Les Cahiers du GERAD ISSN: 0711–2440

JNetMan – An open-source platform for easy implementation and testing of network management approaches

A. Capone, C. Cascone,
L.G. Gianoli, B. Sansò

G–2013–99

December 2013

JNetMan
An open-source platform for easy implementation and testing of network management approaches

Antonio Capone

Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano, Italy

capone@elet.polimi.it

Carmelo Cascone

Luca G. Gianoli

Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano, Italy
and GERAD & Department of Electrical Engineering
Polytechnique Montréal, Canada

carmelo.cascone@polimi.it

gianoli@elet.polimi.it

Brunilde Sansó

GERAD & Department of Electrical Engineering
Polytechnique Montréal, Canada

brunilde.sanso@polymtl.ca

December 2013

Les Cahiers du GERAD

G–2013–99

Abstract: In this paper we present JNetMan, a new Java framework for rapid development of network management solutions based on SNMP (Simple Network Management Protocol). Developed as an open-source project, JNetMan aims at easing network management by offering a set of powerful primitives to directly interact with SNMP-capable devices using a high-level abstraction of the network topology and a set of high-level management libraries (e.g. to obtain the current bit-rate of a link, modify the parameters of the routing protocol, etc.). The result is a reusable and highly configurable software platform that can be used by users to easily develop specific applications for the management of telecommunications networks operated with legacy management protocols, thus allowing fast experimental validation of new solutions. Moreover, thanks to its highly modular structure, JNetMan can be further developed and extended by adding new modules for the management of networks aspects which are not still supported. The implementation of a management application to optimize the network energy consumption is briefly illustrated to clearly point out the potentialities of the platform.

Key Words: Network management; SNMP; Open-source.

1 Introduction

Network management represents for network operators a fundamental mean to efficiently configure the network infrastructure and guarantee its regular functioning. In addition to basic configuration operations, most popular network management practices include resource monitoring, routing optimization, namely *Traffic Engineering* (TE) and fault management [1]. Simple Network Management Protocol (SNMP), thanks to a dedicated secure communication protocol and a hierarchical database structure called *Management Information Base* (MIB) implemented by each SNMP capable device, allows to dynamically modify and retrieve configuration parameters of the network devices.

The growing complexity of the network environments and the rapid spread of bandwidth-hungry applications with very strict Quality of Service (QoS) requirements, push to focus on network management practices to optimize network performance. As result, in the last decade, a very large literature on network management strategies for IP networks operating with different working configurations has appeared (see [2] for a survey). However, while large industrial players typically have both real network test-beds and proprietary or third-part network managements framework (e.g. [3]) to implement and test the developed management solutions, others with fewer means can just adopt a simulation approach which may oversimplify the studied system and hide its real behavior.

To overcome this crucial gap between design and implementation, we present a new open-source Java framework called JNetMan [4, 5], developed as a flexible and configurable tool for rapid deployment and implementation of network management solutions based on SNMP. JNetMan offers a high-level abstraction of the network topology and provides a set of powerful primitives to easily implement typical management operations such as retrieving the current link loads, modifying the IP-address or adjusting the routing protocol parameters. The primitives can be combined in complete freedom by the users to implement the desired management applications and later conduct realistic experimentations to rigorously validate their applicability. It is worth pointing out that thanks to the highly modular structure and the open-source nature of the project, new primitives and modules can be naturally added to implement more sophisticated operations supported by SNMP. Moreover, the programmability and flexibility of JNetMan make it particularly suitable for the integration into Software Defined Networking (SDN) architectures (such as [6]) as evolved plug-in of the Controller southbound interface for advanced orchestration of SNMP functionalities.

The remainder of the paper is organized as follows. In Section 2 we review some state-of-the-art network management proposals and point out how they could have been implemented by using a management framework. In Section 3 we present and discuss general principles and guidelines for the development of general management framework, while in Section 4 we present the detailed features of our new management framework, namely JNetMan. Then, in Section 5 we discuss a practical use of the framework to implement a complex energy-aware network management solution. Finally, we report some concluding remarks in Section 6.

2 Motivations

A large portion of network management operations aims at optimizing the network routing to maximize a certain measure of network performance [2, 7]. Routing optimizations techniques are categorized according to the routing protocol considered, the on-line/off-line nature of the optimization and the hierarchy structure (i.e. centralized or distributed). Despite this very consistent body of work, due to the lack of available programmable management frameworks, the presented applications are typically not practically implemented, but only tested through simulation tools or theoretical experimentations.

The idea of optimizing the link weights of a shortest path routing protocol to perform traffic engineering has been extensively studied (see [7] for a survey and [8] for a particular application). However, although link weight optimization appears to be a scalable and natural way to modify network routing, the real impact of weight adjustment on normal network operation should be carefully analysed. For instance, with a real implementation, it would be possible to monitor delay and packet loss measurements during transitory periods, detect potential service disruption, measure the transient length and test different methodology to

apply the weights change. Other analysis may also include the evaluation of the stability of policies for dynamic weight adjustment [9], or the monitoring of the effectiveness of alternative routing update methods such as *Distributed Loop-free Routing Update* (DLRU) [10]. In all these cases (and others), JNetMan could be potentially used to (i) collect link load measures to evaluate the congestion level or estimate traffic matrices [11] used as input of the optimization algorithms, (ii) update the desired link weights according to a precise policy, (iii) directly modify the router routing tables, (iv) adjust the split ratio parameters in a *Multi Topology Routing*-based (MTR) network [12], and (v) collect real time measurements on delay, packet loss, etc.

The use of JNetMan and the practical implementation of network management strategies are not limited to shortest path based systems. For instance, in case of applications based on the widely used Multi Protocol Label Switching (MPLS), JNetMan may be easily used to test a policy for path selection and resource allocation [13] or evaluate the scalability of the routing path update process [14], which is typically quite critical due to the elevated number of configurable paths. To conclude, note that JNetMan may be exploited in many other alternative domains, such as multicast routing optimization or energy-aware network management, by modifying, for instance, the multicast static group composition, the ID of the network routers (it determines the routing tree structure in case of multiple equal shortest paths), the multicast trees or even the on/off state of a link interface.

3 Guidelines for reusable and programmable network management

Before presenting JNetman features and architecture, it is worth pointing out some crucial aspects concerning the nature, the structure and the requisites of a general reusable and programmable framework for network management.

First of all, let us clarify the distinction between the concept of management framework, which we are focusing on in this paper, and that of management platform. A management framework can be view as a comprehensive set of low level functionalities/primitives, which provides a high level interface between developers and real management operations. This interface is then used to naturally develop what is commonly defined a network management platform, i.e. a top level management tool based on an intelligent controller which enables multiple complex management policies (built combining the primitives of the management framework). Thus, the management framework represents the engine of a more general management platform.

Two important requirements of a management framework are modularity and extensibility, which are crucial to offer the possibility to implement the desired management strategy by quickly adapting the existing modules, or even adding new ones. These requisites are guaranteed by making choices at two different levels: (i) at the lower one, by carefully designing the framework general architecture to clearly identify and isolate each functional block, and, (ii) at the higher one, by keeping the project as open-source, thus allowing anyone who using the framework to personally develop its own extensions to meet its necessity.

A second significant aspect concerns the choice of the underlying network management technologies, which will be exploited to practically implement the management and monitoring capabilities offered by the framework. For instance, as done in JNetMan (see Section 4), due to its wide diffusion in the existing network infrastructure, a possible choice may fall on SNMP, which offers multiple management functions which are device vendor independent and allows to configure both physical devices and network protocols. Other management technologies may include simple command line instructions (e.g. the ping command to measure path round-trip-time) or other management protocols (e.g NETCONF). Note that, due to the extensible and open-source nature of the framework, the integration of a new module to support new management functionalities should be possible at any time.

As for the general architecture, the management framework should provide modules to (i) elaborate a high level abstraction of the real network topology to clearly identify and manage the desired network elements (e.g. link interfaces, chassis, etc.), (ii) establish and manage secure and resilient communications between the network elements according to the rules dictated by the considered management protocol, (iii) offer a set of high level management primitives which allow to easily perform the desired management operations (e.g.

retrieve the load on a given link, modify a parameter on a given node, etc.). These three elements jointly concur to make a framework flexible and programmable.

4 JNetMan

JNetMan is our novel Java-based network management framework, developed to offer users a dynamic and flexible low level baseline to design and practically implement new network management applications. It has been realized by following the main guidelines just mentioned above, both in terms of modularity, extensibility, programmability and general architecture. Being JNetMan an open source project freely distributed under the Apache License 2.0 model, source code and documentation of the first release are available for download at [5].

In its first version, JNetMan has been designed to support SNMP as management protocol. The aim of the framework is to provide a clear separation between the low level SNMP management instructions and the specific high level management strategy. SNMP offer functionalities to perform several management operations, like retrieving the load value of a link and the number of MPLS tunnel carried out on a given interface, or configuring the hello interval of the OSPF protocol. Each specific command requires the execution of a specific group of low level instructions, including, for instance, authenticating the SNMP agents, retrieving the *Object Identifier* (OID) which represents the desired parameter inside the MIB of the selected device, performing the cast between data types adopted in the MIB and variables types used by the management application, generating the packets and triggering the time-outs. However, thanks to the set of powerful primitives offered by JNetMan, all these low level procedures are carried out in a transparent way by the system, allowing the users to directly interact with the desired network devices trough a high-level abstraction of the network and its defined management methods.

The result is a reusable software platform destined to easily implement specific solutions for the management of telecommunications networks and successively conduct extensive experimentations in realistic network scenarios.

The architecture of JNetMan is illustrated in Figure 1. The framework is composed of three main operational layers, the SNMP plug-in, the Topology Abstraction Manager and a collection of Aspect Managers. The latter represents the sets of libraries used to manage different aspects of the network and its devices. At the top of the framework stack stands the management application, which, through the use of JNetMan's API, implements the high level management strategy.

4.1 SNMP plug-in

The SNMP plug-in represents the communication interface between the centralized platform and all the SNMP agents of the network (i.e. the network devices). It implements all the low-level SNMP instructions required to execute the desired management operations with SNMP, including, primarily, the establishment of a reliable SNMP communication channel between the SNMP client of the management host and the SNMP agent of each network device. This block largely relies on the functionalities offered by an existing open-source solution called SNMP4J [15], an enterprise class free open source and state-of-the-art SNMP implementation for Java.

To faster the use and the coordination of the available SNMP management tasks, the *SNMP Plug-in* provides two auxiliary modules, i.e. the *MIB Helper* and the *SMI Helper*, which offer very useful methods to, respectively, process data gathered from specific MIB-modules and safely convert data-types from Java to the MIB format specified by the *Structure of Management Information* (SMI) and vice versa.

4.2 Topology Abstraction Manager

According to the architecture of SNMP, a network is logically composed by the SNMP agents run by the network devices, and each management operation involves the interaction with a specific SNMP agent. However, from the perspective of a user developing a specific high level management application, this logical

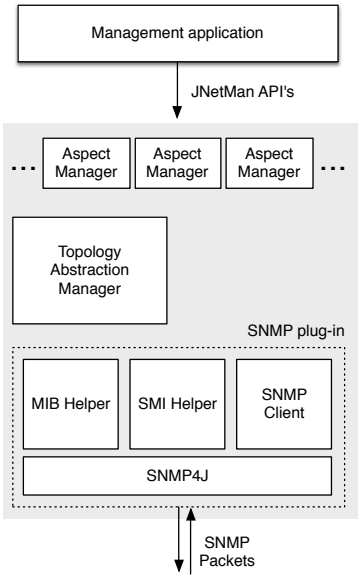

Figure 1: Architecture of JNetMan

representation based on SNMP agents may result unintuitive and misleading. The aim of the *Topology Abstraction Manager* (TAM) is to provide a more natural network topology representation composed by multiple *network entities*. Currently three classes of *network entities* are defined, namely *node*, *interface*, and *link*. A *node* is a network node, and can be intended as an interconnection or an end device. Then, each node is equipped with at least one or multiple network *interfaces*, while a *link* is defined as a connection between two *interfaces*. The TAM provides methods which aim at helping the users to define and characterize the topology, in terms of both use relationships (defined above) and attributes, (e.g. the node IP address used by the SNMP plug-in to set/get values, the ID used by the system to refer to a particular interface, the nominal speed of a link, etc.)

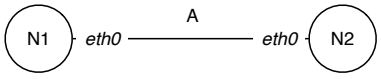

Figure 2: Example of a 2 nodes, 1 link topology

In Figure 3 we show how defining the simple topology of Figure 2, by using the APIs of the TAM:

Once this information has been structured into an high-level abstraction of the network topology, it will be easier for the Aspects Managers to navigate through the network elements and run the desired management operations on the network devices. Likewise, the user will directly refer to these entities to develop its own a management application. A quicker and automatic method to define the topology and its attributes trough plain text files will be shown in the next sections. Furthermore, when the network topology is not known a priori, JNetMan is equipped with an auto-discovery module able to autonomously gather the network information by directly interrogating a set of predefined network devices.

4.3 Aspect Managers

Each network device running an SNMP agent maintains multiple hierarchic database structures known as *Management Information Bases* (MIB), which collect management information of the underlying network system. Each different MIB contains data of a specific device subsystem, such as the routing protocol or the operative status.

```
// First create a network object
Network network = new Network();
// Add two nodes, namely N1 and N2
Node n1 = network.createNode("N1");
Node n2 = network.createNode("N2");
// Add one interface per each node, named eth0
IfCard n1eth0 = n1.createInterfaceCard("eth0");
IfCard n2eth0 = n2.createInterfaceCard("eth0");
// Add one link, named A, and define it as connection between N1.eth0 and N2.eth0
Link linkA = network.createLink("A", n1eth0, n2eth0);
// Set the actual IP address of both nodes.
// This is used for SNMP communications.
n1.setIPAddress("172.16.10.1");
n2.setIPAddress("172.16.10.2");
```

Figure 3: Procedure to define the network topology of Figure 2 through TAM's APIs.

```
/*
 * We want to measure and print the average
 * bit rate (bit/s) of each link defined in
 * the network over an interval of 5000 ms.
 */
for (Link link : network.getLinks()) {
  long br = Monitoring.getAvgBitrate(link, 5000);
  // Print the result, e.g. "linkA: 638000 bit/s"
  System.out.printf("%s: %d bit/s%n", link.getName(), br);
}
```

Figure 4: Procedure to retrieve the load on the network links through the APIs of the *Monitoring* aspect manager.

To guarantee a natural convergence between SNMP and the management framework, JNetMan provides multiple modules called *Aspects Managers* (AM), each one designed to interact with a specific MIB. A single AM is a library of high-level primitives (also commonly known as API) providing functionalities to manage a specific aspect of the system. The provided APIs represent the interface between the management framework (JNetMan) and the top level management application. Their efficient combination allows the users to flexibly implement the desired algorithms. The role of the APIs is to provide complete transparency between the management tasks and their practical implementation. Each AM is defined as an independent module (i.e. a static class in Java) which make use of TAM entities as the object of its management operations. In Figure 4 we show how using the *getAvgBitrate* primitive, defined by the *Monitoring* module, to measure the average bit rate of a link over a fixed interval.

Arguments of the *getAvgBitrate* primitive are, respectively, the link on which to perform the measurement and the interval. In this case the *Monitoring* module exploits the TAM and the SNMP Plug-in functionalities to perform multiple low level operations, such as (i) retrieving from the TAM the IP addresses of the nodes connected to the considered link, (ii) generating specific requests toward the nodes themselves, (iii) demanding the SNMP agents of the nodes to report the number of *bytes* sent and received over the defined interval by the network interfaces attached to the link, and (iv) elaborating these values to compute the average bit-rate. These operations are performed in a way completely transparent for the user who just has to handle the final output values (printing it to stdout in this case).

The AMs are split among *Basic Managers* and *Advanced Managers*. The firsts provide APIs to manage basic elements of the system typically common to all TCP/IP-based devices, while the latter offer function-

alities to control more specific domains, such as the routing or the signaling protocols. The existing *Basic Managers* are:

- **Monitoring Manager**: primitives to perform general monitoring tasks, like measuring the uptime of a network device, the average bit-rate of a link, etc.
- **Interface Manager (IF-MIB)**: primitives to manage network interfaces, e.g. to retrieve or modify the operative state (up/down), obtain the number of bytes sent or received, etc.
- **IP Manager (IP-MIB)**: primitives to retrieve or modify IP-level information, such as IP addresses, IP routing tables, number of IP packets sent or received, etc.

As for the *Advanced Managers*, the only module that has been already developed is the *OSPF manager*, which offer multiple functions to configure the OSPF protocol, e.g. by modifying the link weights, the Hello interval, etc.

While in this work we are presenting the main logic and features of JNetMan, we refer the reader to the updated documentation available at [5] for the complete list of primitives defined for each module and for any compatibility issues.

4.4 Configurability

A key feature of JNetMan is the elevate configurability offered to users during testing phases. Practically speaking, we intend for configurability the possibility of quickly modifying the system parameters without the need of repeatedly adjusting the underlying code. This feature allows the users to validate the developed management solutions under different configurations by easily running multiple executions and quickly modifying time by time the desired parameters. To offer this feature, JNetMan has been developed with the aim of minimizing the number of hard-coded parameters in favor of more practical external configuration input files. To correctly operate, JNetMan requires a specific set of information, including the IP addresses of the network devices, the ID of the interfaces installed for each node, the transmitting and receiving interfaces of each link, plus other additional attributes and configuration parameters such as SNMP user and password, timeout, etc. All these data are provided to JNetMan by multiple plain ASCII files, namely `.properties` files, (usually no more than a few hundreds of kilobytes). These files are read by JNetMan at runtime, thus making it very simple and straightforward to manage an existing network. In Figure 5 we show how defining a network topology in JNetMan by using a very intuitive Java-inspired "*key.attribute = value*" syntax.

4.5 Extensibility

One of the main feature of JNetMan is its open-source nature, which potentially allows any user of the framework to extend and modify the existing modules to meet new requirements and improve the management functionalities. Due to the architecture modularity, a user can easily identify either the existing functional block requiring improvements (e.g. a particular aspect manager or even the entire TAM), or even the entire architecture layer which needs the addition of a completely new module, such as a new aspect manager or a low level plug-in to implement a new management protocol like NETCONF.

Let us have a more accurate look at the possible improvements for the Aspect Managers (AMs), which, being the AM layer the upper one, are the more straightforward to accomplish and potentially the more frequent to be requested. Each AM contains primitives to manage information (both to read and write) contained in a specific MIB class. Due to the very large number of parameters stored in each MIB, only a part of them can be managed by the existing AM primitives. Thus, to develop a management application requiring the interaction with a MIB object not supported by the APIs of the corresponding AM, we offer users the possibility to develop their own new primitives and integrate them inside the AM. The API development can rely on both the network abstraction provided by the TAL and the large set of low level functionalities offered by the SNMP-plug-in.

In some circumstances, the development of a new management procedure may require the interaction with a MIB not yet supported by any one of the existing AMs, like, for instance, the MIBs which stores

```
# nodes.properties example
N1.eth0 = 172.16.1.1/24
N1.eth1 = 172.16.4.1/24
N2.eth0 = 172.16.1.2/24
N2.eth1 = 172.16.2.2/24
N3.eth0 = 172.16.2.3/24


# links.properties example
A = N2.eth0 N1.eth0
A.nominalSpeed = 100000000 # bit/s
B = N2.eth1 N3.eth0
B.nominalSpeed = 100000000
C = N1.eth1 N3.eth1
C.nominalSpeed = 100000000
```

Figure 5: Example of `nodes.properties` and `links.properties` files for a simple network with three nodes, namely N1, N2 and N3, three links, i.e. A, B and C, and two Ethernet interfaces installed on each node. Each link id defined by the corresponding interfaces, while *nominalSpeed* is an optional attribute representing the link capacity.

the configuration data of MPLS or DiffServ. In that case, a user may any way use JNetMan by integrating the current AM stack with a new AM containing the primitives to interact with the desired MIB. Note that, although the development of a new AM may appear complicated, it is worth pointing out the crucial advantages that it would bring w.r.t. the deployment of a new stand-alone management application:

- The addition of a new AM in the existing architecture allows the user to exploit all the APIs already provided and combine them with new ones.
- Writing new stand-alone libraries to manage a particular MIB would force the user to directly take care of all the aspects of the system, including the low level SNMP routines and the network abstraction ones. Conversely, since JNetMan already offers the low level primitives to run SNMP and manage the network topology, developing a new AM would require only the formulation of new high level functionalities relying on efficient and tested low-level management sub-functions.

Finally, let us briefly focus on a particular aspect of SNMP which is highly related to JNetMan extensibility feature. Although SNMP relies on a well documented standard, several vendors have implemented their own proprietary MIB, characterized by a different structure and different identifiers assigned to the parameters. To guarantee the compatibility with this vendor-related MIB, and thus coping with some specific network devices, dedicated AMs may be developed.

4.6 Integration with SDN

In the recent years Software Defined Networking (SDN) has emerged as a new paradigm to control and manage network services. This is done by decoupling the control-plane, that is the system that makes decisions about forwarding traffic, from the data-plane, which is the actual logic that forwards the traffic in a switch. In this scenario OpenFlow has emerged as an open standard that provides a clean interface to program the forwarding plane. In SDN the control-plane is logically centralized in a controller software platform, that, by using protocols like OpenFlow, is able to program forwarding rules in switches, allowing network administrators to have programmable central control of traffic forwarding and processing without requiring physical access to the network's switches.

It is becoming more clear how the real value of SDN is usually found at the edge of the network, that is were most of complex decisions and actions regarding how to handle traffic flows are taken, while the core

transport network is given the duty to only forward and route traffic. There are many examples of limiting the intelligence to the network edge and keeping the core simple with legacy fabric.

In this scenario we believe that solutions like our framework can have a central role in developing SDN controller platforms able to globally orchestrate first, the execution of network services at the network edge, and second, the provisioning and monitoring of the transport core which is based on legacy protocols.

In this perspective JNetMan could be integrated by exposing AM's APIs to applications implemented on top of the SDN controller northbound interface, while having the SNMP Plug-in part of the southbound interface used to communicate with network devices. Moreover, the TAM could be used to both share and collect informations about the topology map, finally enabling a transparent and coordinated management of both the edge and core network.

5 A practical application

Let us quickly analyze with a practical example of how JNetMan can be efficiently exploited to develop a specific management application like the one presented in [4] for energy-aware network management. The green approach discussed in [4] proposes to dynamically adapt the network consumption to the observed traffic levels by efficiently adjusting the OSPF configuration and putting to sleep the unnecessary portions of the network. According to OSPF, an administrative weight is assigned by the network operator to each link and the set of network weights jointly concur to build the shortest path tree of each router. Differently from a typical case wherein the link weight optimization aims at minimizing a measure of network congestion, in this energy-aware context it is smartly exploited to reduce the network energy consumption. Energy savings are achieved by putting to sleep the network elements which appear completely unloaded because excluded by any shortest path routes through the targeted use of very high link weights.

JNetMan has been used to develop a centralized network management platform able to practically implement a dynamic policy for the energy-aware link weight optimization which is based on traffic measurements collected in real-time from the considered routers. This solution has been easily developed by means of the primitives provided by three particular AMs: (i) the *Monitoring* module, which allows to periodically retrieve link load values, (ii) the *OSPF Manager*, which is responsible for modifying the link weights, and (iii) the *IP Manager*, which offers a function to collect statistics on discarded packets due to routing error.

Furthermore, the configurability feature of JNetMan has been efficiently exploited to conduct extensive experimentations by considering multiple network topologies and adjusting time by time the configuration parameters, such as the polling interval for the load measures or the waiting period during the weight update phase.

6 Conclusions

In this paper we presented a novel open-source network management framework, called JNetMan that provides a low level baseline for the flexible and quick implementation of high level network management applications based on SNMP. JNetMan offers to users (i) multiple sets of high level primitives to easily execute the desired management tasks without taking care of the low levels routine necessary to practically complete the operations, (ii) an intuitive logical abstraction of the real network composed of three network entities, namely *nodes*, *interfaces* and *links*, and (iii) a compact configuration interface based on simple ASCII files to rapidly tune the system parameters and run multiple experimentations. Thanks to its modular architecture and to its open-source nature, JNetMan has been predisposed to be further developed by any users requiring new low level functionalities, and easily integrated into network management and programmability platforms including SDN controllers. Finally, as application example, we have pointed out the JNetMan use for the implementation of a network energy management application to reduce the consumption of network domains operated with OSPF.

References

[1] M. Subramanian. *Network Management: An Introduction to Principles and Practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1999.

[2] Fan Wang, A. Ghosh, C. Sankaran, P. Fleming, F. Hsieh, and S. Benes. Mobile WiMAX systems: performance and evolution. *IEEE Communications Magazine*, 46(10):41–49, October 2008.

[3] Riverbed Technology. Network Planning & Simulation (OPNET). http://www.riverbed.com/products-solutions/products/network-performance-management/network-planning-simulation.

[4] A. Capone, C. Cascone, L.G. Gianoli, and B. Sansò. Ospf optimization via dynamic network management for green ip networks. Accepted to SustainIT-13, The Third IFIP International Conference on Sustainable Internet and ICT for Sustainability, 2013.

[5] JNetMan, Java framework for effortless development of SNMP-based, proactive, network management applications [Online]. http://www.jnetman.org/.

[6] Opendaylight project. http://www.opendaylight.org.

[7] A. Altin, B. Fortz, M. Thorup, and H. Ümit. Intra-domain traffic engineering with shortest path routing protocols. *4OR*, 7(4):301–335, December 2009.

[8] B. Fortz and M. Thorup. Increasing internet capacity using local search. *Computational Optimization and Applications*, 29(1):13–48, October 2004.

[9] C. Phillips, A. Gazo-Cervero, J. Galàn-Jimènez, and X. Chen. Pro-active energy management for wide area networks. In *Communication Technology and Application (ICCTA 2011), IET International Conference on*, pages 317–322, 2011.

[10] C. Lee. Securing vehicular ad hoc networks: A research survey. Cryptology and Information Security Conference 2011, Taiwan, May 2011.

[11] M. Rahman, S. Saha, U. Chengan, and A. Alfa. Ip traffic matrix estimation methods: Comparisons and improvements. In *2006 IEEE International Conference on Communications*, volume 00, pages 90–96. IEEE, 2006.

[12] Y. Wang and F. Li. Vehicular ad hoc networks. In *Guide to Wireless Ad Hoc Networks*, Computer Communications and Networks, pages 503–525. Springer London, 2009.

[13] X. Xiao, A. Hannan, and B. Bailey. Traffic engineering with mpls in the internet. *Network, IEEE*, 14(2):28–33, 2000.

[14] A. Elwalid, C. Jin, S. Low, and I. Widjaja. Mate: Mpls adaptive traffic engineering. In *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No.01CH37213)*, volume 3, pages 1300–1309. IEEE, 2001.

[15] The SNMP API for Java (SNMP4J) [Online]. http://www.snmp4j.org.