPAT
	Polyzwis Soumplis	UPAT
	Andrew Lord	BT
	Amanda Azañon	TID
	Ignacio Digon	TID
Checked by:	Juan Fernandez-Palacios	TID

Abstract

D1.2 describes the Network Planning tool in terms of modules, relations among them and external interfaces. Software design of the tool will be specified using UML diagrams.

The proposed planning tool aims to support different routing and restoration mechanisms defined in the project and to be reported in D1.3.

Contents

1 Executive summary

Several planning tools are being developed within WP1, as a result of the wide range of problems to be solved. Those problems are related to the assignment of optical spectrum across specific network routes to meet traffic demands. This Routing and Spectrum Assignment (RSA) is an NP complete problem, and therefore highly computationally intensive. It therefore requires carefully crafted heuristic approaches to give solutions in realistic time scales, especially as the number of network nodes increases.

Two main categories of planning can be distinguished:

- *Off-line planning*. Here we have a network with some traffic requirements and we wish to optimise the location of equipment and the routing and spectrum allocations for the various traffic demands. Time can be taken to explore different scenarios to yield the best solution for the given requirements [1].
- *In-operation planning*. Here, the network is up and running and new real-time RSA decisions need to be made [2].

1.1 Off-line planning

Off-line planning is usually related to periodical planning to update the network. Figure 1 illustrates the classical network planning life-cycle, where a number of steps are performed sequentially. The initial step receives inputs from the service layer and from the state of the resources in the already deployed network and configures the network to be capable of dealing with the forecast traffic, for a period of time. That period is not fixed and actual time length usually depends on many factors, which are operator and traffic type specific. Once the planning phase produces recommendations, the next step is to design, verify and manually implement the network changes. While in operation, the network capacity is continuously monitored and that data is used as input for the next planning cycle. In case of unexpected increases in demand or network changes, nonetheless, the planning process may be restarted.

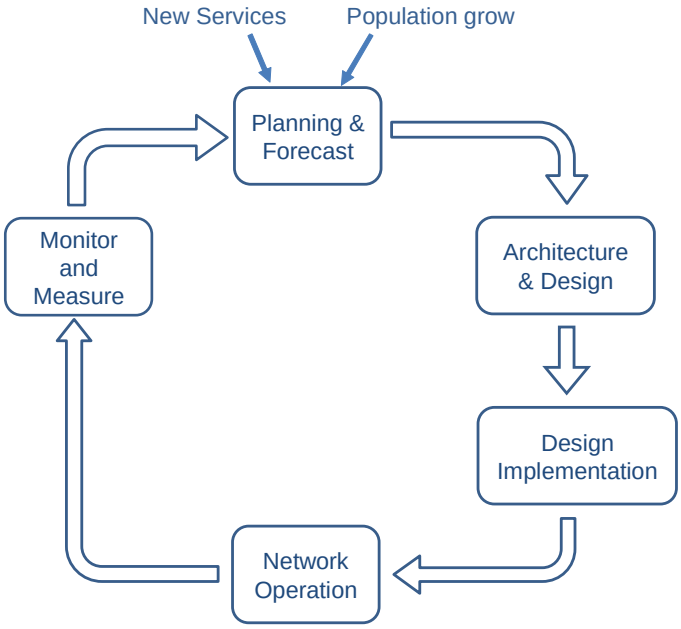


Figure 1: Classical network planning life-cycle.

Off-line planning is performed, probably taking as input the current state of the network, inventory information, etc. The following list of inputs involved in the process can be assumed:

- The Network Management System (NMS) managing the core network, implementing fault, configuration, administration, performance, and security (FCAPS) functions.
- A Planning Department administrating the planning process, i.e., analysing the network performance and finding bottlenecks, receiving potential clients' needs, evaluating network extensions and new architecture, etc.
- An inventory database containing all equipment already installed in the network, regardless they are in operation or not.
- An Engineering Department, performing actions related to equipment installation and set-up.
- A planning tool in charge of computing solutions for each migration step. Several sub-problems related to network reconfiguration, planning, and dimensioning, among others, need to be solved.

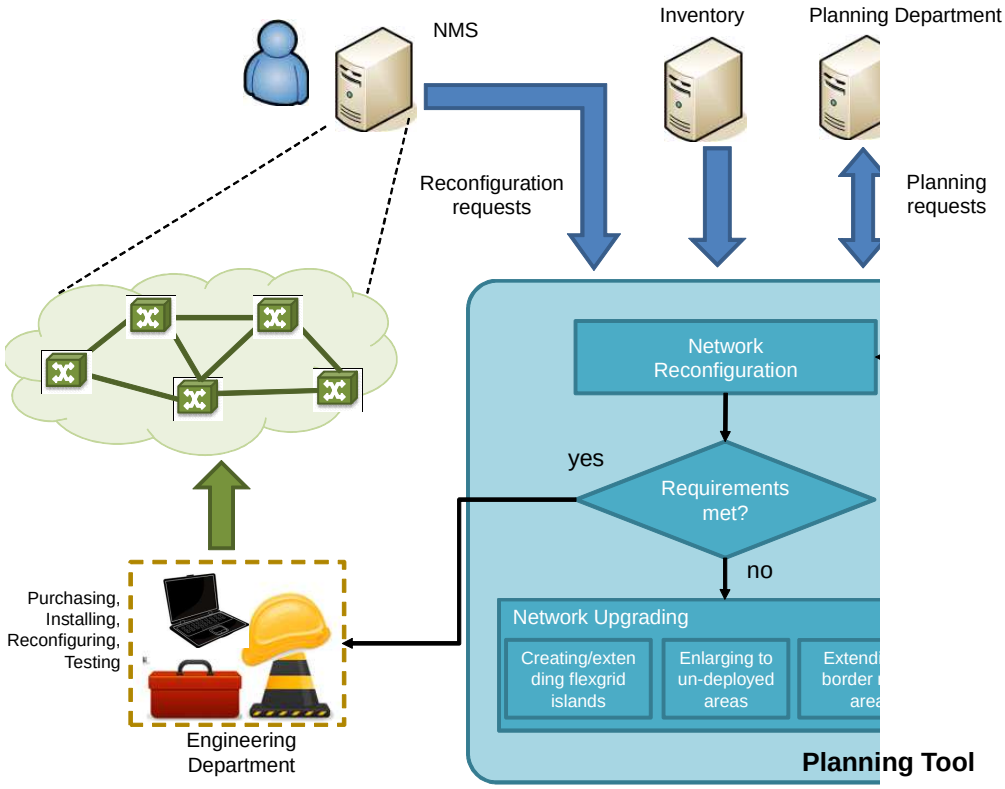


Figure 2: Off-line planning flow chart.

Off-line planning begins when the planning tool receives a request that can be originated in different systems responding to different reasons (Figure 2):

- Operators analysing data gathered by the NMS detect that a migration step can be attempted to improve the performance of the current network. E.g., bottlenecks have been detected in some parts of the network and its current configuration will not be able

to allocate expected traffic, so reconfiguration can be attempted. Note that these triggers arise asynchronously (i.e., without a predefined schedule).

- Planners request network re-planning to serve new clients or cover new areas. Contrary to reconfigurations coming from NMS, planning requests can be better synchronized with other network departments, such as the engineering department.

After a solution is found, new equipment needs to be purchased, installed and configured, before entering into operation.

1.2 In-operation planning

As technologies are developed to allow the network to become more agile, it may be possible to provide response to traffic changes by reconfiguring the network near real-time. In fact, some operators have deployed Generalized Multi-Protocol Label Switching (GMPLS) control planes, mainly for service set-up automation and recovery purposes. However, those control only parts of the network and do not support holistic network reconfiguration. This functionality will require an in-operation planning tool that interacts directly with the data and control planes and operator policies via Operations Support System (OSS) platforms, including the NMS.

Assuming the benefits of operating the network in a dynamic way are proven, the classical network life-cycle has to be augmented to include a new step focused on reconfiguring and re-optimising the network, as represented in Figure 3. We call that step in-operation planning and, in contrast to the traditional network planning, the results and recommendations can be immediately implemented on the network.

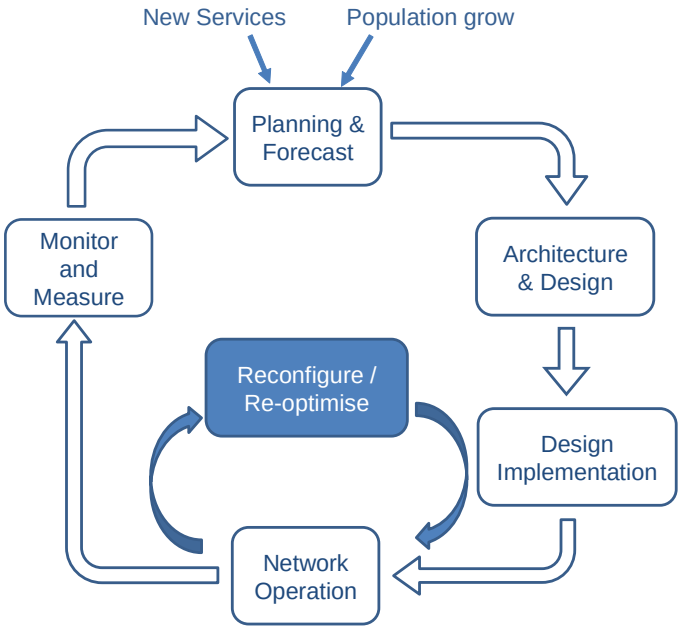


Figure 3: Augmented networks life-cycle.

Figure 4 illustrates the proposed control plane architecture to support flexgrid network re-optimisation, which also facilitates human verification and acknowledgement of network changes. When Router A needs a new connection to Router B, it sends a request to the

control plane of the optical network (1). After checking admission policies, a PCE Protocol (PCEP) message is sent to the PCE (2), which invokes its local provisioning algorithm (3). In the event of insufficient resources being available, e.g. due to spectrum fragmentation, the active PCE recommends the defragmentation of relevant nodes and connections, utilising the right algorithm to provide such re-optimisations. Let us assume that the back-end PCE providing such algorithm will perform the computation (4) upon receipt of a request. When a result is obtained (5), it is sent back to the front-end PCE (6). In case that an operator need to approve implementing the computed solution in the network, a request can be sent to the NMS/OSS (7). When the solution has been verified and acknowledged by the operator, the NMS/OSS informs the PCE (8) and existing connection reallocations are requested. Once the dependent connections have been setup, the responsible PCE will invoke the local provisioning algorithm for the original connection request between routers A and B and sends a PCEP message to the originating control plane node (9).

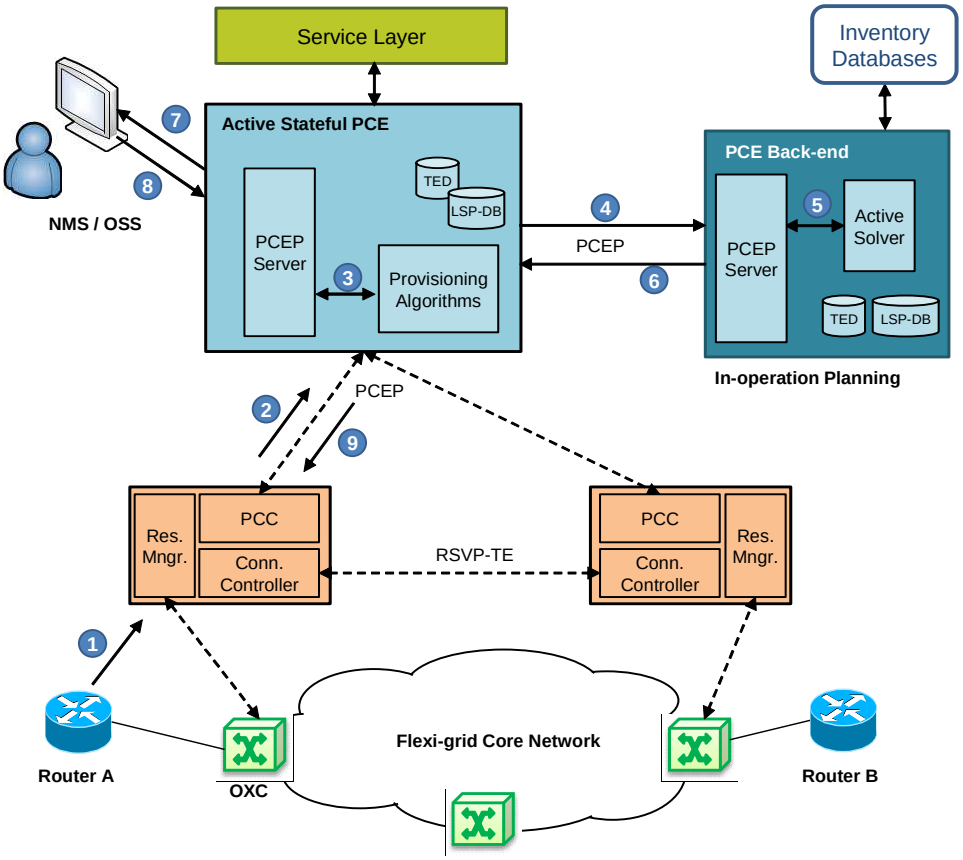


Figure 4: Proposed control plane architecture and network re-optimisation.

1.3 Planning Tools

With the above in mind, two distinct tools are being developed to handle the offline and in-operation planning paradigms:

- **MANTIS**. Predominantly off-line processing tool. Provides a repository for networks and algorithms – thus allowing easy comparison of different approaches and a ready-made benchmarking tool. Potential availability as a Cloud service via a web interface.

- **PLATON**. Off-line and In-operation planning tool. Addressing problems such as flexgrid design, post-repair optimisation and spectrum defragmentation.

Whilst a version of MANTIS existed pre-Idealist, PLATON is a new tool, being developed within the Idealist project to specifically address problems requiring real time decisions. The following table shows the intended development plan for PLATON.

Task	%Done	Year #1	Year #2	Year #3
PLATON	47%			
Cluster Manager	75%			
Requests Database	100%			
Manager	100%			
Management Web Server	100%			
Web-services Server	50%			
PCE Server	75%			
HPC Agent	50%			
Communications	60%			
Optimization Framework	40%			
Deliverable D1.2	95%			
Algorithms year #2	17%			
Single Layer Flexgrid Network Design Problem	50%			
After Failure Repair Optimization (AFRO)	0%			
Spectrum Defragmentation (SPRESSO)	0%			
Algorithms year #3	0%			
Define Algorithm Set	0%			
Implement Algorithms	0%			

This deliverable presents MANTIS and PLATON network planning tools. The several topics are presented for each tool:

a) Objectives and requirements
b) Tool Architecture
c) Software specification and design
d) Developer and user guides

The planning tools will be used to solve network design problems, such as the Single Layer Flexgrid Network Design Problem, or re-optimisation problems such as After Failure Repair Optimization (AFRO) and Spectrum defragmentation (SPRESSO) [3].

2 Introduction

2.1 Purpose and Scope

This is the second deliverable from Work Package 1 of Idealist. The motivation and an overview of this deliverable have been provided in the Executive Summary. The main document is organized into three main sections to cover MANTIS and PLATON tools, as well as an example of algorithm for a specific planning problem.

2.2 Reference Material

2.2.1 Reference Documents

[1] M. Ruiz et al., “Planning Fixed to Flexgrid Gradual Migration: Drivers and Open Issues,” accepted in IEEE Communications Magazine, 2014.

[2] L. Velasco et al., “In-operation Network Planning,” accepted in IEEE Communications Magazine, 2014.

[3] IDEALIST Project, “Deliverable D1.1 – Elastic Optical Network Architecture: reference scenario, cost and planning,” 2013.

[4] Mantis: http://www.mantis-tool.net/

[5] Mantis Python API: api.mantis-too.net

[6] Cython C-Extensions for Python: http://www.cython.org/

[7] Ruby on Rails open source web framework: http://rubyonrails.org/

[8] The Dojo toolkit: http://dojotoolkit.org/

[9] PostgreSQL open source object-relational database system: http://www.postgresql.org/

[10] Qt framework: http://qt-project.org/

[11] Python Programming Language: http://www.python.org/

[12] Raphaël JavaScript Library: http://raphaeljs.com/

[13] Cisco Visual Networking Index -Forecast and Methodology, 2010–2015.

[14] T. Stern, et al. “Multiwavelength Optical Networks: Architectures, Design and Control”, Cambridge University Press, 2nd Edition, 2008.

[15] O. Gerstel et al., “Elastic Optical Networking: A New Dawn for the Optical Layer?”, IEEE Communications Magazine, 2012.

[16] K. Christodoulopoulos, I. Tomkos, E. Varvarigos, “Time-Varying Spectrum Allocation Policies in Flexible Optical Networks”, IEEE Journal on Selected Areas in Communication, 31(1), 2013.

[17] K. Christodoulopoulos, P. Soumplis, E. Varvarigos, “Planning Flexgrid Optical Networks under Physical Layer Constraints”, accepted for publication in IEEE/OSA Journal of Optical Communications and Networking.

[18] A. Patel et al., “Defragmentation of transparent flexible optical WDM (FWDM) networks”, OFC, 2011.

[19] Ll. Gifre, L. Velasco, and N. Navarro, “Architecture of a Specialized Back-end High Performance Computing-based PCE for Flexgrid Networks,” in Proc. IEEE International Conference on Transparent Optical Networks (ICTON), 2013.

[20] JP. Vasseur, JL. Le Roux, “Path Computation Element (PCE) Communication Protocol (PCEP),” IETF RFC 5440, 2009.

[21] I. Nishioka, D. King, “Use of the Synchronization VECtor (SVEC) List for Synchronized Dependent Path Computations,” IETF RFC 6007, 2010

2.2.2 Acronyms

ADT	Abstract Data Type
API	Application Programming Interface
BVT	Bandwidth Variable Transponders
CLI	Command-Line Interface
FCAPS	Fault, Configuration, Administration, Performance, and Security
GMPLS	Generalized Multi-Protocol Label Switching
NMS	Network Management System
OSS	Operations Support System
PCC	Path Computation Client
PCE	Path Computation Element
PCEP	PCE communications Protocol
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
UML	Unified Modelling Language
WDM	Wavelength Division Multiplexing

2.3 Document History

Version	Date	Authors	Comment
Draft 1	6.9.13	Luis Velasco	1st draft. Contains placeholders for all

			contributions
Draft 2	14.10.13	Manos Varvarigos	UPAT contributions on Mantis
Draft 3	14.10.13	Lluís Gifre	UPC contributions on PLATON
Draft 4	22.10.13	Luis Velasco	First Integrated Version
Draft 5	25.10.13	Andrew Lord	Review and comments
Draft 6	28.10.13	Luis Velasco	Second Integrated Version

3 MANTIS

Mantis is one of the two IDEALIST network planning and operation tools for designing the next generation flexgrid optical networks. It includes novel flexgrid optical network algorithms for planning and operation functions. Mantis architecture permits fast execution of the included mechanisms, efficient usage of the computational resources utilized and enables the deployment of the tool in various computing environments (desktop, software as a service).

In what follows we start by outlining our motivation for building such a tool (Section 3.1), and then we focus on the Mantis tool. In Section 3.2 we present the architecture of Mantis tool and then in Section 3.3 we specify the technologies that we used to build it. In Section 3.4 we give details about Mantis execution process and then in Section 3.5 we present a short developers' guide for writing and incorporating in Mantis new algorithms. Finally in Section 3.6 we include a basic users' guide for Mantis, including many screenshots to guide the users to Mantis web-interface.

3.1 Objectives and requirements

Network planning is an important operation for all kinds of networks, and particularly for optical core networks that interconnect the other networks together. This makes network planning and operation tools for core networks a key part of all network performance and design.

Recent research efforts on optical networks have focused on architectures that support variable spectrum connections as a way to increase spectral efficiency and reduce capital costs. Flexgrid optical networks (elastic, or spectrum-flexible are other terms used to describe them) appear as a promising technology for meeting the requirements of next generation networks that will span across both the core and the metro segments, and potentially also across the access segment all the way to the end user. A flexgrid network migrates from the fixed 100 or 50GHz grid that traditional (Dense) Wavelength Division Multiplexing -WDM networks utilize [14], with granularities of 12.5 GHz or 6.25 GHz under discussion at standardization bodies, and can also combine the spectrum units, referred to as *slots*, to create wider channels on an as needed basis. This technology enables a fine-granular, cost- and power- efficient network able to carry traffic that may vary in time, direction and magnitude. Flexgrid networks [15] are built using bandwidth variable switches that are configured to create appropriately sized end-to-end optical paths of sufficient spectrum slots. We will refer to such a connection as a *flexpath*, a variation of the word *lightpath* used in standard WDM networks. Bandwidth variable switches operate in a *transparent* manner for transit (bypassing) traffic that is switched while remaining in the optical domain. Network equipment vendors and operators and the IDEALIST project are already looking into this technology.

A key enabler for the introduction of flexgrid technology will be the creation of network planning and operation tools that will run what-if scenarios and perform scalability and comparison studies to evaluate the benefits and enable the adoption of the candidate technology in real systems.

Also, the optical transport network, which currently relies on the slowly changing circuit switching paradigm, becomes more dynamic and moves towards the software defined networking paradigm. Among others, the transmission parameters that can be controlled include the baud rate (symbols processed per second), the modulation format (number of

bits encoded per symbol), the forward error correction (FEC) used, the spectrum slots employed, the useful bit rate, and the spectrum space left between flexpaths. The multiple degrees of freedom present in future optical networks make connection establishment a more complicated problem, where a centralized controller will make the important networking decisions, in close collaboration with a Network planning and Operation Tool. Figure 5 presents an example of a flexgrid network that is controlled by a centralized Network planning and Operation Tool.

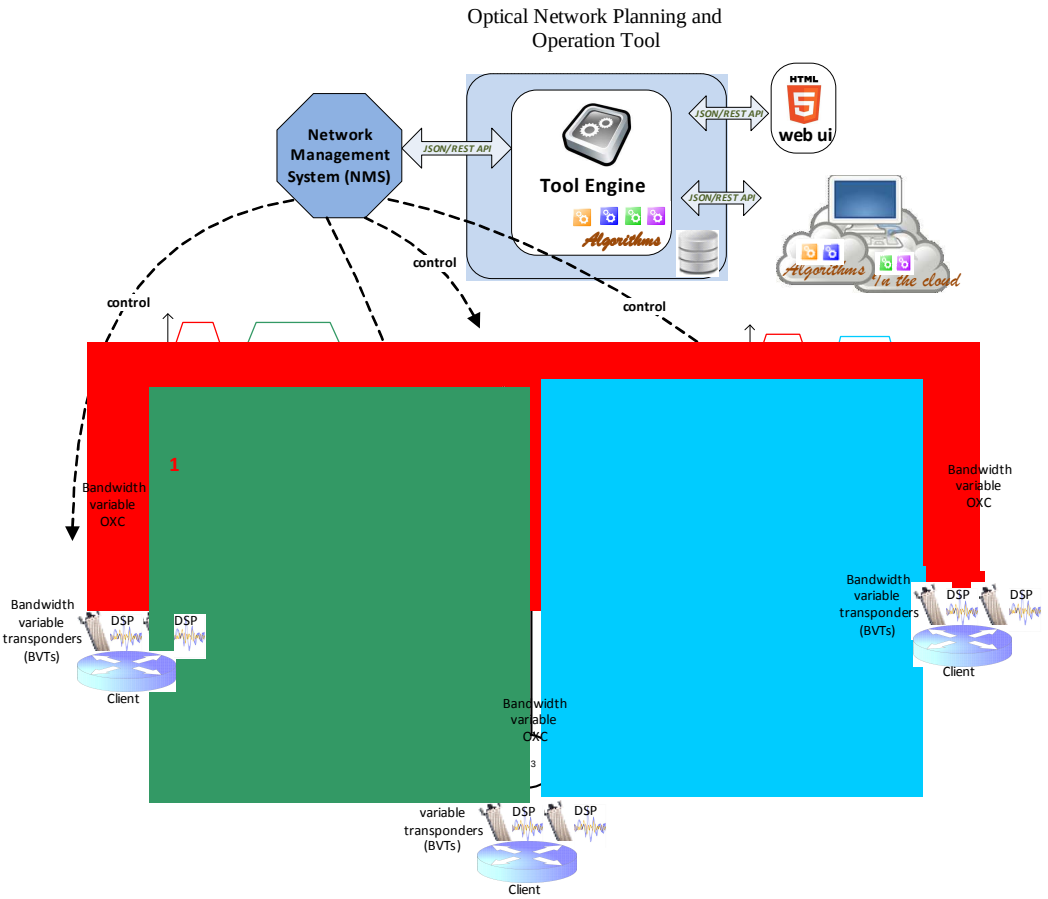


Figure 5: A flexgrid network, comprising bandwidth variable optical switches and transponders that it is operated by an optical network planning and operation tool.

It is evident that the future increase of requested network capacity, the emergence of flex-grid technology and software defined networking paradigm along with new application methodologies (clouds, SaaS, social by design) require not simply the extension of existing network planning tools, currently available from several major players, but the implementation of new ones. In what follows, we describe the basic algorithmic challenges that a network planning and operational tool for flex-grid optical networks should address, and present such a tool, called Mantis, that contains a rich library of algorithms that is currently extended to meet the majority of related challenges. Mantis was designed and implemented so as to accommodate both desktop and cloud execution.

3.1.1 Algorithmic Issues in Flexgrid Optical Networks

Network planning typically occurs before a network is deployed and focuses on how to better accommodate the current and foreseen network traffic. Greenfield planning refers to the case where the network, including its physical topology, is designed from scratch, and can be very demanding, since many costs, such as the cost of digging ducts to lay the fiber cables, and the cost of the fibers, amplifiers, switches, transponders and regenerators to be installed, have to be accounted for. More common in optical transport networks is to plan the network assuming that the fibers are already installed, which is the reality for tier-1 and tier-2 operators. In this case the topology is given and the purpose is to decide the equipment (transponders, regenerators) to be purchased and where to deploy it, and the switches that may have to be replaced. The objective is to minimize the equipment cost and the resources used for serving the given traffic (saving resources for future use or, in case of infrastructure leasing, for cost minimization). In the network operation phase, the demands are generally processed upon their arrival, one or a small set at a time, and the traffic is accommodated using the equipment already deployed in the network or when necessary some little additional equipment. Therefore, the operation process must take into account any constraints posed by the current state of the network. Following the above, the algorithms for optical transport networks can be broadly classified into (i) static (planning) and (ii) dynamic (operating) algorithms. In both cases, the issues that have to be addressed for flexgrid optical networks are:

- Accounting for physical layer impairments
- Routing, spectrum allocation, and choice of modulation format
- Serving dynamic traffic fluctuations and spectrum fragmentation

3.1.1.1 Accounting for Physical layer impairments

Optical transport networks have evolved over recent years from opaque (point-to-point) to transparent networks, as a way to reduce CAPEX and OPEX costs. In the latter case, optical switches are configured to transparently handle transit traffic; the signal remains in the optical domain, saving on the cost of transponders used in the past to terminate and retransmit traffic at intermediate hops. Since optical connections may span over many and long links, physical layer impairments (PLIs), such as noise, dispersion, interference, accumulate and affect the quality of transmission (QoT). Accounting for PLIs is a challenge for algorithm designers, especially with respect to their exact modelling and the interdependencies introduced.

PLIs affect both fixed grid WDM and flexgrid networks, but there are distinct differences between the two cases. When tuneable transponders are present in flexgrid networks, there are a number of transmission parameters that can be controlled that change the effects of the PLIs and directly or indirectly affect the connection establishment decision. Note that flexgrid networks are expected to use coherent detection and DSP, implying that impairments, particularly those related to dispersion, will be substantially reduced or fully compensated. However, the transmission configurations of the transponders have to be included in the RSA problem, and the multiple degrees of freedom present in flexible optical networks and their interdependencies make connection establishment in such networks more complicated than in traditional fixed grid networks. On the other hand, in WDM networks, PLIs, even though more significant under the assumption that coherent detection is not used, can be accounted for quite accurately, since fewer parameters are involved (non-tunable transponders and constant guard-band) and analytical models successfully capture these effects.

So the most difficult part in accounting for the physical layer in flexgrid networks is the interrelation between the physical layer, the transponders' configuration parameters, and the connection establishment process.

3.1.1.2 Routing and Spectrum Allocation

The problem of establishing connections in fixed grid WDM networks is typically referred to as the Routing and Wavelength Assignment (RWA) problem, known to be NP-complete. Connection establishment in flexgrid networks is more complicated for several reasons. First, in contrast to WDM networks where each connection is assigned a single wavelength, in flexgrid networks, spectrum slots can be combined to form variable width channels, leading to the so-called Routing and Spectrum Allocation (RSA) problem. Additionally, the transponders' (BVTs') adaptability yields many transmission options, each with different transmission reach and spectrum used (see discussion in previous subsection). Algorithms that try to capture, to varying degrees, the problem of jointly allocating recourses and selecting the transmission configurations are referred to as Distance Adaptive Spectrum Allocation or as Routing, Modulation Level and Spectrum Allocation (RMLSA) algorithms.

Another difference between the RWA and RSA problems relates to the number of connections served by a transponder. In WDM networks, a transponder is allocated to a single connection. In flexgrid networks, researchers are considering powerful BVT, called sliceable bandwidth variable transponders (SBVT), which can be shared to serve more than one connection. SBVTs introduce an additional degree of flexibility in that multiple flexpaths can be assigned to SBVTs, in order to keep them highly utilized and reduce their number and total cost.

Both RWA and RSA include as a sub-problem the placement of regenerators in the network. Given the BVTs' limited transmission reach, due to PLIs, regenerators are used to establish lengthy connections. However, in contrast to WDM networks where the capabilities of the BVT and regenerators are given, in flexgrid networks the transmission reach depends on the transmission configuration of the tunable transponder and can be controlled. Thus, regenerator placement in flexgrid networks also involves choosing the BVTs' configurations making it more complicated than in fixed grid networks.

3.1.1.3 Dynamic network operation

The network is typically initiated with an offline/planning algorithm assuming an oversubscribed traffic matrix, meaning that actual traffic is on average as low as 30% of the traffic described in the matrix, to absorb short term fluctuations (e.g. daily cycles) and avoid frequent network upgrades (long term traffic increases eventually require upgrades, of course). Thus, the network is operated in an incremental manner, with new connections added sporadically, when utilization between endpoints exceeds a certain percentage, and existing connections rarely (if ever) terminated. This practice is rather different than the dynamic scenarios usually considered in the literature that assume the dynamic establishment and release of connections and measure the blocking probability over time. Thus, a more appropriate model for fixed grid WDM networks would be to incrementally add connections and observe when the first connection blocking occurs. Flexgrid networks using adjustable transponders (BVT) require a different approach as their operation will be more dynamic, having time scales at which optical connection rate changes occur probably 1-2 orders of magnitude smaller than in fixed grid networks. Flexgrid can bring the optical

layer closer to the IP layer, making the IP layer able to “dial”/control the bandwidth that it uses.

Dynamic traffic variations in flexgrid networks can be accommodated at two different levels. We consider the first level to be the establishment of new connections, as in fixed grid networks. Given the high capacity that flexgrid BVTs are expected to transmit (designs of 400 Gb/s or higher have appeared in the literature), relatively long periods of time will pass until a new connection is established, probably longer than in WDM networks. A second level is to absorb changes in the requested rate that are short- or medium-term by adapting the BVT, e.g., tuning the modulation format and/or the number of spectrum slots they use, a feature not available in WDM systems.

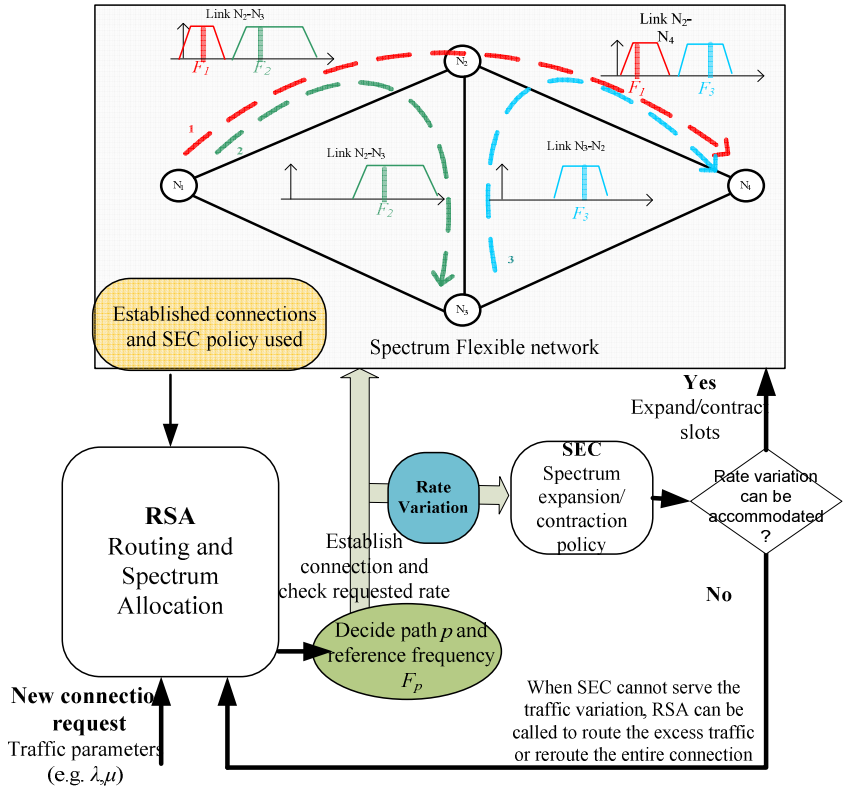


Figure 6: Flow diagram of a generic approach to operate a flexgrid Network.

Figure 6 describes a generic approach for operating a flexgrid network. The offline algorithm used to initialize the network, or the dynamic online algorithm that subsequently adds flexpaths, assigns to each flexpath a path and a reference frequency. A flexpath occupies a certain amount of spectrum slots around that reference frequency, and traffic variations can be absorbed by the BVT by tuning the modulation format or expanding/contracting the spectrum they use. Slots that are freed by a flexpath can be assigned to different flexpaths at different time instants, obtaining statistical multiplexing gains. To enable the dynamic sharing of spectrum, we need Spectrum Expansion/Contraction (SEC) policies [16] to regulate how this is performed. An example of such a policy would be to increase and decrease the spectrum slots in a symmetric way, i.e., alternate between expanding towards the higher and lower spectrum slots of the

reference frequency of the flexpath, until it reaches a slot already occupied by another connection, at which point we allocate slots from the side that we can. An RSA operation is performed again when a SEC policy cannot absorb traffic variations by granting additional spectrum slots, or when the requested rate exceeds the transponder capabilities. Then, RSA is called to route the excess traffic over a different flexpath, or reroute the entire connection (to save in guard-bands).

After establishing and tearing down multiple flexpaths in a flexgrid network the spectrum slowly becomes fragmented, reducing its ability to accommodate new connections [18]. This problem is much less significant in fixed grid WDM networks, where each connection is assigned a single wavelength. A number of defragmentation solutions have been proposed for flexgrid networks, broadly classified into proactive and reactive approaches. A proactive approach can be an RSA algorithm that avoids fragmentation by trying to leave spectrum voids that are usable, while a reactive approach can be a special defragmentation algorithm that reroutes/re-optimizes the network and frees spectrum for new connections.

3.2 Mantis architecture description

Mantis is a network planning and operation tool for designing the next generation optical networks. Our approach was to build a tool that provides fast execution, efficient computation resources usage, basic fault tolerance, scalable with the demand and can be easily extended with new algorithms and features. Another key feature of Mantis is that it has been designed to run as a desktop application but also in the cloud as a software as a service (SaaS).

Mantis components are organized in three layers: the **access layer**, the **application layer** and the **execution layer**. In addition, there are two common interfaces whose primary purpose is to provide loose coupling between the application layer and the other two layers. By using these interfaces we can have the same access and execution layers for both versions of the tool (desktop and cloud) while we can extend their functionality without breaking the implementation of the other components. Figure 7 shows Mantis architecture and its main components.

The **access layer** handles the interaction with the users through a web-based user interface and its exposed RESTful API. Through the Mantis simple web-based interface, users can have access to all tool's functionalities, perform easily all the supported operations and collaborate with other users. Furthermore, there is available a Python library that utilizes the RESTful API for communicating with the tool and provides almost the same functionality with the web-based interface. More interfaces to the Mantis functionalities can be added by extending the access layer to include a command-line interface (CLI) and to provide appropriate interfaces in order to expose Mantis's online algorithms to network management tools for optical networks providing path computation element (PCE) functionalities.

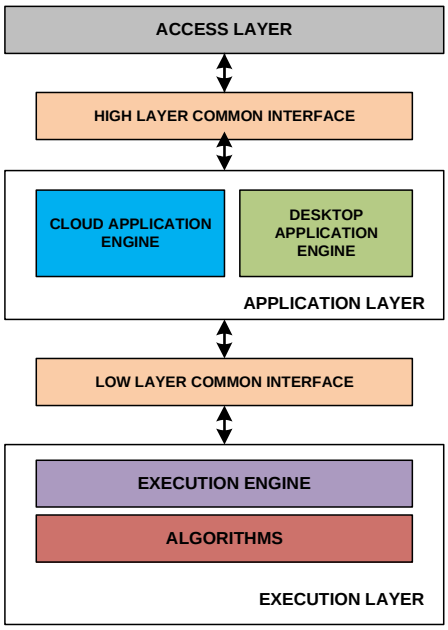


Figure 7. Mantis general architecture and its main components.

The **execution layer** consists of the execution engine and the library of available network planning and operation algorithms. Execution engine receives requests, for starting or terminating algorithm executions, through the common interface from the application layer and is responsible for performing all the required actions, including the preparation of the execution environment, the monitoring of the execution progress and the handling of the final results or possible failures. Furthermore, it is responsible to monitor the resources of the node where is executed and to return details such as the number of available cores, the used cores, the total and used memory etc. There is also a limit on the number of concurrent executions at every execution node which depends on the computation capacity and the number of the available cores.

The **application layer** implements the application logic and orchestrates the execution of user requests. It is the only layer that differs between desktop application and cloud service deployment as there are different requirements and operations that should be performed.

When Mantis is deployed as desktop application there is a server that contains the desktop application engine and the execution layer implementations (Figure 8). The desktop application engine receives requests from the access layer and stores them in a local queue and a disk file that provides a simple fault tolerance mechanism, eliminating the possibility of requests getting lost or not served due to server problems. Then the users' requests are forwarded to the local execution node. Furthermore, the desktop application engine is designed to limit the number of concurrent executions based on the capabilities of the hosting machine in order to avoid resources saturation since the algorithms are executed only in the machine where the server is deployed.

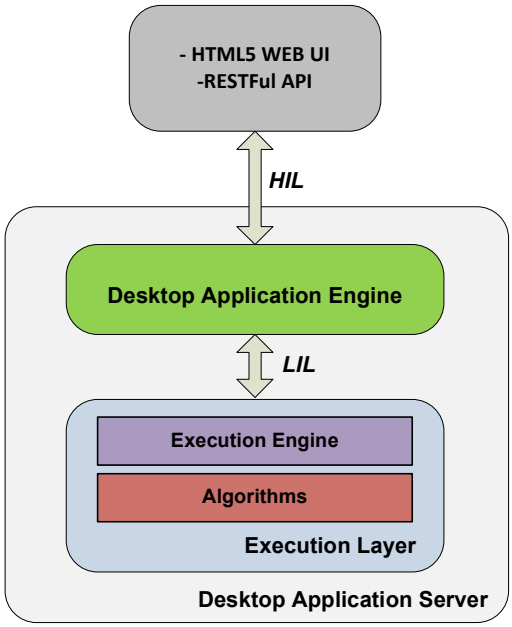


Figure 8. Mantis as desktop application.

When Mantis is deployed as cloud service, the application layer should implement the cloud application engine that handles the interaction with the cloud infrastructure. Figure 9 presents Mantis when deployed as cloud service. In this case, there is available one cloud engine but multiple execution engines, one at every computing node in the available cloud resources.

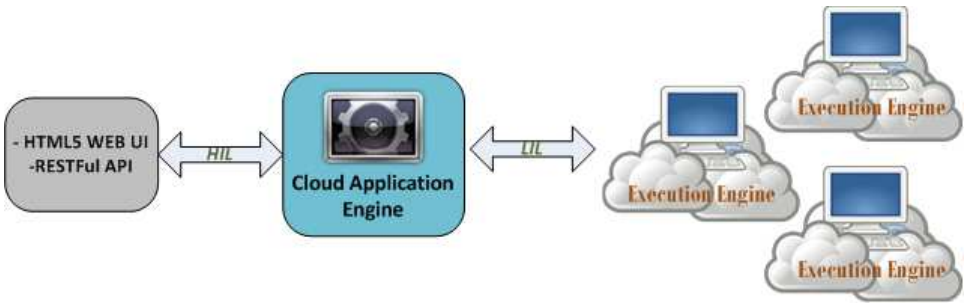


Figure 9. Mantis as Cloud service.

3.3 Software Implementation Technologies

In this section we present the technologies that we used in order to implement the various components of the Mantis architecture. Since, there are many different technical solutions available today it was necessary to select one of them based on some criteria. For Mantis implementation, we used software technologies and toolkits that are open source, mature, widely supported, have good performance and are able to scale in order to have a robust software system that fulfils all the design goals.

The access layer implementation is based on Rails [7] web application development framework, which is written in Ruby language. We have chosen Rails since is well-matched to the practices of agile software development, particular in its emphasis on software

testing and “convention over configuration”, as a means of avoiding writing the same code repeatedly. In Rails we have implemented both the RESTful API and the access layer’s application logic that handles the interaction with the database and other tool’s layers. The web-based user interface is a client-side rich internet application based on the Dojo [8] JavaScript toolkit. Dojo is an open source high quality toolkit where everything is customizable and which provides all the necessary modules to create a rich and full functional user interface. Regarding the system database we have selected the PostgreSQL [9], an open source and robust object-relational database system that features all the necessary characteristics and which is compatible with all the other implementation technologies.

In the execution layer, the execution engine is written in C++ programming language using the cross-platform Qt [10] framework that is widely used for developing application software. Qt runs on the major desktop platforms and includes among others a unified cross-platform API for SQL database access, thread management and network support.

Mantis algorithms are written either in C++ or in Cython [6] and are accessible from the tool through the execution engine’s custom plug-in mechanism. This mechanism leverages the Qt capabilities for extending an application’s functionality by defining and implementing a common interface through which various modules interact with the application and expose their functionalities.

3.4 Software Design

In this section we give details on the execution of Mantis process through various basic use cases diagrams. Furthermore, we present the Mantis database structure.

In Mantis users can interact with the following main entities:

- **Network topology:** represents a fixed network, providing details for the nodes, links and their lengths.
- **Traffic demands:** a square matrix that includes the request demands between the network nodes.
- **Configuration:** a set of parameters required for the execution of a planning or operation function (algorithm).
- **Instance:** an executed configuration.
- **Projection:** a parameterized version of some configuration, which causes a set of instances to be generated that differ in the value of one initial parameter (e.g., number of wavelengths).
- **Charts:** Visualize the results from various executions.

We provide examples of use case diagrams that describe the various interactions between a user and Mantis. Generally, users are able to create, edit, delete, share and clone network topologies, traffic demands, configurations and charts (Figure 10).

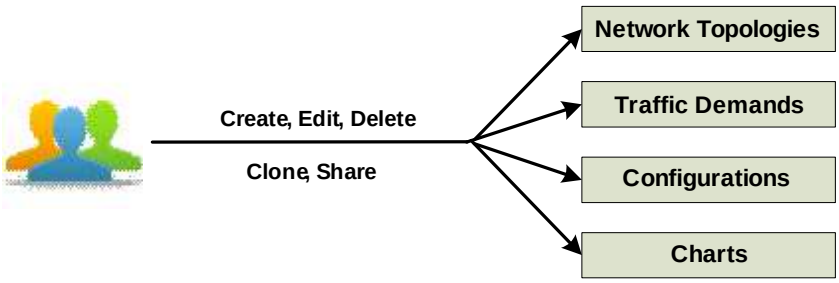


Figure 10. Available actions for the main entities.

Before users can request the execution of any algorithm it is necessary to create a configuration that provides the values for all required parameters. Initially, a user should select the algorithm for which the configuration will be defined and then should select a network topology, traffic demands and specify the other configuration parameters (Figure 11).

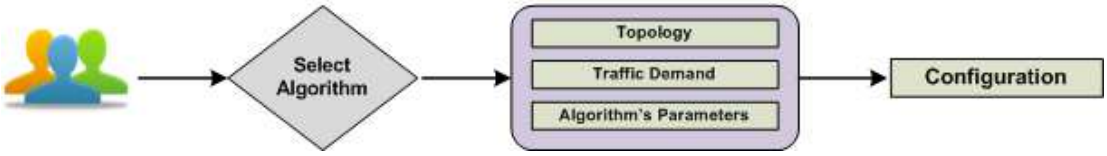


Figure 11. Procedure to create a new configuration.

For every configuration, users can request either its execution or the creation of a new projection (Figure 12). In the second case, it is necessary to select the parameter in which the projection's instances will differ and define also a range of values for it.

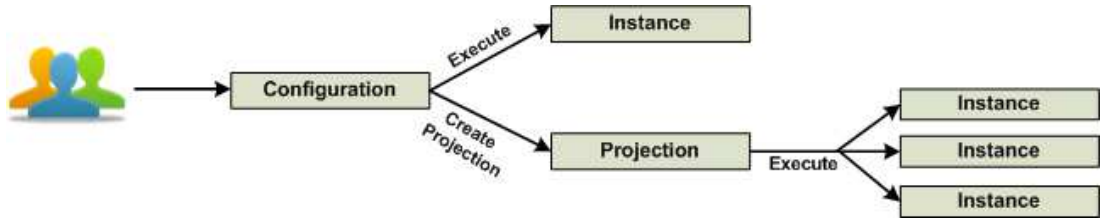


Figure 12. A configuration can be either executed or used to create a projection.

For every successfully executed instance users can view its analytical results, export the proposed solution and use its results to create charts (Figure 13).

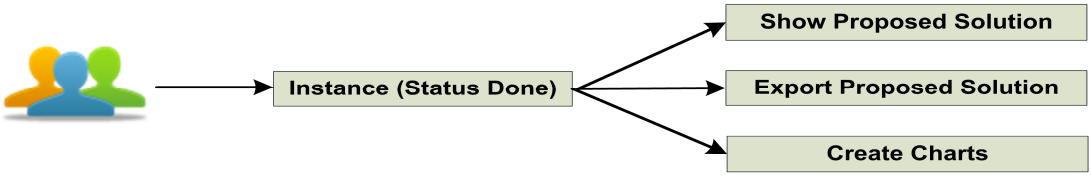


Figure 13. Available actions for successfully executed instances.

Furthermore, users can create charts to visualize the results from various executions by combining the results of successfully executed instances in various data series, while each chart can contain one or more data series (Figure 14).

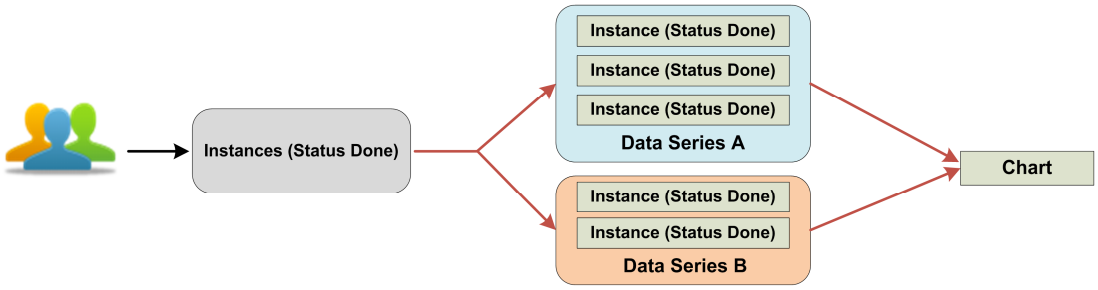


Figure 14. Procedure to create charts using the results from various instances.

Database is an essential component for the Mantis operation, since it contains both system and user data. This information is accessed mainly from the access layer and the available client library. Furthermore, access to database has also the application layer in order to setup an algorithm execution or to store its results. Mantis database model includes ten tables and is presented in Figure 15.

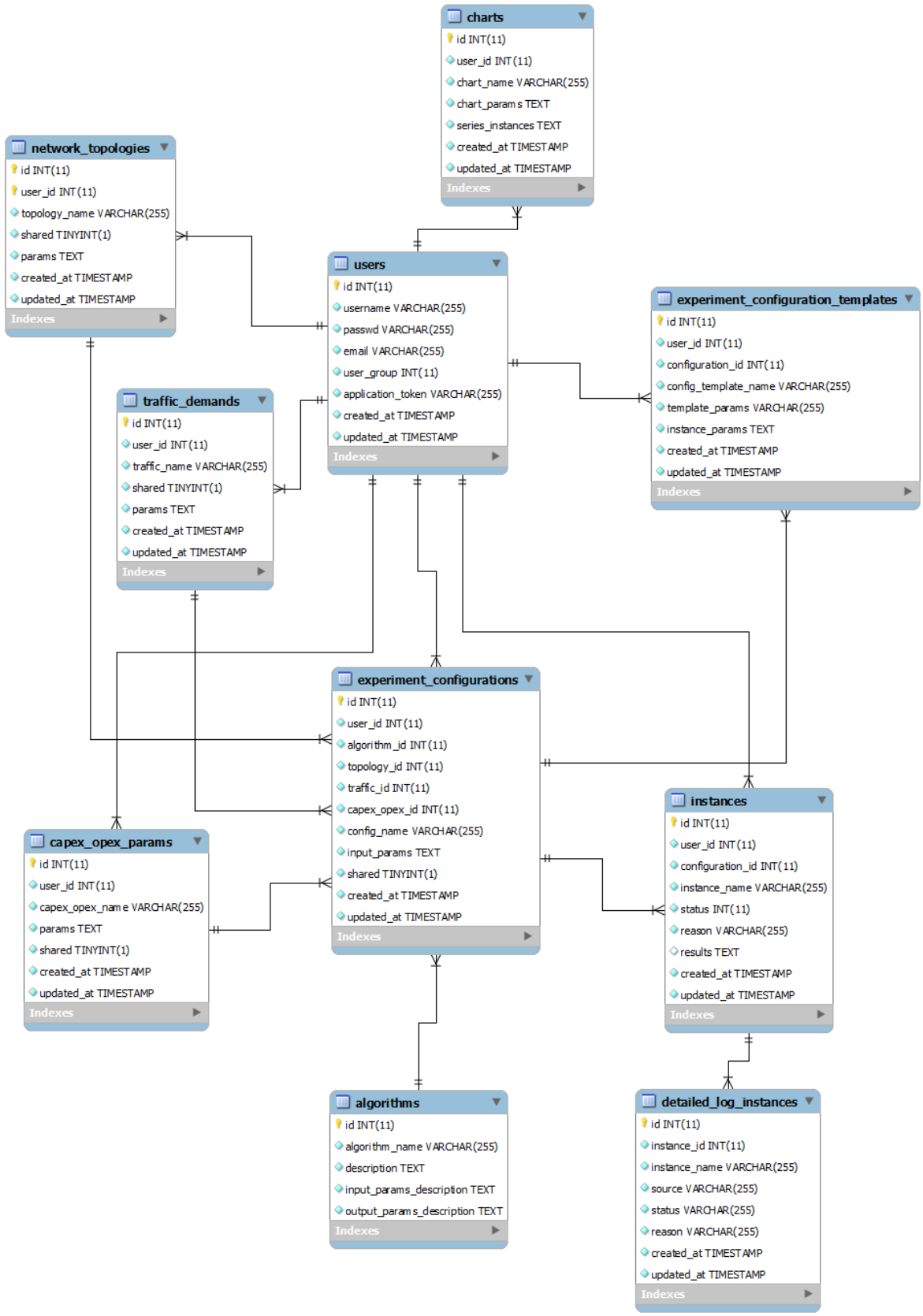


Figure 15. Mantis database model.

3.5 Developer Guide

In this section we outline the process that developers have to follow to write and integrate to Mantis new algorithms. Also we present the Mantis Python library that users can utilize to interact programmatically with the tool.

3.5.1 The Plug-in Mechanism

One primary concern during Mantis design was to build a system that not only will execute the algorithms in reasonable time but also will provide all the necessary functionality and features so as to be easily extendable with new algorithms and features without breaking the current implementation of its components.

According to Mantis architecture the execution layer is composed from the execution engine and available network planning and operation algorithms. Mantis algorithms are accessed from the execution engine through a custom plug-in mechanism. This mechanism enables new algorithms to be added in the tool without any modification in the application layer and the execution engine.

In the current version of the tool, algorithms can be implemented either as Python/Cython modules or as Qt plug-ins written in C++.

Generally, the procedure for adding a new algorithm in Mantis is relatively simple and comprises the following steps:

1. Implement an algorithm either as Python/Cython module or as Qt plug-in using the common interface.
2. Create input parameters and results descriptions based on the Mantis JSON schemas (presented next) and provide any dependencies in software packages and libraries.
3. Insert into tool's database the necessary information about the new algorithm (name, short description, description in JSON format for its input parameters and results).
4. Add new algorithm in the execution engine's library.
5. Update the web-based interface with the appropriate forms to create configurations for the new algorithm.

The first four steps are mandatory, while the fifth one that involves the implementation of the appropriate forms in the user interface can be omitted. However, in this case the users will be able to interact with this algorithm only through the Python Mantis library (Section 3.5.2). Developers of new algorithms perform only the first two steps while the remaining steps are performed by the Mantis administrator. In the current version of the tool the necessary steps for adding a new algorithm are executed manually. Nevertheless, we plan in future releases to automate as much as possible the execution for steps 3 and 4 as Mantis vision is to enable users to develop their own algorithms and plug them into the core platform, for evaluating their performance and comparing them with existing ones.

The execution engine's plug-in mechanism exposes a common interface, independently from the implementation technology, which determines explicitly the syntax of the input

parameters and the results for all algorithms. This common interface uses the JavaScript Object Notation (JSON) as data-interchange format for providing the input and output parameters for all Mantis algorithms.

In particular, every algorithm in Mantis returns the output results as JSON object and takes as input the following four parameters, using predefined JSON schemas; in the order they are presented:

1. Network topology description
2. Traffic demands description
3. Algorithm specific parameters
4. CAPEX/OPEX parameters

The JSON schema used for describing the traffic demands for a network topology of N nodes, contains an object with one variable named "***traffic_matric***", of array type; this array contains N arrays of number type with size 1xN. This variable describes the traffic demands between each pair of adjacent nodes (Figure 16). A network topology with N nodes is defined in JSON format, as an object with two elements the "***number_of_nodes***" and the "***links***" of array type. Each element of the array is another object that describes a unidirectional link in the network topology. These objects have two elements the "***length***" that describes the link's length in kilometers and the "***nodes***" which is a 1x2 array that contains the link's nodes.

In Figure 16 we present an example for a network topology with six nodes and nine unidirectional links, while Table 1 shows the corresponding Mantis JSON schemas descriptions for the network topology and the traffic demands.

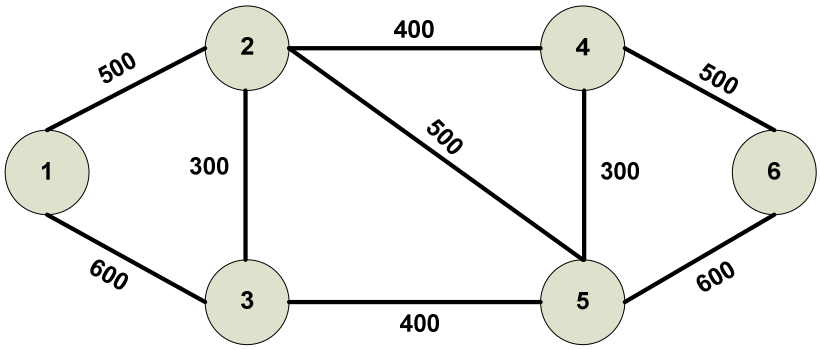


Figure 16. A 6-node network topology.


```
{
   "number_of_nodes": 6,
    "links": [
       { "nodes":[0,1] , "length":500 }, { "nodes":[0,2] , "length":600 },
       { "nodes":[1,0] , "length":500 }, { "nodes":[1,2] , "length":300 },
       { "nodes":[1,3] , "length":400 }, { "nodes":[1,4] , "length":500 },
       { "nodes":[2,0] , "length":600 }, { "nodes":[2,1] , "length":300 },
       { "nodes":[2,4] , "length":400 }, { "nodes":[3,1] , "length":400 },
```

```
{
   "traffic_matrix": [
       [0, 300, 200, 300, 100, 50],
       [10, 0, 100, 20, 80, 90],
       [430, 340, 0, 40, 20, 20],
       [10, 10, 20, 0, 120, 140],
       [130, 320, 220, 220, 0, 80],
```

{ "***nodes***":[3,4] , "***length***":300 }, { "***nodes***":[3,5] , "***length***":500 }, { "***nodes***":[4,1] , "***length***":300 }, { "***nodes***":[4,2] , "***length***":400 }, { "***nodes***":[4,3] , "***length***":300 }, { "***nodes***":[4,5] , "***length***":600 }, { "***nodes***":[5,3] , "***length***":500 }, { "***nodes***":[5,4] , "***length***":600 }] }	[130, 210, 120, 10, 40, 0]] }
(a)	(b)

Table 1. (a) 6-node topology in Mantis JSON (b) traffic demands description example.

Furthermore, the access layer and the execution engine need to know for each algorithm the detailed description of its input and output parameters in order to be able to prepare the execution and handle its results. However, this information is different for each algorithm and cannot be automatically determined by the tool. For this reason, whenever we integrate a new algorithm in Mantis execution layer, it is necessary to provide the description of its input and output parameters.

The JSON schema used for describing the input parameters contains an object with three members named ***"container_name"***, ***"input_parameters"*** and ***"output_parameters"***. The first member provides the name of the module or plug-in that contains the algorithm implementation, while the other two that are arrays of objects describe the input and output parameters. Each object has the following three members:

- ***name:*** object's member name that contain the parameter
- ***description:*** a short description for the input or output parameter
- ***type***: parameter's data type, supported types *integer*, *float*, *string*, *dictionary*, *list* and *list-2* (two-dimensional arrays)

Table 2 presents the description for an algorithm which expects its input parameters to be members of an object with names ***"param1"*** and ***"param2"***. The algorithm's results are returned as an object with three members named as ***"result_1", "result_2"*** and ***"result_3"***.

```
{
        "container_name": "new_algorithm",
        "input_parameters": [
                {"name":"param_1", "description":"short description", "type":"integer" },
                {"name":"param_2", "description":"short description", "type":"float" }
        ],
        "output_parameters": [
                {"name":"result_1","description":"short description", "type":"integer"},
                {"name":"result_2","description":"short description", "type":"float"},
                {"name":"result_3","description":"short description", "type":"list"}
         ]
}
```

Table 2. Algorithm's input and output parameters description

3.5.2 Mantis Python Library

The access layer exposes all Mantis functionality through a RESTful API which enables the Mantis usage directly over HTTP. The RESTful API can be useful when users want to utilize Mantis services without using the existing web-based interface and Python library or when want to integrate their own existing tools and environments. All requests to the RESTful API are authenticated with HTTP Basic Authentication, which is based on the users' username and password in Mantis, while the responses are formatted in JSON, which is the default option, or in XML.

We have developed a Python library that utilizes this RESTful API for communicating with the tool. Users can easily install it and interact programmatically with the tool, while the only requirement in order to use the library is to have a valid account in the online tool.

Mantis Python library is composed of four main modules *traffic*, *topology*, *configuration* and *instance* that handle the interaction with the corresponding entities in the tool. These modules contain all the necessary methods for retrieving network topologies, traffic demands, configurations, instances and for retrieving detailed information regarding their characteristics and their status.

A user through the Mantis Python library can create, edit, delete, clone and make public or private network topologies, traffic demands and configurations. In addition, the library contains methods for filtering the returned information based on various parameters, monitoring the execution of all running instances, executing configurations and querying their current status. The complete documentation for the Mantis Python library is available on [5].

For example, the method *get_instance* with argument an instance identifier *'1010',* returns (in JSON format) the following information about the specified instance:

- instance unique identifier (*id*)
- configuration name from which the instance has been created (*config_name*)
- instance name that is automatically created from the system (*instance_name*)
- execution results (results)
- current execution status code of the instance (*status*)
- current status description (*reason*)
- timestamps for the creation (*created_at*) and the last update (*updated_at*)


```
{
   'id': 1010, 'config_name':'ofdm_test_1', 'config_id': 21,
   'instance_name': 'ofdm_test_1_6175', 'results': { }, 'status': 3, 'reason': 'RUNNING',
   'created_at': '2013-06-26T08:42:55Z', 'updated_at': '2013-06-26T08:43:04Z'
}
```


Table 3. Details about some instance.

In the current version of the tool, users cannot directly deploy and execute their own developed algorithms. However, they can still compare the results from their own algorithms, which are executed locally in their own machines, against the results from the ones already incorporated in the tool by using the provided Python library.

In Table 4 we present an example of using the library for that purpose. Initially, a user identifies from the user interface an instance whose proposed solution he wants to compare with his algorithm. Using the available methods he is able to retrieve all the configuration details (network topology, traffic demands, CAPEX/OPEX and input parameters) along with the detailed solution for the specified instance. Hence, he is able to compare the results of his own locally executed algorithm with the initial instance using the same network topology, traffic demands and any other input parameters.

```
import mantis.traffic
import mantis.topology
import mantis.instance
import mantis.configuration

# Return the details for the instance with name instance_111
instance_details = instance.get_instance({"name":"instance_111" })

# From the available information, keep detailed solution and configuration id
config_id = instance_details["config_id"]
instance_results = instance_details["results"]

# Get details about the instance's  configuration
configuration_details = configuration.get_configuration({id:config_id})

# Get network topology and traffic demands ids
topology_id = configuration_details["topology_id"]
traffic_id = configuration_details["traffic_id"]

# All configure parameters and their values
configuration_input_params = configuration_details["input_params"]

# Get details and description for network topology with the specified id
# topology_details["params"] contains the description of the requested network topology
topology_details = topology.get_topology_by_id(topology_id)

# Get details and description for traffic demands with the specified id
# traffic_details["params"] contains the description of the requested traffic demands
traffic_details = traffic.get_traffic_by_id(traffic_id)
```

Table 4. Get all the required information using the Mantis Python library.

3.6 User Guide

Mantis comes with a functional User Interface (UI) that is web-based. In what follows we outline this interface, including many snapshot images so as to guide the users throughout the process of defining experiments, running them, and collecting the produced outputs.

A primary purpose of Mantis is to create a common benchmarking environment with social characteristics where researchers share topologies, traffic matrices and CAPEX/OPEX parameters, and evaluate their algorithms under common conditions. In this way, Mantis could also evolve as an online collaboration platform for optical network researchers,

improving the comparability, quality and reliability of the results presented in various research articles and projects.

In Mantis users can define various parameters (e.g., network topology, traffic demands, equipment, devices monetary and energy cost) and select among algorithms for routing and wavelength assignment, routing and spectrum allocation for spectrum-flexible networks, for impairment awareness, for equipment placement, such as regenerators and transponders. Algorithms evaluate future network plans and demands and report a detailed solution including the required bandwidth to serve the demands, the number and configurations for transponders and regenerators, total monetary cost and total required energy, connections that could not be established either due to physical layer impairments or due to bandwidth unavailability.

3.6.1 Algorithms included in the current version of Mantis

The current Mantis version includes network planning and operation algorithms for fixed-grid single-line-rate (SLR) and mixed-line-rate (MLR) WDM, and flexgrid optical networks that can be used for both transparent (without regenerators) and translucent (with regenerators) networks.

The OFDM IA-RSA (IA stands for Impairment Aware) algorithm, presented in [17], considers the planning problem of a flexgrid optical network under physical layer impairments and addresses the problems identified in the Sections 3.1.1.1 and in 3.1.1.2 in a unified manner. The algorithm takes as input a network topology, a traffic matrix and the feasible configurations of the used transponders. It serves the demands for their requested rates by choosing the route, breaking the transmission in multiple connections, placing regenerators if needed, and allocating spectrum to them.

In a similar manner, the IA-RWA-MLR and IA-RWA-SLR algorithms consider respectively the planning problem of mixed-line-rate and single-line-rate fixed-grid WDM optical networks under physical layer impairments.

Online versions of these algorithms are also available in the current version of Mantis. An online algorithm takes as input the output of the offline case and a new demand and serves the new demand in an incremental manner. The output of the online algorithm is saved and can be used as an offline starting point to serve more new demands.

More algorithms that address dynamic network operation of flexgrid networks (see Section 3.1.1.3), including dynamic spectrum expansion/contraction of established connections, grooming of traffic, spectrum defragmentation and others, are currently implemented and evaluated, and will be incorporated in future releases of Mantis.

3.6.2 User Interface

Mantis comes with a clean and simple web-based user interface though which the users can have access to all tool's functionalities. The main Mantis' entities are the configurations, the instances and the projections. A configuration consists of a set of parameters that define the execution of a specific algorithm, while an instance is an executed configuration. The projections are parameterized versions of a configuration that result in a set of instances to be generated at the execution nodes. These instances differ only in the value of one initial parameter (e.g., number of wavelengths). Figure 17 shows the relation between configurations, projections and instances.

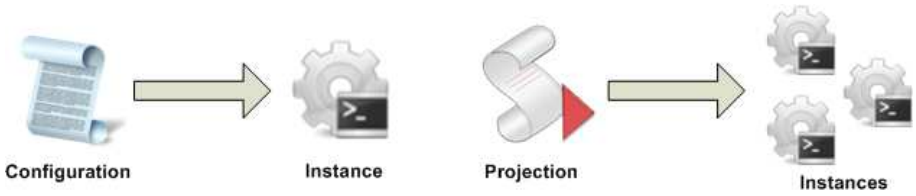


Figure 17. Relation between configuration, projections and instances.

3.6.2.1 Main UI areas

The user interface is divided into three primary areas, the main toolbar in the top that provides immediate navigation between tool's pages, the main work space in the middle and the vertical accordion menu in the left that contains seven entries. Figure 18 presents Mantis main page and its primary areas.

In the user interface there is a separation of the different steps: network topology and traffic demands creation, algorithms selection and configuration, execution and results presentation. All these operations are implemented in their own pages that are presented in the main work space, while the main toolbar and the accordion menu are always available to users for switching between the various operations.

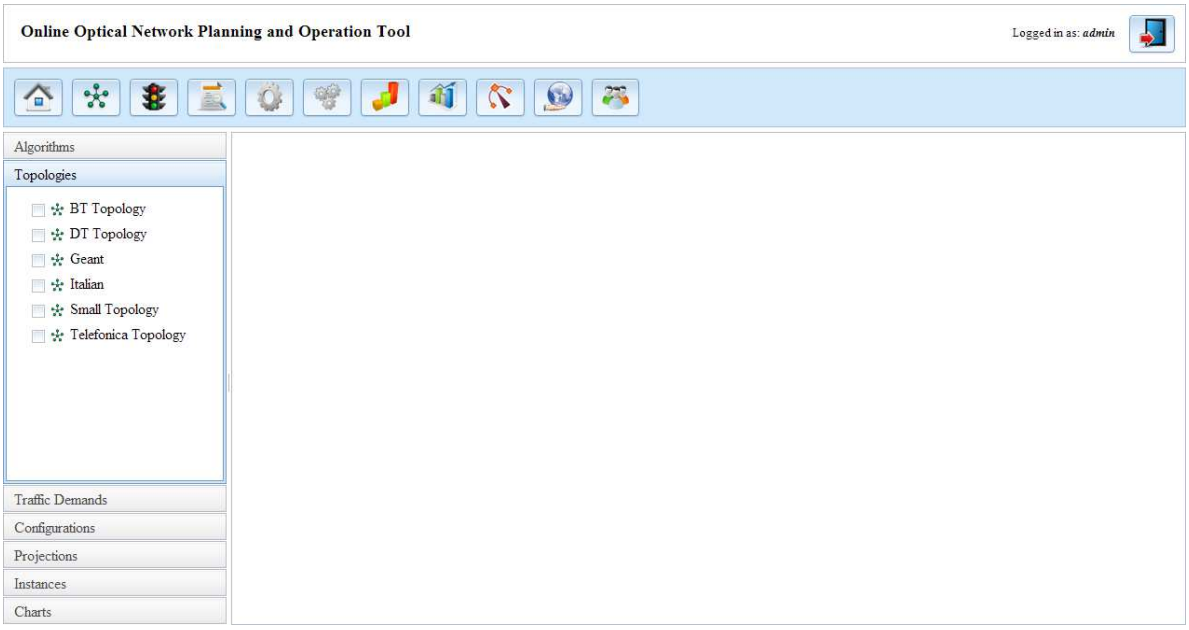
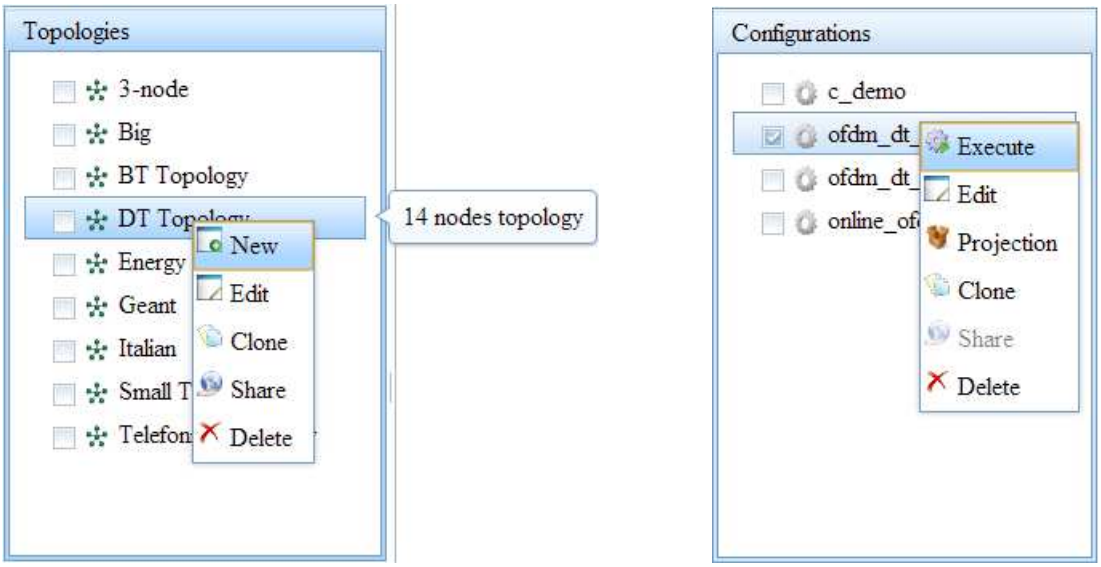


Figure 18. Mantis main page and its primary areas.



(a) (b)

Figure 19. Context menu for topologies and configurations.

The entries in the left menu have a tree like structure and functionality through context menus. The left menu provides access to the following system entities:

- network planning and operation algorithms
- network topologies
- traffic demands
- configurations
- projections
- instances
- graphs

These menus offer a limited set of actions that are available for each selected object. For example, in Figure 19a and Figure 19b we present the actions that are available for network topologies and configurations respectively.

3.6.2.2 Defining topologies, traffic matrices and costs

Mantis user interface enables users to easily and graphically design network topologies by defining network nodes and links between them along with their length, save, edit, and use the existing topologies at any time. Figure 20a shows the available interface for working with network topologies in which users can navigate either by using the main toolbar or by selecting a network topology from the left menu. In this page the workspace contains a number of control buttons and the main canvas where users can design their network topologies. The network topologies implementation is based on the Raphaël [12] JavaScript library that provides methods for working with vector graphics on the web.

Users can easily create and customize their network topologies since every graphical object (nodes and links) provides a context menu with the available actions. By selecting a

node (Figure 20b) users can edit its label, delete it, add a link starting from this node or add a new node in the topology. Similarly, there is available a context menu for interacting with links (Figure 20c), which provides to users actions for adding a link, changing its current length or deleting it.

Traffic demands, which are expressed as square matrices, can be created either graphically by using an editable table (Figure 20d) or can be imported from files. For the latter case, Mantis currently supports two formats in order to import traffic demands, the Mantis JSON format (described in Section 3.5) and the comma-separated values (CSV) format.

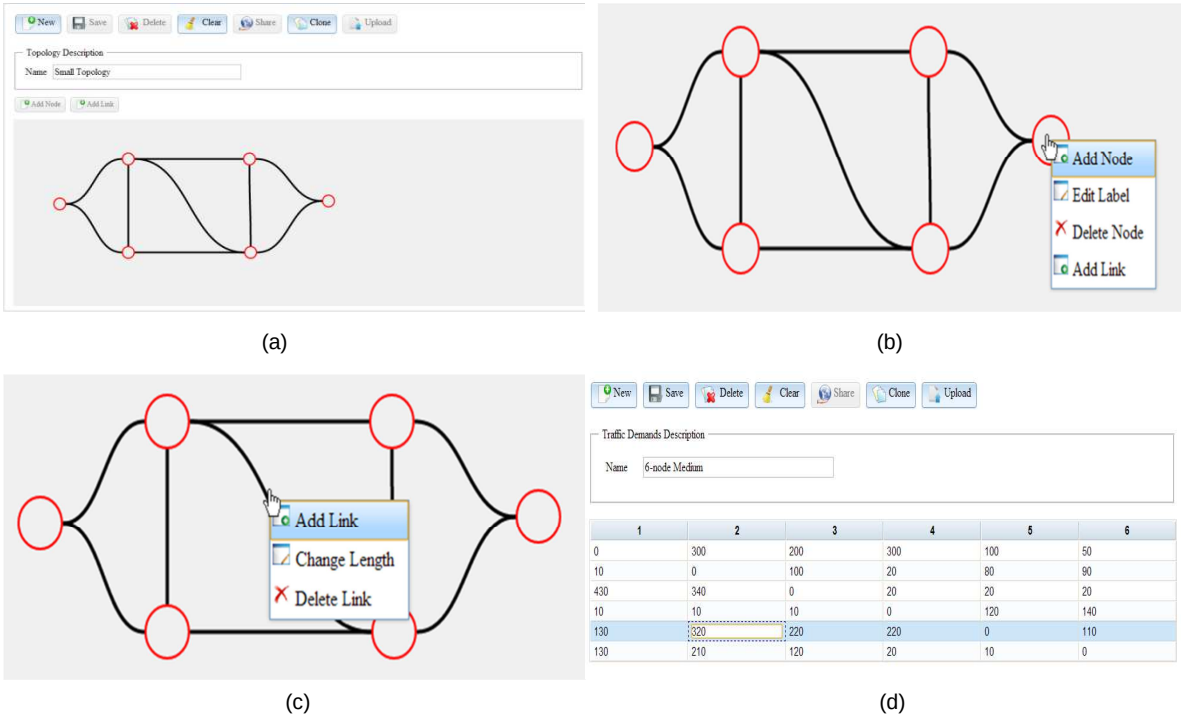


Figure 20. (a) network topology workspace (b) actions for network nodes (c) actions for network links (d) traffic demands workspace.

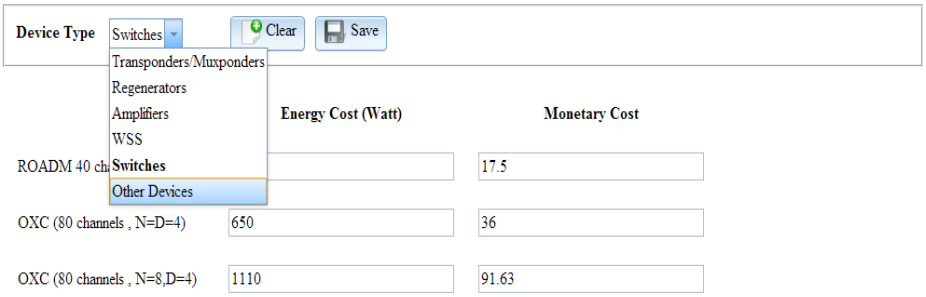


Figure 21. Define CAPEX/OPEX parameters for various devices.

Additionally, users can define the energy and monetary cost (Figure 21) for various devices used in optical transport networks, like transponders/muxponders, regenerators, amplifiers, switches. All Mantis algorithms use these values to calculate the monetary and energy cost for their solutions. This information is also available through the custom plug-in mechanism (Section 3.5) to every algorithm that is integrated to Mantis.

3.6.2.3 Setting up a configuration

Before users can request the execution of any algorithm it is necessary to create a configuration that contains the values for all parameters, which are required for the execution of a planning or operation algorithm. In the user interface there is a specific configuration page for every algorithm since they require different input parameters. Initially, a user should select the algorithm for which the configuration will be defined. Then, in the algorithm's configuration page (Figure 22) should define a configuration name, select a network topology, traffic demands and specify the other configuration parameters that are specific to the selected algorithm. Mantis automatically checks all provided parameters and informs the users for possible mistakes, before permanently saving any configuration. The performed validation ensures that every configuration in the database contains all the necessary parameters with valid values. Hence, it eliminates any possible resources wasting when a user requests the execution of an invalid configuration.

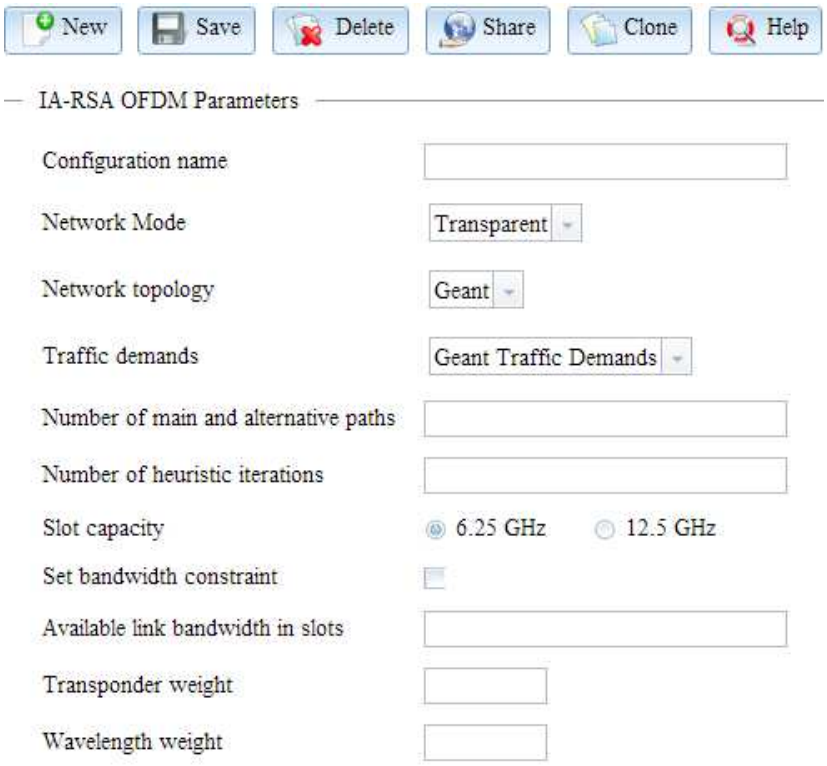


Figure 22. Configuration page for IA-RSA OFDM algorithm.

3.6.2.4 Setting up a projection

For every configuration, users can request either its execution or the creation of a new projection. When a user wants to create a new projection initially should select some configuration from the left accordion menu and execute the projection action from its context menu (Figure 19). Next, the user through the available interface should select the parameter in which the projection's instances will differ and define also a range of values for it. Figure 23 shows this interface where users can also view the configuration's input parameters values. A projection's value range will also determine the number of instances that will be created for its execution.

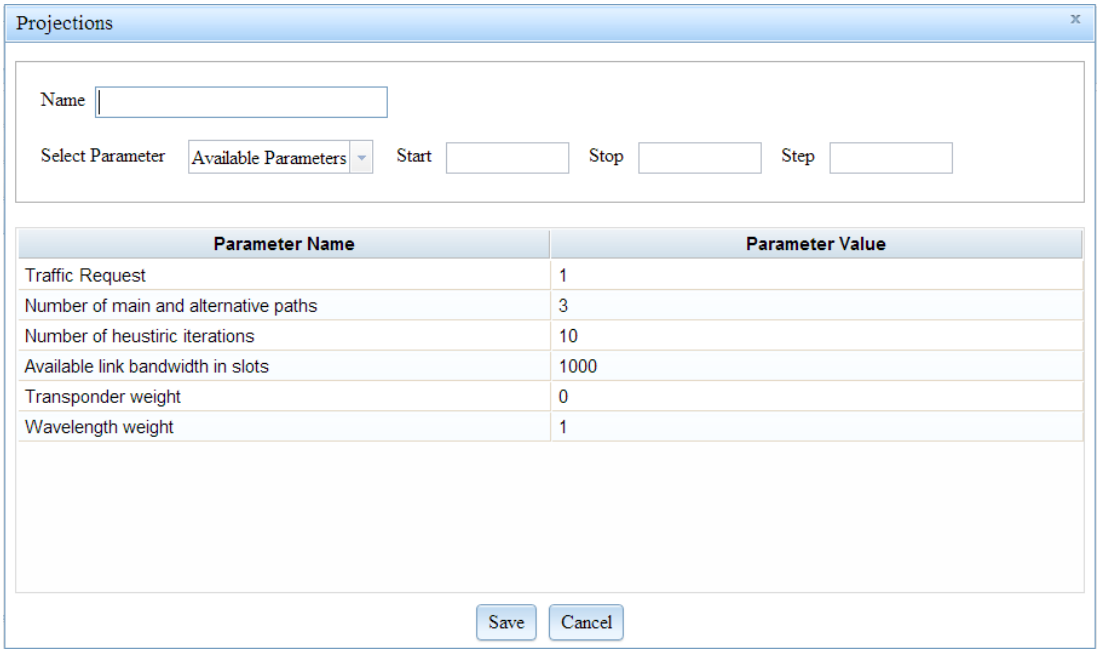


Figure 23. Define projection using an IA-RSA OFDM algorithm configuration.

3.6.2.5 Status and results

Users can always check the status of their running or finished instances and have access to useful details including the instance name, configuration name, execution status, creation date and execution date through the history page (Figure 24). In addition, users can terminate running instances, view the results from successfully executed instances or filter the displayed instances either by their execution status or their configuration name.

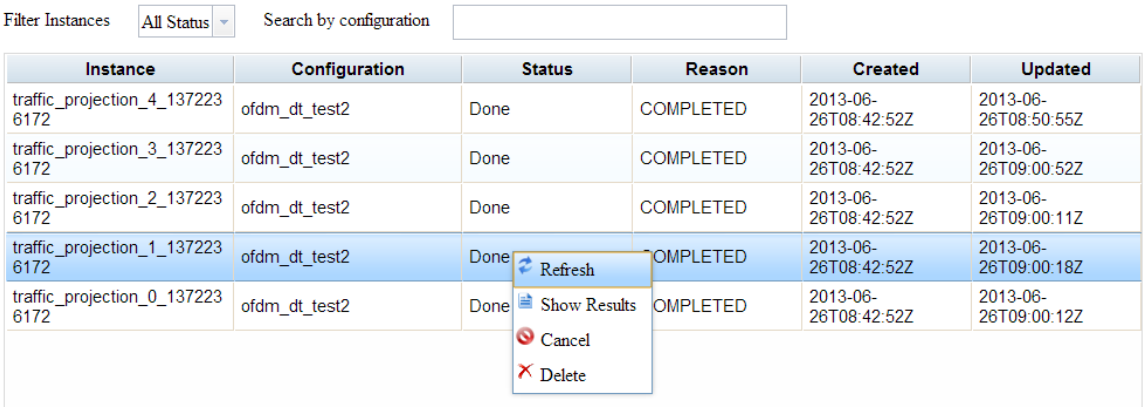
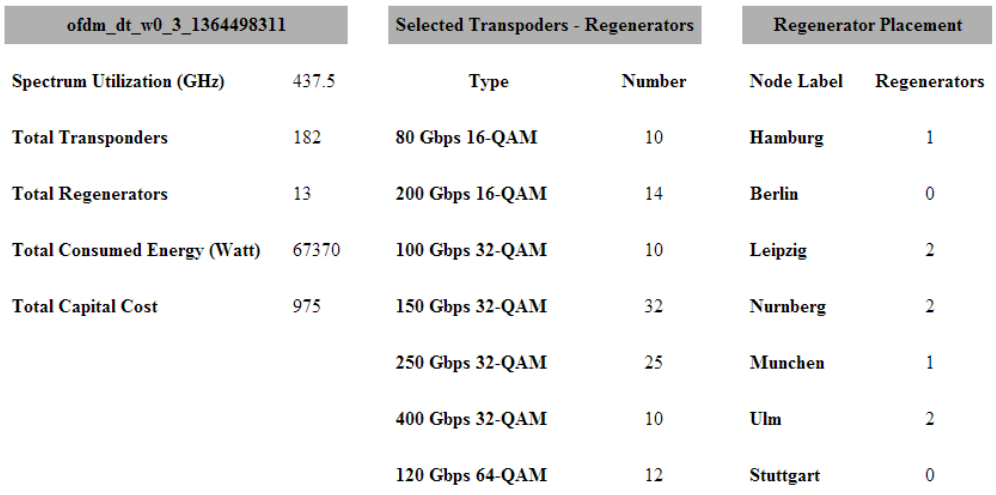


Figure 24. History page where a user can view details for his instances.

For every successfully executed instance users can view its analytical results and export the proposed solution in JSON format, including details for each connection, for further analysis. In Figure 25 we present part of the detailed solution, including required transponders and regenerators and where regenerators should be placed, after the execution of a configuration for the IA-RSA OFDM algorithm on the DT network topology.

Generally, Mantis is designed to report a detailed solution, which according to the executed algorithm may include:

- required bandwidth to serve the demands
- established lightpaths
- number and configurations for the required transponders and regenerators
- where transponders and regenerators are placed
- total monetary cost and total required energy
- connections that could not be established due to physical layer impairments or bandwidth unavailability



ofdm_dt_w0_3_1364498311	
Spectrum Utilization (GHz)	437.5
Total Transponders	182
Total Regenerators	13
Total Consumed Energy (Watt)	67370
Total Capital Cost	975

Selected Transpoders - Regenerators	
Type	Number
80 Gbps 16-QAM	10
200 Gbps 16-QAM	14
100 Gbps 32-QAM	10
150 Gbps 32-QAM	32
250 Gbps 32-QAM	25
400 Gbps 32-QAM	10
120 Gbps 64-QAM	12

Regenerator Placement	
Node Label	Regenerators
Hamburg	1
Berlin	0
Leipzig	2
Nurnberg	2
Munchen	1
Ulm	2
Stuttgart	0

Figure 25. Part of detailed solution, including required transponders and regenerators and where regenerators should be placed for the DT network.

Furthermore, users can create charts to visualize the results from various executions in order to have a better evaluation of the different scenarios. Charts can be created by combining the results of various completed instances and consist of one or more data series. Figure 26a shows the interface for creating and editing charts in which users can easily add, edit or delete data series. A user selects the values that will contain each data series through the interface presented in Figure 26b. A user should choose a configuration to view its instances and then selects from them the values that he wants to include to the data series. The values in each data series are displayed in the same order that are selected from the user. Users can view the created charts (Figure 26c) either by selecting them from the left accordion menu or from the corresponding page in the user interface where it contains a drop-down list with all charts that each user can access.

Figure 26. (a) Create a chart with three data series (b) select values for data series (c) chart presentation.

3.6.2.6 Social Characteristics

Since, one of Mantis primary purposes is to create a common benchmarking environment tool provides a private and a public workspace. The private workspace includes user's

network topologies, traffic demands, configurations, results and charts. Network topologies, traffic demands, configurations and charts can be shared with other users. The shared items are available in the public workspace while the other users have only read access to them. However, they can create their own copies.

The public workspace initially contains a number of system defined network topologies and traffic demands along with the public items from the other users. In the current version of Mantis when a user defines an item as public this becomes automatically accessible from all users. However, in following versions we plan to introduce the concept of working groups that will be created from the users. Then, a user will share his items either with all users or with a group's members only. The user interface provides a separate page (Figure 27) where users can view and manage the network topologies, traffic demands, configurations and charts that have been defined as public.

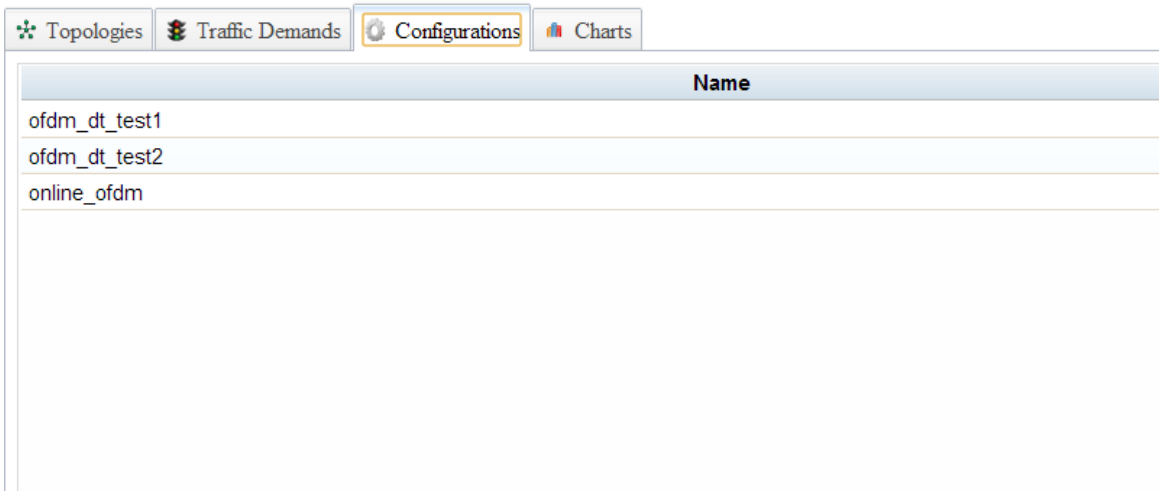


Figure 27. View and manage shared items.

4 PLATON

This section describes UPC's PLAnning Tool for Optical Networks (PLATON) from a non-formal to formal level. Section 4.1 summarizes in a non-formal manner which are the PLATON objectives and requirements, that later are extended and grouped in section 4.2 to define the different tool modules. Each tool module is then formally specified in section 4.3 using UML Use Case diagrams that describe the user interactions with the tool and later, in section 4.3.2, its behaviour is formally modelled using UML sequence diagrams and class diagrams. Finally, section 4.4.1 describes how to integrate PLATON into a third party tool, and section 4.6 gives a user manual about using PLATON.

4.1 Objectives and requirements

4.1.1 Objectives

The objective of PLATON is to provide an environment where a defined set of optical network planning algorithms can be executed using high performance and state-of-the-art hardware and software technologies.

The selected hardware to be used in PLATON are Graphic Processing Unit (GPU) devices because they provide notable speedups in terms of computation time and consume only a fraction of the energy required in comparison with an equivalent computation power high performance computing cluster. Additionally, GPUs require less physical space than a computers cluster and are cheaper than a cluster. The drawback of GPUs is the fact that the algorithms must be specifically adapted to be executed into the GPU, so a specialized developer must implement those algorithms.

4.1.2 Requirements

The PLATON requirements are defined below:

- **Performance:** Implement the defined set of algorithms using high performance technology and algorithm optimizations.
- **Manager-Agent architecture:** Design the application to maintain a queue of pending jobs scheduled by its priority. When an agent becomes idle, the manager will assign the first pending job from the queue to that agent. Each agent executes one job (i.e. an algorithm over a user dataset). Multiple jobs must run concurrently on different agents. System must be extensible in terms of number of agents in case of higher degree of concurrency is required.
- **Test-bed Connectivity:** Provide PCE Protocol interface to real network test-beds allowing execution of high performance algorithms.
- **Automated Connectivity:** Provide XML Web Services interface to network simulators allowing execution of high performance algorithms.
- **Web User Interface:** Allow the users to login into a web application to manually schedule and monitor its high performance computations over specific manually uploaded datasets.
- **Algorithm Set Extensibility:** Design the tool to allow the administrators to implement and integrate new algorithms.

- **Extensible PCEP-based framework:** Provide the base PCEP framework used by PLATON to allow third party users to implement their own network planning tool.

4.2 Architecture description

PLATON is composed of two main modules, the Cluster Manager, and the set of HPC Agents [19]. PLATON can be operated in two different scenarios: the off-line and the in-operation scenario.

In the off-line operation scenario, illustrated in Figure 28, the Cluster Manager receives user jobs through the web application or the web services interfaces. Jobs are introduced into a priority queue and the Job Scheduler is responsible for assigning those jobs to the

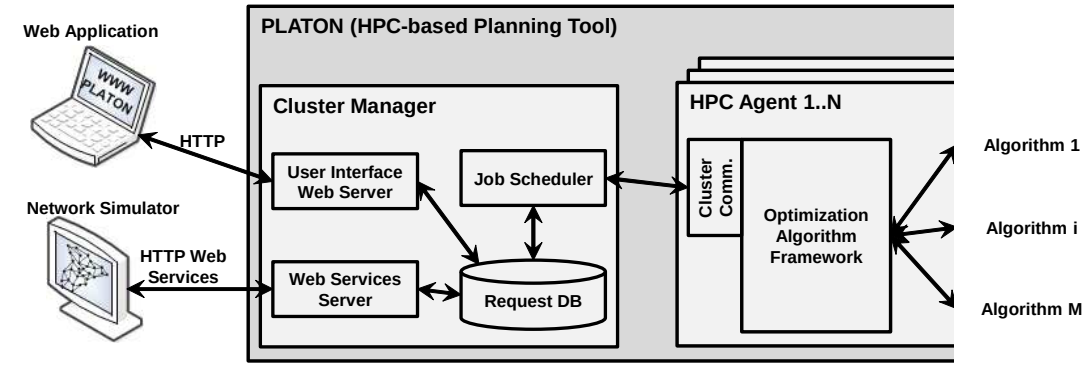


Figure 28. PLATON Architecture – Off-line planning scenario

In the in-operation scenario, illustrated in Figure 29, the Path Computation Element (PCE) sends the computation jobs directly to the HPC agent. This second scenario does not add the computation jobs into the priority queue because the Job Scheduler's delay is not

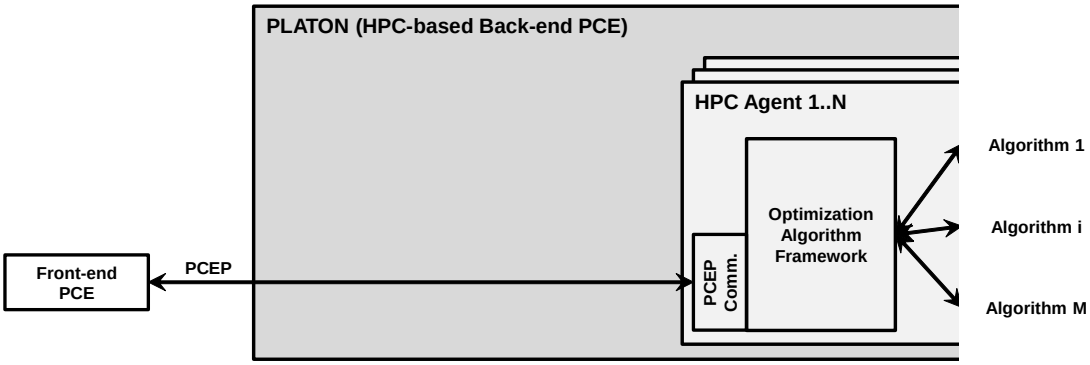


Figure 29. PLATON Architecture – In-operation planning scenario

The Cluster Manager is divided into 4 blocks:

- **Request DataBase** that stores all jobs to be executed. The Request DB provides information to the rest of Cluster Manager's modules. When a job is created or modified into the Request DB, a trigger is thrown to the Job Scheduler to notify the update.
- **User Interface Web Server** that provides a web application to allow users' access to manage their computation jobs and administrators' access to manage the whole

cluster. Information retrieved by the web application is taken from the Request DB. Jobs created using the web application are directly stored into the Request DB.

- **Web Services Server** that allows remote network simulation applications to automatically connect to the cluster and request computation jobs. Information retrieved by the web services is taken from the Request DB. Jobs created using the web services are directly stored into the Request DB.
- **Job Scheduler** is responsible for monitoring all the HPC Agents and requested jobs. Each time it receives a trigger from the Request DB, its internal job's priority queue should be updated.

Each HPC Agent is divided into 4 blocks:

- **Cluster Communications** manages the communication between the Cluster Manager and the HPC Agent when the agent is operated in the off-line planning scenario.
- **PCEP Communication** attends for PCEP Request messages and computes the requested algorithms when the agent is operated in the in-operation planning scenario. When computation is completed, emits a PCEP Reply message with the results.
- **Optimization Algorithm Framework** is the main agent's meta-heuristic loop. This framework is common to all algorithms, and depending on the requested algorithm, this framework executes specific algorithms from the algorithm set.
- **Algorithms 1..M** is the set of specific algorithms that can be executed by the agent. When a computation is executed, one of them is selected by the meta-heuristic depending on the requested computation to be done. All the algorithms must be designed using the same interface (i.e. the same functions and classes signatures) so that the same agent code must be able to use any of the algorithms. These algorithms are executed into the GPU and must be specifically designed for them.

4.3 Software specification

For each module previously defined we specify: a) which are the use cases (i.e. the actions that can be done in the module), b) the related actors (i.e. the events, users, etc. that trigger the execution of the use cases), and c) the lifecycle of the involved entities when needed to describe its complexity. Additionally we provide the communications protocol specification used between the Cluster Manager and the HPC Agents in the off-line operation scenario. We define these elements using UML use case diagrams to specify the actions available to each user, UML life cycle diagrams to describe an overview of complex entities' behaviour, and UML activity diagrams to describe the interaction between the Cluster Manager and the HPC Agents.

4.3.1 Cluster manager specification

The PLATON Cluster Manager can be operated through two different user interfaces: the web application for human users and the XML web services for automated users. Both interfaces offer the same functionality and the same internal entities; and only differ in the technology used to interact with the user.

Cluster Manager has 3 privilege level-structured actors: the non-registered user (lower privileges), the registered user (regular privileges) and the administrator user (higher privileges).

Figure 30 illustrates the use cases related to non-registered user. The non-registered user is able to:

- **Do login:** logging in into the cluster manager. When properly logged in, it becomes a registered user or an administrative user depending on its user configuration.
- **Change lost password:** introduce user's email and if a user exists having those email address, the system sends a link to the email address. If the email address owner clicks the link, the new user's access password can be provided. The old user's password is discarded.

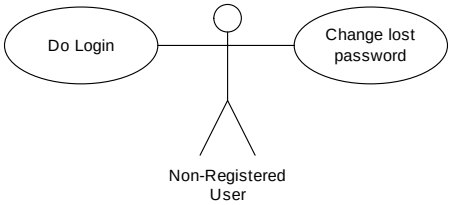
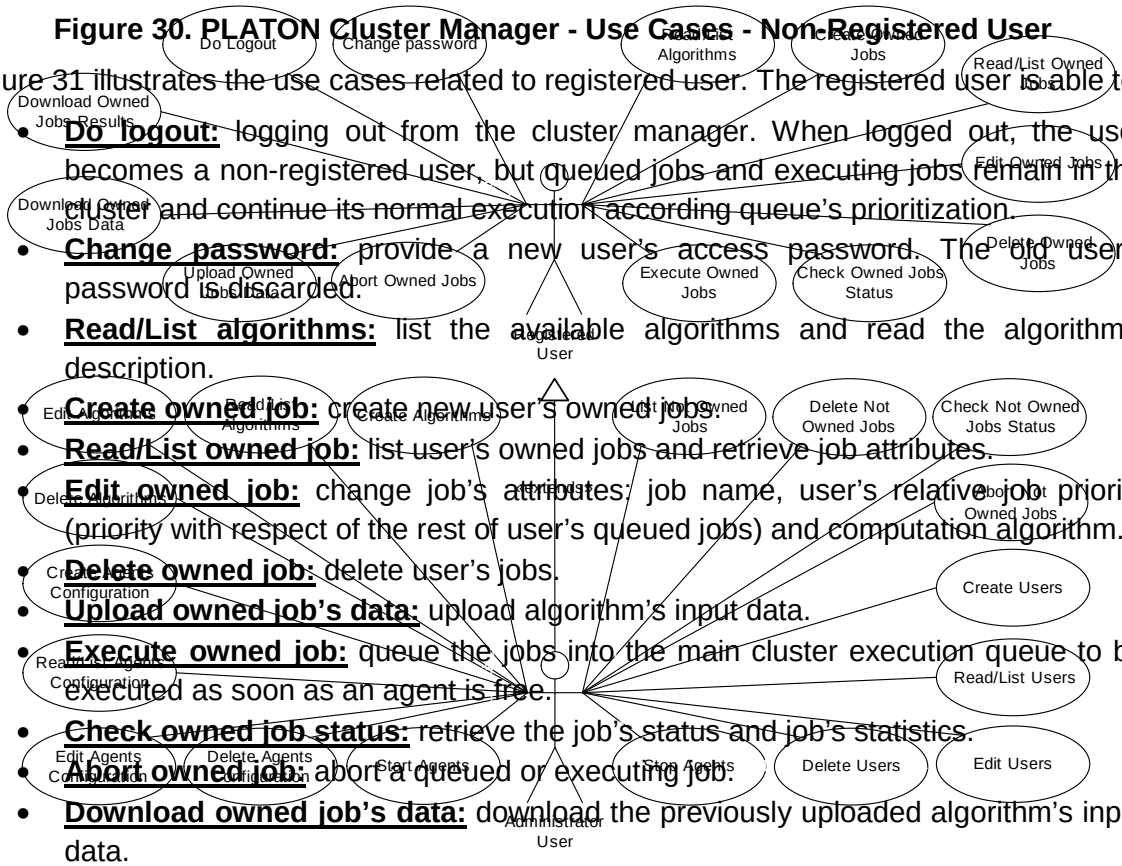


Figure 30. PLATON Cluster Manager - Use Cases - Non-Registered User

Figure 31 illustrates the use cases related to registered user. The registered user is able to:

- **Do logout:** logging out from the cluster manager. When logged out, the user becomes a non-registered user, but queued jobs and executing jobs remain in the cluster and continue its normal execution according queue's prioritization.
- **Change password:** provide a new user's access password. The old user's password is discarded.
- **Read/List algorithms:** list the available algorithms and read the algorithm's description.
- **Create owned job:** create new user's owned jobs.
- **Read/List owned job:** list user's owned jobs and retrieve job attributes.
- **Edit owned job:** change job's attributes: job name, user's relative job priority (priority with respect of the rest of user's queued jobs) and computation algorithm.
- **Delete owned job:** delete user's jobs.
- **Upload owned job's data:** upload algorithm's input data.
- **Execute owned job:** queue the jobs into the main cluster execution queue to be executed as soon as an agent is free.
- **Check owned job status:** retrieve the job's status and job's statistics.
- **Abort owned job:** abort a queued or executing job.
- **Download owned job's data:** download the previously uploaded algorithm's input data.
- **Download owned job's results:** download the algorithm's result data.



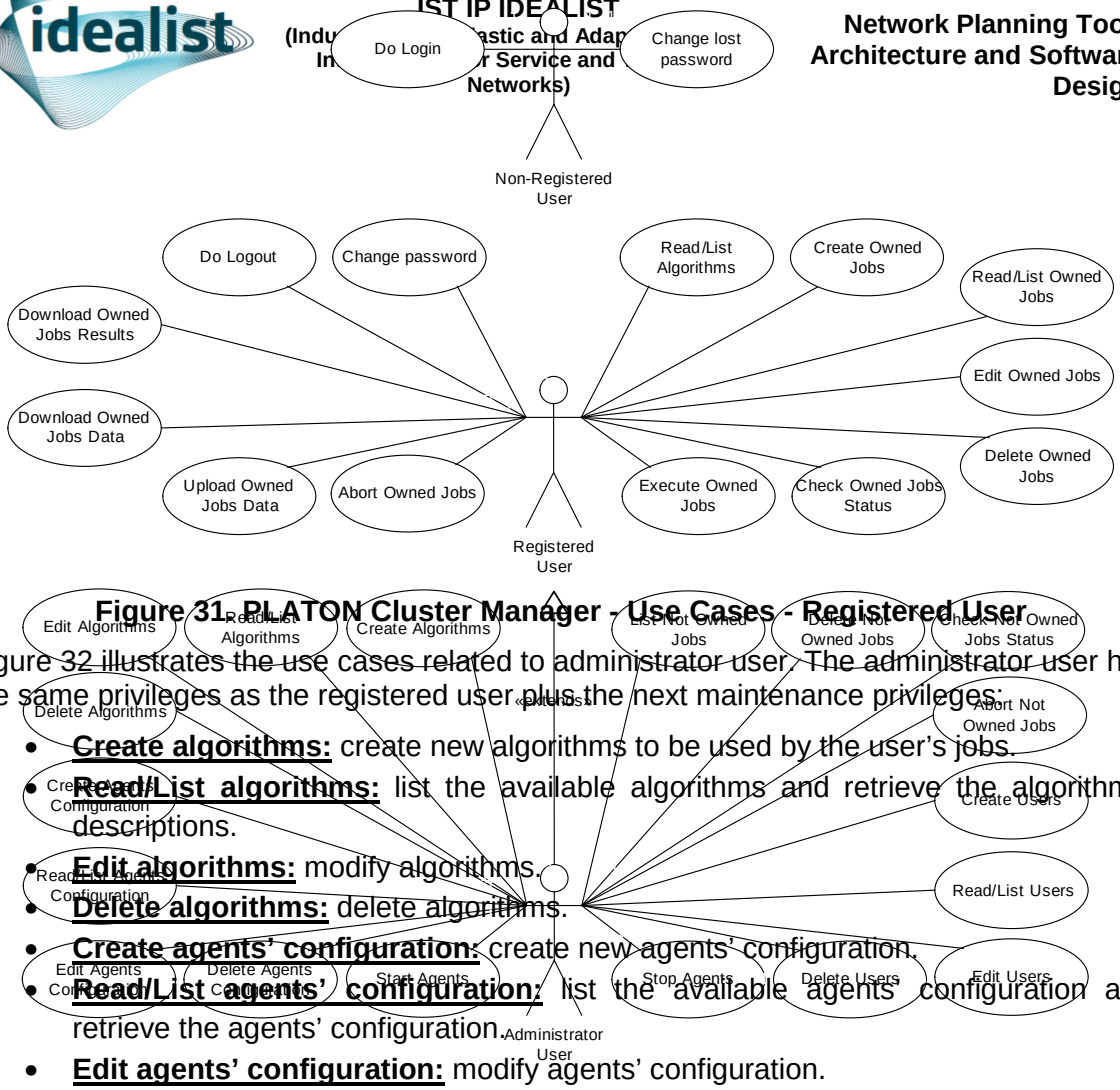


Figure 31. PLATON Cluster Manager - Use Cases - Registered User

Figure 32 illustrates the use cases related to administrator user. The administrator user has the same privileges as the registered user plus the next maintenance privileges:

- **Create algorithms:** create new algorithms to be used by the user's jobs.
- **Read/List algorithms:** list the available algorithms and retrieve the algorithm's descriptions.
- **Edit algorithms:** modify algorithms.
- **Delete algorithms:** delete algorithms.
- **Create agents' configuration:** create new agents' configuration.
- **Read/List agents' configuration:** list the available agents configuration and retrieve the agents' configuration.
- **Edit agents' configuration:** modify agents' configuration.
- **Delete agents' configuration:** delete agents' configuration.
- **Start agents:** activate stopped agents.
- **List agents' status:** list all agents detailing its status.
- **Stop agents:** deactivate started agents.
- **Read/List not owned jobs:** list jobs from all users and retrieve job's attributes including the job's owner.
- **Delete not owned jobs:** delete not owned jobs.
- **Check not owned jobs status:** retrieve not owned job's status.
- **Abort not owned jobs:** abort not owned jobs.
- **Create users:** create new cluster users.
- **Read/List users:** list and retrieve cluster user's attributes.
- **Edit users:** modify cluster users including its access password and administration permission.
- **Delete users:** delete existing cluster users.

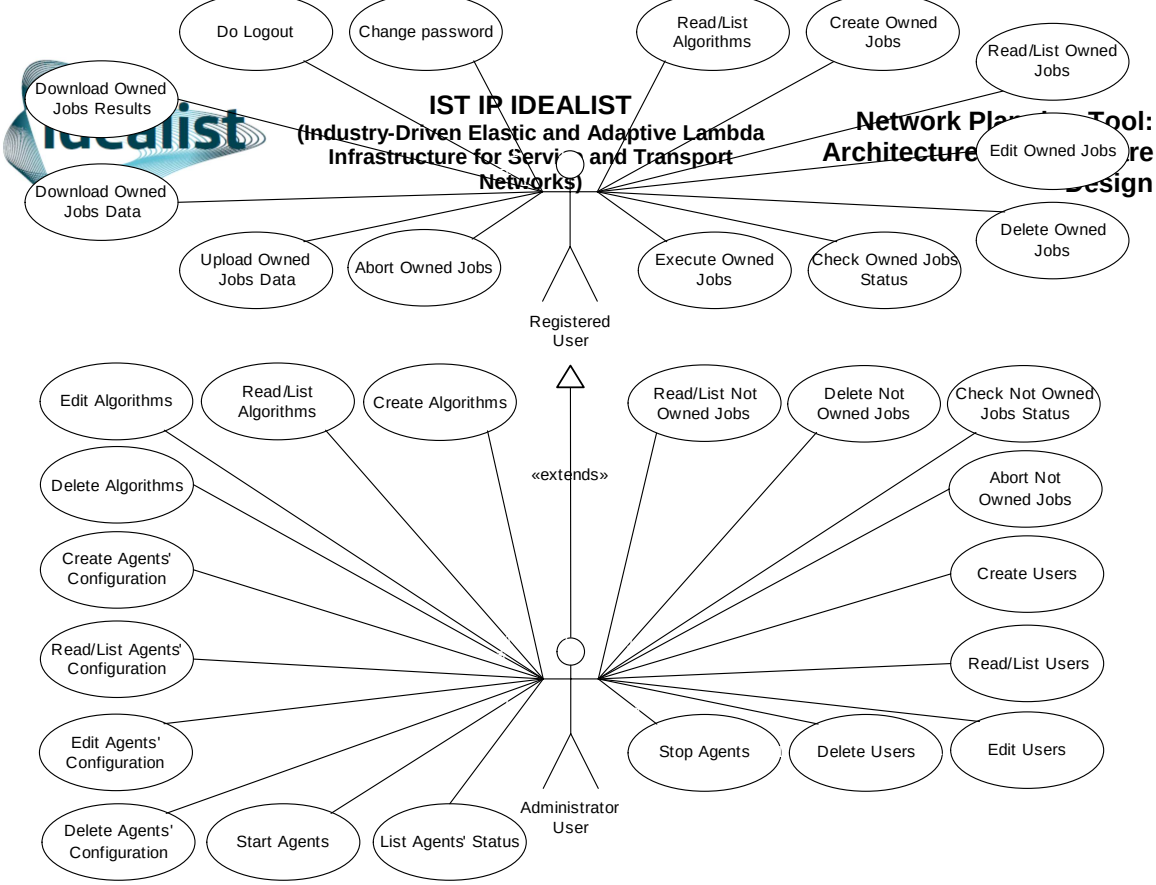


Figure 32. PLATON Cluster Manager - Use Cases - Administrator User

To illustrate how the use cases interact with the entities, we use UML life cycle diagrams. There are life cycle diagrams for each entity used in PLATON, but they are trivially simple, so we describe only the jobs life cycle diagram illustrated in Figure 33.

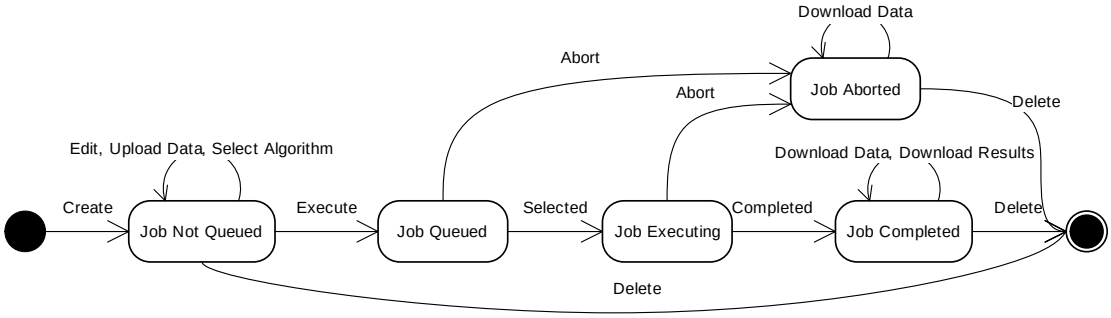


Figure 33. PLATON Cluster Manager - Life Cycles - Jobs

When created and not executed, the initial job's state is "Job Not Queued" and the available actions that can be done over the job are: a) change its attributes (name, priority, algorithm to be executed and input data) and the job remains in the same state; b) execute, so all attributes become fixed, the state changes to "Job Queued" and the job is inserted into the execution's priority queue; or c) deleted so the job is deleted from the database.

When the job's state is "Job Queued", two actions can be done: a) abort, that removes the job from the execution's priority queue and changes the state to "Job Aborted"; or b) selected by the priority queue to be executed, so the job is removed from the priority queue, the state is changed to "Job Executing", and the execution is initiated.

When the job's state is "Job Executing", two actions can be done: a) abort, that aborts the job's computation and changes the state to "Job Aborted"; or b) completed that implies that computation has been completed and changes the state to "Job Completed".

When the job's state is "Job Aborted", two actions can be done: a) download the input data provided to the algorithm; or b) delete the job, so the job is deleted from the database.

Finally, when the job's state is "Job Completed", three actions can be done: a) download the input data provided to the algorithm; b) download the output results generated by the algorithm; or c) delete the job, so the job is deleted from the database.

4.3.2 HPC agent specification

The PLATON HPC agents can be operated through two different scenarios: the off-line planning scenario, and the in-operation planning scenario. Both scenarios use the same algorithms and input data format, and produce the same output results format.

When the active scenario is the off-line planning, both input data and output results are encoded as XML request and response messages respectively. However, the in-operation planning scenario take as input data one PCEP Request message, and output results necessarily must be a PCEP Reply message. This data format divergence requires translating the input/output data from/to PCEP and XML messages into/from internal messages representations.

HPC Agent has two possible actors, the Cluster Manager, and a PCEP Request, corresponding to both possible operation scenarios. Figure 34 illustrates the use cases corresponding to both actors.

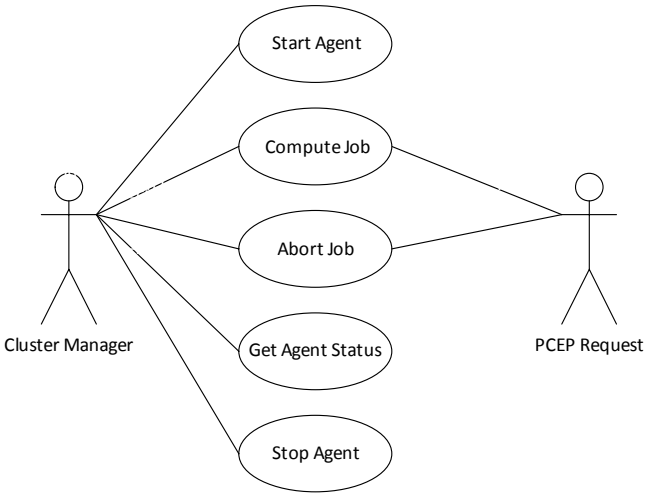


Figure 34. PLATON HPC Agent - Use Cases

When the HPC Agent is operated using a Cluster Manager, additional use cases exist because the Cluster Manager is able to monitor, start and stop the HPC Agents.

The use cases correspond to:

- **Start agent:** instantiate a new agent as a member of a PLATON cluster.
- **Compute job:** execute a new job. This use case can be initiated by the Cluster Manager and by a PCEP Request.
- **Abort job:** abort a job that is currently being executed. This use case can be initiated by the Cluster Manager and by a PCEP Request.
- **Get agent status:** retrieves to the Cluster Manager the current status of an agent.
- **Stop agent:** abort the current computation if any and shutdown the agent.

4.3.3 Communication protocol specification

To specify the communication protocol used between the Cluster Manager and each HPC Agent, a UML activity diagram is illustrated in Figure 35. Note that the diagram corresponds to a Cluster Manager plus single HPC Agent communication, but multiple agents can be attached to the Cluster Manager. There are other interactions between the Cluster Manager and the HPC Agent (i.e. start and stop agent, transfer files, etc.), but they are trivially simple or consist on executing commands through an SSH console so they are avoided and only the communication protocol is described. Note that file transfers between the Cluster Manager and the HPC Agent are done via SSH commands so there is no need to extend the protocol to support file transfers.

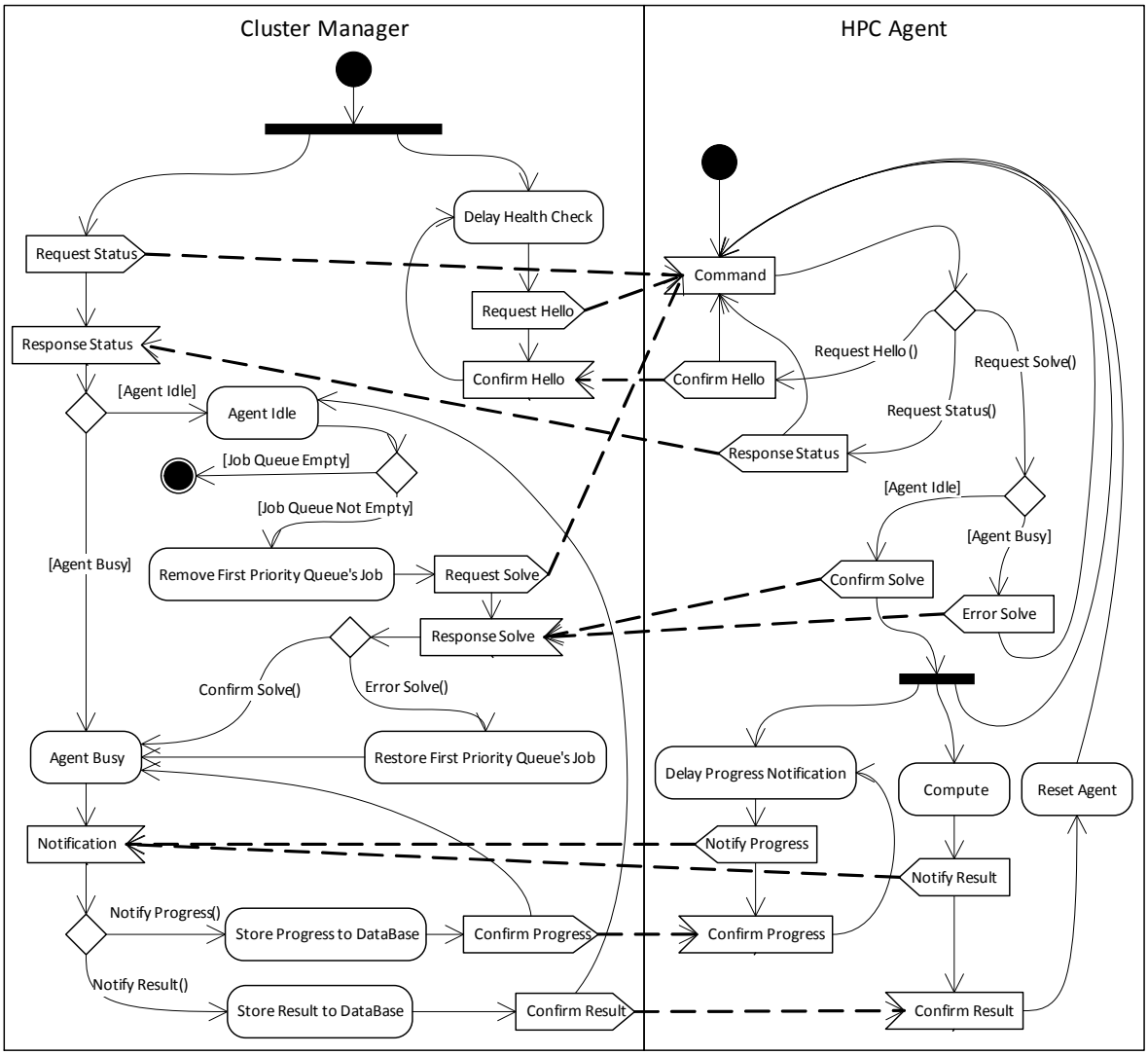


Figure 35. PLATON Communication Protocol Cluster Manager vs. HPC Agent – Activity Diagram

When HPC Agent is initialized, it stands for an input command and a specific action is executed depending on the received command. When the Cluster Manager is initialized, it initializes two concurrent activities.

The first Cluster Manager activity consists on periodically sending “Hello” messages to the HPC Agent and stand for the HPC Agent confirmation to control the agent’s health. When

the HPC agent receives a “Hello” command, it must respond as soon as possible to the Cluster Manager. When the Cluster Manager receives acknowledge for the “Hello” command, it delays the next “Hello” request for a specific amount of time.

The second Cluster Manager activity begin sending a “Status” command to the HPC Agent and waiting for a response to decide if the HPC Agent is initially idle or busy. An HPC Agent is idle when the Cluster Manager is initialized only when the Cluster Manager is restarted without ensuring that the HPC Agent is idle. When the HPC Agent receives the “Status” command, it responds with its status. When the Cluster Manager receives the response, if the agent is idle and the job’s priority queue is not empty, the next priority queue job is removed from the queue and assigned to the HPC Agent by sending a “Solve” command and wait for a “Solve” confirmation. When the confirmation is received, the Cluster Manager stands for incoming notifications (i.e. progress notification and final result notification) from the HPC Agent. If the HPC Agent is busy, the Cluster Manager also stands for incoming notifications until current job is completed. Then, the Cluster Manager continues with the next queued job.

When the HPC Agent receives the “Solve” command, if it is busy, an error message is sent to the Cluster Manager and the job that has been removed from the priority queue is restored to be taken as the next computation job; if the agent is idle, it confirms the job assignment sending a solve confirmed notification and it initializes three concurrent activities: a) periodically notifying the computation progress to the Cluster Manager; b) initializing the job’s computation and notifying the result to the Cluster Manager when the job is completed; and c) immediately returning activity flow control to receive command to attend other commands received during the job’s computation.

When the HPC Agent notifies its job’s computation progress it attaches updated statistics related to the computation progress that are stored by the Cluster Manager into the database. When properly stored, the Cluster Manager confirms the progress message reception to the HPC Agent and the HPC Agent delays the next progress notification.

When the HPC Agent notifies its job’s computation completion it attaches the final computation statistics that are also stored into the database by the Cluster Manager. After storing the final statistics into the database, the Cluster Manager confirms the reception of the results and self prepares to assign the next job to the idle HPC Agent.

When the HPC Agent receives the results confirmation, it self-reset destroying the send progress activity and redirects flow control to receive next command. Now, both the Cluster Manager and the HPC Agent are ready to begin the next computation job.

4.4 Software design

PLATON software design has been detailed using UML conceptual diagrams to describe relations between entities and UML sequence diagrams to describe the application execution flows. As we did with the software specification section, we separately describe the Cluster Manager and HPC Agent modules when possible.

With the aim of clarifying two simplifications has been done into the model:

- Some entities have been duplicated between diagrams, and only the fields used on such diagram are kept. For this reason the “Algorithm” entity has the attribute “name” on the Cluster Manager diagram, because the user needs to know the name of the algorithm selected on a specific job, but there isn’t on the HPC Agent diagram because the HPC Agent only needs to know the “id” of the algorithm to be executed, not its name.

- Trivial and not important methods have been avoided and only the most important methods are kept into the diagrams.

4.4.1 Cluster manager design

The cluster manager conceptual diagram shows that the principal entity in the Cluster Manager is the Job. The rest of entities are linked to the job in one way or another. Next we describe each entity and its attributes from the Cluster Manager point of view.

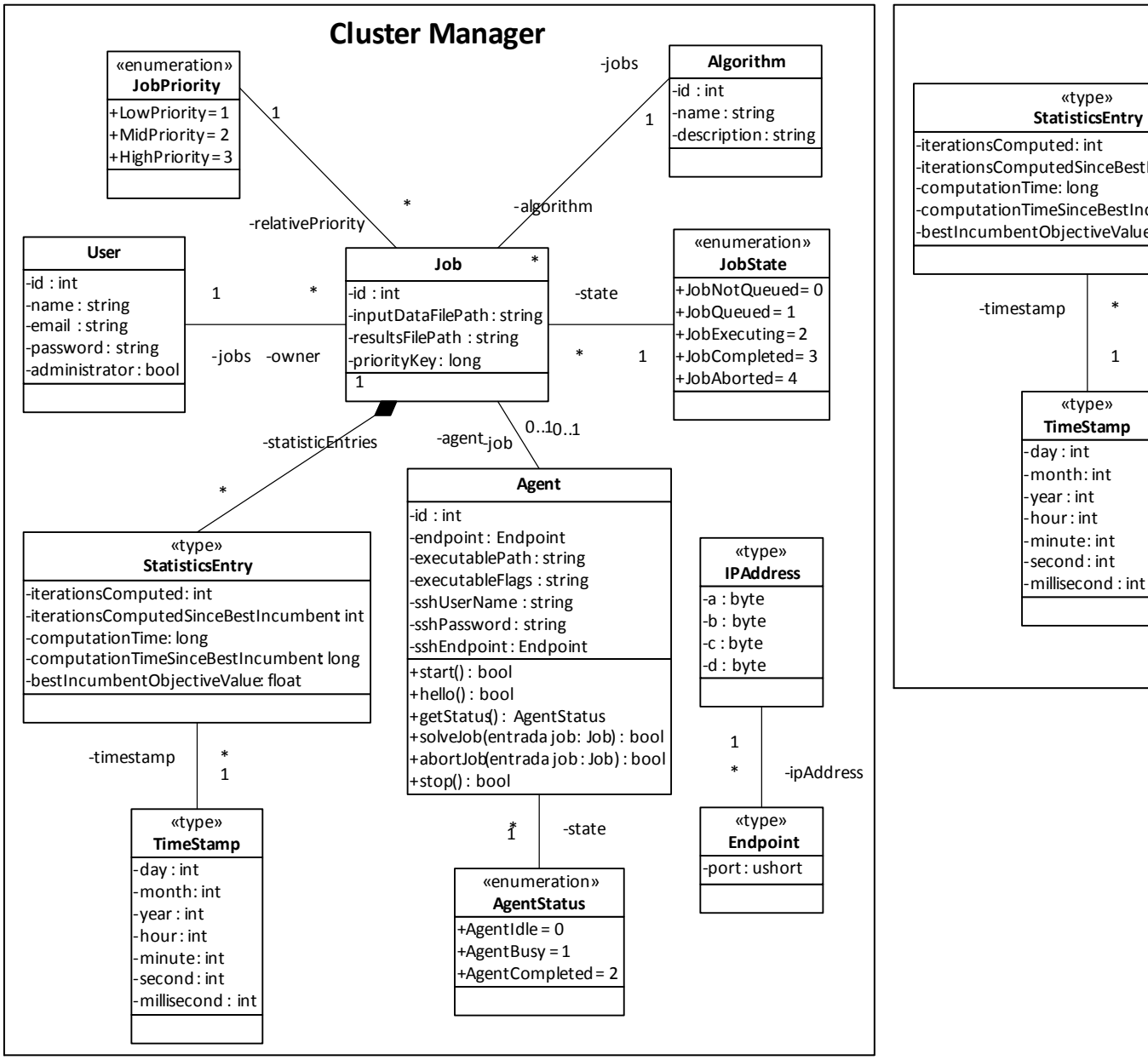


Figure 36. PLATON Cluster Manager - Conceptual Diagram

The job entity has the following attributes:

- id: is the job's ID.
- inputDataFilePath: is the path to the input file to be used by the algorithm.

- resultsFilePath: is the output file to be used by the algorithm to write results.
- priorityKey: is the key used by the job's scheduler to select the next job to be executed.
- relativePriority: is the job's relative priority used to adjust the job's priority key.
- owner: is the user that owns the job.
- algorithm: is the selected algorithm to be used to compute the job.
- state: is the current state of the job.
- agent: is the agent that is computing the job.
- statisticEntries: is the set of statistic entries each one corresponding to one progress notification or a results notification.

The JobPriority enumeration entity is a set of possible job relative priorities. This enumeration can be extended if necessary and its default possible values are:

- LowPriority coded with integer value 1.
- MidPriority coded with integer value 2.
- HighPriority coded with integer value 3.

The Algorithm entity represents the system algorithms and its attributes are:

- id: is the algorithm's ID.
- name: is the algorithm's name.
- description: is the algorithm's description in plain text.

The User entity represents the system users. When some jobs are linked to the user, the user can list all its owned jobs. In case the user is an administrator, it can also list the not owned jobs. The User attributes are:

- id: is the user's ID.
- name: is the user's full name.
- email: is the user's email address.
- password: is the user's password.
- administrator: is a flag that informs to the system if the user has administrative rights or it is a regular user.

The JobState enumeration entity is the set of possible job states and its possible values are:

- JobNotQueued coded with integer value 0.
- JobQueued coded with integer value 1.
- JobExecuting coded with integer value 2.
- JobCompleted coded with integer value 3.
- JobAborted coded with integer value 4.

The StatisticsEntry entity represents each of the job's statistics notifications received from the agent. It represents execution progress during time and its attributes are:

- iterationsComputed: is the number of iterations computed.

- iterationsComputedSinceBestIncumbent: is the number of iterations computed since the best incumbent has been updated.
- computationTime: is the amount of computation time elapsed in milliseconds.
- computationTimeSinceBestIncumbent: is the amount of computation time elapsed in milliseconds since the best incumbent has been updated.
- bestIncumbentObjectiveValue: is the objective value of the best incumbent found.
- timestamp: is the date and time when the statistic entry has been notified.

The IPAddress entity represents an IP address v4. It only has the 4 bytes required to encode the IP address.

The EndPoint entity represents an IP address v4 plus the connection port represented as an unsigned short number.

The AgentStatus enumeration entity is the set of possible agent states from the Cluster Manager and its possible values are:

- AgentIdle coded with integer value 0. No job is assigned to the agent.
- AgentBusy coded with integer value 1. An assigned job is under computation.
- AgentCompleted coded with integer value 2. An assigned job has been completed and the agent is waiting for results reception

The Agent entity represents each HPC Agent connected to the Cluster Manager and its attributes are:

- id: the agent's ID.
- endpoint: is the IP address and port where the agent listens for Cluster Manager connections.
- executablePath: is the path to the agent's executable on its remote file system.
- executableFlags: is the set of flags used to execute the agent through SSH connection.
- sshUserName: is the username used by SSH to access to the agent's machine.
- sshPassword: is the password used by SSH to access to the agent's machine.
- sshEndpoint: is the IP address and port where the agent's SSH daemon is listening for incoming connections.

The important agent's methods are:

- start(): used to establish a remote connection to the agent's SSH terminal using the sshEndpoint, sshUserName and sshPassword. When connection is established, the executablePath and executableFlags are used to initialize the HPC Agent.
- hello(): used to send a "Hello" message to the agent and stand for the answer. This function is used by the Job Scheduler.
- getStatus(): used to send a "Status" message to the agent and stand for the answer. This function is used by the Job Scheduler.
- solveJob(): used to send a "Solve" message to the agent attaching the job to be solved and stand for the answer. This function is used by the Job Scheduler.
- abortJob(): used to send an "Abort" message to the agent attaching the job to be aborted and stand for the answer. This function is used by the Job Scheduler.

- stop(): used to establish a remote connection to the agent's SSH terminal using the sshEndpoint, sshUserName and sshPassword. When connection is established, the agent is located and killed sending Shell commands.

4.4.1.1 Job Scheduler's priority queue

The Job's Scheduler is the responsible of deciding which job is going to be executed each time. To schedule the jobs, a job priority queue is used, and a priority key is assigned to each job. The next job to be executed each time is the job with lower priority key.

This priority key is a scaled timestamp, and is computed using the time instant when the job has been set to "JobQueued" and the relative priority requested by the job's owner.

The relative priority is a number that represents a priority level and can be 1 for low priority, 2 for mid priority and 3 for high priority. If needed, additional priority levels can be added. The default relative priority level is mid priority.

The priority key is computed as follows:

rp = Relative priority is one of {1=Low, 2=Mid, 3=High}

n = Number of relative priority levels = 3

ts = Time stamp computed as job's queue insertion date and time in milliseconds

pk = Priority key used by the priority queue to sort the jobs

pk = ts* (n + 1 - rp)

This priority key computation ensures that any job is going to remain into the queue indefinitely. New jobs are introduced into the queue at least with current date and time in milliseconds, so, if one job remains into the queue for a while, its priority key is going to be lower than newly introduced high priority jobs, so a low priority ancient job can become prior than a new high priority job.

Note that if all jobs are set with the same priority, the priority queue acts as a FIFO queue.

4.4.2 HPC agent design

The HPC Agent's conceptual diagram is composed by 3 main entities: the Job, the Algorithm and the StatisticsEntry. These entities and the rest of components are described below from the point of view of the HPC Agent.

The StatisticsEntry is exactly equal than the Cluster Manager respective entity so it is not further described. The only difference is that in the HPC Agent, instead of having a set of statistics entries, there is a single statistics entry used to store the current statistics.

The Job entity on the HPC Agent only uses the attributes id, inputDataFilePath, resultsFilePath and algorithm. The rest of attributes are not required. One extra attribute not used by the Cluster Manager is required in the HPC Agent:

- currentStatistics: is the current statistics entry. On the next progress/results notification this StatisticsEntry is going to be sent.

The AlgorithmInputData and AlgorithmOutputData entities wrap each one a generic object used to store in-memory the algorithm input and results data.

The Algorithm entity represents the algorithm to be executed in the agent and has two additional attributes and three methods used by the agent. The additional Algorithm's entity attributes are:

- jobInputData: is an AlgorithmInputData entity containing the input data that has been read from the input file.
- jobResultsData: is an AlgorithmOutputData entity containing the results data generated from the algorithm that is going to be written into the results file.

The additional Algorithm's entity methods are:

- initialize(): is responsible of initializing the algorithm using the input data read from input file. This method returns true when the initialization has been done correctly or false if the input data is incorrect.
- executeIteration(): is responsible of executing one algorithm's iteration and updating the currentStatistics attribute with the partial results. The function returns true when additional iterations must be computed, or false if the execution has been completed.
- getResults(): is the function called by the agent when the computation has been completed to obtain the results data that must be written into the algorithm's results file.

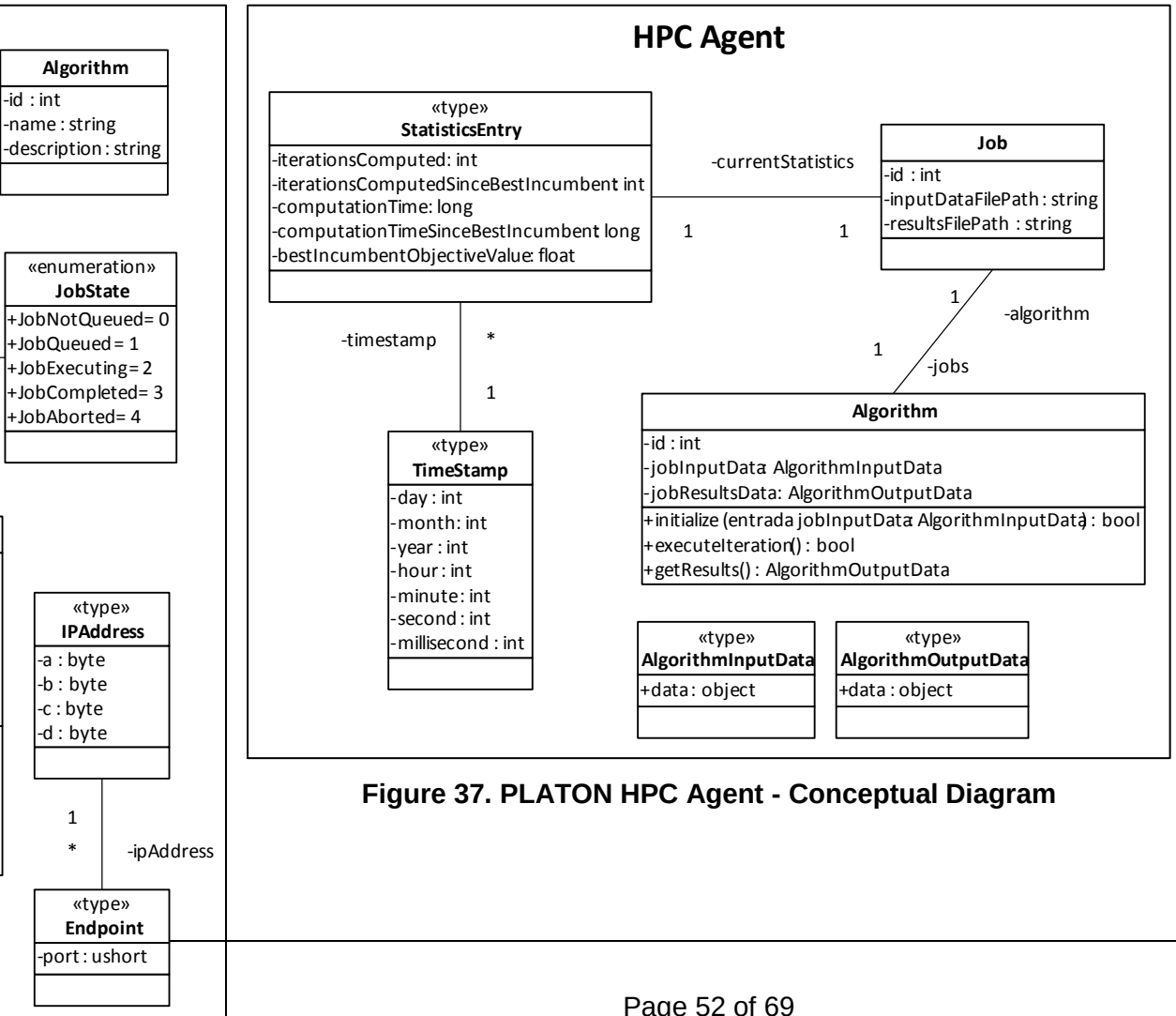


Figure 37. PLATON HPC Agent - Conceptual Diagram

4.4.3 Create/Execute/Retrieve Job sequence diagram

To describe operations done by the Cluster Manager and the HPC Agent when a job is created, modified and executed, an UML sequence diagram is provided. The rest of operations also have sequence diagrams, but are so simple and does not provide important information; or are equal than parts of the provided diagram.

The process, shown in Figure 38, begins when the user requests to the interface to create a new job. This job is created with the default job's state that is JobNotQueued. The interface requests a Job insertion into the database, and when the database confirms the job insertion, the interface also confirms to the user such situation.

Then, the user requests the list of algorithms to the interface and this request is forwarded to the database. The retrieved set of algorithms is then shown to the user who selects the algorithm it wants to associate to the job. When the user selects the algorithm the interface requests to the database to update the job associating the algorithm.

The next step is to provide the job's input data file through the interface. The interface stores the file into the network file system shared between the Cluster Manager and the HPC Agent, and also updates the file path into the job that is being created.

Now the job is ready to be executed. The user requests a job execution to the interface that updates the job by setting the job's state to "JobQueued". This specific update into the database triggers the Job Scheduler to update its internal priority queue inserting the newly created job and computing its priority as described in section 4.4.1.1.

When the HPC Agent becomes idle, the job scheduler removes the first (i.e. the most prior job) and assigns it to the HPC Agent by sending a "Solve" command that the agent confirms by sending a "Confirm Solve" response. When the Job Scheduler receives the "Confirm Solve" response it updates the job's state to "JobExecuting".

The job's execution begins by initializing the algorithm and reading the job's initial data. Then, one or more algorithm's heuristic iterations are executed and periodically the agent sends "Notify Progress" commands to the Cluster Manager that contains partial statistical results that are stored into the database to plot the algorithm's evolution on the interface. When the notifications are stored, a "Confirm Progress" response is sent to the agent.

When the computation is completed the agent stores the results into the file pointed by the results file path and it sends a "Notify Results" to the Cluster Manager that contains the final statistics of the algorithm's execution. Receiving this command implies that the execution has been completed, so the Cluster Manager stores the final statistics received and changes the job's state to "JobCompleted".

Finally, when the user wants to view the progress of the computation, it simply connects to the Cluster Manager's interface and requests a job retrieving. The information retrieved contains the job's data, the job's state, a link to download the job's input data file and the job's statistical progress of the computation. Additionally, if the job is completed, the interface retrieves a link to download the results file. Because both the input data file and the results file are stored into the shared file system, the interface of the Cluster Manager is able to directly retrieve the files to the user.

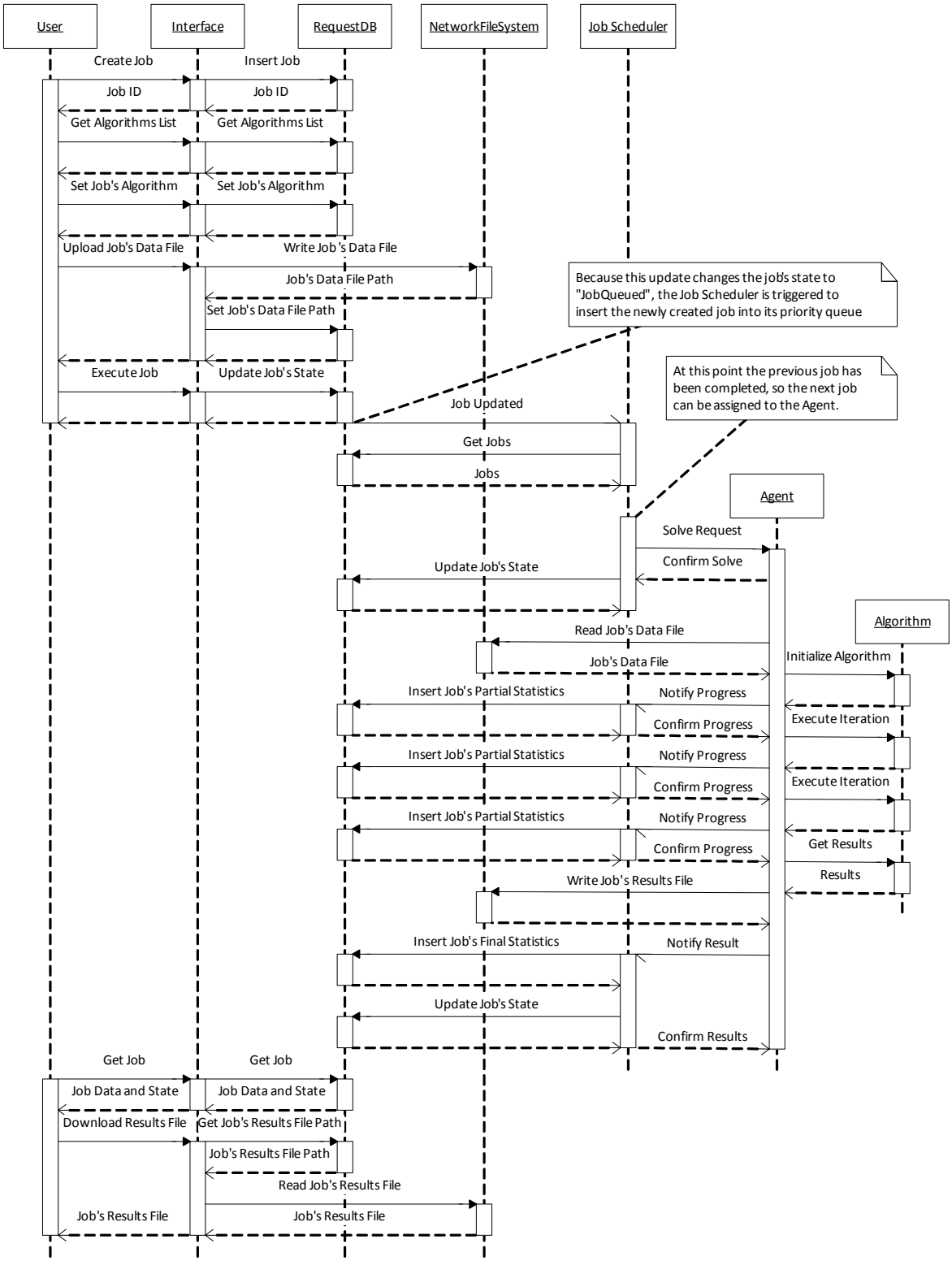


Figure 38. PLATON Create/Execute/Retrieve Job - Sequence Diagram

4.5 Developer guide

We provide an in-operation planning tool that allows the partners to use their own algorithms. This planning tool contains both an extendable PCE interface, i.e. usually partners will implement their algorithms there, and a PCC test application used to send requests to the PCE interface.

The interface provided and described in this section is based on PCE Communications Protocol [20] and the Synchronization VECtor [21] standards.

4.5.1 Path Computation Element API

In this section, we review the public PCEP structures and methods specified in the `pcep.h` file.

4.5.1.1 Session configurable parameters

PCEP peers negotiate a set of parameters before a session is established. At the time of writing this document, two parameters are specified relater to timers for session refresh and session expiration: *KeepAliveTimer* and *DeadTimer*, both in seconds.

The `PCEPConfig` structure allows specifying the range of the above timers.

```
typedef struct {
      u_int32_t defaultKeepAlive;
      u_int32_t minKeepAlive;
      u_int32_t maxKeepAlive;
      u_int32_t minDeadTimer;
      u_int32_t defaultDeadTimer;
      u_int32_t maxDeadTimer;
} PCEPConfig;
```

`PCEPConfig` ADT has defined the following methods:

`PCEPConfig *`**`PCEPConfig_new`**`()`

Constructor. Returns a valid pointer to a new `PCEPConfig` structure if success or `NULL` if an error happened while allocating memory.

`void` **`PCEPConfig_destroy`**`(PCEPConfig *config)`

Destructor. Releases the allocated memory pointed by the `config` argument.

4.5.1.2 PCEP

PCEP structure is the front-end of the PCEP API. It contains a number of private fields and the following methods to work with it:

`PCEP *`**`PCEP_new`**` (char *str_pceIPAddr, char *str_pccIPAddr, int isPCE, PCEPConfig *config)`

Constructor. Returns a valid pointer to a new `PCEP` structure if success or `NULL` if an error happened while allocating memory. The following arguments are needed:

- `char *str_pceIPAddr`: IP address of where the PCE runs. E.g. “10.10.1.1”
- `char *str_pccIPAddr`: In case of this instance is a PCC, this parameter should contains its local IP address.

• `int isPCE`: specifies whether this instance will act as a PCE (=1) or not (=0). • `PCEPConfig *config`: configuration parameters for PCEP sessions.
`void` **`PCEP_destroy`** `(PCEP *pcep)` Destructor. Releases the allocated memory pointed by the `pcep` argument.
`void` **`PCEP_startSession`** `(PCEP *pcep)` Starts a new session to the PCE. Note that two sessions between two peers is not allowed, so this method must be invoked only once.
`void` **`PCEP_registerCB`** `(PCEP *pcep, int event, int (*Client_notifyEvent) (void *, int))` Registers a callback function to be invoked after the specified event arises. The set of registrable events, are defined in the FSM section. The following arguments are needed: • `PCEP *pcep`: pointer to the PCEP structure. • `int event`: One of the above. • `(*Client_notifyEvent) (void *, int)`: Pointer to the callback function, where the first argument points to a `PCEPSession`. The second argument specified the condition that raised the event and it has not meaning in the current version of the API
`void` **`PCEP_unRegisterCB`** `(PCEP *pcep, int event)` Unregisters from the specified event. The following arguments are needed: • `PCEP *pcep`: pointer to the PCEP structure. • `int event`: One of the specified in the FSM section.

4.5.2 PCEP Finite State Machine (FSM)

In this section, we review the events that are available for registering callback functions, from those specified in the `pcep_fsm.h` file.

- `PCEP_EV_UP`: A new session has been established.
- `PCEP_EV_CLOSE`: The session has been closed.
- `PCEP_EV_PCREQ`: A new PCReq has been received at the PCE.
- `PCEP_EV_PCREP`: A new PCRep has been received from the PCE.

4.5.3 PCEP Session

In this section, we review the public `PCEPSession` methods specified in the `pcep_session.h` file. `PCEPSession` contains all the data regarding a specific PCEP session. It contains a number of private fields and the following methods to work with it:

`int` **`PCEPSession_newTransaction`** `(PCEPSession *session, PCEP_TransactionType type)`

Creates a new transaction to the peered PCE. Only one transaction type has been defined in this PCEP version; the one for request and response path computations. The following arguments are needed: • `PCEPSession *session`: Pointer to a valid PCEP session. • `PCEP_TransactionType type`: type of transaction. Defined in `pcep_transaction.h`.
`PCEP_PathComputation *`**`PCEPSession_getPathComputation`** `(PCEPSession *session)` Return a valid pointer to an initialized `PCEP_PathComputation` structure if success or `NULL` if an error happened while allocating memory. The following argument is needed: • `PCEPSession *session`: Pointer to a valid PCEP session.
`int` **`PCEPSession_sendReq`** `(PCEPSession *session)` Sends a PCReq message towards the peered PCE with the data in the `PCEP_PathComputation` structure. Returns 0 if success, -1 if error.
`int` **`PCEPSession_sendRep`** `(PCEPSession *session)` Sends a PCRep message towards the PCC that requested a path computation. It uses the data in the `PCEP_PathComputation` structure. Returns 0 if success, -1 if error.

4.5.4 Transaction

In this section, we review the public methods specified in the `pcep_transaction.h` file. It contains a number of data structures, type definitions and methods.

Type definitions:

`typedef enum {PATHCOMPUTATION} PCEP_TransactionType` Only one transaction type is defined in this PCEP version: • `PATHCOMPUTATION`: Specified a Req-Rep transaction type.

Data structures:

The following data structures are defined. Note that many of the fields are PCEP objects.

`PCEP_PathComputation`: Contains all the data regarding a PCReq and PCRep messages within a transaction.

```
typedef struct {
// public read only. USE the right add function for writing
      unsigned int requestCount;
      unsigned int svecCount;
      unsigned int responseCount;
      PCEP_Request **requestArray;
      SVEC ** svecArray;
      PCEP_Response **responseArray;
} PCEP_PathComputation
```

PCEP_Request: Contains *partial* data regarding a PCReq message within a transaction.

```
typedef struct {
// public read/write
      RP *rp;
      EndPoints *endPoints;
      LSPA*lspa;
      Bandwidth *reqBw;
      RRO *rro;
      Bandwidth *currentBw;
      IRO *iro;
      LoadBalancing *loadBal;

// public read only. USE the right add function for writing.
      unsigned int metricCount;
      Metric **metricArray;
} PCEP_Request
```

PCEP_Response: Contains all the data regarding a PCRep message within a transaction.

```
typedef struct {
// public read/write
      RP *rp;
      No_Path *noPath;
      LSPA *lspa;
      Bandwidth *bw;
      IRO *iro;

// public read only. USE the right add function for writing
      unsigned int metricCount;
      unsigned int pathCount;
      Metric **metricArray;
      PCEP_Path **pathArray;
} PCEP_Response
```

PCEP_Path: Contains the data regarding a path for a PCRep message within a transaction.

```
typedef struct {
// public read/write
      ERO *ero;
      LSPA *lspa;
      Bandwidth *bw;
      IRO *iro;

// public read only. USE the right add function for writing
      unsigned int metricCount;
      Metric **metricArray;
} PCEP_Path
```

Methods:

`PCEP_Request *`**`PCEP_PathComputation_addRequest`** `(PCEP_PathComputation *pcomp)` Adds a new `PCEP_Request` to the path computation data structure and returns a valid pointer to that request or `NULL` if an error happened. The following arguments are needed: • `PCEP_PathComputation *pcomp`: Pointer to a valid `PCEP_PathComputation` structure.
`SVEC *`**`PCEP_PathComputation_addSVEC`** `(PCEP_PathComputation *pcomp)` Adds a new `SVEC` object to the path computation data structure and returns a valid pointer or `NULL` if an error happened. The following arguments are needed: • `PCEP_PathComputation *pcomp`: Pointer to a valid `PCEP_PathComputation` structure.
`PCEP_Response *`**`PCEP_PathComputation_addResponse`** `(PCEP_PathComputation *pcomp)` Adds a new `PCEP_Response` to the path computation data structure and returns a valid pointer or `NULL` if an error happened. The following arguments are needed: • `PCEP_PathComputation *pcomp`: Pointer to a valid `PCEP_PathComputation` structure.
`PCEP_Path *`**`PCEP_Response_addPath`** `(PCEP_Response *response)` Adds a new `PCEP_Path` to the path response data structure and returns a valid pointer or `NULL` if an error happened. The following arguments are needed: • `PCEP_Response *response`: Pointer to a valid `PCEP_Response` structure.
`Metric *`**`PCEP_Response_addMetric`** `(PCEP_Response *response)` Adds a new `Metric` object to the path response data structure and returns a valid pointer or `NULL` if an error happened. The following arguments are needed: • `PCEP_Response *response`: Pointer to a valid `PCEP_Response` structure.
`Metric *`**`PCEP_Path_addMetric`** `(PCEP_Path *path)` Adds a new `Metric` object to the `PCEP_Path` data structure and returns a valid pointer or `NULL` if an error happened. The following arguments are needed: • `PCEP_Path *path`: Pointer to a valid `PCEP_Path` structure.

4.5.5 PCEP objects

In the current version of this document, the following objects are specified:

Object Name	Description	Definition
RO (ERO, RRO, IRO)	Route Object	`pcep_obj_ro.h`
RP	Request Parameters	`pcep_obj_rp.h`
NO_PATH	No Path Object	`pcep_obj_no_path.h`
LOAD_BALANCING	Load Balancing	`pcep_obj_load_balancing.h`
METRIC	Mertric Object	`pcep_obj_metric.h`
BANDWIDTH	Bandwidth object	`pcep_obj_bandwidth.h`
END_POINTS	End Points Object	`pcep_obj_end_points.h`
LSPA	LSP Attributes	`pcep_obj_lspa.h`

SVEC	Synchronization VECtor	pcep_obj_svec.h

The interface for the objects is really similar, where OBJECT needs to be replaced by the name of the desired object.

`OBJECT * OBJECT_new ()`

`OBJECT` constructor. Returns a valid pointer to a new `OBJECT` structure if success or `NULL` if an error happened while allocating memory.

`void OBJECT_init (OBJECT *object, …)`

Initializes `object`. The arguments depend on the specific object.

`int OBJECT_toString (OBJECT *object, char *str)`

Prints the contents of the object in the string. The following arguments are needed:

- `OBJECT *object`: Pointer to a valid `OBJECT` structure.
- `char *str`: allocated memory where the object in to be written.

Returns length of the string if positive or a processing error:

- `NOT_SUPPORTED_OBJ_ERROR`
- `BAD_OBJ_FMT_ERROR`

`int OBJECT_clone (OBJECT *source, OBJECT *dest)`

Clone the object from a source object. The following arguments are needed:

- `OBJECT * source`: Pointer to a valid `OBJECT` structure, which is the source of the object contents.
- `OBJECT * dest`: Pointer to a valid `OBJECT` structure, which are the destination of the object contents.

In addition to the above public methods, some others are available, as detailed in the specific section describing the object.

4.5.6 Examples

4.5.6.1 PCC test program

The following C code presents an example of PCC that sends requests to the in-operation planning tool integrated in a back-end PCE.

```
#include <zebra.h>
#include "memory.h"
#include "log.h"

#include "pcc.h"
#include "pcep_api.h"
#include "pcep_session.h"
#include "pcep_transaction.h"

extern PCC *pcc;

// Call back functions
int PCC_PCEPSessionDwnCB (void *arg, int condition);
int PCC_PCEPSessionUpCB (void *arg, int condition);
int PCC_ProcessPCRepCB (void *arg, int condition);

PCC *PCC_new() {

   PCC *pcc = (PCC *)XMALLOC(MTYPE_PCC, sizeof(PCC));
   if (!pcc) {
```

```
      zlog_err("[PCC] couldnt allocate memory");
      return NULL;
   }

   // Defining configuration parameters
   PCEPConfig *config = PCEPConfig_new();
   if (!config) {
      XFREE (MTYPE_PCC, pcc);
      zlog_err("[PCC] couldnt allocate memory");
      return NULL;
   }

   config->maxKeepAlive = 100;
   config->minKeepAlive = 0;
   config->defaultKeepAlive = 30;
   config->maxDeadTimer = 300;
   config->minDeadTimer = 0;
   config->defaultDeadTimer = 120;

   // Creating a PCEP acting as a PCC
   pcc->pcep = PCEP_new("10.10.1.1", "10.10.1.2", 0, config);
   PCEPConfig_destroy(config);
   if (!pcc->pcep) {
      XFREE (MTYPE_PCC, pcc);
      return NULL;
   }

   // Call back registering
   PCEP_registerCB (pcc->pcep, PCEP_EV_CLOSE, PCC_PCEPSessionDwnCB);
   PCEP_registerCB (pcc->pcep, PCEP_EV_UP, PCC_PCEPSessionUpCB);
   PCEP_registerCB (pcc->pcep, PCEP_EV_PCREP, PCC_ProcessPCRepCB);

   zlog_info("[PCC_new] running on %s", pcc->str_PCCIPAddr);

   // PCC must initiate session establishment
   PCEP_startSession(pcc->pcep);

   return pcc;
}

void PCC_destroy(PCC *pcc) {

   if (pcc) {
      PCEP_destroy (pcc->pcep);
      XFREE (MTYPE_PCC, pcc);
   }
}

int PCC_PCEPSessionDwnCB (void *arg, int condition) {

   if (!arg) return -1;

   PCEPSession *session = (PCEPSession *) arg;
   zlog_info ("[PCC_PCEPSessionDwnCB] Session 0x%08x is DOWN",
            session->peerIPAddr);
   pcc->enabled = 0;

   return 0;
}

int PCC_PCEPSessionUpCB (void *arg, int condition) {

   if (!arg) return -1;
```

```
   PCEPSession *session = (PCEPSession *) arg;

   zlog_info ("[PCC_PCEPSessionUpCB] Session 0x%08x is UP",
            session->peerIPAddr);

   // Sets a flag to indicate that the session is up
   pcc->enabled = 1;

   // Example of New Path Computation Request. This fragment should be in another
initiating method. It is in this method just as an example.

   // Create a new Transaction to send a PCReq.
   PCEPSession_newTransaction (session, PATHCOMPUTATION);

   // Get the Path Computation structure and fill in the request.
   PCEP_PathComputation *pathComp = PCEPSession_getPathComputation (session);
   PCEP_Request *request = PCEP_PathComputation_addRequest (pathComp);

   request->rp = RP_new();
   RP_init(request->rp, 0, 1, 0, 12);

   request->endPoints = EndPoints_new();
   EndPoints_init(request->endPoints, 0x0101A8C0, 0x0501A8C0);

   // Send the PCReq message to the peered PCE.
   PCEPSession_sendReq (session);

   return 0;
}

int PCC_ProcessPCRepCB (void *arg, int condition) {

   if (!arg) return -1;

   PCEPSession *session = (PCEPSession *) arg;

   zlog_info ("[PCC_ProcessPCRepCB] Received PCRep Message for session 0x%08x",
            session->peerIPAddr);

   // Get the Path Computation structure
   PCEP_PathComputation *pathComp = PCEPSession_getPathComputation (session);

   // Reads the response for each request
   int i;
   char str[1024];
   for (i=0; i<pathComp->responseCount; i++) {

      RP_toString(pathComp->responseArray[i]->rp, str);
      zlog_info ("[PCC_ProcessPCRepCB] %s", str);

      int j;
      for (j=0; j<pathComp->responseArray[i]->pathCount; j++) {
          ERO_toString(pathComp->responseArray[i]->pathArray[j]->ero, str);
          zlog_info ("[PCC_ rocessPCRepCB] %s", str);
      }
   }

   return 0;
}
```

4.5.6.2 Planning Tool with integrated PCEP interface

The following C code illustrates how algorithms can be integrated in PLATON.

```
#include <zebra.h>
#include "memory.h"
#include "log.h"

#include "pce.h"
#include "pcep_api.h"
#include "pcep_session.h"

extern PCE *pce;

// Call back functions
int PCE_PCEPSessionUpCB (void *arg, int condition);
int PCE_PCEPSessionDwnCB (void *arg, int condition);
int PCE_ProcessPCReqCB (void *arg, int condition);

PCE *PCE_new() {
   PCE *pce = (PCE *)XMALLOC(MTYPE_PCE, sizeof(PCE));
   if (!pce) {
      zlog_err("[PCE] couldnt allocate memory");
      return NULL;
   }

   // Defining configuration parameters
   PCEPConfig *config = PCEPConfig_new();
   if (!config) {
      XFREE (MTYPE_PCE, pce);
      zlog_err("[PCE] couldnt allocate memory");
      return NULL;
   }

   config->maxKeepAlive = 100;
   config->minKeepAlive = 0;
   config->defaultKeepAlive = 30;
   config->maxDeadTimer = 300;
   config->minDeadTimer = 0;
   config->defaultDeadTimer = 120;

   // Creating a PCEP acting as a PCE
   pce->pcep = PCEP_new((“10.10.1.1”, NULL, 1, config);
   PCEPConfig_destroy(config);
   if (!pce->pcep) {
      XFREE (MTYPE_PCE, pce);
      return NULL;
   }

   // Call back registering
   PCEP_registerCB (pce->pcep, PCEP_EV_CLOSE, PCE_PCEPSessionDwnCB);
   PCEP_registerCB (pce->pcep, PCEP_EV_UP, PCE_PCEPSessionUpCB);
   PCEP_registerCB (pce->pcep, PCEP_EV_PCREQ, PCE_ProcessPCReqCB);

   zlog_info("[PCE_new] running on %s", pce->str_PCEIPAddr);

   return pce;
}

void PCE_destroy(PCE *pce) {

   if (pce) {
      PCEP_destroy (pce->pcep);
      XFREE (MTYPE_PCE, pce);
   }
}

int PCE_PCEPSessionDwnCB (void *arg, int condition) {
```

```
   if (!arg) return -1;

   PCEPSession *session = (PCEPSession *) arg;

   zlog_info ("[PCE_PCEPSessionDwnCB] Session 0x%08x is DOWN",
            session->peerIPAddr);

   return 0;
}

int PCE_PCEPSessionUpCB (void *arg, int condition) {

   if (!arg) return -1;

   PCEPSession *session = (PCEPSession *) arg;

   zlog_info ("[PCE_PCEPSessionUpCB] Session 0x%08x is UP",
            session->peerIPAddr);

   // Sets a flag to indicate that the session is up
   pce->enabled = 1;

   return 0;
}

int PCE_ProcessPCReqCB (void *arg, int condition) {

   if (!arg) return -1;

   PCEPSession *session = (PCEPSession *) arg;

   zlog_info ("[PCE_ProcessPCReqCB] Received PCReq Message for session 0x%08x",
            session->peerIPAddr);

   // Get the Path Computation structure
   PCEP_PathComputation *pathComp = PCEPSession_getPathComputation (session);

   // Creates a response for each request
   int i;
   for (i=0; i<pathComp->requestCount; i++) {

      PCEP_Response *response = PCEP_PathComputation_addResponse (pathComp);

// Each response MUST contains one RP object and a PATH with an ERO if success
// or a NO_PATH if no path was found.
      response->rp = RP_new();
      RP_clone(pathComp->requestArray[i]->rp, response->rp);

      PCEP_Path *path = PCEP_Response_addPath (response);

      path->ero = ERO_new();
      RO_addSubObj (path->ero,
          Unnumbered_newSubObj (pathComp->requestArray[i]->endPoints->sourceIP,
                    0x01));
      RO_addSubObj (path->ero,
          Unnumbered_newSubObj (0x0201A8C0, 0x01));
      RO_addSubObj (path->ero,
          Unnumbered_newSubObj (pathComp->requestArray[i]->endPoints->destinationIP,
          0x02));
   }

   // Send the PCRep message
   PCEPSession_sendRep (session);
```

```
   return 0;
}
```

4.6 User guide

In this section we describe the process to create and execute jobs in PLATON. Note that being PLATON's architecture highly complex, the previous UML diagrams show a simplified vision of the tool. Some entities such as projects has been avoided and simplified in the UML diagrams and related explanations.

Some remarks to better understand this guide:

- Jobs are grouped into projects.
- Input files are related to projects and can be shared between jobs in the same project.
- Jobs are added directly to the queue after being created so they are executed as soon an agent is free.

The first step is to log in PLATON typing a valid username and password.

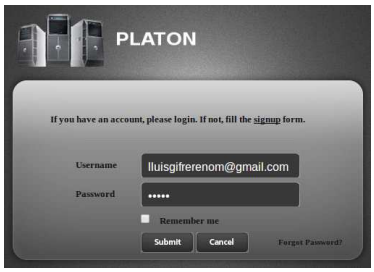


Figure 39. PLATON - Login

After logging into PLATON, the user is redirected to its projects list. To create a new project, click the "Projects -> New Project" menu. See Figure 40

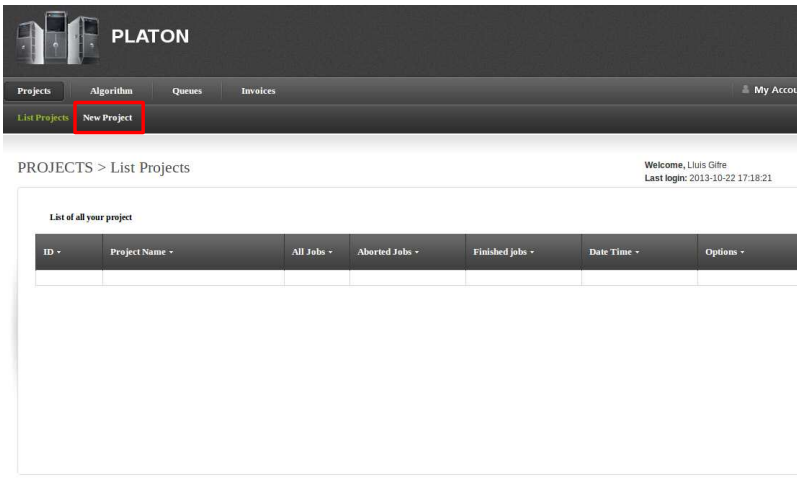


Figure 40. PLATON - User's projects list

When selected the "New Project" menu, the user is redirected to the input form shown in Figure 41 where the project's name must be filled in and then the user must click the submit button.

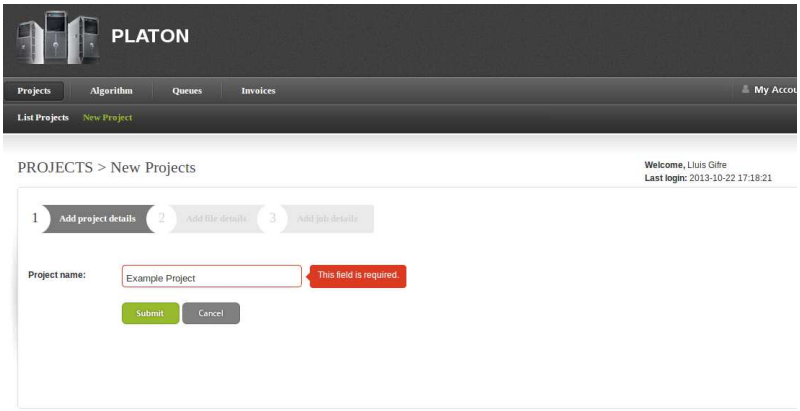


Figure 41. PLATON - Create Project

After clicking the submit button the user is redirected to the second project creation step where the input data file must be selected. Click submit button to continue. Additional input data files can be uploaded to the project latter, but one is mandatory to create the project.

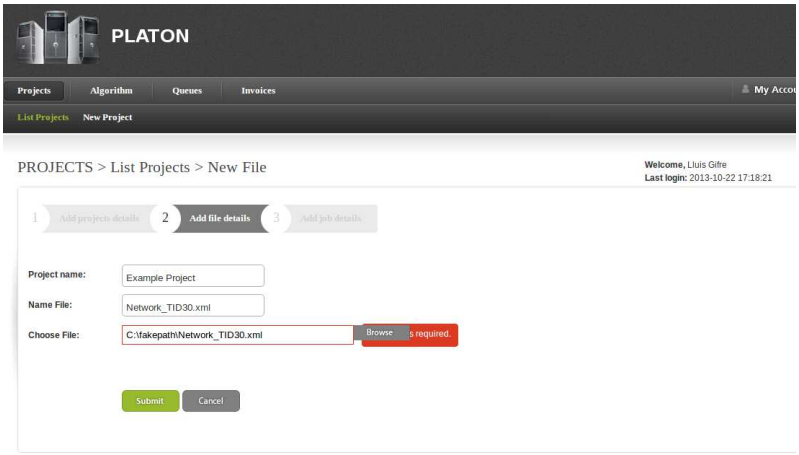


Figure 42. PLATON - Select Input Data File

After clicking the submit button, the user is redirected back to its project's list where the new project is now shown. Some buttons related to the project are shown in the right hand side of the project row. These buttons allow the user to edit the project, delete the project, create a new job, upload a new input data file, list the jobs, and list the input data files. In our case, to create a new job, the "NJ" (for New Job) button must be clicked and the user is redirected to the new job form.

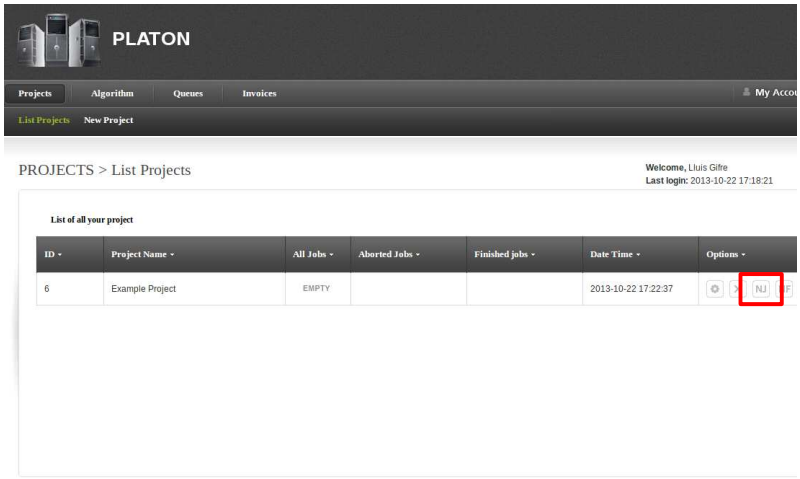


Figure 43. PLATON - User's projects list having a project

The user is redirected to the form shown in Figure 44, where the computing algorithm, the input data file from the project, the job name and the job priority are selected.

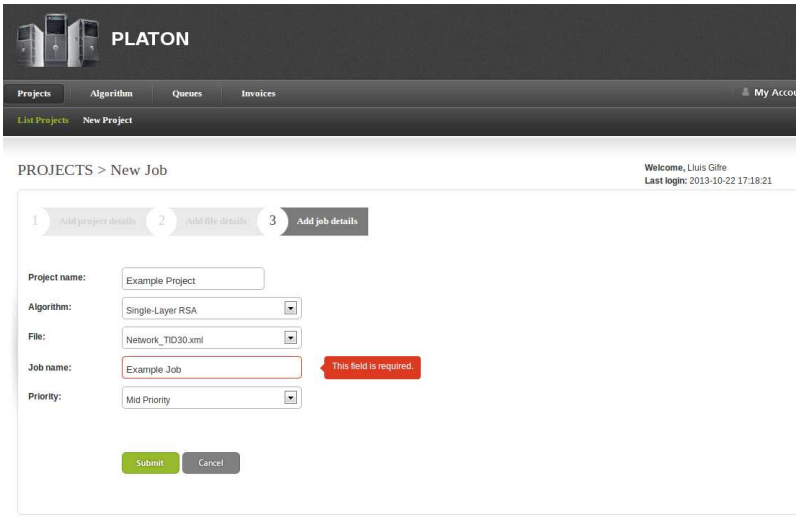


Figure 44. PLATON - Create Job

After clicking the submit button, the user is redirected to the project's job list that contains the newly created job in "Job Queued" status. As soon an agent is free, the job is executed and status is changed to "Job Executing" as shown in Figure 45. Additionally in this list the jobs have several options that consists on edit and delete the job if it is not under execution, view the input file, view the results file if the job has been completed, and view the current execution status details shown in Figure 46.

The list of jobs and the job's execution detail shown respectively in Figure 45 and Figure 46 are refreshed dynamically each 20 seconds.

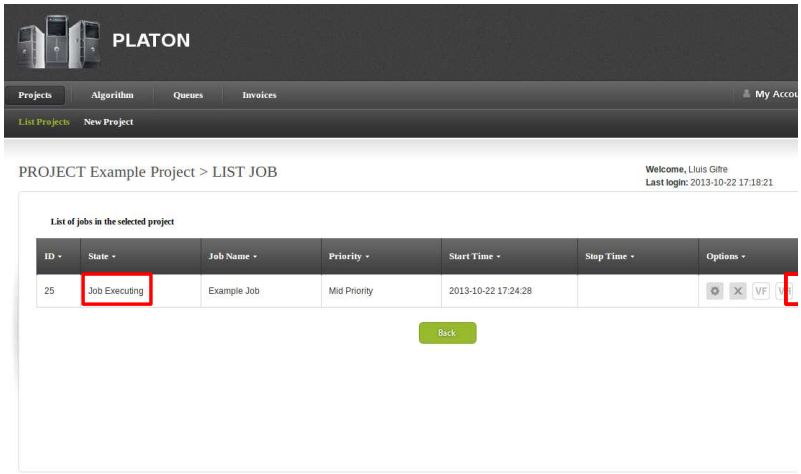


Figure 45. PLATON - Job Executing

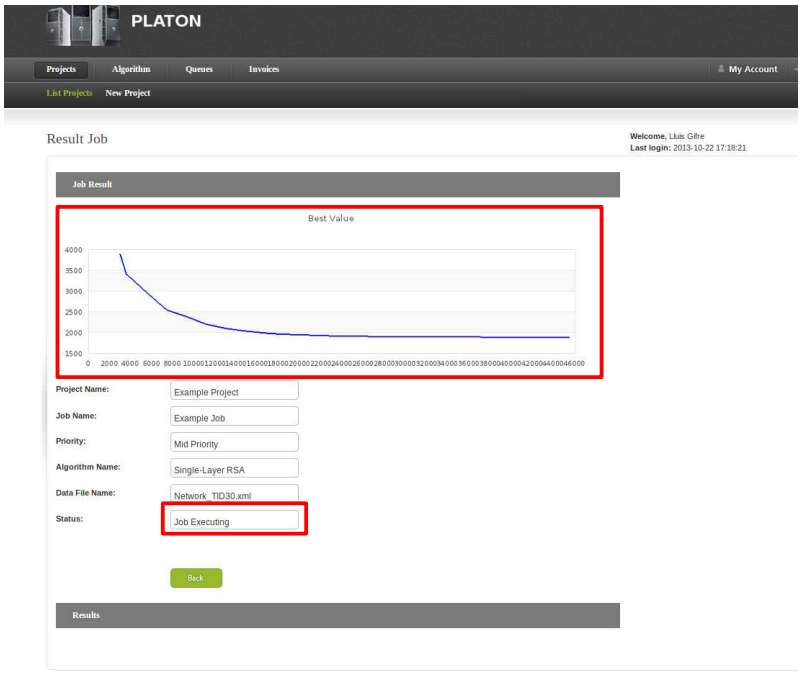


Figure 46. PLATON – Job's Execution Status – Job Executing

Figure 46 show the job's progress. This information is dynamically refreshed and the best solution found is plotted dynamically. When the computation is completed, the job's status changes dynamically to "Job Completed" and the bottom section is updated with the results file download links as shown in Figure 47.

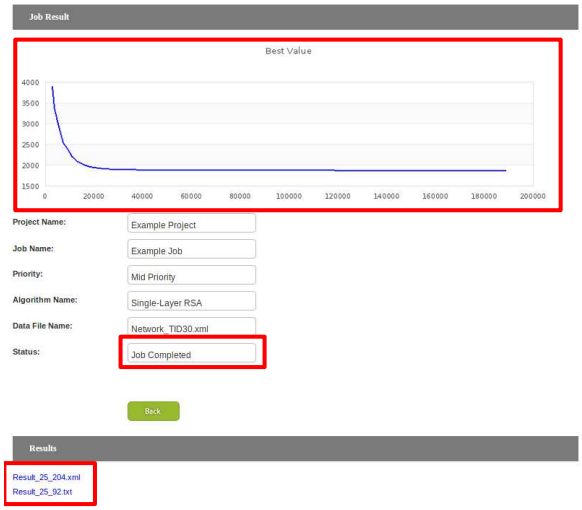


Figure 47. PLATON – Job's Execution Status – Job Completed

5 Conclusions

This deliverable focused on two network planning tools for next generation flexgrid optical networks.

The first one, named MANTIS, is intended for off-line planning and testing algorithms that can be designed for dynamic network operation.

The second one, named PLATON, focuses on off-line and in-operation network planning. For off-line planning, PLATON can rely on state-of-the-art computation hardware, such as GPUs, which enable the design of highly parallel algorithms. As for the in-operation planning, PLATON includes a PCEP interface so as to receive PCEP requests from a PCE.

END OF DOCUMENT