

net2plan

Net2Plan 0.4.0

User's manual

15TH FEBRUARY 2016

Pablo Pavón Mariño

Contents

1	Introduction	3
1.1	A brief overlook of Net2Plan	3
1.1.1	Organization of this document	4
1.2	Accompanying book and teaching materials	4
1.3	Videotutorials	5
1.4	Installing instructions	5
1.4.1	Directories	5
1.5	Licensing	6
1.6	Authors	6
1.6.1	Net2Plan	6
1.6.2	Java Optimization Modeler (JOM)	7
1.7	Citing Net2Plan and JOM in research works	7
1.8	Release notes	7
2	The Net2Plan network model	9
2.1	A network - <code>NetPlan</code> object	9
2.2	Nodes - <code>Node</code> object	10
2.3	Links - <code>Link</code> object	10
2.4	Traffic demands - <code>Demand</code> object	11
2.5	Multicast traffic demands - <code>MulticastDemand</code> object	12
2.6	Multicast trees - <code>MulticastTree</code> object	12
2.7	Routing of unicast traffic: source-routing vs. hop-by-hop routing	13
2.7.1	Routing loops	14
2.7.2	Source-routing of the traffic - <code>Route</code> object	15
2.7.3	Source-routing of the traffic - <code>ProtectionSegment</code> object	16
2.8	Shared-risk groups - <code>SharedRiskGroup</code> object	17
2.9	Multilayer networks	18
2.9.1	Network layers - <code>NetworkLayer</code> object	18
2.10	The default failure model in Net2Plan	19
2.10.1	Default failure model in multilayer networks	20
3	The Net2Plan Graphical User Interface (GUI)	21
3.1	Menu File	21
3.1.1	File → Options	22
3.1.2	File → Classpath editor	23
3.1.3	File → Java error console	23
3.1.4	File → Java error console	24
3.2	Offline network design	24
3.2.1	Network topology panel	25
3.2.2	Warnings panel	26
3.2.3	View/edit network state tab	26
3.2.4	Algorithm execution tab	29

3.2.5	View reports tab	30
3.3	Traffic matrix design	31
3.3.1	Traffic generation: general traffic models	31
3.3.2	Traffic generation: population-distance traffic model	32
3.3.3	Manual matrix introduction/edition	34
3.3.4	Traffic normalization	34
3.3.5	Creating a set of traffic matrices from a seminal one	35
3.4	Online network simulation	35
3.4.1	The event driven simulation framework	36
3.4.2	Graphical User Interface	38
3.5	Help menu	42
4	The Net2Plan Command-Line Interface (CLI)	43
4.1	Examples	43
5	Development of algorithms and reports in Net2Plan	45
5.1	Net2Plan Library, Built-in Examples and Code Repository	46
5.2	JOM: Java Optimization Modeler	46
5.3	Preparing a Java IDE for Net2Plan programming	46
6	Technology-specific libraries	48
	References	49

Chapter 1

Introduction

1.1 A brief overlook of Net2Plan

Net2Plan is an open-source and free to use Java-based software, licensed under the GNU Lesser General Public License (LGPL). Net2Plan has its origins in September 2011, as an accompanying resource for new network optimization courses at Telecommunications Engineering degrees in the Technical University of Cartagena (Spain). After its creation, Net2Plan has spread to other Universities, and is applied in a number of works in the academia and industry.

Net2Plan was designed with the aim to overcome the barriers imposed by existing network planning tools in two forms: (i) users are not limited to execute non-disclosed built-in algorithms, but also can integrate their own algorithms, applicable to any network instance, as Java classes implementing particular interfaces, and (ii) Net2Plan defines a network representation, so-called network plan, based on abstract and technology-agnostic concepts such as nodes, links, traffic unicast and multicast demands, routes, multicast trees, forwarding rules, protection segments, shared-risk groups and network layers.

Network instances can have an arbitrary number of layers, arranged in arbitrary forms. Technology-specific information can be introduced via user-defined attributes attached to nodes, links, routes, layers etc. in the network plan. The combination of a technology-agnostic substrate and technology-related attributes provides the required flexibility to model any network technology within Net2Plan, an added value from a didactic point of view. In this respect, current Net2Plan version provides specific libraries to ease the design of IP, wireless and optical networks.

Net2Plan provides both a graphical user interface (GUI) and a command-line interface (CLI). In either mode, Net2Plan includes four tools:

- *Offline network design*: Targeted to execute offline planning algorithms, that receive a network design as an input and modify it in any form (e.g. optimize the routing, the capacities, topology etc.). Algorithms based on constrained optimization formulations (e.g. ILPs or convex formulations) use the open-source freeware Java Optimization Modeler

<http://www.net2plan.com/jom>

to interface from Java to a number of external solvers such as GLPK, CPLEX or IPOPT, that produce a numerical solution. The modeling syntax of JOM is human-readable, and capable of handling arrays of decision variables and constraints of arbitrary dimensions, facilitating the definition and solving of complex models directly from Java in a few lines of code.

- *Online simulation*: Permits building simulations of online algorithms that code how the network *reacts* to different events generated by built-in or user-developed event generation modules. For instance, it can be used to evaluate network recovery schemes that react to failures and repairs or dynamic provisioning algorithms that allocate resources reacting to time-varying traffic demands. Several built-in algorithms exist coding e.g. how IP/OSPF networks or some types of IP over WDM multilayer networks react to traffic fluctuations and failures. Also, some distributed algorithms for congestion control, capacity allocation in wireless networks and other contexts, are implemented as online algorithms, where nodes asynchronously iterate to adapt to network conditions.
- *Automatic report generation*: Net2Plan permits the generation of built-in or user-defined reports, from any network design.
- *Traffic matrix generation*: Net2Plan assists users in the process of generating and normalizing traffic matrices, according to different models.

1.1.1 Organization of this document

The rest of this chapter is devoted to describe some basic information about:

- Section 1.2 introduces an accompanying book published by the author with the theoretical fundamentals of network optimization, with a practical approach based on Net2Plan examples. The majority of built-in algorithms come from examples thoroughly described in the book. Also, teaching materials available in the website are introduced.
- Section 1.3 introduces the videotutorials available in the Net2Plan website.
- Section 1.4 comments on how to install and run Net2Plan.
- Section 1.5 comments on the license of Net2Plan and JOM.
- Section 1.6 comments on the Net2Plan origins and authorship.
- Section 1.7 informs on how we would prefer the work to be cited in research publications.
- Section 1.8 describes the release notes.

Then, Chapter 2 is devoted to describe the network representation in Net2Plan, and is an unavoidable reading for understanding the tool.

Chapter 3 describes the functionalities in the graphical user interface. Chapter 4 is focused on the command-line interface. Chapter 5 is focused on the development of new algorithms and reports in Net2Plan. Finally, Chapter 6 inform on some libraries and available algorithms in Net2Plan, specific to networks technologies like IP/OSPF, wireless or WDM (optical).

1.2 Accompanying book and teaching materials

Net2Plan has been extensively used as an accompanying resource of the network optimization book:

Pablo Pavón Mariño, *Optimization of computer networks. Modeling and algorithms. A hands-on approach*. Wiley, May 2016.

The book is targeted to grow in the reader the ability to model a multitude of network optimization problems, and create algorithms for them. The book materials indexed in the repository

<http://www.net2plan.com/ocn-book>

include all the examples of models and algorithms in the book implemented as Net2Plan offline and online algorithms, and reports. They can be used to find numerical solutions to multiple real-life network problems. The book also includes exercises where the reader can develop and test their own Net2Plan algorithms, applying the techniques described.

A number of teaching materials for some courses using Net2Plan are provided in the Net2Plan website (e.g. lab work wordings).

1.3 Videotutorials

In the website, there is a video tutorial section in which users can see instructions and examples of how to use Net2Plan, and how to develop algorithms for it. The interested reader is encouraged to go through them.

1.4 Installing instructions

For installing the software, just uncompress the `.rar` file provided in any folder. For running the software just click in the `Net2Plan.jar` file. The software does not modify any registry information in the computer. For uninstalling, just remove the folder.

Net2Plan requires Java Runtime Environment 7 or higher versions and a screen resolution of, at least, 800x600 pixels. Since it is developed in Java, it works in the most well-known operation systems (Microsoft Windows, Linux, Mac OS X).

To install Net2Plan, just save the compressed file in any directory. Then, extract all the files and folders into a new directory. The software does not modify any registry information in the computer. For uninstalling, just remove the folder.

To execute Net2Plan in Graphical User Interface (GUI) mode (see Chapter 3), just double click on `Net2Plan.jar`, or execute the following command in a terminal: `java -jar Net2Plan.jar`.

To execute Net2Plan in Command-Line Interface (CLI) mode (see Chapter 4), execute the following command in a terminal: `java -jar Net2Plan-cli.jar`.

Important: Net2Plan makes use of the Java Optimization Modeler (JOM) library for solving optimization models interfacing to external solvers. JOM is shipped with Net2Plan, and used in a number of built-in algorithms and some functionalities. Please, follow the instructions in the JOM website (<http://www.net2plan.com/jom>) to install the external solvers needed. If these solvers are not installed, Net2Plan still works correctly, but the user cannot access the subset of functionalities and algorithms using JOM.

1.4.1 Directories

The directories in the Net2Plan installation are:

- **doc/help**: includes this user's guide.
- **doc/javadoc**: the Javadoc of Net2Plan (needed by algorithm developers).
- **lib**: Includes auxiliary libraries needed by Net2Plan.
- **src**: Includes the Net2Plan code, and some examples.
- **plugins**: Includes the Java classes and source code of some of the plugins in which Net2Plan is organized.
- **workspace**: includes example code and data.
- **workspace/data**: includes example topologies and traffic matrices.

1.5 Licensing

Net2Plan is free and open-source. It is licensed under the GNU Lesser General Public License Version 3 or later (the “LGPL”).

1.6 Authors

1.6.1 Net2Plan

Net2Plan tool has its origins in 2011, during the preparation of the teaching materials for two new courses at Universidad Politécnica de Cartagena (Spain) taught by Prof. Pablo Pavón Mariño, in Telecommunications Engineering degrees:

- *Telecommunication networks theory* (2nd year, 2nd quarter).
- *Network planning and management* (3rd year, 2nd quarter).

Up to version 0.3.1, Net2Plan was also a part of the Ph.D. work of José Luis Izquierdo Zaragoza, supervised by Prof. Pablo Pavón Mariño.

The authors and developers of Net2Plan are:

- Pablo Pavón Mariño and José Luis Izquierdo Zaragoza up to version 0.3.1.
- Pablo Pavón Mariño, from version 0.4.0 onwards. Naturally, the credit for the code inherited from initial Net2Plan versions is shared with José Luis!

Prof. Pablo Pavón would like to credit José Luis for his extraordinary commitment and efforts in Net2Plan origins, and to the members of GIRTEL research group (<http://girtel.upct.es>), as well as the students, practitioners and researchers using Net2Plan, for their fruitful feedback.

1.6.2 Java Optimization Modeler (JOM)

Prof. Pablo Pavón Mariño is the author of JOM (Java Optimization Modeler), an open-source Java library for modeling and solving optimization problems in a simple MATLAB-like syntax. JOM is a library extensively used in Net2Plan algorithms that numerically solve network problems by means of optimization solvers.

The JOM website is:

<http://www.net2plan.com/jom>

1.7 Citing Net2Plan and JOM in research works

In research works, Net2Plan can be cited using the publication:

P. Pavon-Marino, J.L. Izquierdo-Zaragoza, “Net2plan: an open source network planning tool for bridging the gap between academia and industry”, IEEE Network, vol. 29, no 5, p. 90-96, October/November 2015.

To cite JOM, please use the web site link:

<http://www.net2plan.com/jom>

Many built-in algorithms and reports in Net2Plan (many of them using JOM) are thoroughly described in the book:

Pablo Pavón Mariño, *Optimization of computer networks. Modeling and algorithms. A hands-on approach*. Wiley, May 2016.

1.8 Release notes

- Net2Plan 0.4.0 (February 15, 2016)
 - Major changes in the form in which the network model is programmed. Now **NetPlan** object gives access to a number of other elements in their own classes: **Node**, **Link**, **Demand**, **MulticastDemand**, ... Also, all the elements have both an identifier (**long**, maybe non-consecutive, never changes along time) and an index (0,1,...) (0-indexed and consecutive, removing an element rennumbers the rest). Link, node, demand etc. indexes are amenable as array indexes. Then, the laborious **Map**-based organization of the elements in the network model is not longer needed. In general, the algorithms' code get much simpler and clearer.
 - Full support for multicast traffic.
 - Some new options in the graphical user interface.
 - Full re-elaboration and reorganization of the built-in examples and reports, including plenty of new algorithms, to make Net2Plan an accompanying resource of [1].
- Net2Plan 0.3.1 (November 23, 2015)
 - Minor changes.

- Improved documentation.
- New: A new plugin architecture (undocumented) has been created to support the integration of external CLI/GUI tools or I/O filters. Original plugins are detached from the kernel and are also located into plugins folder.
- Net2Plan 0.3.0 (June 29, 2015)
 - Major changes in network model:
 - * New: Complete multilayer support, including layer coupling (links at an upper layer become demands at a lower layer, or viceversa).
 - * New: Identifiers for nodes, links, demands, and so on, are now long values. `get()` methods, whose output were arrays, now return maps.
 - * New: Internal speed-up via caching of common `get()` methods.
 - New: *Online simulation* is a new tool that merges the previous simulators into a common one, and it uses the same network model than for network design.
 - Improved documentation
- Net2Plan 0.2.3 (March 7, 2014)
 - Minor changes.
 - Improved documentation.
- Net2Plan 0.2.2 (October 16, 2013)
 - Minor changes
 - Improved documentation
- Net2Plan 0.2.1 (May 23, 2013)
 - New: *Time-varying traffic* simulator.
 - New: *Network design* is now able to execute multilayer algorithms.
 - Minor changes.
- Net2Plan 0.2.0 (March 18, 2013)
 - Initial Java version, many major changes from latest MATLAB version

In its very early stage Net2Plan was designed as a MATLAB toolbox. From version 0.2.0, previous MATLAB versions were discontinued, thus backward-compatibility is not ensured at all. However, interested users can find them in the website.

Chapter 2

The Net2Plan network model

This chapter describes the network model used in Net2Plan, and the key Java classes used to represent it.

2.1 A network - NetPlan object

A network design is stored into a data structure so-called network plan, the class:

```
com.net2plan.interfaces.networkDesign.NetPlan
```

in the Net2Plan library). The **NetPlan** object is used in all the Net2Plan functionalities:

- In the offline network design tool, algorithms receive a network plan and return a modified network plan. E.g. an algorithm optimizing the routing, expects to receive a **NetPlan** object with nodes, links and traffic, and adds the routing information to it.
- In the online tool, the **NetPlan** object contains the state of the network at a particular simulation moment. Online algorithms receive the current design, an event to react to (e.g. new traffic, a failure or reparation...) and produce the new network state by modifying the given **NetPlan** object.
- Reports are just built-in or user-defined that receive a design (**NetPlan** object) and produce a HTML file.

There is one and only one **NetPlan** object in each network representation, that gives access through its methods to all the network elements (nodes, links, demands, etc.), which are represented in the model by specific classes like **Node**, **Link**, **Demand**, **MulticastDemand**...

Specific network-wide information included in the **NetPlan** object is:

- A network name (an arbitrary **String**).
- A network description (an arbitrary **String**).
- User-defined name-value attributes, as a **Map<String,String>**.

The structure of the **NetPlan** class is cornerstone to understand how Net2Plan works. Below, we describe separately each of the elements that make up a design.

2.2 Nodes - Node object

Nodes are the basic entity of a network design. They can be the end points of the links, the sources or destinations of traffic, and also forward traffic not targeted to them.

Each node in a network is represented by a **Node** object, contained inside the **NetPlan** object. Specific node information contained in this object is:

- *Id* (**long**): A unique identifier or serial number assigned by the kernel, that never changes along the life of the *NetPlan* object. Nodes created later receive higher numbers, but not necessarily consecutive.
- *Index* (**int**): An identifier (**int**) assigned by the kernel 0,1,2,... to the nodes. Node indexes are renumbered when a node is removed (e.g. when node with index 0 is removed, all the other nodes reduce their index in one).
- *Name* (**String**): An arbitrary **String** with name of the node.
- *(X, Y) coordinates*: The coordinates of the node in a bidimensional Cartesian plane. Serves for visualization and can be optionally used to automatically compute the length of the links between the nodes.
- *Up/down state*: A node can be *up* (working correctly) or *down* (failed). In the latter case, Net2Plan assumes that it is not able to forward traffic. The carried traffic and occupied link capacity of all the traversing routes, multicast trees or protection segments is then set to zero, and is set back to its previous value when the resource gets up again.
- User-defined name-value attributes, as a **Map<String,String>**.

2.3 Links - Link object

Links are elements connecting the nodes, with the capability of carrying traffic between them. Links are always unidirectional. A link starts in one node, and ends in a *different* node (self-links are not allowed). Two nodes can be connected by zero, one or more links.

Each link in a network is represented by a **Link** object. Each link is characterized by:

- *Id* (**long**): A unique identifier or serial number assigned by the kernel, that never changes along the life of the *NetPlan* object. Links created later receive higher numbers, but not necessarily consecutive.
- *Index* (**int**): An identifier (**int**) assigned by the kernel 0,1,2,... to the links inside a given network layer. Link indexes are renumbered when a link in the same layer is removed (e.g. when link with index 0 is removed, all the other links reduce their index in one).
- *Origin node* (**Node**): The node where the link starts.
- *Destination node* (**Node**): The node where the link ends.
- *Network layer* (**NetworkLayer**): (Of interest in multilayer designs) The layer where the link belongs to. A link belongs to one and only one network layer. Note that in multilayer networks, nodes are not attached to any particular layer, while links are. Then, nodes are the elements in charge of moving traffic from one layer to other (traffic entering a node through a link at a given layer, and leaving it through other link at other layer).

- *Capacity* (**double**): The capacity of the link, measured in the link's layer capacity units (the capacity of all the links in the same layer is measured in the same units).
- *Length* (**double**): The length in km of the link.
- *Propagation speed* (**double**): The propagation speed of the signal along the link. Typically 200,000 km/s in wired networks, and 300,000 km/s in wireless. This is used in delay calculations.
- *Up/down state*: A link can be *up* (working correctly) or *down* (failed). In the latter case, Net2Plan assumes that it is not able to forward traffic. The carried traffic and occupied link capacity of all the traversing routes, multicast trees or protection segments is then set to zero, and is set back to its previous value when the resource gets up again.
- *Coupled demand* (**Demand**): (Of interest in multilayer designs) In multilayer networks, a link in an upper layer can be coupled to a demand in a lower layer with the same end nodes, to reflect that the lower layer demand is realizing the link. In these cases, the link capacity is no longer defined by the user, but automatically made equal to the coupled demand carried traffic. See Section 2.9 for further information.
- User-defined name-value attributes, as a `Map<String,String>`.

2.4 Traffic demands - Demand object

Unicast traffic is modeled through a set of demands (or *commodities*). Each demand represents an *offered* end-to-end unidirectional traffic flow to the network, between two different particular nodes (self-demands are not allowed). Two nodes can have zero, one or more demands between them.

Each demand in a network is represented by a **Demand** object. Each demand is characterized by:

- *Id* (**long**): A unique identifier or serial number assigned by the kernel, that never changes along the life of the *NetPlan* object. Demands created later receive higher numbers, but not necessarily consecutive.
- *Index* (**int**): An identifier (**int**) assigned by the kernel 0,1,2,... to the demands inside a given network layer. Demand indexes are renumbered when a demand in the same layer is removed (e.g. when demand with index 0 is removed, all the other demands reduce their index in one).
- *Ingress node* (**Node**): The node where the demand starts.
- *Egress node* (**Node**): The node where the demand ends.
- *Network layer* (**NetworkLayer**): (Of interest in multilayer designs) The layer where the demand belongs to. A demand belongs to one and only one network layer.
- *Offered traffic* (**double**): The amount of offered traffic, measured in the demands' layer traffic units (the offered traffic of all the demands in the same layer is measured in the same units). The traffic that is actually carried depends on how the demand traffic is routed (see Section 2.7).
- *Coupled link* (**Link**): (Of interest in multilayer designs) In multilayer networks, a demand in the lower layer can be coupled to a link in an upper layer with the same end nodes, to reflect that the lower layer demand is realizing the link. In these cases, the link capacity is no longer defined by the user, but automatically made equal to the coupled demand carried traffic. See Section 2.9 for further information.
- User-defined name-value attributes, as a `Map<String,String>`.

2.5 Multicast traffic demands - MulticastDemand object

Multicast traffic is modeled through a set of multicast demands. Each demand represents an *offered* multicast traffic flow, starting in a particular ingress node, and ending in a particular set of egress nodes (different to the ingress node). The number of multicast demands affecting a node is arbitrary.

Each multicast demand in a network is represented by a **MulticastDemand** object. Each multicast demand is characterized by:

- *Id* (**long**): A unique identifier or serial number assigned by the kernel, that never changes along the life of the *NetPlan* object. Multicast demands created later receive higher numbers, but not necessarily consecutive.
- *Index* (**int**): An identifier (**int**) assigned by the kernel 0,1,2,... to the multicast demands inside a given network layer. Multicast demand indexes are renumbered when a multicast demand in the same layer is removed (e.g. when multicast demand with index 0 is removed, all the other demands reduce their index in one).
- *Ingress node* (**Node**): The node where the multicast demand starts.
- *Egress nodes* (**Set<Node>**): The set of nodes where the multicast demand ends.
- *Network layer* (**NetworkLayer**): (Of interest in multilayer designs) The layer where the multicast demand belongs to. A multicast demand belongs to one and only one network layer.
- *Offered traffic* (**double**): The amount of offered traffic, measured in the demands' layer traffic units (the offered traffic of all the demands in the same layer is measured in the same units). The traffic that is actually carried depends on how the multicast demand traffic is routed through its associated multicast trees.
- *Coupled links* (**Set<Link>**): (Of interest in multilayer designs) In multilayer networks, a multicast demand in the lower layer can be coupled to a set of links in the upper layer, all of them starting in the demand ingress node, and ending in each of the multicas demand egress nodes. The coupling reflects that the lower layer multicast demand is realizing the set of links. In these cases, the link capacities are no longer defined by the user, but automatically made equal to the coupled multicast demand carried traffic. See Section 2.9 for further information.
- User-defined name-value attributes, as a **Map<String,String>**.

2.6 Multicast trees - MulticastTree object

A multicast tree is an element carrying the traffic of a multicast demand. It is composed of a set of links comprising a unidirectional tree, starting in the associated multicast demand ingress node, and ending at its egress nodes. Multicast trees must connect the ingress node and each of the egress nodes without loops. This results in that the number of links in the tree will be equal to the number of nodes minus one.

A multicast tree can be assigned to only one multicast demand, but a multicast demand can be carried by zero, one or more trees. When the number of trees is two or more, we say that the multicast routing is bifurcated.

Each multicast tree in a network is represented by a **MulticastTree** object. Each multicast tree is characterized by:

- *Id* (**long**): A unique identifier or serial number assigned by the kernel, that never changes along the life of the *NetPlan* object. Multicast trees created later receive higher numbers, but not necessarily consecutive.
- *Index* (**int**): An identifier (**int**) assigned by the kernel 0,1,2,... to the multicast tree inside a given network layer. Multicast tree indexes are renumbered when a multicast tree in the same layer is removed (e.g. when multicast tree with index 0 is removed, all the other trees reduce their index in one).
- *Associated multicast demand* (**MulticastDemand**): The multicast demand of which this tree is carrying traffic. The ingress and egress nodes of the demand must be the ones of the tree.
- *Carried traffic* (**double**): The amount of traffic that this tree is carrying, measured in the tree layer traffic units.
- *Occupied link capacity* (**double**): The amount of capacity in the traversed links that this multicast tree occupies, measured in the tree layer link capacity units. Typically, the demand traffic and link capacity are measured in the same units (e.g. Gbps), and the tree carried traffic equals its occupied link capacity.
- User-defined name-value attributes, as a `Map<String,String>`.

2.7 Routing of unicast traffic: source-routing vs. hop-by-hop routing

The routing inside a network layer is the form in which the offered traffic represented by the demands is carried on the layer links. In Net2Plan there is only one form of routing the multicast traffic: using multicast trees. However, Net2Plan permits defining *two* different forms of routing the *unicast* traffic inside a layer:

- *Source-routing*: In source routing, each traffic demand is assigned a set of routes, from its ingress to its egress node. A route defines a sequence of traversed links, the amount of traffic of the demand that it carries (in traffic units) and the amount of capacity that consumes in each link (in link capacity units). Source routing is characteristic of connection-oriented technologies like MPLS, ATM, OTN or SONET/SDH, where a flow completes a connection establishment stage before sending any data. During this stage, the network decides and preconfigures the flow routing in the traversed nodes.
- *Hop-by-hop routing*: In hop-by-hop routing, nodes in the network define the routing using so-called forwarding rules. Forwarding rules are triples (d, e, f) , where d is a demand, e a link and f a number between 0 and 1. f represents the fraction of traffic that appears in the origin node of link e (either is generated by it if it is the ingress of d , or enters it through the input links), that is forwarded through link e .

Typically, the forwarding rules of the output links of a node are configured in structures called *routing tables* or *forwarding tables* in it. Given a node n , and a demand d , the forwarding rules of d associated to the node output links must sum at most 100%:

- If they sum 100%, the node forwards all the traffic of the demand. This is the typical case when the node is not the egress node of the demand.
- If n is the end node of the demand d , the output forwarding rules typically sum zero. Net2Plan assumes that the non-forwarded traffic was successfully received.
- If they sum less than 100% (e.g. 0%), and the node is *not* the end node of the demand, Net2Plan assumes that non-forwarded traffic is dropped in the node.

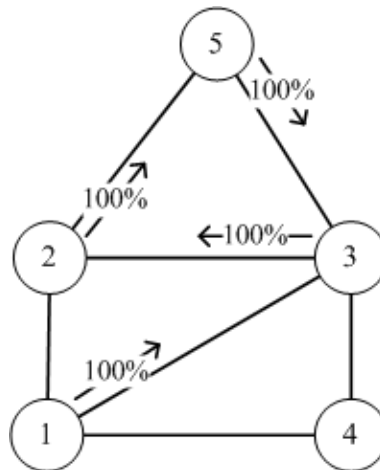


Figure 2.1: Example of a closed loop (3-2-5), for a demand from node 1 to node 4.

Forwarding rules reflect better the behavior of connectionless network layers, where a source can inject traffic without a previous connection establishment. The two common cases are IP and Ethernet networks. These networks are based on forwarding rules defined in the nodes, with the particular aspect that forwarding decisions typically depend solely on the destination of the traffic. To model this with Net2Plan, all the forwarding rules in a node, for the demands which have the same egress node (e.g. whatever its ingress node is) should be the same.

2.7.1 Routing loops

In both source-routing and hop-by-hop routing looping situations can occur.

In source routing, a route is defined by specifically determining the sequence of links to traverse. Net2Plan allows routes which traverse a node and/or a link more than once, and thus routings with arbitrary loops can be defined, as long as they are of finite length.

The definition of the forwarding rules can also create loops, as occurs in reality. However, the routing loops in hop-by-hop networks have a defining aspect: the traffic can potentially make an infinite number of hops when it enters a routing loop. We distinguish two different situations:

- Closed loops where the traffic entering in them never reaches the destination (e.g. Fig. 2.1). In this case, the traffic only enters the loop and never leaves it, and thus accumulates and saturate the link capacity.
- Open loops where the traffic entering a loop can make a number of cycles, and eventually leave it reaching the destination- Fig. 2.2 shows an example. In this case, a traffic unit to node 4 reaching node 2, could enter the cycle between nodes 2-5 or nodes 2-3-5. If the forwarding decision in each node is randomly taken, the probability of staying in the loop after k hops is positive for any k .

Naturally, looping situations in hop-by-hop networks would reflect a wrong design of the routing tables. Net2Plan is equipped with the appropriate functions to detect them, and to determine the traffic in the links from the offered traffic and forwarding rules also in this cases. Further details on the techniques to do that can be consulted in Chapter 4 of [1].

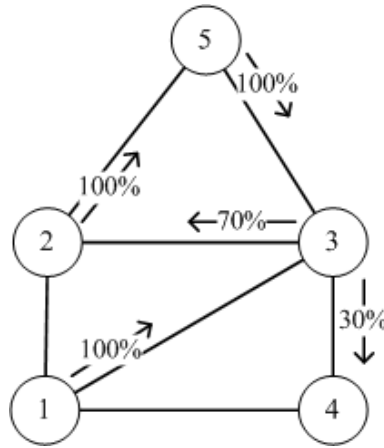


Figure 2.2: Example of an open loop (3-2-5), for a demand from node 1 to node 4.

2.7.2 Source-routing of the traffic - Route object

In the source-routing case, each unicast traffic demand is assigned an arbitrary set of routes, which determine how the demand traffic is carried. Each route is defined by a sequence of links of the same layer where the demand belongs to, that make up a path from the demand ingress node to the demand egress node.

A route can be assigned to only one demand, but a demand can be carried by zero, one or more routes. When the number of routes is two or more, we say that the unicast routing is bifurcated.

A route can be assigned an arbitrary number of *protection segments*. These are fractions of capacity in the links that are reserved, and can be used to reroute some routes in case of failure.

Each route in a network is represented by a **Route** object. Each route is characterized by:

- *Id* (**long**): A unique identifier or serial number assigned by the kernel, that never changes along the life of the *NetPlan* object. Routes created later receive higher numbers, but not necessarily consecutive.
- *Index* (**int**): An identifier (**int**) assigned by the kernel 0,1,2,... to the route inside a given network layer. Route indexes are renumbered when a route in the same layer is removed (e.g. when route with index 0 is removed, all the other routes reduce their index in one).
- *Associated demand* (**Demand**): The demand of which this route is carrying traffic. The ingress and egress nodes of the demand must be the ones of the route.
- *Initial sequence of links* (**List<Link>**): The sequence of links of the route, when the route was created. This must be a sequence of links forming a path from the demand ingress to the demand egress node.
- *Current sequence of links and/or protection segments* (**List<Link>**): The current sequence of links of the route. This may be different to the initial path if some rerouting was performed. In particular, the rerouting can now decide to traverse protection segments, which are represented by **ProtectionSegment** objects, subclasses of **Link** (see Section 2.7.3 for details).
- *Carried traffic* (**double**): The amount of traffic that this route is carrying, measured in the route layer traffic units.

- *Occupied link capacity* (**double**): The amount of capacity in the traversed links that this route occupies, measured in the route layer link capacity units. Typically, the demand traffic and link capacity are measured in the same units (e.g. Gbps), and the route carried traffic equals its occupied link capacity.
- *Usable backup protection segments* (**Set<ProtectionSegment>**): The set of protection segments that could be used by the route to reroute the traffic (e.g. to react to a link failure).
- User-defined name-value attributes, as a **Map<String,String>**.

2.7.3 Source-routing of the traffic - ProtectionSegment object

In source-routing, a protection segment is a sequence of links forming a path (that is, the end of a link is the start of the next link), with a given link capacity reserved in each. The reserved capacity is subtracted from the link capacity that is usable by the routes and multicast trees traversing the links.

Each protection segment in a network is represented by a **ProtectionSegment** object, which is a subclass of **Link** (where the link end nodes correspond to the segment end nodes, and the segment reserved capacity is the link capacity).

The typical use of protection segments, is ease the modeling of network recovery schemes that reserve link capacity in advance, to be used in the case of a network failure. In particular, Net2Plan offers built-in algorithms where a route affected by a failure (a traversed link or node failed), is rerouted using (i) the surviving links in the route, and (ii) the reserved capacity in the associated protection segments. Then, by using these algorithms and an appropriate definition of protection segments, it is possible to fast prototype recovery schemes like:

- In 1+1 *dedicated protection* schemes, each route has a backup path with the same reserved capacity as the route. To model them, we would create a protection segment per route, with the same end nodes (although typically with a link-disjoint path) and a reserved capacity equal to the occupied link capacity.
- In shared protection schemes, a protection segment is associated as the backup of more than one route. For instance, two routes can share a common backup path.
- In link protection schemes, the failure of a link is protected using a pre-established subpath from the link initial to the link end node. This can be modeled defining a protection segment for the path protecting a link, and associate all the routes traversing the link to it.

Recall that since a protection segment is also a link (subclass of **Link**), a **List<Link>** object reflecting the current path of a route, can host an arbitrary sequence of traversed links and/or protection segments to carry the traffic in the case of failure.

Protection segments are characterized by:

- *Id* (**long**): A unique identifier or serial number assigned by the kernel, that never changes along the life of the *NetPlan* object. Protection segments created later receive higher numbers, but not necessarily consecutive.
- *Index* (**int**): An identifier (**int**) assigned by the kernel 0,1,2,... to the protection segment inside a given network layer. Segment indexes are renumbered when a segment in the same layer is removed (e.g. when segment with index 0 is removed, all the other segments reduce their index in one).

- *Sequence of links* (**List<Link>**): The sequence of links (not protection segments) forming a path (a link starts where the previous link ends) that make up the path of this segment. The segment end nodes do not have to be equal to the end nodes of any route it is backing up, but all the links must belong to the same layer, which will be the layer of the protection segment.
- *Associated routes* (**Set<Route>**): The set of routes that this segment can be backup to, all of them of the layer of the protection segment.
- *Link reserved capacity* (**double**): The amount of capacity in the traversed links that is reserved for protection, measured in the layer link capacity units.
- User-defined name-value attributes, as a **Map<String,String>**.

2.8 Shared-risk groups - SharedRiskGroup object

A shared-risk group (SRG) represents a particular risk of failure for the network that, if happens, creates a simultaneous failure in a particular set of links and/or nodes. For instance, a SRG can be associated to the risk of accidentally cutting a particular duct that holds the links between two nodes (e.g. one in each direction). If this cut occurs, the two links would fail simultaneously. Then, they would remain unavailable until a reparation of the damage is completed.

Each SRG in a network is represented by a **SharedRiskGroup** object. Each SRG is characterized by:

- *Id* (**long**): A unique identifier or serial number assigned by the kernel, that never changes along the life of the *NetPlan* object. SRGs created later receive higher numbers, but not necessarily consecutive.
- *Index* (**int**): An identifier (**int**) assigned by the kernel 0,1,2,... to the SRG. SRG indexes are renumbered when a SRG is removed (e.g. when SRG with index 0 is removed, all the other SRGs reduce their index in one).
- *Mean Time To Fail (MTTF)* (**double**): The average time between two consecutive failures.
- *Mean Time To Repair (MTTR)* (**double**): The average time between the moment the failure occurs, until it is repaired.
- *Associated set of nodes* (**Set<Node>**): The nodes that simultaneously fail when the risk associated to the SRG occurs.
- *Associated set of links* (**Set<Link>**): The links that simultaneously fail when the risk associated to the SRG occurs.
- User-defined name-value attributes, as a **Map<String,String>**.

SRGs are used to model the failure risks that threat the network, and eases the design and evaluation of the network recovery mechanisms. As an example, SRG information is used by some built-in online algorithms that create failure and reparation events in the network according to the statistical information in the SRGs, and send these events to the recovery algorithms that must react to them. Also, some built-in Net2Plan reports can estimate analytically the availability of the network for arbitrary network recovery mechanisms (protection or restoration based), from the SRG information.

2.9 Multilayer networks

Communication networks are organized into layers, governed by different protocols and potentially managed by different companies or institutions, such that the links in an upper layer appear as traffic demands carried by the lower layer in an underlying topology. For instance, in IP over WDM optical networks:

- The upper layer is composed of a set of IP routers, connected through optical connections of fixed capacity (e.g. 10 Gbps, 40 Gbps, 100 Gbps) called lightpaths. The IP routers see each lightpath as a direct link or pipe to other router, and the traffic is routed on top of the lightpaths according to the IP nodes routing tables.
- In the upper layer, each lightpath is a demand to carry traversing a path of optical fibers in the underlying topology of optical fibers. Each lightpath is assigned a wavelength, that cannot be changed along its route, unless wavelength conversion devices are available. The optical switching nodes forwarding the lightpaths are called Optical Add/Drop Multiplexers (OADMs).

Thus, in the previous example, a lightpath appears to the IP layer as a direct link between two routers of a fixed capacity, irrespective of the actual route of the lightpath across the fibers. The topology of IP links (each corresponding to a lightpath) is usually referred to as the *virtual topology*, since each link is not backed by an actual wire, but by a lightpath that follows an arbitrary optical path across the fiber topology.

Multiple other examples exist of multilayer networks. For instance, a common three-layer structure is that of IP routers connected through a topology of MPLS virtual circuits, that are routed on top of a topology of lightpaths, that are routed on top of a topology of optical fibers.

2.9.1 Network layers - NetworkLayer object

To be able to represent arbitrary multilayer designs, in Net2Plan, a network is composed of a number of layers, at least one. One out of them is defined as the so-called default layer. In many methods indicating the layer of a link, demand, route etc. is optional, and if not specified, the default layer is assumed. Thanks to this, common users not interested in multilayer designs can work without really knowing the possible complexities of multilayer networks. Also, it is possible to use (without any change) algorithms or reports for single layer networks in a selected layer of a multilayer design, by just setting our layer of interest as the default (e.g. selecting it in the graphical user interface).

According to the Net2Plan multilayer model, the following elements are associated to one and only one network layer:

- Link
- Demand
- Multicast demand
- Route
- Protection segment
- Forwarding rules

That is, the offered traffic, the links, and how the traffic is routed on them can be defined differently for each network layer. Actually, a layer can define the routing in a hop-by-hop form, and other using source-routing, as often occurs in reality.

The following elements are not associated to a particular network layer:

- Nodes: a node can have input/output links at different layers.
- Shared risk groups: Represent a risk of failure that can affect e.g. links at different layers.

Each layer in a network is represented by a **NetworkLayer** object. Each network layer is characterized by:

- *Id* (**long**): A unique identifier or serial number assigned by the kernel, that never changes along the life of the *NetPlan* object. Network layers created later receive higher numbers, but not necessarily consecutive.
- *Index* (**int**): An identifier (**int**) assigned by the kernel 0,1,2,... to the network layer. Layer indexes are renumbered when a layer is removed (e.g. when layer with index 0 is removed, all the other layers reduce their index in one).
- *Name* (**String**): An arbitrary network layer name (e.g. "IP").
- *Description* (**String**): An arbitrary network layer description (e.g. "The IP layer of my network").
- *Traffic units* (**String**): A string defining the units in which the offered and carried traffics in the layer are measured (e.g. "Mbps").
- *Link capacity units* (**String**): A string defining the units in which the capacity of all the links in the layer is measured (e.g. "Mbps"). Typically, traffic and link capacity units are the same. However, in some occasions it may be interesting to have different units (e.g. traffics in Mbps and link capacities in "number of channels", or in "MHz").
- User-defined name-value attributes, as a **Map<String,String>**.

2.10 The default failure model in Net2Plan

In order to evaluate network resilience mechanisms, the network model allows setting *up* or *down* nodes and links. The default behavior of Net2Plan corresponds to a network that makes nothing to adapt to the failures, and just the affected traffic is dropped. Naturally, in the online network design, the user can use other built-in algorithms that react to network failures according to particular network recovery schemes, or implement its own one.

The default reaction of Net2Plan to link and node failures is described below.

- When a link sets its state to *down* (failed), then:
 - If the routing type in the link layer is *source-routing*, all the traversed routes set their carried traffic and occupied capacity in the traversed links to zero. This also affects to the routes using protection segments that traverse the link. Calling the methods **getCarriedTraffic** or **getOccupiedLinkCapacity** for these routes would return a zero. However, calling the methods **getCarriedTrafficInNoFailureState** and **getOccupiedCapacityInNoFailureState** would return the nominal carried traffic and occupied link capacities, the one that would exist if no failed resources were traversed.

- If the routing type in the link layer is *hop-by-hop routing*, all the forwarding rules of the link are set to zero, for all the demands, and the routing for all the network is recomputed.
- When a link sets its state to *up* (is repaired), then:
 - If the routing type in the link layer is *source-routing*, all the traversed routes are checked. If thanks to this repair, now they traverse only up nodes and links, its carried traffic and occupied link capacities get their nominal values (the ones returned by `getCarriedTrafficInNoFailureState` and `getOccupiedCapacityInNoFailureState` methods of the route).
 - If the routing type in the link layer is *hop-by-hop routing*, all the forwarding rules of the link take the nominal values, for all the demands, and the routing in the whole network is recomputed.

The failure and repair of a node is equivalent to simultaneous failure and repair of all the in/out links of the node.

2.10.1 Default failure model in multilayer networks

Recall that in the multilayer representation of Net2Plan, upper layer links are implemented as traffic demands in the lower layer, and the upper layer link capacity is made equal to the coupled demand carried traffic.

A failure in the links of a lower layer, can make the carried traffic of a demand in the same lower layer drop, even become zero. This capacity update is automatically seen by the coupled *upper layer* link. However, the upper layer link *does not become automatically down*. Users developing network recovery algorithms for multilayer networks should take this into account (e.g. if they want to propagate the failure to the upper layer, by setting the upper layer links as down, they have to program this behavior themselves).

Chapter 3

The Net2Plan Graphical User Interface (GUI)

The graphical user interface of Net2Plan is launched by double clicking in the `Net2Plan.jar` file after decompressing the file with the Net2Plan distribution. It is also possible to launch it with the command `java -jar Net2Plan.jar` from a console.

Fig. 3.1 shows the initial user interface, where a welcome message is printed. From it, the user can choose among the following menus:

- **File**, to access some general configuration options. They will be described in Section 3.1.
- **Tools**, gives access to the three main tools within Net2Plan: *offline network design*, *traffic matrix manipulation*, and *online simulator*. They will be described in Section 3.2, 3.3 and 3.4 respectively.
- **Help**, gives access to Net2Plan documentation and the welcome(about) screen. This menu is described in Section 3.5.

3.1 Menu File

This menu has four options: *Options*, *Classpath Editor*, *Show Java console*, and *Exit*.

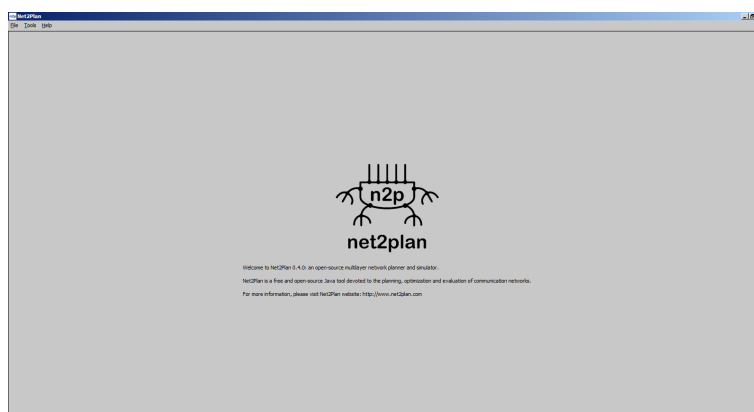
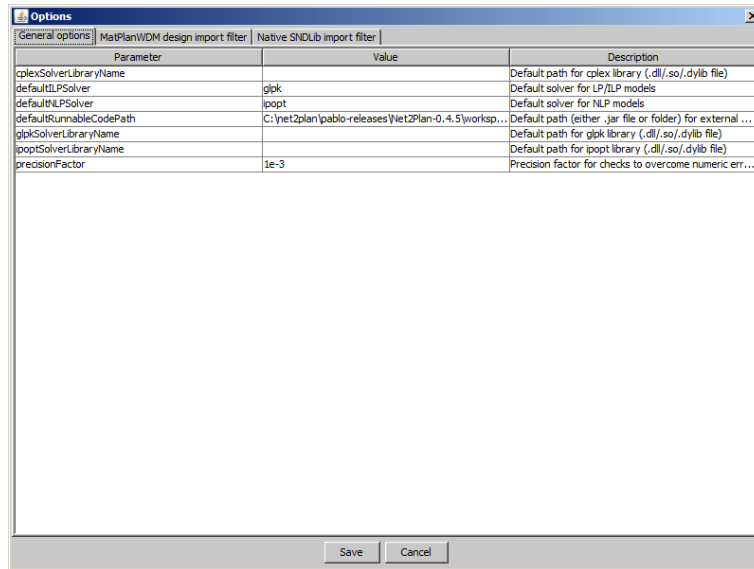


Figure 3.1: Net2Plan welcome screen.

Figure 3.2: Configurable options in the **Options** menu.

3.1.1 File → Options

Use **Options** to set Net2Plan-wide parameters. These options have a global scope to all Net2Plan modules: are used within the kernel, and, for instance, to compute delay metrics in built-in reports. The description of each parameter is included next to its name. Users implementing their own algorithms/reports have read access to these parameters, as a map that links the parameter name and its current value.

In this version the general configurable options (Fig. 3.2) are:

- **precisionFactor**: Precision factor for checks to overcome numeric errors. This parameter allows considering in the kernel small tolerances in the sanity-checks of the network designs. It avoids situations in which numerical inaccuracies (e.g. caused by finite precision of the solvers) would be interpreted as errors. For instance, if an algorithm returns a design where the traffic carried by a link is 10.0000001 and link capacity is 10, the kernel may show a warning. The precision factor applies since the actual check performed has a margin given by the precision factor. Default value of precisionFactor is 10^{-3} , and its value is constrained to be in range (0,1).
- **defaultRunnableCodePath**: Default path that will be used by tools in the GUI as the first option to load external code (e.g. algorithms). It can be either a .jar file or a folder. Default value is the **BuiltInExamples.jar** included within Net2Plan.
- **defaultILPSolver**: Default solver to be used for solving Linear Programs (LP) or Mixed Integer Linear Programs (MILP). Default: *glpk*.
- **defaultNLPsSolver**: Default solver to be used for solving Non-Linear Programs (NLP). Default: *ipopt*.
- **cplexSolverLibraryName**: Default path for cplex library (.dll/.so file). Default: None
- **glpkSolverLibraryName**: Default path for glpk library (.dll/.so file). Default: None
- **ipoptSolverLibraryName**: Default path for ipopt library (.dll/.so file). Default: None

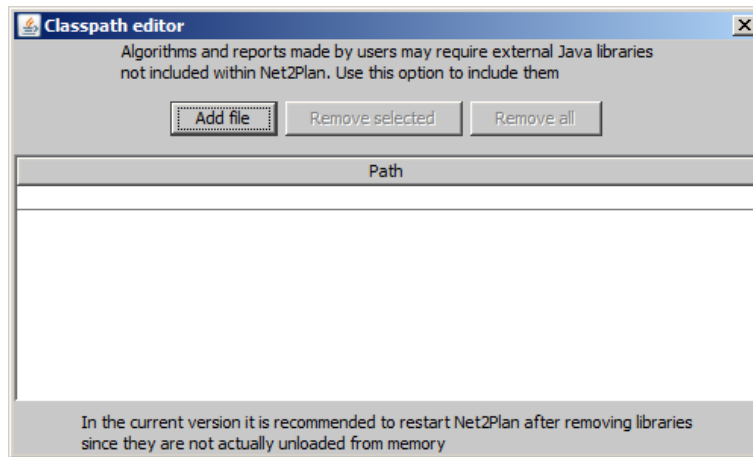


Figure 3.3: Configurable options in the `Options` menu.

Default solvers are used for a few internal operations requiring a solver (e.g. the option of multi-cast tree automatic creation, or some traffic normalization features in the traffic design tool). External algorithms from users, or even built-in examples may have their own solver-related parameters. Regarding the solver library names, they are used if, and only if, an algorithm specifies two solver-related parameters, `solverName` (i.e. `cplex`) and `solverLibraryName` (i.e. `cplex125.dll`), and the solver library name is empty. Otherwise, the non-empty default value for the algorithm will be used by default.

The tabs *MatPlanWDM design import filter* and *Native SNDLib import filter* are specific options of two plugins of Net2Plan, that permit reading network files in the old MatPlanWDM format (MatPlanWDM is a MATLAB-based planning tool developed by the author, now discontinued), and in the SNDLib format (files in the repository of network topologies <http://sndlib.zib.de/>).

3.1.2 File → Classpath editor

Although a moderate library set is provided within Net2Plan, users may require extra Java libraries (`.jar` files) to develop their own algorithms or reports (e.g. mathematical or graph theory libraries). So, the classpath editor (Fig. 3.3) avoids the tedious task of including Java libraries in environment variables (i.e. `CLASSPATH` in Windows).

Important: In the current version of Net2Plan Java libraries can be included in run-time, but unfortunately it is not possible to do the same to remove libraries. In this case, user is forced to restart Net2Plan.

3.1.3 File → Java error console

This feature centralizes the error handling within Net2Plan. When an error is thrown, for example, due to invalid input parameters in an algorithm, the error and the stack trace is shown there. Moreover, `System.out/System.err` is redirected there also, allowing users to debug their Java code. The console can be accessed also using the combination `ALT+F12`.

Important: Due to limitations in Java Virtual Machines, when JNI/JNA for native library access is used, the native output (i.e. `stdout` in C/C++) is not equivalent to the Java output, thus such information will not appear in the Java error console. A workaround is to start the GUI from the command-line.

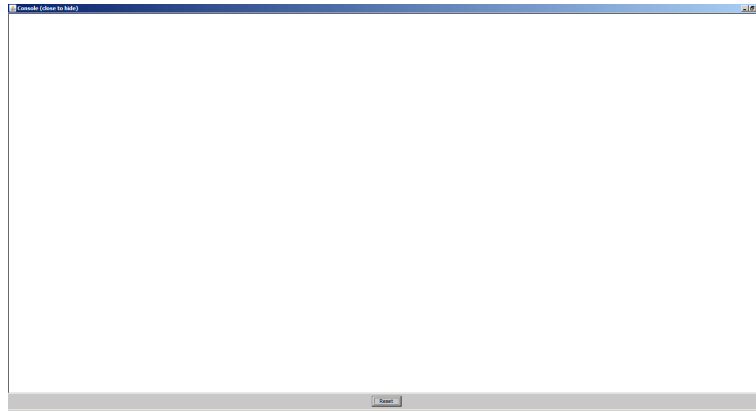


Figure 3.4: Java error console.

3.1.4 File → Java error console

Quits Net2Plan.

3.2 Offline network design

Selecting the submenu **Tools** → **Offline network design** (or ALT+1) opens the offline network design tool. This tool is targeted to create a static or offline network design, internally represented by **NetPlan** object, containing the elements: network nodes, links, unicast and multicast traffic demands, routes, protection segments, multicast trees, network layers and SRGs described in Chapter 2. The word offline here means that all the variables in the network plan are supposed to be static (do not change along time). For instance, offered traffics are assumed to be constants representing the average traffic volumes, although in reality the traffic can fluctuate around this average according to statistical patterns.

The key functionalities of this tool are:

- Create, load, save and/or manually modify the network designs using the GUI.
- Observe several statistics and performances automatically calculated from the network designs.
- Apply algorithms, that receive the current network design and modify it in any form (e.g. routing algorithms that receive a design with nodes, links and offered traffic, and add the routing information to it).
- Apply reports to the current design, which are functions that produce an HTML file from the current design.

Fig. 3.5 shows a workspace of the window. Three main areas exist:

- The *Network topology panel* (top-left area).
- The *Warnings* panel (bottom-left area).
- The input data, execution and reporting panel (right area).

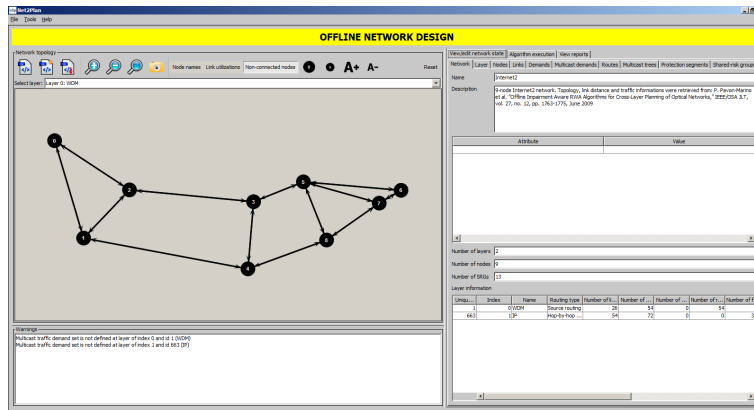


Figure 3.5: Network design window.




3.2.1 Network topology panel







This panel shows graphically the current network design, and permits modifying some parts of it. Users are able to add or remove nodes and links, zoom out and zoom in (and reset zoom), save a screenshot of the currently shown topology (in PNG format), and also show/hide the name of the nodes and link identifiers, as well as show/hide the non-connected nodes to clarify the display. When a multilayer design is loaded, a combobox appears to allow users selecting the current layer, as shown in Fig. 3.5.

Some basic topology manipulations are possible in the topology panel:

- *Add nodes.* Nodes are inserted by right clicking into the canvas and using the option *Add node here*.
- *Remove nodes.* Nodes can be removed by right clicking on them and using the option *Remove node*.
- *Move nodes.* It is possible to move nodes by dragging them, while pressing the **CTRL** key.
- *Add link.* Links are inserted by clicking first in the origin node and then in the destination node. It is possible to insert unidirectional links or bidirectional ones (in this latter case, the user must press **SHIFT** key during that process). Note that in Net2Plan all the links are unidirectional, and with *bidirectional link* we mean the automatic creation of two unidirectional links of opposite directions. Finally, links can be also inserted by right clicking over the origin node and selecting the destination node in the popup menu.

The icons at the *Network Topology panel* permit (left-to-right):

1. Button  loads a network design from a **.n2p** file. Loaded design becomes the current network design, previous design is lost.
2. Button  loads a **.n2p** file, but only extracting the set of offered demands from it (ignoring any other information). The loaded traffic demands replace the set of demands in the current network design, and leaves unchanged the rest of the current network design. Typically, the loaded file was generated using the *Traffic matrix design* functionality. If the number of nodes in the loaded file and the current network design are different, the operation is not completed and an error message is shown.
3. Button  saves current design into a **.n2p** file.

4. Buttons ,  and  make a zoom-in, zoom-out and zoom-all respectively of the shown design.
5. Button  allows to take a snapshot of the canvas and save it to a `.png` file.
6. Button *Node names* toggles between showing or not node name.
7. Button *Link utilizations* toggles between showing or not the link utilizations next to the links (measured as the total traffic divided the total capacity, including if any the traffic and reserved capacity of the link protection segments).
8. Button *Non-connected nodes* toggles between showing or not non-connected nodes (those without input nor output links at the shown layer).
9. Buttons  and  increase/decrease the node sizes respectively.
10. Buttons **A+** and **A-** increase/decrease the font sizes respectively.
11. Button *Reset* erases the current network design, which becomes an empty design.

3.2.2 Warnings panel

In this panel you can see some short messages of warning about the current network design e.g. if node, link or demand sets are defined, if routing is defined or not, if all the offered traffic is routed or there are traffic losses, and so on. No warning messages means that the current network design has nodes, links and offered traffic, that is routed without losses in the network, so that no link is oversubscribed (an oversubscribed link, is the one which is assigned more carried traffic than its capacity).

3.2.3 View/edit network state tab

The **View/edit network state** tab (accessible with **CTRL+1**) shows complete information about the current network design, including some basic statistics and warnings that permit to visually fast-check the design and its performances.

Also, this tab permits completing some simple modifications in the design, like adding/removing any element layers, nodes, links, unicast and multicast demands, routes, protection segments, multicast trees, forwarding rules and SRGs), setting the capacity of links, the offered traffic of demands, or the carried traffic of the routes.

The *View/edit network state* tab is organized into ten or nine sub-tabs, depending on the routing type of the active layer. Each sub-tab corresponds to each of the elements in the Net2Plan network representation: **Network**, **Layer**, **Nodes**, **Links**, **Demands**, **Multicast demands**, **Routes**, **Protection segments**, **Forwarding rules** and **Shared-risk groups**. When layer routing is source routing the **Forwarding rules** sub-tab is hidden. When layer routing is hop-by-hop, the **Routes** and **Protection segments** sub-tabs are hidden. Some general statements applicable to all the sub-tabs:

- In the tables shown, fields coloured in gray are not editable, since they show information calculated from other base fields.
- Right-clicking in the sub-tabs provide fast-access to popup menus with some element-related specific actions.
- Placing the mouse in a column or a field shows a help message with detailed information of its content.

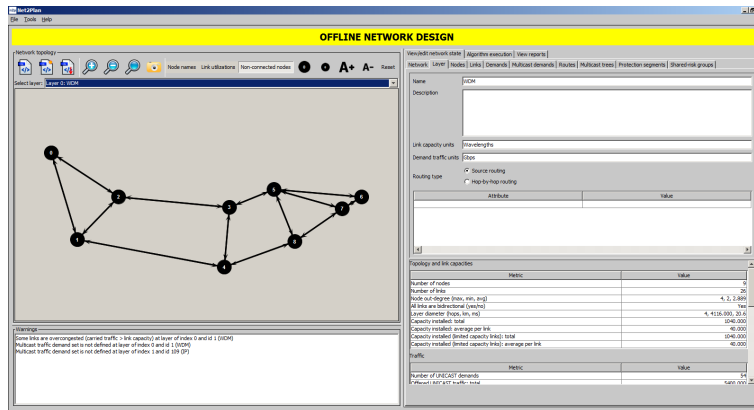


Figure 3.6: Network design window. Layer tab.

- Clicking on a column name reorders the table rows according to it (more clicks toggle between ascending and descending ordering).

A more detailed description of the information in each sub-tab follows.

View/edit network state → Network tab

This tab shows statistical information describing the current network design at a network level (e.g. see Fig. 3.5).

- *Name*, *Description* and *Attributes*: Shows the network name, description message, and set of user-defined key-value parameters associated to the **NetPlan** object in the design. Right-clicking in the **Parameters** panel permits adding/removing/editing attributes.
- **Number of layers**, **Number of nodes**, and **Number of SRGs** are read-only fields displaying this information.
- **Layer information**: This panel shows basic information about all the layers in the network such as name, description, link and demand units, attributes, and number of items for each layer-dependent element (links, demands, and so on). The user can add/remove layers by right clicking on the table and using the corresponding option.

View/edit network state → Layer tab

This tab shows statistical information describing the current network design at a layer level (see Fig. 3.6).

- Upper part of this tab shows and permits editing the layer name, a layer description message, link and capacity units, and the set of user-defined key-value parameters associated to the **NetworkLayer** element in the design. Right-clicking in the **Parameters** panel permits adding/removing/editing attributes.
- *Routing type*. Allows setting the routing type of the layer between source routing and hop-by-hop routing. Note that users may change between them, and the kernel automatically will translate a routing into the other one. Source routing can be always translated into hop-by-hop routing.

However, the other situation is not always possible. If forwarding rules are defined such that traffic gets trapped into an open or close loop, their equivalent routes cannot be found.

- Four tables showing network performance metrics associated to the layer are provided. By placing the mouse over a metric name, a full description is provided. The tables are:
 - *Topology and link capacities* table contain statistics related to the network nodes and links.
 - *Traffic* table provides statistics regarding the unicast and multicast offered and blocked traffic.
 - *Routing* table provides information regarding the routing, like the average number of hops, symmetry, bifurcation, or existence of loops.
 - *Resilience information* table is only active when the routing is of the *source-routing* type. It provides information regarding the protection segments defined.

View/edit network state → Nodes tab

This sub-tab shows the information related to network nodes. Clicking on each node highlights it in the left panel.

It is possible to show/hide a node and setting them as up/down. The up/down option allows users to play with the network to see the effect, in terms of traffic losses, of a failure. We would like to remark, that when setting up/down a node the standard Net2Plan reaction described in Section 2.10 is applied (and not any user-defined recovery mechanism).

View/edit network state → Links tab

This sub-tab shows the information related to network links. Clicking on each link highlights it in the left panel.

Similarly to the nodes, it is possible to show/hide a link and setting it as up/down. The behavior of Net2Plan in both cases is analogous to that with the nodes.

View/edit network state → Demands tab

This sub-tab shows the information related to traffic demands. Clicking on each demand highlights its associated ingress and egress nodes in the left panel, and all the links in the network carrying traffic of the demand.

View/edit network state → Multicast demands tab

This sub-tab shows the information related to multicast traffic demands. Clicking on each demand highlights its associated ingress and set of egress nodes in the left panel, and all the links in the network carrying traffic of the demand.

View/edit network state → Routes tab

This sub-tab appears only if the layer routing is of the source-routing type. It shows the information related to the routes defined. Clicking on each route highlights its traversed links in blue, and the links

belonging to any protection segment which is an eligible backup to the route, are shown in yellow. If the route traverses a protection segment, these links are shown in orange.

View/edit network state → Multicast trees tab

This sub-tab shows the information related to the multicast trees defined. Clicking on each tree highlights its traversed links in blue.

View/edit network state → Protection segments tab

This sub-tab appears only if the layer routing is of the source-routing type. It shows the information related to the protection segments defined. Clicking on each segment highlights its associated links.

View/edit network state → Forwarding rules tab

This sub-tab appears only if the layer routing is of the hop-by-hop type. It shows all the forwarding rules defined in the network layer. Clicking on each forwarding rule highlights the associated demand ingress and egress nodes, and the link.

View/edit network state → Shared-risk groups tab

This sub-tab shows the information related to the SRGs defined. Clicking on each SRG highlights its associated links and/or nodes (the ones that simultaneously fail when the SRG fail).

3.2.4 Algorithm execution tab

This panel (accessible also from **CTRL+2**) permits the users executing network design algorithms that receive as an input (i) the current network design, (ii) a set of algorithm-defined parameters, and (iii) Net2Plan-wide parameters, and produce as an output a new network design that becomes the current one, and an output message string. Fig. 3.7 shows an example.

To execute an algorithm, users should specify the Java class (implementing the `IAlgorithm` interface) containing the network design algorithm. A `.class` file can be selected using the **Load** button. In addition, a `.jar` file can be also selected. In that case, the pull-down menu below permits selecting one among the `.class` files in the `.jar`, that implement the `IAlgorithm` interface.

Once an algorithm is selected, the **Description** text field shows the algorithm description as returned by the `getDescription()` method of the algorithm. The **Parameters** panel shows the set of input parameters of the algorithm. Net2Plan invokes the algorithm `getParameters()` method to obtain the list of input parameters, with a name, a default value and a description message for each. This information is displayed in the **Parameters** panel. Then, the graphical interface allows the user modifying the value of any parameter before running the algorithm.

The algorithm is executed pressing the **Execute** button. At this moment, Net2Plan invokes the `executeAlgorithm()` method of the algorithm, passing as inputs the current network design, the values of the input parameters (as `String` objects, any parsing should be done by the algorithm), and the current values of the Net2Plan-wide parameters (see Section 3.1.1). The `executeAlgorithm()` method returns a `NetPlan` object that becomes the current network design. If the method raises a

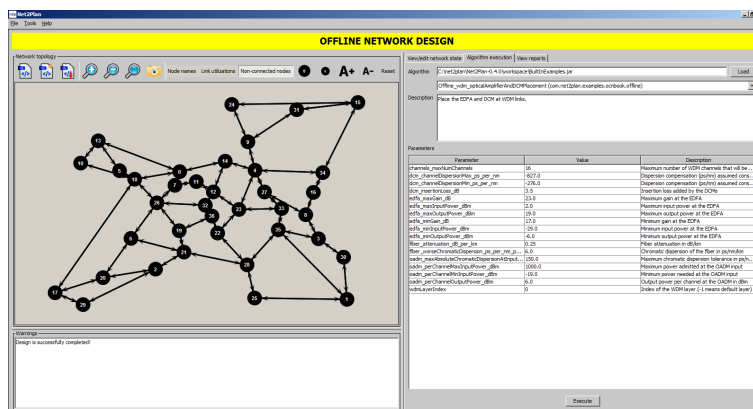


Figure 3.7: *Algorithm execution tab.*

Net2PlanException, it is shown in the window. If the method raises any other **Exception**, the stack trace is printed in the Java console for helping the users to debug their algorithms. In either case, if the `executeAlgorithm()` raises any exception, the current design is unchanged, whatever changes to it were made in the algorithm before the exception was raised.

To see more information about how to develop user-made offline network design algorithms see the Chapter 5.

Important: In multilayer networks, the default layer of the **NetPlan** object passed to the algorithm is the one shown in the network topology panel when the **Execute** button is pressed.

3.2.5 View reports tab

In this panel (accessible from **CTRL+3**) users can select a report to apply to the network plan. The structure is similar to that for executing algorithms.

To run a report, users should specify the Java class (implementing the `IReport` interface) containing the report code. A `.class` file can be selected using the `Load` button. In addition, a `.jar` file can be also selected. In that case, the pull-down menu below permits selecting one among the `.class` files in the `.jar`, that implement the `IReport` interface.

Once a report is selected, the **Description** text field shows the report description as returned by the `getDescription()` method of the report. The **Parameters** panel shows the set of input parameters of the report. Net2Plan invokes the `getParameters()` method which returns the list of input parameters, with a name, a default value, and a description message for each. This information is displayed in the **Parameters** panel. Then, the graphical interface permits the user modifying the value of any parameter before running the report.

The report is executed pressing the **Show** button. At this moment, Net2Plan invokes the report `executeReport()` method, passing as inputs the current network design, the values of the input parameters (as **String** objects, any parsing should be done by the algorithm), and the current values of the Net2Plan-wide parameters (see Section 3.1.1). The `executeReport()` method returns a **String**, which is interpreted as an HTML file, and shown in the tabs in the lower part of the tab.

Users can see a report in a browser using the option `View in navigator`, or even saving it to an external HTML file.

Reports can be closed individually using the CTRL+W combination.

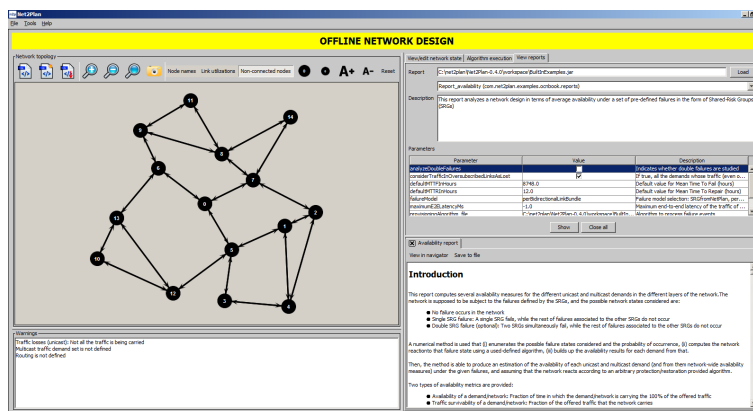


Figure 3.8: *View reports* tab.

3.3 Traffic matrix design

As stated in Chapter 2, network design contains a set of unicast traffic demands, representing the unicast offered traffic in the network. Each demand is characterized by an ingress and egress node, and a traffic volume representing an average of the demand traffic load. In occasions, the set of traffic demands is composed of one demand for each node pair. That is, there are no two traffic demands with the same ingress and egress nodes. In these cases, it is possible to represent the demand set using a compact matrix representation, so-called traffic matrix. A traffic matrix for a network of N nodes is a $N \times N$ matrix with zeros in the diagonal. The coordinate in the i -th row and j -th column of the matrix, contains the amount of traffic generated in node i that is targeted to node j . In other words, the traffic volume of the demand associated with the (i, j) node pair.

The traffic matrix design tool assists users in the process of generating user-defined traffic matrices. It permits generating new matrices manually, or following several models found in the literature (e.g. random-uniform, population-distance models...). Created matrices can be saved in `.n2p` format to be further applied in Net2Plan. In addition, some popular traffic matrix generation models are available in the Net2Plan library and thus can be directly integrated into Java-based design algorithms. For more information, see the class `TrafficMatrixGenerationModels` in the Javadoc.

Selecting **Traffic matrix design** under **Tools** menu (or using ALT+2) activates the Traffic matrix design window. Fig. 3.9 displays the workspace window for this option. The upper part of the left panel gives access to a set of general traffic generation models. Below, the user can generate matrices using one particular method: the population-distance traffic model. The right panel shows the traffic matrices generated, and permits saving (as `.n2p`), loading (as `.n2p`), resizing the matrices and some other simple modifications in the buttons above the matrices. In the lower right side, the traffic normalization panel permits applying a normalization method to one or all of the traffic matrices in the upper panel. The panel below, permits selecting a method for producing a set of traffic matrices from a seminal one, in different forms.

3.3.1 Traffic generation: general traffic models

In this panel, the user can generate one or a batch of traffic matrices, selecting one of the following traffic generation patterns:

- *Constant*. Generates a traffic matrix with a given constant value in all its coordinates.

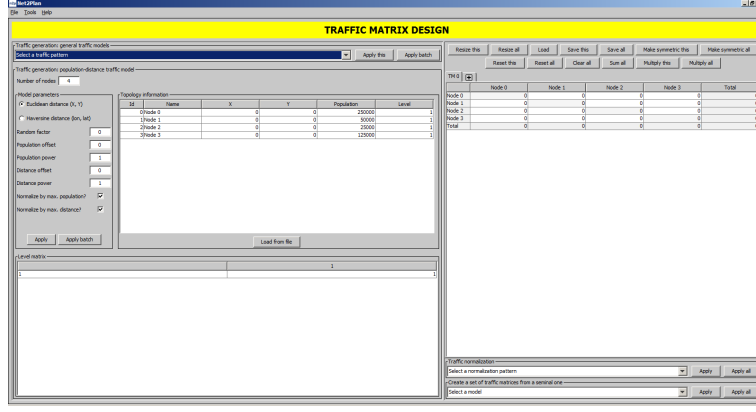


Figure 3.9: Traffic matrix design panel tab.

- *Uniform (0,10)*. Generates a traffic matrix with random values in the range (0,10) .
- *Uniform (0,100)*. Generates a traffic matrix with random values in the range (0,100).
- *50% Uniform (0,10) & 50% Uniform (0,100)*. Generates a traffic matrix with 50% of its entries with random values in the range (0,100), and the rest of the entries with random values in the range (0,10).
- *25% Uniform (0,10) & 75% Uniform (0,100)*. Generates a traffic matrix with 25% of its entries with random values in the range (0,100), and the rest of the entries with random values in the range (0,10) .
- *Gravity model*: Generates a traffic matrix according to the gravity model. The user should provide for each node n , its total traffic generated $OUT(n)$ and received $IN(n)$. Naturally, the sum of all the traffic generated by all the nodes should be equal to the sum of the traffic received by all the nodes (we denote it as H). From this information, the coordinate (i, j) of the traffic matrix is given by $OUT(i)IN(j)H$. Then, the traffic from node i to node j is proportional to the total traffic produced at i and proportional to the total traffic received at j .

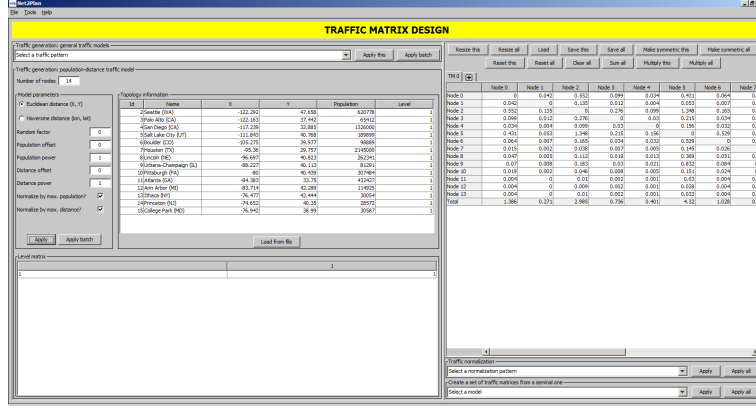
In any method, diagonal values of the traffic matrix are always zero, since self-demands are not allowed. Pressing the **Apply** button, one traffic matrix is created, and shown in the right panel. In its turn, the **Apply batch** button permits creating an arbitrary number of matrices, which are also shown in the right panel.

3.3.2 Traffic generation: population-distance traffic model

In this panel, the user can generate one or a batch of traffic matrices, using the population-distance traffic matrix model described in [2]. This model receives as an input the number of nodes in the network, and for each node, its (X, Y) position, its population and a factor called node level. This information can be introduced manually in the **Topology information** panel, or loaded from a **.n2p** file, where the nodes have the population and level attribute defined.

In the population-distance model, the traffic γ_{ij} from node i to node j is calculated following the expression:

$$\gamma_{ij} = (1 - rf + 2 \times rf \times rand()) \times Level(L_i, L_j) \times \frac{\left(\frac{Pop_i Pop_j}{Pop_{max}^2} + Pop_{off} \right)^{Pop_{power}}}{\left(\frac{dist_{ij}}{dist_{max}} + dist_{off} \right)^{dist_{power}}}$$

Figure 3.10: *Traffic matrix design* panel tab. Population-distance model.

This expression is explained in the following points:

- *Random factor* $(1 - rf + 2 \times rf \times rand())$: $rand()$, is a sample of a uniform (0,1) distribution, and $rf \in [0, 1]$ is a method parameter controlling the randomness, since expression $(1 - rf + 2 \times rf \times rand())$ is a uniform sample in the interval $[1 - rf, 1 + rf]$. Then, if $rf = 0$, there is no randomness (the sample is always one), if $rf = 1$, the randomness is maximum, since the random becomes a uniform sample in the range (0,2).
- *Node level factor* $Level(L_i, L_j)$. This factor permits multiplying the traffic between two nodes by a constant dependent on the level of each node. The $Level(L_i, L_j)$ values are defined in a $L \times L$ matrix (being L the number of levels or node types defined by the user). For instance, imagine we have a network with two types of nodes, *clients* and *servers*. We want to create a traffic matrix for this network where clients do not send traffic to clients, and servers do not send traffic to servers. This can be done defining two node levels in the network for separating *client nodes* and *server nodes*. Then, we can use a level matrix with 0s in the diagonals, so the client-client traffic and server-server traffic is multiplied by 0 in the model.
- *Population factor* $\left(\frac{Pop_i Pop_j}{Pop_{max}^2} + Pop_{off} \right)^{Pop_{power}}$. The population factor makes the traffic between two nodes proportional to the product of the population of both nodes normalized by the maximum node population Pop_{max}^2 . The factor Pop_{off} is used to smooth the effects of the product (e.g. if a population is 0, the traffic is still not zero). The Pop_{power} factor controls the effect of the population in the matrices. Typical values are:
 - $Pop_{power} = 1$ in models based on so-called gravitation attraction.
 - $Pop_{power} = 0$ if site traffic is independent of the population.
- *Distance factor* $\left(\frac{dist_{ij}}{dist_{max}} + dist_{off} \right)^{dist_{power}}$. The distance factor makes the traffic between two nodes inversely proportional to the distance between them. The $dist_{off}$, $dist_{max}$ and $dist_{power}$ values have a similar function as in the population factor. Typical values are $dist_{power} = 2$ or 3.
- Note that the population and/or distance normalization factors can be disabled. In such cases, the factors $dist_{max}$ and pop_{max} are set to one.

Pressing the **Apply** button, one traffic matrix is created, and shown in the right panel. In its turn, the **Apply batch** button permits creating an arbitrary number of matrices, which are also shown in the right panel.

3.3.3 Manual matrix introduction/edition

The right panel shows the traffic matrices generated by any of the previous models, in different tabs numbered as TM0, TM1... The user can manually modify the matrices directly typing the coordinate values. The *Resize this* button permits resizing (changing the number of nodes) of the traffic matrix in the active tab. The *Resize all* button permits resizing all of them in one step. The *Load* button permits loading a traffic matrix from a **.n2p** file. The buttons *Save this* and *Save all* permits saving the traffic matrix in the active tab, or all the traffic matrices in **.n2p** files. The buttons *Make symmetric this* and *Make symmetric all* produce symmetric matrices. The traffic between two nodes becomes the average between the traffic in both directions. The buttons *Reset this* and *Reset all* sets 0s in all the coordinates of active/all matrices. The button *Clear all* eliminates all the matrices. The button *Sum all* adds a new traffic matrix to the panel, which is the sum of all the traffic matrices shown (all of them must have the same number of nodes). The buttons *Multiply this* and *Multiply all* permits multiplying all the coordinates of this/all matrices by a constant factor.

3.3.4 Traffic normalization

The traffic normalization pull-down menu in the lower-right side of the window, permits selecting a normalization method to apply to the traffic matrix in the active tab (button *Apply*) or to all the traffic matrices (button *Apply all*). Four types of normalization methods are implemented: total, row and column normalization [2]:

- *Total normalization.* The target of total normalization, is modifying a traffic matrix M , so that in the normalized matrix M' , the total amount of traffic generated equals a user-defined value S . This is done by multiplying all the elements of the original traffic matrix by a constant factor according to the expression:

$$M'_{ij} = M_{ij} \frac{S}{\sum_{ij} M_{ij}}$$

- *Row normalization.* The target of row normalization, is modifying a traffic matrix M , so that in the normalized matrix M' , the total amount of traffic generated by each node i , equals a user-defined value S_i . This is done by multiplying all the elements in the i -th row of the original traffic matrix by the same constant factor according to the expression:

$$M'_{ij} = M_{ij} \frac{S_i}{\sum_j M_{ij}}$$

- *Column normalization.* The target of column normalization, is modifying a traffic matrix M , so that in the normalized matrix M' , the total amount of traffic received by each node j , equals a user-defined value S_j . This is done by multiplying all the elements in the j -th column of the original traffic matrix by the same constant factor according to the expression:

$$M'_{ij} = M_{ij} \frac{S_j}{\sum_i M_{ij}}$$

- *Normalize to the maximum traffic that can be carried.* The target is to multiply the traffic matrix M by the maximum factor α so that the matrix αM can still be carried by a given network using the optimum routing (without oversubscribing the links). The user can choose between two forms of calculating this: an estimated method, and an exact method:
 - *Estimated method.* This method actually produces a matrix which is an upper bound to the maximum traffic that an optimal routing could carry. For each input/output pair in

the network (i, j) , it calculates the number of hops that the shortest path (either in hops or in km) between those nodes has SP_{ij} . For each node pair (i, j) , the quantity $\alpha M_{ij} SP_{ij}$ is the minimum possible amount of link bandwidth that the traffic between those nodes could consume in any routing. Then, the normalized matrix αM is such that the sum of this quantity along all the coordinates, equals the total amount of bandwidth summing the links in the network $\sum_e u_e$ (where u_e stands for the capacity of link e). In other words:

$$\alpha = \frac{\sum_{ij} M_{ij} SP_{ij}}{\sum_e u_e}$$

For further details, see Exercise 4.2 in [1].

- *Exact method.* This method solves the linear program that exactly computes the maximum factor α for which the matrix αM still has a feasible routing in the network. The formulation is solved using the JOM library. The solver used and its `.DLL/.so` location is obtained from the default values in the Options menu (see Section 3.1.1).

3.3.5 Creating a set of traffic matrices from a seminal one

In some occasions, network design studies require a set of traffic matrices, instead of a single one. For instance, when we need to produce a sequence of traffic matrices that reflect the forecasted traffic in following years, according to a expected traffic growth (this is called multi-period planning). Also, it may be necessary to produce random variations of a single traffic matrix, to check how the network performances vary if the traffic fluctuates. For this or other purposes, Net2Plan offers the following methods:

- *New matrices with a compound annual growth rate.* A sequence of traffic matrices is created, representing one for each of the incoming years, being the seminal matrix the traffic today. Each matrix is equal to the matrix of the previous year, multiplied by a factor $(1 + CAGR)$, where *CAGR* is the *Compound Annual Growth Rate*.
- *Uniform random variations.* A set of matrices is created from a seminal one. For each new matrix, each coordinate is given by a sample of a uniform random variable with average x (the coordinate value in the seminal matrix), and a user-defined maximum relative variation (in the range $[0, 1]$). For instance, a value 0.2 of the maximum relative variation means that the new coordinate is taken uniformly between $(1 - 0.2)x$ and $(1 + 0.2)x$.
- *Gaussian random variations.* A set of matrices is created from a seminal one. For each new matrix, each coordinate is given by a sample of a gaussian random variable, with average x (the coordinate value in the seminal matrix), and a user-defined coefficient of variation (quotient between standard deviation and average), and a user-defined maximum relative variation. This latter value is used to truncate the sample. For instance, a value 0.2 of the maximum relative variation means that if the sample is below $(1 - 0.2)x$ then the value $(1 - 0.2)x$ is produced, and if the sample is greater than $(1 + 0.2)x$, the value $(1 + 0.2)x$ is produced.

In any method, if negative values in the coordinates appear, they are set to 0.

3.4 Online network simulation

In a real-world environment, network conditions vary during its operation, according to different phenomena. Failures in nodes and links, establishment of new virtual circuits, or variation in traffic

volumes are some examples. In this case, users could be interested in analyzing, using an event-driven simulation, how their networks react to those changes and how their designs are consequently adapted for them.

Net2Plan provides a post-analysis simulation tool that allows to the user the (joint) evaluation of the availability performance of protection and restoration algorithms in the network, the performance of on-line provisioning schemes that allocate resources to incoming connections (e.g. virtual circuits requests, lightpath requests, phone calls, multimedia sessions), the performance of dynamic allocation algorithms which react to variations in traffic demand volumes, or in general any allocations during network operation. Allocation is not only referred to modify traffic routing, but also it means that the network topology may change along time (e.g. adding new links, updating the link capacities...) for adapting to the new traffic condition.

3.4.1 The event driven simulation framework

The architecture of the simulator is based on the well-known discrete-event simulation paradigm. The network operation is modeled as a discrete sequence of events in time. Each event occurs at a particular time instant and marks a change of state in the system. Between consecutive events, no change in the network occurs; thus the simulation can directly jump in time from one event to the next.

Event generator and event processor modules

An online simulation is governed by two objects in Net2Plan. For both, the user can develop its own algorithm, or use a built-in one:

- *Event generator.* The event generator is an object implementing the interface `IEventGenerator`. Its typical use is implementing the code that generates the external events that the network is going to react to: e.g. traffic variations, failure/repairs. Event generators can send and receive events, but *cannot* change the current state of the network, represented by a `NetPlan` object.

Example. The built-in class `Online_evGen_generalGenerator`, implements a generator which can produce fast traffic variations mixed or not with slow (multi-hour) traffic variations and/or failure/repair events in the network.

- *Event processor.* The event processor is an object implementing the interface `IEventProcessor`. Its typical use is consuming the events produced by the event generator, implementing the particular form in which the network state will change. Then, when consuming an event, the event processor will receive from the kernel the event to react to, and the current network state as a `NetPlan` object, and should return the network reaction by modifying the given `NetPlan` object, which will become the new network state.

Example. The built-in class `Online_evProc_generalProcessor`, implements a processor which allows the user to choose among some allocation schemes to react to the events created by the `Online_evGen_generalGenerator` module.

The SimEvent object

Both event generator and processor can send events to themselves and to the other module. Their difference is mainly that event generators cannot modify the current network state.

An event is an object implementing the interface **SimEvent**, which at least contains the event time, event priority (for ordering the simultaneous events among them) the destination module (generator or processor), and any **Object** containing specific information of the event.

Some built-in classes extending **SimEvent** exist, that are used by the built-in algorithms provided as examples in the Net2Plan repository. These are basic events like events to add/remove traffic demands, or to signal a change in the up/down state (fail/repair) of nodes and links.

The user can implement their own **SimEvent** classes, or use these ones. The full list of classes implemented can be seen in the Javadoc, as public classes inside **SimEvent**.

The simulation cycle

The Net2Plan kernel is in charge of governing the simulation. The complete process is described below:

1. The user loads the event generator and event processor modules. The kernel prints the description of each module and their lists of input parameters by calling the methods **getDescription** and **getParameters**.
2. The user can set the values of the input parameters in the GUI. Also, can set some simulation wide parameters (not algorithm dependent) like the simulation duration, transitory time or how some internal kernel statistics should be computed.
3. The user starts the simulation pressing the *Run* button. The **NetPlan** object corresponding to the network design in the topology panel becomes the current network state. The kernel calls the **initialize** method of the event generator, and the event processor. It passes them the current network state (the event generator cannot modify it), and the parameters. Any initialization routines that these modules need should be coded there. Also, at least one of these methods should produce events using the **scheduleEvent** method. If not, there will be no events to consume, and the simulation will end.
4. The Net2Plan kernel handles the simulation loop. It keeps a list of future events to consume (FEL, *Future Event List*), ordered according to the global simulation clock (which initially started in zero), and repeats the *event scheduling loop*:
 - Advance the clock to the time of the first event in the list (the one with the lowest time, using the event priority to order among simultaneous events).
 - Call the **processEvent** method of the event destination module (generator or processor), passing to it the event to consume, and the current network state. Both event generator or processor can produce new events (only with a time equal or higher than the current simulation time). However, only the event processor can call the methods in the **NetPlan** object of the current state that modify it. A **UnsupportedOperationException** is raised if they are called in the event generator.
5. In the event scheduling loop, when the clock reaches the time configured as transitory time in the simulation-wide parameters, the kernel calls the method **finishTransitory** of both event generator and event processor. Typically, these methods reset any internal variables for computing statistics that they may have. Also the kernel computed statistics are reset. The event generator or event processor can also force the kernel to call the **finishTransitory** methods and reset the kernel statistics, by invoking the method **endTransitory**.
6. In the event scheduling loop, when the clock reaches the time configured as simulation end time in the simulation-wide parameters, the kernel calls the method **finish** of both event generator and event processor. These methods return a **String**, which is later printed by Net2Plan in

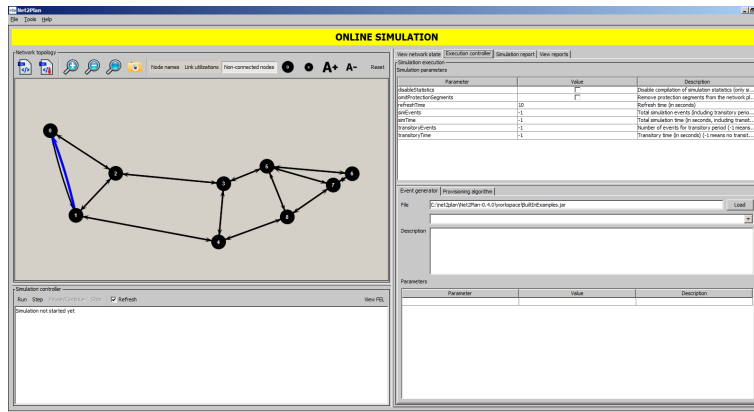


Figure 3.11: *Online simulation* tool. Execution controller panel.

the simulation report. Then, **finish** methods typically return a short report with any internal statistics computed. The event generator or event processor can also force the kernel to end the simulation and call the **finish** methods, by invoking the method **endSimulation**.

Simulation statistics

Two types of statistics are collected during the simulation, and then printed in the final report:

- Net2Plan general statistics. If the **disableStatistics** option is not set, Net2Plan computes a complete set of statistic on the simulation evolution. They are technology-agnostic statistics like the blocked traffic (observing the demands offered and carried traffics), average link occupations, etc.
- Event generator or event processor internal statistics. Typically, these modules can collect and then return in the **finish** method algorithm-specific statistics.

3.4.2 Graphical User Interface

Selecting *Online simulation* (also from ALT+3) opens the corresponding window of the online simulation tool. The workspace of the window is divided into three areas, in a similar way to the network design mode: network topology visualization (top-left area), execution and reporting (right area), and simulation controller and information area (bottom-left area). Fig. 3.11 shows an example of this screen. Next subsections describe the main panels.

Network topology panel

The **Network Topology** panel is the same as in the offline network design tool. The only difference is that the network cannot be edited (e.g. add/remove node). To do so, the *Offline Network Design* tool must be used.

Typically, to perform a simulation, a network design is loaded and becomes the initial network state. During the simulation, changes in the topology (e.g. failed nodes and/or links) are shown in the canvas.

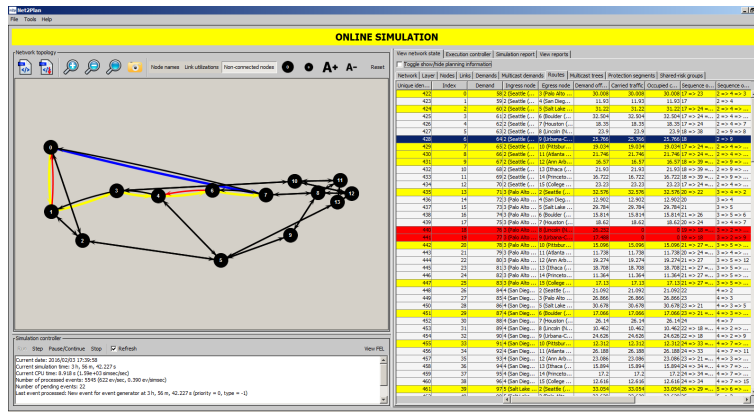


Figure 3.12: Online network design.

View network state tab

In the *View network state* tab (accessible also from CTRL+1), users can see information related to the network state. Optionally, using the checkbox *Toggle show/hide planning information*, users can also see the same information as it was in the *initial* network design before running the simulation, for comparisons.

The following code color is used to highlight the network state regarding to the routes (see Fig. 3.12):

- In the route table, the routes which do not carry traffic (could not be recovered) are printed in red. The ones that follow a different sequence of links than at the start of the simulation (and it is likely that were rerouted to survive a failure), are printed in yellow.
- In the canvas, the failing links are in red. The currently traversed links of a route are printed in blue, or in orange if the traversed links belong to a protection segment. Finally, the links belonging to protection segments available to the route, but not currently traversed by it are printed in yellow.
- If the initial information is used as comparison, the traversed links and usable protection segments of the route are printed like in the current network state, but using dashed lines.

Execution controller

With this panel, users are able to execute simulations and view the current state of the network. In the **Execution controller** (accessible also from CTRL+2), users can load network designs and execute simulations. To execute a simulation the user should specify the following parameters:

- Simulation parameters: general parameters for the simulation.
 - **disableStatistics**. In some occasions, users might be interested in collecting only their own statistics, and would prefer avoiding the overhead that requires statistics collection by the kernel. This can be done with this parameter.
 - **omitProtectionSegments**. As stated in Section 2.7.3, protection segments reserve a certain amount of bandwidth to provide (partial) backup-paths for primary routes. However, in some situations, users might be interested in executing their simulations assuming that no

protection segments are defined. If this parameter is set to true, then protection segments are removed from the network plan just before the simulation starts, and their bandwidth in the links is available for carrying common routed traffic.

- **refreshTime**. If the option *Refresh* has been activated in the simulation controller panel, information about the current simulation, (number of processed events, simulation and CPU time...) will be shown there. This information is refreshed every number of seconds given by this parameter
- **simEvents**. Total number of events (including transitory events) to be simulated. If the parameter **simTime** is also specified, the simulation will automatically finish when the first stopping condition is met. Allowed values are integers greater than zero, or -1 for no limit (simulation must be manually stopped, or stopped by the algorithm calling the **endSimulation** method).
- **simTime**. Total simulation time (including transitory events) in seconds. If the parameter **simEvents** is also specified, the simulation will automatically finish when the first stopping condition is met. Allowed values are numbers greater than zero, or -1 for no limit (simulation must be manually stopped).
- **transitoryEvents**. Number of events for the transitory period. If **transitoryTime** is also specified, the transitory period will finish when the first condition is met. Allowed values are integers greater than 0, or -1 for no transitory period.
- **transitoryTime**. Transitory time in seconds. If **transitoryEvents** is also specified, the transitory period will finish when the condition is met. Allowed values are integers greater than 0, or -1 for no transitory period.
- **Event generator**. the event generator algorithm, as a Java class extending the **IEventGenerator** class. A description of the algorithm will be shown in this panel as well as any input parameter that may be modified by the user.
- **Provisioning algorithm**. The event processor algorithm, as a Java class extending the **IEventProcessor** class. A description of the algorithm will be shown in this panel as well as any input parameter that may be modified by the user.

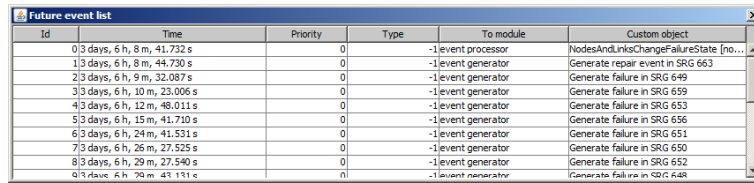
During the simulation, users are able to save the **NetPlan** object collecting the current network state using the corresponding button in the network topology view.

Important: Once the simulation is started, none of the above options can be changed. Users should stop and reset the simulation (clicking the **Reset** button) to perform changes.

Simulation controller

In this panel users can control the execution and flow of the simulation. Several buttons can be manipulated here:

- **Run**. This button will start the simulation. Information about the current simulation will be shown in the text panel (the refresh time is defined by the **refreshTim** parameter). If a previous simulation has already been started it must be stopped and reset before starting a new one.
- **Step**. Instead of running continuously, the simulation can be advanced in steps. Each time the **Step** button is clicked the simulation will consume one event of the future event list, and then pause and refresh the text panel.
- **Pause/Continue**. The simulation will be paused or resumed each time this button is clicked. A simulation must be started in order for this button to work.



Id	Time	Priority	Type	To module	Custom object
0	3 days, 6 h, 8 m, 41.732 s	0	-1 event processor	NodesAndLinksChangeFailureState	NodesAndLinksChangeFailureState [no...]
1	3 days, 6 h, 8 m, 44.730 s	0	-1 event generator	Generate repair event in SRG 663	
2	3 days, 6 h, 9 m, 32.087 s	0	-1 event generator	Generate failure in SRG 649	
3	3 days, 6 h, 10 m, 23.006 s	0	-1 event generator	Generate failure in SRG 659	
4	3 days, 6 h, 12 m, 48.011 s	0	-1 event generator	Generate failure in SRG 653	
5	3 days, 6 h, 15 m, 41.710 s	0	-1 event generator	Generate failure in SRG 656	
6	3 days, 6 h, 24 m, 41.531 s	0	-1 event generator	Generate failure in SRG 651	
7	3 days, 6 h, 26 m, 27.525 s	0	-1 event generator	Generate failure in SRG 650	
8	3 days, 6 h, 29 m, 27.540 s	0	-1 event generator	Generate failure in SRG 652	
9	3 days, 6 h, 30 m, 45.151 s	0	-1 event generator	Generate failure in SRG 648	

Figure 3.13: Online network design. Future event list

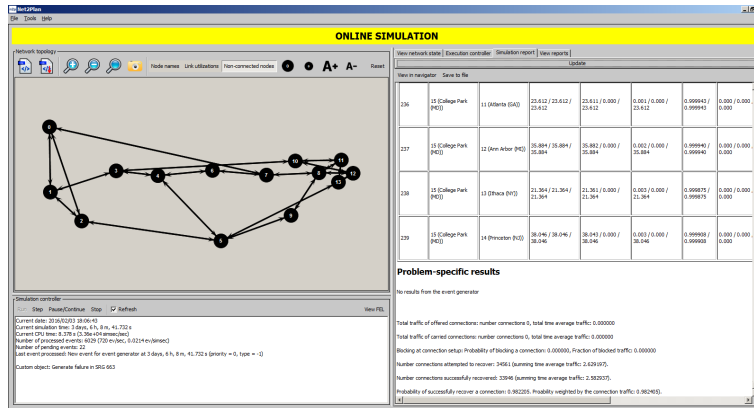


Figure 3.14: Online network design. Simulation report

- **Stop.** Users can stop the simulation at any time, but then the simulation cannot continue.
- **Refresh.** The information in the text panel will be refreshed if this check-box is activated.
- **View FEL:** Clicking this option will show the *Future Event List*, where users can examine the future events to be processed by the provisioning algorithm (see Fig. 3.13)

Also, the text panel shows some brief simulation state information: simulation time, CPU time, last event processed... This information is updated according to the refresh time parameter.

Simulation report

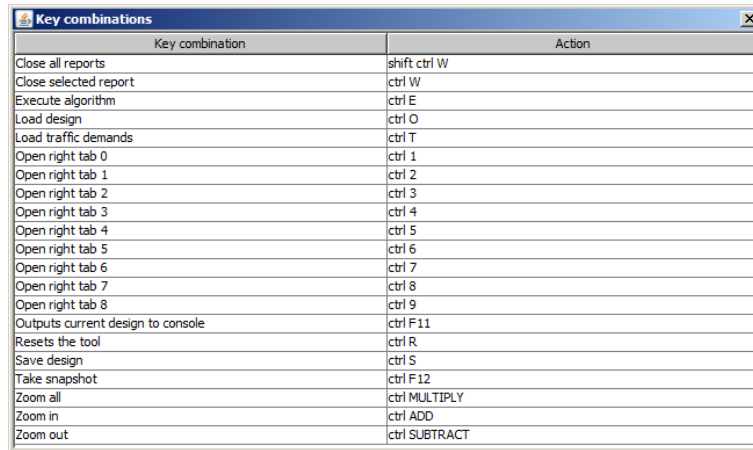
In this panel (accessible also from **CTRL+3**), users can obtain the statistics collected by the kernel, and also those collected by their algorithms (event generator and event processor). Users can see the report at any moment (clicking the **Update** button) while the simulation is running, paused or stopped, and can be opened by a web browser, as happened with the reports for network designs.

In some occasions, users might be interested in collecting only their own statistics, and they might want to eliminate the overhead that requires statistics collection. This can be done by checking the simulation parameter `disableStatistics`. In this case, only algorithm-specific statistics are shown.

Net2Plan statistics include different network-wide, per-layer, per-node, per-link and per-demand information. Fig. 3.14 shows an example.

View reports

This panel (accessible also from **CTRL+4**) offers the same functionality as the reporting tool in the offline network design tool. Users can select a report to apply to the *current* network state. Upon selection



Key combination	Action
Close all reports	shift ctrl W
Close selected report	ctrl W
Execute algorithm	ctrl E
Load design	ctrl O
Load traffic demands	ctrl T
Open right tab 0	ctrl 1
Open right tab 1	ctrl 2
Open right tab 2	ctrl 3
Open right tab 3	ctrl 4
Open right tab 4	ctrl 5
Open right tab 5	ctrl 6
Open right tab 6	ctrl 7
Open right tab 7	ctrl 8
Open right tab 8	ctrl 9
Outputs current design to console	ctrl F11
Resets the tool	ctrl R
Save design	ctrl S
Take snapshot	ctrl F12
Zoom all	ctrl MULTIPLY
Zoom in	ctrl ADD
Zoom out	ctrl SUBTRACT

Figure 3.15: Key combinations for the offline network design tool.

and clicking **Show** the report is shown in the bottom panel. Again, reports can be opened by a web browser using the option **View in navigator**.

3.5 Help menu

This menu has the following options:

- **About**. Gives access to the same welcome screen as the one shown in Fig. 3.1.
- **User's guide**. Shows the local copy of this document. It can be accessed also using **F1**.
- **Library API Javadoc**. Shows the local copy of the library API Javadoc.
- **Examples in website**. Shows the examples available in the website (requires an Internet connection).
- **Key combinations**. Shows the key combinations or shortcuts of the active tool (if any). For example, the key combinations for the offline network design tool are shown in Fig. 3.15. Key combinations can also be accessed with **ALT+K**

Chapter 4

The Net2Plan Command-Line Interface (CLI)

Contrary to the GUI, the command-line interface allows users to make use of batch processing or large-scale simulation features, thus it is specifically devoted to in-depth research studies.

All features available for GUI mode are also available here. The execution is controlled via command-line arguments. To run Net2Plan in CLI mode, users must execute the following command in a terminal:

```
java -jar Net2Plan-cli.jar [more options]
```

Help information can be obtained through the help argument in the following ways:

- To obtain a brief information, including only the execution modes, users should execute the CLI without arguments: `java -jar Net2Plan-cli.jar`
- To obtain the complete information, including individual help for every execution mode, users should type: `java -jar Net2Plan-cli.jar -help`.
- To obtain the information about a certain execution mode, users should type: `java -jar Net2Plan-cli.jar -help mode-name`

Next, the command to execute every mode is shown:

- *Network design.* `java -jar Net2Plan-cli.jar -mode net-design [more options]`
- *Traffic matrix design.* `java -jar Net2Plan-cli.jar -mode traffic-design [more options]`
- *Online simulation.* `java -jar Net2Plan-cli.jar -mode online-sim [more options]`
- *Reporting.* `java -jar Net2Plan-cli.jar -mode report [more options]`

4.1 Examples

We show here some examples:

- To execute the `MyAlgorithm` algorithm in the Jar file `myFolder\myJar` for the NSFNet network and its reference traffic matrix, setting the parameter `myParam` to the value 3:

```
java -jar Net2Plan-cli.jar --mode net-design
--input-file workspace\data\networkTopologies\NSFNet_N14_E42.n2p
--traffic-file workspace\data\trafficMatrices\NSFNet.n2p
--output-file workspace\data\NSFNet.n2p
--class-file myFolder\mJar.jar
--class-name MyAlgorithm
--alg-param myParam=3
```

- To execute the built-in availability report over the previous design:

```
java -jar Net2Plan-cli.jar --mode report
--input-file workspace\data\NSFNet.n2p
--output-file workspace\data\report.html
--class-file workspace\BuiltInExamples.jar
--class-name Report_availability
```

- To generate a series of 4 10×10 traffic matrices using a (0, 100) random uniform model:

```
java -jar Net2Plan-cli.jar --mode traffic-design
--num-nodes 10
--traffic-pattern uniform-random-100
--output-file workspace\data\trafficMatrices\tm10nodes.n2p
--num-matrices 4
```

- To execute a simulation using `myEventGenerator` and `myEventProcessor` classes in `myFolder\myJar` file:

```
java -jar Net2Plan-cli.jar --mode online-sim
--input-file workspace\data\networkTopologies\NSFNet_N14_E42_complete.n2p
--output-file workspace\data\simReport.html
--generator-class-file myFolder.myJar.jar
--processor-class-file myFolder.myJar.jar
--generator-class-name myEventGenerator
--processor-class-name myEventProcessor
--sim-param simEvents=100000
--sim-param transitoryEvents=10000
```

Important: To avoid excessive verbosity in the CLI, package names for Java classes can be omitted. Note that in case of `.jar` files the first class matching the class name will be selected, if multiple cases in different packages share the same class name.

Important: Restrictions to the path of `.class` files must follow the guidelines in Chapter 5.

Chapter 5

Development of algorithms and reports in Net2Plan

One of the most important features of Net2Plan is that it allows users to execute their own code (algorithms, reports... in general we refer to them as runnable code). Here, we briefly describe how to integrate users' code into Net2Plan.

Runnable code is implemented as Java classes, using single `.class` files or integrated into `.jar` files, with a given signature:

- Algorithms for offline network design should implement the interface `com.net2plan.interfaces.networkDesign.IAlgorithm`
- Reports should implement the interface `com.net2plan.interfaces.networkDesign.IReport`
- Online algorithms that process network events, used in the online simulation tool and some reports, should implement the interface `com.net2plan.interfaces.simulation.IEventProcessor`
- Modules that generate events to be consumed by online algorithms should implement the interface `com.net2plan.interfaces.simulation.IEventGenerator`.

A complete information of each interface can be found in the Library API Javadoc. Integration of runnable code simply requires saving it into any directory of the computer, although it is a good practice to store them in the workspace directory of Net2Plan.

In addition, in order to improve the user experience, the kernel is able to catch any exception thrown by runnable code, and print exception messages in the Java console. Recall that any information printed to the Java console by any runnable code (e.g., exception messages, and also any messages printed on purpose into `System.out`), can be seen in the Net2Plan Java console (see Section 3.1.3). This is a valuable resource for debugging the algorithms ran in Net2Plan. In particular, messages from exceptions include a full trace of the error (files, line number of the exception...).

When the runnable code wants to stop its execution raising an exception that needs to be informed to the user in a more clear form (not through the Java console), we recommend to throw the `Net2PlanException` class (see Library API Javadoc for more information). The message associated to this exception is printed in a pop-up dialog instead of the Java console, and thus is more visible to the user. For instance, let us assume an algorithm that receives an input parameter from the user, that should be positive. A good programming practice is starting the algorithm checking if the received parameters are within their valid ranges. If a negative number is received (or something that is not a

number), it is better to raise a `Net2PlanException` that shows the information message in a pop-up in Net2Plan, than a general Java `RuntimeException` whose message can be read only if the user checks the Java console.

Important: When runnable code is implemented as a Java `.class` files, the full path to the class must follow the package name of the class, in order to successfully load the code. For example, if we create an algorithm named `TestAlgorithm` in the package `test.myAlgorithms`, the full path to the class must be like `...any.../test/myAlgorithms/TestAlgorithm.class`. For `.jar` files there is not any restriction on the full path, they can be stored at any folder.

Important: Net2Plan allows to make *online* changes in the runnable code, that is, users can modify their runnable code, recompile and reexecute it (just clicking the **Execute** button at the graphical interface) without the need to restart Net2Plan.

5.1 Net2Plan Library, Built-in Examples and Code Repository

Net2Plan assists the task of creating and evaluating algorithms by providing built-in example algorithms and a set of libraries (e.g., k -loopless shortest paths, candidate path list creation for unicast and multicast traffic...). An exhaustive list of built-in algorithms and the Library API Javadoc can be found in Net2Plan repository and in the Javadoc. Net2Plan web site is expected to become a valuable repository for network planning algorithms. The algorithms in the repository will be open for validation and verification, improving the trustworthiness of planning results.

5.2 JOM: Java Optimization Modeler

Often, some network design problems are solved by modeling them as optimization problems (i.e. integer linear problems, linear problems, convex problems, ...), and then calling an optimization solver to obtain its numerical solution. In this context, optimization modeling tools are targeted to ease the definition of the problem decision variables, constraints and objective function, and become an interface with the (usually complex) solver libraries. AMPL and GAMS are examples of commercial modeling tools. JOM (Java Optimization Modeler) is an open-source Java library developed by Prof. Pablo Pavón Mariño, which can interface with a number of solvers using a vectorial MATLAB-like syntax, which e.g. permits the addition of sets of constraints in one line of code. Current JOM version can interface with GPLK (free) and CPLEX (commercial) solvers for mixed integer linear problems, and IPOPT (free) for non-linear differentiable problems. JOM directly interfaces with compiled solver libraries (.DLLs in Windows and .SOs in Linux), via Java Native Access (JNA). JOM is independent from Net2Plan and can be used for any type of optimization problem. However, Net2Plan uses JOM in all the network design algorithm examples based on solving formulations that are included in the Net2Plan distribution.

5.3 Preparing a Java IDE for Net2Plan programming

For users interested in integrating their own algorithms to Net2Plan, it is required to prepare the Java IDE to program the runnable code. Essentially, users just have to configure their preferred Java IDE to use Java 7 (or later), and to include the libraries in the `lib` subfolder of the Net2Plan folder in the Java build-path. In Eclipse, the latter can be done in the option:

```
Project => Properties => Java Build Path => Libraries => Add External JARs...
```

In Netbeans, the option can be found in:

`Run => Set Project Configuration => Customize => Libraries => Add JAR/Folder...`

Additionally, Javadoc and sources can be attached using the corresponding options in the IDEs. Once the Java IDE is configured, users can start programming their own Net2Plan code.

Chapter 6

Technology-specific libraries

As stated in previous sections, the `NetPlan` object contains only the base minimum member elements corresponding to a *technology-agnostic* view of the network. For instance, for the links it permits setting the length in km, the propagation speed of the signal or its capacity, which are concepts applicable to any network technology. However, if the link e.g. corresponds to an IP link in a network with a routing controlled by the OSPF protocol, we would be interested in storing in the design the link weight to apply in shortest path computations. This is an example of technology-specific information. For this, `Net2Plan` permits the user to add/remove/edit a `Map<String,String>` of key-value attributes per link, node, demand, etc. For instance, this link weight information could be stored in an attribute with key `ospf-linkWeight`, and value the associated weight converted to `String` (actually this is the name chosen by the `IPUtils` library in `Net2Plan`).

Current `Net2Plan` version, provides libraries with specific routines used in the following technologies:

- *IP networks*: The `IPUtils` class incorporates some routines of interest in IP networks. Among them, the computation of the traffic according to OSPF/ECMP weights. Please see the Javadoc of `IPUtils` class for the conventions used in `Net2Plan` when introducing IP specific information in the network model. Also, some built-in algorithms are provided for IP networks, which can be consulted in the `Net2Plan` code repository under keywords *IP* and *OSPF*.
- *Optical WDM networks*: The `WDMUtils` class incorporates some routines of interest in optical WDM networks. Among them, the routines to handle the occupation of the wavelengths, and simple algorithms for routing and wavelength assignment. Please see the Javadoc of `WDMUtils` class for the conventions used in `Net2Plan` when introducing WDM specific information in the network model. Also, some built-in algorithms are provided for WDM networks, which can be consulted in the `Net2Plan` code repository under keyword *WDM*.
- *Wireless networks*: The `WirelessUtils` class incorporates some routines of interest in wireless networks. Among them, the routines to compute coverage and interference matrices, according to some models of Chapter 5, and algorithms in Part II of [1]. Please see the Javadoc of `WirelessUtils` class for the conventions used in `Net2Plan` when introducing wireless specific information in the network model. Also, some built-in algorithms are provided for wireless networks, which can be consulted in the `Net2Plan` code repository under keyword *wireless*.

Remember: All key-value pairs in the attribute maps are stored as `String` values. Users are responsible to make the proper conversions. For example, you can store an `int` array as a succession of numbers separated by spaces.

References

- [1] Pablo Pavón Mariño, *Optimization of computer networks. Modeling and algorithms. A hands-on approach*. Wiley, May 2016. *Accompanying website*: <http://www.net2plan.com/ocn-book>.
- [2] Robert S. Cahn, *Wide area network design: concepts and tools for optimization*, Morgan Kaufmann Publishers Inc., 1998.