An Architecture for Experimenting with Traffic Engineering and QoS Routing Algorithms

Stefano Avallone, Simon Pietro Romano and Giorgio Ventre
COMICS Lab, Dip. Informatica e Sistemistica
Università di Napoli Federico II
Via Claudio 21, 80125 Napoli, Italy
Email: {stavallo, spromano, giorgio}@unina.it

***Abstract*—The Internet research community is making a great effort in order to define efficient network management and control functions. The goal is to make the best use of the network infrastructure, while meeting the users' requirements at the same time. One of the keys to achieve such a goal is the routing algorithm. A wide variety of traffic engineering and QoS routing algorithms have been proposed aiming to optimize resources and provide QoS to the users. However, the performace of such algorithms have usually been evaluated through simulations. In this paper, we present an architecture which enables to experiment with different routing algorithms in a real network. The performance of the routing algorithms can then be evaluated in terms of number of admitted flow requests, average delay, packet loss and jitter. We describe how we implemented all the components of the proposed architecture and how we setup a testbed consisting of Linux PCs. Finally, we illustrate an experiment we conducted to demonstrate the operation of our architecture.**

I. Introduction

Recently, a great deal of research efforts have been devoted to optimizing network management and control. In this context, a major challenge is to define a routing algorithm which meets the users' requirements, at the same time optimizing network resources utilization. A wide variety of intra-domain routing algorithms have been proposed for this purpose (e.g., [1][2][3][4][5][6]). They are mainly centralized algorithms which assume the complete knowledge of the topology is available and individually route each flow request. Such routing algorithms can be roughly divided into QoS routing and traffic engineering algorithms. The former represent QoS requirements as constraints and address the problem of finding a multi-constrained optimal path. The latter target resource optimization and consider the bandwidth (or the *effective* bandwidth) as the sole QoS requirement.

The performance of the routing algorithms discussed above has been typically evaluated through simulations. Indeed, setting up a testbed for experimenting with such algorithms presents many challenging issues. Firstly, such QoS routing and traffic engineering algorithms are not suitable for a distributed hop-by-hop implementation, where each router autonomously determines the next hop for a packet based on the knowledge of the network topology and the destination of the packet. Indeed, such algorithms also require the knowledge of the QoS requirements of the flow the packet belongs to. Furthermore, in case multiple additive QoS constraints are considered, the shortest path from an intermediate node to the destination node may not be a sub-path of the shortest path from the source node to the destination node. Indeed, it has been shown [7] that subsections of shortest paths are not shortest paths in multiple dimensions. As a consequence, an intermediate node should also be aware of the source node of the packet inside the domain, i.e., the ingress node through which the packet entered the domain, to forward the packet along the shortest path from the source node to the destination node.

A centralized approach combined with explicit routing is instead more efficient. The explicit path for each flow is computed just once by a single entity (e.g., the ingress node or a centralized manager) and established using a signaling protocol. MPLS (Multi-Protocol Label Switching) [8], e.g., is a network infrastructure which enables to setup virtual circuits (denoted as LSPs – Label Switched Paths) along explicit paths. A packet filter installed on the ingress node classifies the packets and forwards them along the appropriate LSP.

In the centralized approach, however, we need to provide the single entity computing the explicit paths with the network topology and the vector of link weights associated with every link. Traffic engineering algorithms usually require just one link weight (the available bandwidth), while QoS routing algorithms typically require multiple additive link weights (e.g., delay and jitter). The network topology may be built by exploiting the operation of link state routing protocols (e.g., OSPF – Open Shortest Path First [9]), which provide that each router stores the messages exchanged with its peers in a link state database. As far as the vector of link weights, we may extend a link state routing protocol to have it carry information about the link weights. Such an approach is certainly required in case the link weights are dynamic and reflect the current QoS values experienced by packets crossing the link. However, in [10] we show that while an accurate knowledge of the current available bandwidth is needed, using dynamic additive link weights requires complex routing strategies to assure that the QoS granted to admitted flows is still preserved after new flows are routed. Instead, static link weights based on upper bounds to the QoS values that can be experienced across a link require the routing algorithm to perform no other operation than computing the explicit paths.

In the implementation of our testbed, we opted for the

strategy to set the additive QoS link weights defined in [10]. Since they are static, there is no need to extend a link state protocol in order to propagate them among routers. As far as the available bandwidth, if we assume that all the flows entering the network are managed by the ingress nodes or the centralized manager, then we do not need to propagate either the information about the available bandwidth. Indeed, on every link the available bandwidth initially equals the physical capacity and is then decremented each time a flow is routed on the link by an amount equal to the bandwidth requirement of the flow. Such a strategy implies that, in case the explicit paths for the flows entering the network through an ingress node are computed by the same ingress router, the ingress nodes need to exchange the information about the explicit paths they have computed and the bandwidth requirement of the flows in order to have a consistent view of the bandwidth availability on the network links. In case of a centralized manager which computes the path for all the flows, instead, there is clearly no need for such exchange of information. However, the centralized manager has to communicate the computed explicit paths to the ingress nodes, so they can establish the corresponding LSPs.

Our architecture comprises a centralized manager, denoted as Network Controller, which manages all the flows entering the network. The Network Controller may be configured to use one of the routing algorithms we have implemented to find an explicit path for each flow request. Another component (the Service Manager) produces the requests for the arrival and termination of flows. If there are not enough resources to guarantee the QoS requirements of a flow (i.e., the routing algorithm does not return any feasible path), our architecture does not admit the flow. Otherwise, the computed explicit path is established and the traffic corresponding to the request being served is generated. We underline that all the processing of the flows is done automatically. For this reason, we refer to our architecture as the automated manager.

To demonstrate the operation of the automated manager we implemented all of its constituent modules in a real experimental testbed made of Linux PCs. Each component communicates with the others by means of standard protocols (e.g. LDAP, SNMP, etc.). This property allows for modularity, i.e. it is possible, for example, to replace a Linux router with a commercial router, provided that the latter implements the required functionality.

The paper is structured as follows. Section II describes the component of the automated manager in detail. Section III summarizes the operation of the routing algorithms we have implemented. Section IV illustrates the operation of our automated manager through an experiment we conducted. Finally, Section V concludes the paper.

II. Description of the Overall Architecture

The overall architecture of our automated manager is illustrated in Fig. 1. This section presents an overview of its constituent components and their inter-relations, while the following subsections dig into the details of the implementation

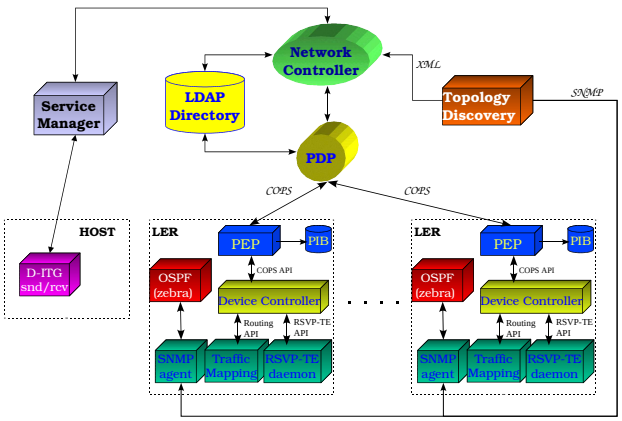

Fig. 1. Architecture of the automated manager

of the single blocks.

The Service Manager informs the Network Controller about the arrival and departure of flow requests. For each request, the Network Controller performs admission control and computes an explicit path using the selected routing algorithm. The routing algorithm requires the knowledge of the network topology, which is provided by the Topology Discovery component. We set up an LDAP (Lightweight Directory Access Protocol) [11] server to permanently store information to be accessed by multiple components at different stages (e.g. the network topology). If the user request is admitted, the selected path must be established (if not existent yet) by its ingress node. Also, a traffic filter has to be installed on the ingress node in order to make the specified traffic flow across the corresponding path. We chose COPS (Common Open Policy Service) [12] as the protocol to communicate the selected path (as a list of IP addresses of its constituent nodes) and the traffic specification to the ingress node. The ingress node is in charge of establishing the corresponding explicit LSP through the MPLS network. The signaling protocol used to accomplish this task is RSVP-TE [13], i.e. the Resource reSerVation Protocol with the extensions for Traffic Engineering. If the LSP is correctly setup, an acknowledgment is propagated back to the Service Manager, which informs the traffic generator on how to transmit the requested traffic. By collecting statistics about the generated flows, it is possible to retrieve information on the experimented throughput, delay, jitter and packet loss.

A. Topology Discovery

The task of the Topology Discovery module is to determine the network topology. This information is vital to the operations of the automated manager. There are several methods proposed in literature to accomplish such task. Some of them assume no knowledge about the network whose topology has to be discovered. Thus, they necessarily come out with an approximated topology of the network. This work assumes a different viewpoint, since we have the control of the experimental testbed. We have implemented [14] a module

that discovers the topology of an OSPF network through SNMP (Simple Network Management Protocol) queries. The Topology Discovery accesses the OSPF MIB (Management Information Base) [15] of a randomly chosen router to retrieve a copy of the LSAs (Link State Advertisements) stored and some other information such as the area(s) which the router belongs to, the area border routers (i.e. routers belonging to multiple areas) identifiers, etc. In case the contacted router does not belong to the backbone area (area 0), the Topology Discovery determines the area border router belonging to area 0 and accesses its OSPF MIB through SNMP. Then, all the area border routers of the backbone area are contacted. Each of them provides information on a single area of the network. By collecting such information, the Topology Discovery is able to reconstruct the overall topology of the network. After that, the Topology Discovery queries (using SNMP) all the routers of the network in order to retrieve information on the IP addresses of all the interfaces, the capacity of each interface, etc.

The result of all these inquiries is an XML (eXtensible Mark-up Language) file constituted by two parts: a list of router identifiers (including IP address and capacity of each interface) and a list of links.

Such implementation of the Topology Discovery requires an SNMP agent and an OSPF process running on each router. As stated before, all the architecture blocks have been implemented in a Linux environment. Linux PCs act as routers and *zebra*, a free routing software, provides an implementation of the OSPF (and other) routing protocol.

B. Service Manager

The Service Manager is in charge of managing flow requests. It forwards them one-by-one to the Network Controller, so they can be served and assigned a path if admitted.

The Service Manager needs to know the IP addresses of the hosts connected to the network and acting as sources and destinations. For this purpose, it retrieves the network topology from the LDAP server. The nodes having at least an interface which is not connected to another node of the network are considered to be edge nodes. Such interfaces are supposed to be connected to source/destination nodes. To determine the IP addresses of the hosts, we also suppose that the network which the host and the edge node belong to is subnetted in order to allow just two hosts (i.e. the network mask is 255.255.255.252). The Service Manager uses the obtained IP addresses to fill the source and destination address fields of the flow requests sent to the Network Controller.

The value of the other fields of a flow request (e.g. flow duration, bandwidth requirement, QoS constraints) are described by random variables with constant, uniform or exponential distribution. The parameters of such random variables can be specified in a configuration file.

The Network Controller serves each request and replies to the Service Manager informing it about the result of processing the request. In case the request is accepted and the network correctly configured, the Service Manager requests

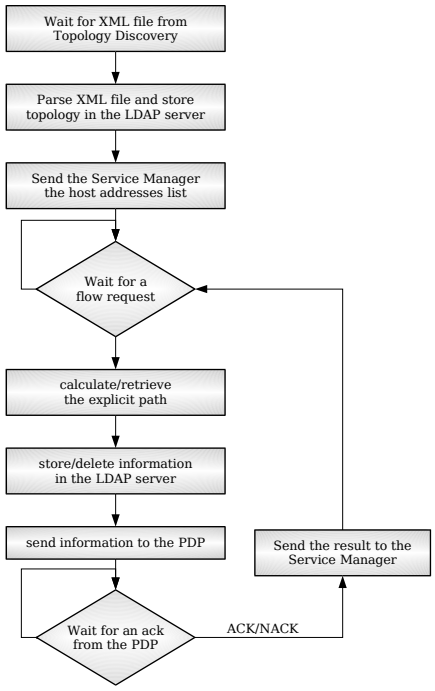

Fig. 2. Network Controller flow chart

the generation of the flow. For this purpose, we used D-ITG (Distributed Internet Traffic Generator) [16], our platform for traffic generation, which enables to launch the sender component in *server* mode. This means that the sender stays idle waiting for messages that instructs it on the flows to be generated. The sender component has to be launched in server mode on every host, together with the receiver component. The Service Manager uses the C++ API (Application Programming Interface) included in the D-ITG package to request the generation of a flow corresponding to the user request. The D-ITG sender notifies the Service Manager about both the reception of the message and the end of the flow generation. When the Service Manager is notified about the end of the flow generation, it informs the Network Controller, which can release the resources allocated to the flow.

C. Network Controller

The Network Controller is the brain of the whole automated manager. Based on resource availability and the selected routing algorithm, it admits or rejects flow requests and selects an explicit path for those admitted. The routing algorithms described in Sect. III have been implemented and are available for use. A scheme of the operation of the Network Controller is illustrated by the flow chart in Fig.2.

The Network Controller cannot operate without the knowledge of the underlying network. In the initialization phase, the Network Controller stays idle until it receives this information from the Topology Discovery module as a set of XML tags. An XML parser is used to extract the network topology and the description of each node (IP addresses of the interfaces,

capacity of the links, etc.). The extracted information is stored in two directories (*Topology* and *Network*) of the LDAP server. The introduction of a permanent storage component like the LDAP server is justified by a couple of reasons. First, some information needs to be accessed by multiple components at different stages. Second, a single component (the Network Controller) in charge of managing the network topology and the established LSPs may not scale.

Besides storing the network information in the LDAP server as described, the Network Controller creates a data structure of its own to contain network topology. Such data structure is used by the routing algorithm routines and has been defined for this purpose. Also, it provides for the information (not stored in the LDAP server) on the available bandwidth of the links at any time. When a flow is admitted (terminates), the available bandwidth is decreased (increased) accordingly. We underline that our automated manager currently does not make any bandwidth reservation, i.e. there is no traffic differentiation and all the traffic is treated in a best-effort fashion. The Network Controller is also arranged for supporting QoS constraints like delay, jitter and packet loss. For each link, the value of those QoS measures is considered as a function of the available bandwidth. Some routing algorithms that explicitly consider QoS constraints have been already implemented. It is therefore possible to manage requests having a bandwidth requirement and some QoS constraints by selecting one of such routing algorithms. The definition of the relationships between QoS measures and available bandwidth will be further investigated in the future.

After the initialization phase, the Network Controller is ready to serve flow requests sent by the Service Manager. There are two kinds of such requests: flow arrival and flow departure requests. In case of a flow arrival request, the Network Controller uses the selected routing algorithm to determine whether the flow can be admitted and, in the positive case, the path it has to follow. The *LspTable* directory of the LDAP server is used to keep track of all the established LSPs. Such directory is made of as many sub-trees as the number of edge nodes, each sub-tree representing the LSPs originating in an edge node. The leaves are entries with two attributes, the LSP identifier and the number of flows mapped on the LSP. When a new flow is admitted, the LspTable directory is scanned to determine whether an LSP along the computed path has already been established. If so, the number of flows mapped on the LSP is increased by one and there is no need to ask the PDP (Policy Decision Point) to establish a new LSP. Otherwise, the identifiers of the nodes constituting the selected path are stored in the LspTable directory (with a new LSP identifier) and the PDP (Policy Decision Point) is asked to establish the new LSP. The PDP receives just the LSP identifier and returns the result of the operation. In any case, the new flow needs to be mapped on the selected LSP. First, an entry with the flow specification (source and destination addresses and ports, protocol, requested bandwidth) is stored in the *FilterTable* directory of the LDAP server. Such an entry has two additional attributes, a filter identifier and the identifier of the LSP the flow is mapped on. Then, the identifier of the filter to be installed is communicated to the PDP, which takes the appropriate actions and replies with a positive or negative acknowledgment. The Network Controller thus inform the Resource Manager about the result of the operation.

When the Network Controller receives a flow departure request, it has to find the path along which the flow has been routed, in order to deallocate resources. For this purpose, the Network Controller scans the FilterTable directory searching for the entry corresponding to the flow to be removed. Such entry contains the identifier of the LSP assigned to the flow, which can be used to access the LspTable directory and obtain the sequence of hops constituting the LSP and the number of flows mapped on it. The information on the sequence of hops of the LSP enables to properly deallocate resources. If no other flow is mapped on the same LSP, the Network Controller deletes the corresponding entries in the LspTable directory and asks the PDP to remove the LSP. Then, the Network Controller deletes the flow specification entry in the FilterTable directory and asks the PDP to uninstall the filter.

D. PDP (Policy Decision Point)

The decisions taken by the Network Controller must be communicated to the nodes concerned in order to be enforced. We use the COPS protocol to rule such communication. The COPS protocol provides for a unique Policy Decision Point (PDP) and a Policy Enforcement Point (PEP) for each node to be configured. The task of the PDP is to code the action to be undertaken into a set of rules called *policies* and send them to the PEP on the selected node. The Network Controller requires the PDP to perform four kinds of operations: to establish/remove an LSP and to install/uninstall a traffic filter. When the Network Controller determines that an LSP must be established or removed, it informs the PDP about the type of operation to be performed and the LSP identifier. Using such identifier, the PDP accesses the LDAP server to retrieve the sequence of nodes constituting the LSP. Then, it prepares the corresponding policies following the structure defined for the MPLS traffic engineering policies [17]. In case a traffic filter must be installed/uninstalled, the Network Controller communicates the type of operation to be performed and the filter identifier. Such identifier is used to retrieve the flow specification, which is the basis of the policies created by the PDP. Such policies are compliant with the structure defined for framework policies [18].

The PEP that receives the policies performs the required actions and sends the result of such actions to the PDP. The PDP, in its turn, propagates such result back to the Network Controller.

E. PEP (Policy Enforcement Point)

Two components of our automated manager run on each ingress/egress node: the Policy Enforcement Point (PEP) and the Device Controller. The PEP listens for policies sent by the PDP and passes them to the Device Controller, whose task is to enforce the required actions. The reason behind the separation

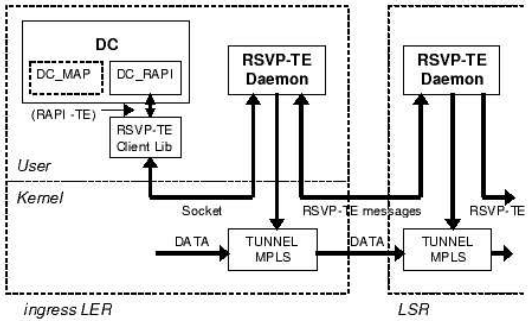

Fig. 3. DC_RAPI subsystem

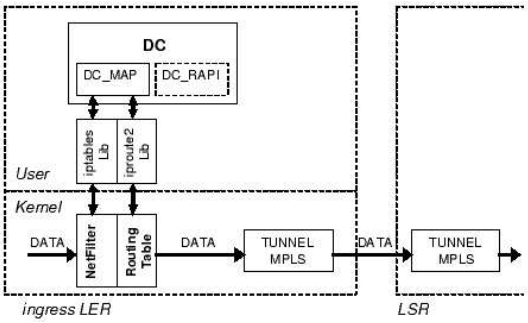

Fig. 4. DC_MAP subsystem

of receipt and enforcement functions is the portability to other systems. The PEP is a simple C program and can be easily ported to other systems. The Device Controller, instead, is highly dependent on the system being used. We have implemented the Device Controller for Linux operating system. Porting the automated manager to another system will only require to implement another Device Controller, while the PEP will require at most minor changes.

A PEP receives policies from the PDP, filters (if necessary) and passes them to the Device Controller. Then, it awaits for the Device Controller to communicate the result of the operation and propagates such result back to the PDP.

F. Device Controller

The Device Controller is the component in charge of configuring network devices. It is therefore strictly dependent on the system where it is running. We describe here our implementation for Linux operating system. The Device Controller

TABLE I
EVENTS LOGGED BY THE SERVICE MANAGER

Name	Description
SM2NC_OKSENDREQ	The Service Manager sent a flow arrival request to the Network Controller
NC2SM_ACCEPTED	The Network Controller admitted a flow
SM2ITG_SENTREQ	The Service Manager sent a request to a D-ITG sender
ITG2SM_STARTFL	A D-ITG sender acknowledged the start of traffic generation
ITG2SM_ENDFLOW	A D-ITG sender communicated the end of a flow generation
SM2NC_OKSENDSTOP	The Service Manager sent a flow departure request to the Network Controller

Listing 1. Network Controller output: network topology

```
- FINDING EDGE NODES
  DevId  192.168.1.34   IfId  192.168.1.14
  DevId  192.168.1.6    IfId  192.168.1.22
  DevId  192.168.2.2    IfId  192.168.2.6

- CREATING DevId_map
  0 192.168.1.1
  1 192.168.1.10
  2 192.168.1.18
  3 192.168.1.2
  4 192.168.1.34
  5 192.168.1.6
  6 192.168.2.2

- Printing topology...
Edge nodes:     4  5  6
Link (0,3): Cap:1000000.00bps  BwdAvail:1000000.00bps  QoSmtr:[415.00]
Link (0,6): Cap:1000000.00bps  BwdAvail:1000000.00bps  QoSmtr:[415.00]
Link (1,2): Cap:1000000.00bps  BwdAvail:1000000.00bps  QoSmtr:[415.00]
Link (1,5): Cap:1000000.00bps  BwdAvail:1000000.00bps  QoSmtr:[415.00]
Link (2,1): Cap:1000000.00bps  BwdAvail:1000000.00bps  QoSmtr:[415.00]
Link (2,4): Cap:1000000.00bps  BwdAvail:1000000.00bps  QoSmtr:[415.00]
Link (2,6): Cap:1000000.00bps  BwdAvail:1000000.00bps  QoSmtr:[415.00]
Link (3,0): Cap:1000000.00bps  BwdAvail:1000000.00bps  QoSmtr:[415.00]
Link (3,4): Cap:1000000.00bps  BwdAvail:1000000.00bps  QoSmtr:[415.00]
Link (3,5): Cap:1000000.00bps  BwdAvail:1000000.00bps  QoSmtr:[415.00]
Link (4,2): Cap:1000000.00bps  BwdAvail:1000000.00bps  QoSmtr:[415.00]
Link (4,3): Cap:1000000.00bps  BwdAvail:1000000.00bps  QoSmtr:[415.00]
Link (5,1): Cap:1000000.00bps  BwdAvail:1000000.00bps  QoSmtr:[415.00]
Link (5,3): Cap:1000000.00bps  BwdAvail:1000000.00bps  QoSmtr:[415.00]
Link (6,0): Cap:1000000.00bps  BwdAvail:1000000.00bps  QoSmtr:[415.00]
Link (6,2): Cap:1000000.00bps  BwdAvail:1000000.00bps  QoSmtr:[415.00]

- CONNECTING TO THE PDP
  Successfully connected to the PDP

- WAITING FOR THE SM
  Sending information to the SM...

- WAITING FOR FLOW REQUESTS FROM THE SERVICE MANAGER
```

can be logically divided into three subsystems:

interface PEP/Device Controller: It is the subsystem in charge of communicating with the PEP. It listens for policies and replies with the result of the operation.

DC_RAPI: It is responsible for establishment/removal of LSPs (Fig.3). For this purpose, it uses the RSVP-TE API (RAPI-TE) to communicate with the RSVP-TE daemon [19] running on the same node. In case of LSP establishment, the sequence of hops extracted from the policies received is passed as parameter of the *rapi_sender* function. This causes the RSVP-TE daemon to include an Explicit Route Object in the PATH message sent to the egress node. To remove an LSP, the *rapi_release* function is invoked with the LSP identifier as parameter.

DC_MAP: It provides the mapping of flows onto LSPs (Fig.4). It exploits the *netfilter* infrastructure and the policy routing capability of the Linux kernel. The netfilter infrastructure allows to define how packets matching some specified conditions are manipulated. The policy routing enables to define multiple routing tables. For each packet, one of the routing table will be consulted based on the *fwmark* value of the packet. Netfilter enables to mark a packet with a specified value. DC_MAP uses the API available to configure both the netfilter infrastructure and the policy routing. First, when an LSP is established, a new routing table is added to route packets having an *fwmark* value equal to the LSP identifier. Such routing table contains just one rule, that causes all

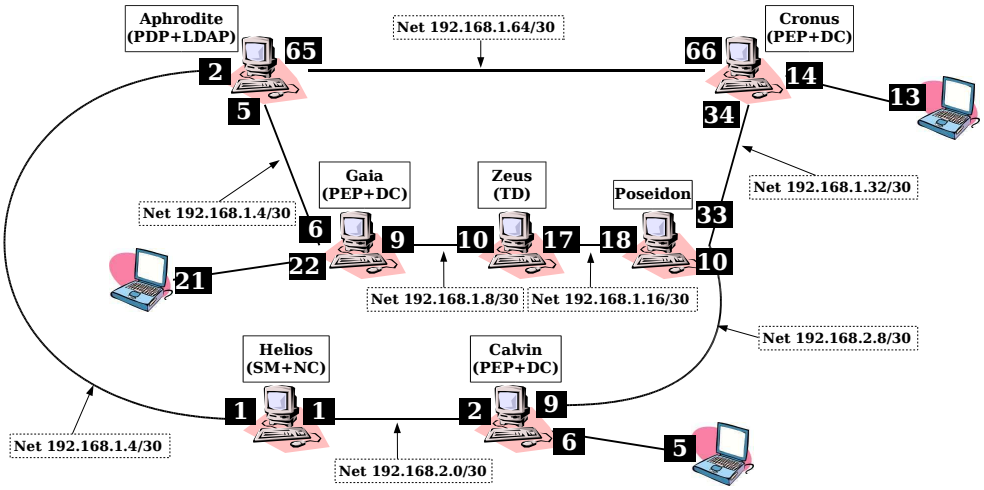

Fig. 5. Experimental testbed

packets to be sent across the LSP. This is done by specifying the virtual interface associated to the LSP as the outgoing interface. When a flow has to be mapped onto the LSP, a rule is added to the netfilter infrastructure so as the packets matching the flow specification are marked with the LSP identifier. The result is that the packets of the flow are routed on the selected LSP.

III. Routing algorithms

The Network Controller can be launched by specifying the routing algorithm to be used. The following algorithms have been implemented and can be evaluated:

Shortest-inverse: Each link is assigned a cost equal to the inverse of the available bandwidth. The selected path is the least cost one (where the cost of a path is the sum of the costs of its constituent links) among those having sufficient available bandwidth

Widest-shortest path [1]*:* It selects the path with the minimum hop count among all paths having sufficient residual bandwidth. If there are several such paths, the one with the maximum residual bandwidth is selected.

MIRA [2]*:* The idea is that a new request must follow a path that does not "interfere excessively" with a route that may be critical to satisfy a future demand. The amount of interference on a particular source-destination pair (s, d) due to routing a flow between some other source-destination pair is defined as the decrease in the maxflow between s and d. The maxflow [20] value is an upper bound on the total amount of bandwidth that can be routed between two edge nodes. The minimum interference path between a particular source-destination pair is the path which maximizes the minimum maxflow between all other source-destination pairs.

SAMCRA [3]*:* It finds the shortest path subject to m additive QoS constraints. It is based on three fundamental concepts: (a) a non-linear path length function, (b) the k-shortest path approach and (c) the principle of non-dominated paths. The path length is a function of QoS constraints and link weights.

Q-BATE [6]*:* It attempts to provide QoS guarantees while still optimizing network resources. It is based on the link weights setting strategy defined in [10]. The basic concepts of Q-BATE are look-ahead, depth-first approach and a path length definition as a function of both the available bandwidth and other additive QoS measures.

TE-DB [4]*:* It considers a delay constraint and introduces three objectives for traffic engineering: (a) reducing the blocking of flows, (b) minimizing network cost and (c) distributing network load. Objective functions (a) and (b) are transformed into constraints. TE-DB make use of TAMCRA [21], the predecessor of SAMCRA [3], to find a candidate set of k paths satisfying the set of constraints and then select the one with the shortest length according to objective (c).

SMIRA [5]*:* SMIRA stands for simple minimum-interference routing algorithms. These algorithms evaluate the interference on an source-destination pair by means of a k-shortest-path-like computation instead of a maxflow computation.

IV. Experimental results

We describe an experiment we conducted to illustrate the operation of our management system. The testbed is depicted in Fig.5. Three PCs (Gaia, Cronus and Calvin) act as ingress/egress nodes, which are connected to three laptops acting as source/destination of traffic. The other four PCs (Helios, Aphrodite, Zeus and Poseidon) act as core nodes. Aphrodite also hosts the PDP and the LDAP server, while the Service Manager and the Network Controller run on Calvin. Both the sender and receiver components of our traffic generator D-ITG run on each laptop. The clocks of the laptops are synchronized with the clock of Helios by means of the Network Time Protocol (NTP) [22]. This ensures to evaluate the packet one-way-delay with a good approximation.

Listing 2. Service Manager output

```
SM2NC_OKSENDREQ     A 192.168.1.21 192.168.1.13 275230 17 30000 20000 2239.69
NC2SM_ACCEPTED      A 192.168.1.21 192.168.1.13 275230 17 30000 20000 2239.69
SM2ITG_SENTREQ   Sender  192.168.1.21  Command  -a 192.168.1.13 -C 34.40 -c 1000  -T UDP -sp 30000 -rp 20000 -t 96000
ITG2SM_STARTFL   Sender  192.168.1.21  Command  -a 192.168.1.13 -C 34.40 -c 1000 -T UDP -sp 30000 -rp 20000 -t 96000
SM2NC_OKSENDREQ     A 192.168.1.21 192.168.2.5 358258 17 30001 20000 2569.22
NC2SM_ACCEPTED      A 192.168.1.21 192.168.2.5 358258 17 30001 20000 2569.22
SM2ITG_SENTREQ   Sender  192.168.1.21  Command  -a 192.168.2.5 -C 44.78 -c 1000 -T UDP -sp 30001 -rp 20000 -t 63000
ITG2SM_STARTFL   Sender  192.168.1.21  Command  -a 192.168.2.5 -C 44.78 -c 1000 -T UDP -sp 30001 -rp 20000 -t 63000
SM2NC_OKSENDREQ     A 192.168.2.5 192.168.1.13 341069 17 30000 20001 2756.59
NC2SM_ACCEPTED      A 192.168.2.5 192.168.1.13 341069 17 30000 20001 2756.59
SM2ITG_SENTREQ   Sender  192.168.2.5   Command  -a 192.168.1.13 -C 42.63 -c 1000 -T UDP -sp 30000 -rp 20001 -t 90000
ITG2SM_STARTFL   Sender  192.168.2.5   Command  -a 192.168.1.13 -C 42.63 -c 1000 -T UDP -sp 30000 -rp 20001 -t 90000
SM2NC_OKSENDREQ     A 192.168.2.5 192.168.1.21 366839 17 30001 20000 2753.21
NC2SM_ACCEPTED      A 192.168.2.5 192.168.1.21 366839 17 30001 20000 2753.21
SM2ITG_SENTREQ   Sender  192.168.2.5   Command  -a 192.168.1.21 -C 45.85 -c 1000 -T UDP -sp 30001 -rp 20000 -t 67000
ITG2SM_STARTFL   Sender  192.168.2.5   Command  -a 192.168.1.21 -C 45.85 -c 1000 -T UDP -sp 30001 -rp 20000 -t 67000
SM2NC_OKSENDREQ     A 192.168.2.5 192.168.1.21 301573 17 30002 20001 2051.71
NC2SM_ACCEPTED      A 192.168.2.5 192.168.1.21 301573 17 30002 20001 2051.71
SM2ITG_SENTREQ   Sender  192.168.2.5   Command  -a 192.168.1.21 -C 37.70 -c 1000 -T UDP -sp 30002 -rp 20001 -t 97000
ITG2SM_STARTFL   Sender  192.168.2.5   Command  -a 192.168.1.21 -C 37.70 -c 1000 -T UDP -sp 30002 -rp 20001 -t 97000
SM2NC_OKSENDREQ     A 192.168.2.5 192.168.1.21 310192 17 30003 20002 2628.30
NC2SM_ACCEPTED      A 192.168.2.5 192.168.1.21 310192 17 30003 20002 2628.30
SM2ITG_SENTREQ   Sender  192.168.2.5   Command  -a 192.168.1.21 -C 38.77 -c 1000 -T UDP -sp 30003 -rp 20002 -t 58000
ITG2SM_STARTFL   Sender  192.168.2.5   Command  -a 192.168.1.21 -C 38.77 -c 1000 -T UDP -sp 30003 -rp 20002 -t 58000
SM2NC_OKSENDREQ     A 192.168.2.5 192.168.1.13 418162 17 30004 20002 2577.21
NC2SM_REJECTED      A 192.168.2.5 192.168.1.13 418162 17 30004 20002 2577.21
SM2NC_OKSENDREQ     A 192.168.1.13 192.168.2.5 297720 17 30000 20001 2988.55
NC2SM_ACCEPTED      A 192.168.1.13 192.168.2.5 297720 17 30000 20001 2988.55
SM2ITG_SENTREQ   Sender  192.168.1.13  Command  -a 192.168.2.5 -C 37.22 -c 1000 -T UDP -sp 30000 -rp 20001 -t 85000
ITG2SM_STARTFL   Sender  192.168.1.13  Command  -a 192.168.2.5 -C 37.22 -c 1000 -T UDP -sp 30000 -rp 20001 -t 85000
SM2NC_OKSENDREQ     A 192.168.2.5 192.168.1.13 340137 17 30005 20003 2030.39
NC2SM_ACCEPTED      A 192.168.2.5 192.168.1.13 340137 17 30005 20003 2030.39
SM2ITG_SENTREQ   Sender  192.168.2.5   Command  -a 192.168.1.13 -C 42.52 -c 1000 -T UDP -sp 30005 -rp 20003 -t 83000
ITG2SM_STARTFL   Sender  192.168.2.5   Command  -a 192.168.1.13 -C 42.52 -c 1000 -T UDP -sp 30005 -rp 20003 -t 83000
SM2NC_OKSENDREQ     A 192.168.1.21 192.168.2.5 264725 17 30002 20002 2149.91
NC2SM_ACCEPTED      A 192.168.1.21 192.168.2.5 264725 17 30002 20002 2149.91
SM2ITG_SENTREQ   Sender  192.168.1.21   Command  -a 192.168.2.5 -C 33.09 -c 1000 -T UDP -sp 30002 -rp 20002 -t 85000
ITG2SM_STARTFL   Sender  192.168.1.21  Command  -a 192.168.2.5 -C 33.09 -c 1000 -T UDP -sp 30002 -rp 20002 -t 85000
ITG2SM_ENDFLOW   Sender  192.168.1.21  Command  -a 192.168.2.5 -C 44.78 -c 1000 -T UDP -sp 30001 -rp 20000 -t 63000
SM2NC_OKSENDSTOP    R 192.168.1.21 192.168.2.5 358258 17 30001 20000 2569.22
SM2NC_OKSENDREQ     A 192.168.1.21 192.168.2.5 429574 17 30003 20003 2935.44
NC2SM_ACCEPTED      A 192.168.1.21 192.168.2.5 429574 17 30003 20003 2935.44
SM2ITG_SENTREQ   Sender  192.168.1.21   Command  -a 192.168.2.5 -C 53.70 -c 1000 -T UDP -sp 30003 -rp 20003 -t 50000
ITG2SM_STARTFL   Sender  192.168.1.21  Command  -a 192.168.2.5 -C 53.70 -c 1000 -T UDP -sp 30003 -rp 20003 -t 50000
ITG2SM_ENDFLOW   Sender  192.168.2.5   Command  -a 192.168.1.21 -C 45.85 -c 1000 -T UDP -sp 30001 -rp 20000 -t 67000
SM2NC_OKSENDSTOP    R 192.168.2.5 192.168.1.21 366839 17 30001 20000 2753.21
ITG2SM_ENDFLOW   Sender  192.168.2.5   Command  -a 192.168.1.21 -C 38.77 -c 1000 -T UDP -sp 30003 -rp 20002 -t 58000
SM2NC_OKSENDSTOP    R 192.168.2.5 192.168.1.21 310192 17 30003 20002 2628.30
ITG2SM_ENDFLOW   Sender  192.168.1.21  Command  -a 192.168.1.13 -C 34.40 -c 1000 -T UDP -sp 30000 -rp 20000 -t 96000
SM2NC_OKSENDSTOP    R 192.168.1.21 192.168.1.13 275230 17 30000 20000 2239.69
```

The Topology Discovery (installed on Zeus) collects information about the topology and the configuration of the interfaces and organizes it in an XML file. This file is sent to the Network Controller, which parses it and stores the relevant information into the appropriate directories of the LDAP server.

Then, the Network Controller analyses the network topology to determine the edge nodes. Listing 1 witnesses that it succeeds in recognizing all the edge nodes. Their interfaces which are not connected to other network nodes are reported. The IP addresses of such interfaces are used to calculate the host IP addresses (assuming 255.255.255.252 as netmask) to be sent to the Service Manager. Each node is then assigned an integer identifier, as shown in Listing 1. We recall that links are unidirectional, i.e. there are a couple of links (one for each way) between each pair of connected nodes.

This demonstrative experiment illustrates the possibility to account for additive QoS constraints. Here, we consider just one QoS measure, namely delay. As shown in listing 1, the Network Controller assigns each link a QoS link weight equal to 415 milliseconds. This value has been obtained by generating a constant bit rate traffic between two directly connected nodes and measuring the maximum one-way-delay. Clearly, this is an unpretentious attempt to estimate an upper bound to the experienced delay through a link.

Once connected to the PDP, the Network Controller is ready to serve flow requests. Therefore, it sends the host addresses list to the Service Manager, which starts sending requests. For this experiment, the Network Controller makes use of Q-BATE to route flows. The Service Manager has been configured to generate 30 UDP flow requests, having a bandwidth requirement uniformly distributed between $250kbps$ and $450kbps$. The inter-arrival time between flow requests is exponentially distributed with a mean of 5 seconds, while the flow duration is uniformly distributed between 50 and 100 seconds. The delay constraint is uniformly distributed between $2s$ and $3s$.

During the execution, the Service Manager logs all the occurring events. Some of them are explained in Table I. The output of the Service Manager related to the first dozen of flows is reported in listing 2. The syntax of the messages

Listing 3. First request served by the Network Controller

```
******************************** REQUEST RECEIVED ********************************

New flow request: S=192.168.1.21 D=192.168.1.13 B=275230 Proto=17 SrcPort=30000 DstPort=20000 TypeOp=A QoS[1]=2239.69

Selected path from Ingress LSR to Egress LSR:
5  DevId=192.168.1.6
3  DevId=192.168.1.2
4  DevId=192.168.1.34

***** Adding LSP *****
Searching for device, dn: DevId=192.168.1.6, dc=LSPTable ...NOT FOUND
Adding LSP...
Adding LSPEntry...
dn: LSPId=1, DevId=192.168.1.34, DevId=192.168.1.2, DevId=192.168.1.6, dc=LSPTable
Result: success

--- New LSP with LSPId=1 has been added ---

***** Transmitting request to create new LSP to PDP (1) *****
Operation type <Install LSP> sent to PDP
LSPId <1> sent to PDP

Waiting for Acknowledgement from PDP...
Received ACK: LSP created correctly

Searching for Filter, dn: dc=FilterTable ...
No Filter found

***** Adding Filter *****
Result: success

--- New Filter with FilterId=1 for LSP with LSPId=1 has been added ---

***** Transmitting request to create new IP Filter to PDP (1) *****
Operation type <Install Filter> sent to PDP
FilterId <1> sent to PDP

Waiting for Acknowledgement from PDP...
Received ACK: Flow mapped on LSP correctly

Sending message <A 192.168.1.21 192.168.1.13 275230 17 30000 20000 2239.69*ACCEPTED> to the Service Manager...

Link status
Edge nodes:     4  5  6
Link (0,3):  Cap:1000000.00bps  BwdAvail:1000000.00bps  QoSmetr:[415.00]
Link (0,6):  Cap:1000000.00bps  BwdAvail:1000000.00bps  QoSmetr:[415.00]
Link (1,2):  Cap:1000000.00bps  BwdAvail:1000000.00bps  QoSmetr:[415.00]
Link (1,5):  Cap:1000000.00bps  BwdAvail:1000000.00bps  QoSmetr:[415.00]
Link (2,1):  Cap:1000000.00bps  BwdAvail:1000000.00bps  QoSmetr:[415.00]
Link (2,4):  Cap:1000000.00bps  BwdAvail:1000000.00bps  QoSmetr:[415.00]
Link (2,6):  Cap:1000000.00bps  BwdAvail:1000000.00bps  QoSmetr:[415.00]
Link (3,0):  Cap:1000000.00bps  BwdAvail:1000000.00bps  QoSmetr:[415.00]
Link (3,4):  Cap:1000000.00bps  BwdAvail:724770.00bps   QoSmetr:[415.00]
Link (3,5):  Cap:1000000.00bps  BwdAvail:1000000.00bps  QoSmetr:[415.00]
Link (4,2):  Cap:1000000.00bps  BwdAvail:1000000.00bps  QoSmetr:[415.00]
Link (4,3):  Cap:1000000.00bps  BwdAvail:1000000.00bps  QoSmetr:[415.00]
Link (5,1):  Cap:1000000.00bps  BwdAvail:1000000.00bps  QoSmetr:[415.00]
Link (5,3):  Cap:1000000.00bps  BwdAvail:724770.00bps   QoSmetr:[415.00]
Link (6,0):  Cap:1000000.00bps  BwdAvail:1000000.00bps  QoSmetr:[415.00]
Link (6,2):  Cap:1000000.00bps  BwdAvail:1000000.00bps  QoSmetr:[415.00]

- WAITING FOR FLOW REQUESTS FROM THE SERVICE MANAGER
```

displayed depends on whether the message is exchanged with the Network Controller or ITGSend. In case of a message sent to (received by) the Network Controller, the first letter indicates whether it refers to a flow arrival request ('A') or a flow departure request ('R'). Then, these fields follow: source host IP address, destination host IP address, bandwidth requirement (bps), protocol number, source port, destination port and delay constraint (ms). In case of a message sent to (received by) ITGSend, the IP address of the host where ITGSend is running is first reported. Then, the string used to instruct the sender about the flow to be generated follows. We note that the Service Manager is programmed to request the generation of a constant bit rate traffic with equally sized packets (1000 bytes). The '-C' option is used to specify the requested rate (packets per second), while the '-t' option enables to specify the flow duration.

We now illustrate how the Network Controller handles the first flow arrival request (listing 3). The Network Controller selects a path (lines 5–8) and searches the LSPTable directory of the LDAP server to determine whether an LSP along that path already exists. In this case, the result is obviously negative and the entries related to the new LSP are stored in the LSPTable directory (lines 10–17). Then, the Network Controller requires the PDP to create such a new LSP. The PDP acknowledges the success of the operation (lines 19–24) and therefore the Network Controller moves on with the

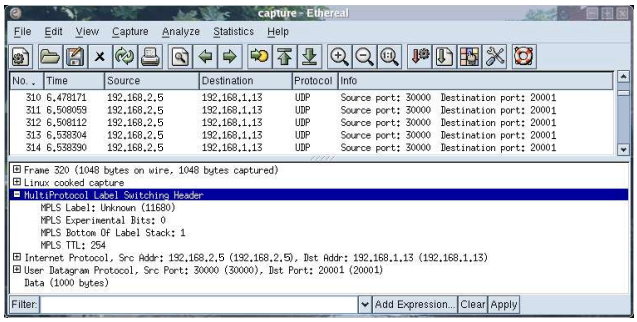

Fig. 6. Snapshot of packets captured by Ethereal

installation of the traffic filter. Again, the corresponding entries are stored in the LDAP server (FilterTable directory, lines 26–32) and then a request is sent to the PDP. The positive acknowledgement from the PDP confirms the success of the network configuration phase (34–39). The Network Controller can thus inform the Service Manager that the request has been accepted.

The Service Manager receives the message of the Network Controller announcing the admission of the first request and the successful network configuration (listing 2, line 2). Then, it instructs ITGSend to generate the corresponding flow (line 3). ITGSend acknowledges the request (line 4) and, later, the end of the transmission (line 49). The Service Manager then informs the Network Controller (line 50), which deallocates the resources reserved to the flow.

Clearly, in case there is no path satisfying the bandwidth requirement and the delay constraint of a flow, the flow is rejected. In our experiments, not all the flows could be admitted, as the output of the Service Manager (listing 2) witnesses (e.g., line 26).

For each flow, the corresponding traffic generated by ITGSend travels across the selected LSP. Each packet is therefore labeled at the ingress node and then subject to the label swapping mechanism at each intermediate node up to the egress. In order to verify that packets actually cross the selected LSP, we used a packet analyser (Ethereal) to *sniff* the traffic crossing the different network nodes. As an example, Figure 6 shows some labeled packets captured on Poseidon, generated by the host connected to Calvin and destined to the host connected to Cronus.

At the end of the experiment, the ITGDec utility can be used to analyse the log files stored by D-ITG senders and receivers. Bit rate, dropped packets, min/max/average delay of each generated/received flow can be evaluated. Listing 4 reports the results displayed by ITGDec when processing the log file stored by ITGRecv on Gaia. It comes out that Gaia has received three flows, for each of which detailed information are provided.

ITGDec also evaluates the average bitrate (delay, jitter and packet loss) over time intervals of the desired duration and stores the values in a text file that can be easily read by

Listing 4. Sample ITGDec output

```
------------------------------------------------------------
Flow number: 2  from 192.168.2.2:30001  ---> To  192.168.1.21:20000
------------------------------------------------------------
Total time                =     66.976796 s
Total packets             =          3072
Minimum delay             =      0.000169 s
Maximum delay             =      0.106901 s
Average delay             =      0.001427 s
Delay standard deviation  =      0.006231 s
Average jitter            =      0.001086 s
Bytes received            =       3072000
Average bitrate           =    366.933049 Kbit/s
Average packet rate       =     45.866631 pkt/s
Packets dropped           =             0
------------------------------------------------------------
Flow number: 3  from 192.168.2.2:30002  ---> To  192.168.1.21:20001
------------------------------------------------------------
Total time                =     96.977636 s
Total packets             =          3657
Minimum delay             =      0.000183 s
Maximum delay             =      0.116835 s
Average delay             =      0.007324 s
Delay standard deviation  =      0.009277 s
Average jitter            =      0.004019 s
Bytes received            =       3657000
Average bitrate           =    301.677801 Kbit/s
Average packet rate       =     37.709725 pkt/s
Packets dropped           =             0
------------------------------------------------------------
Flow number: 4  from 192.168.2.2:30003  ---> To  192.168.1.21:20002
------------------------------------------------------------
Total time                =     57.988307 s
Total packets             =          2249
Minimum delay             =      0.000144 s
Maximum delay             =      0.096688 s
Average delay             =      0.001384 s
Delay standard deviation  =      0.006034 s
Average jitter            =      0.001145 s
Bytes received            =       2249000
Average bitrate           =    310.269448 Kbit/s
Average packet rate       =     38.783681 pkt/s
Packets dropped           =             0
------------------------------------------------------------
**********  TOTAL RESULTS   **********
------------------------------------------------------------
Number of flows           =             3
Total time                =    102.345971 s
Total packets             =          8978
Minimum delay             =      0.000144 s
Maximum delay             =      0.116835 s
Average delay             =      0.003818 s
Delay standard deviation  =      0.090103 s
Average jitter            =      0.002353 s
Bytes received            =       8978000
Average bitrate           =    701.776526 Kbit/s
Average packet rate       =     87.722066 pkt/s
Packets dropped           =             0
------------------------------------------------------------
```

programs like octave, matlab and gnuplot. By using octave, we produced the plot shown in Figure 7. Such plot represents the bit rate of each flow generated by Calvin during the experiment.

The demonstrative experiment described in this section has illustrated the capabilities of our automated manager. We have shown that limited manual configuration is needed, as most of the work is automatically done. Also, we have provided instruments to easily analyse the results of the experiments. We believe that interesting results can be drawn from a thourogh performance evaluation of routing algorithms.

V. Conclusion and Future Work

We presented an automated manager for experimenting with traffic engineering and QoS routing algorithms. We implemented all of its component in a Linux testbed and

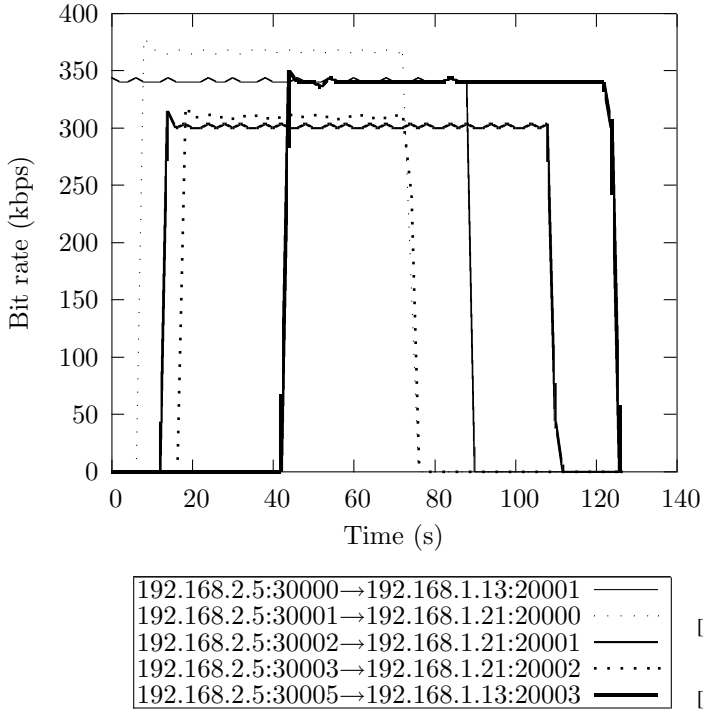

Fig. 7. Traffic generated by Calvin

used standard protocols for their interactions. This paper is mainly devoted to the description of the components of our automated manager. An experiment carried out in our testbed has been also illustrated to demonstrate the operation of our architecture. A detailed performance comparison of the implemented routing algorithms will follow. Such analysis will enable to compare them from different viewpoints: the ratio of admitted requests, the network throughput and the QoS delivered to the flows.

ACKNOWLEDGMENT

Research outlined in this paper has been supported by the European Union under the NETQOS Project FP6-033516 and the CONTENT Network of Excellence FP6-0384239.

REFERENCES

[1] R. Guerin, D. Williams, and A. Orda, "QoS routing mechanisms and OSPF extensions," in *Proc. Globecom*, 1997.

[2] K. Kar, M. Kodialam, and T. Lakshman, "Minimum Interference Routing of Bandwidth Guaranteed Tunnels with MPLS Traffic Engineering Applications," *IEEE Journal on Selected Areas in Communications*, vol. 18, no. 12, pp. 2566–2579, December 2000.

[3] P. Van Mieghem and F. Kuipers, "Concepts of Exact QoS Routing Algorithms," *IEEE/ACM Transactions on Networking*, vol. 12, no. 5, pp. 851–864, October 2004.

[4] G. Banerjee and D. Sidhu, "Comparative analysis of path computation techniques for MPLS traffic engineer," *Computer Networks*, vol. 40, pp. 149–165, 2002.

[5] I. Iliadis and D. Bauer, "A New Class of Online Minimum-Interference Routing Algorithms," in *Proc. of NETWORKING 2002*, ser. LNCS 2345, 2002, pp. 959–971.

[6] S. Avallone and G. Ventre, "Q-BATE: A QoS Constraint-based Traffic Engineering Routing Algorithm," in *Proceedings of NGI 2006*. Valencia (Spain): IEEE, April 2006, pp. 94–101.

[7] P. Van Mieghem, H. De Neve, and F. Kuipers, "Hop-by-Hop Quality of Service Routing," *Computer Networks*, vol. 37, pp. 407–423, October 2001.

[8] E. Rosen, A. Viswanathan, and R. Callon, "Multiprotocol Label Switching Architecture," IETF, RFC 3031, January 2001.

[9] J. Moy, "OSPF Version 2," IETF, RFC 2328, April 1998.

[10] S. Avallone and G. Ventre, "A simple framework for QoS provisioning in traffic engineered networks," in *Proceedings of IEEE IWQoS 2006*. New Haven, CT, USA: IEEE, June 2006, pp. 279–280.

[11] J. Hodges and R. Morgan, "Lightweight Directory Access Protocol (v3): Technical Specification," IETF, RFC 3377, September 2002.

[12] D. Durham, J. Boyle, R. Cohen, S. Herzog, R. Rajan, and A. Sastry, "The COPS (Common Open Policy Service) Protocol," IETF, RFC 2748, January 2000.

[13] D. Awduche, L. Berger, D. Gan, T. Li, V. Srinivasan, and G. Swallow, "RSVP-TE: Extensions to RSVP for LSP Tunnels," IETF, RFC 3209, December 2001.

[14] S. Avallone, S. D'Antonio, M. Esposito, A. Pescapè, and S. Romano, "A Topology Discovery Module based on a Hybrid Methodology," in *Proceedings of IPS 2004*, Budapest, March 2004, pp. 25–32.

[15] F. Baker and R. Coltun, "OSPF Version 2 Management Information Base," IETF, RFC 1850, November 1995.

[16] S. Avallone, D. Emma, A. Pescapè, and G. Ventre, "Performance evaluation of an open distributed platform for realistic traffic generation," *Performance Evaluation: An International Journal*, vol. 60, no. 1–4, pp. 359–392, March 2005, special Issue on Performance Modeling and Evaluation of High-Performance Parallel and Distributed Systems.

[17] M. Li, "MPLS Traffic Engineering Policy Information Base," Internet draft, February 2003.

[18] R. Sahita, S. Hahn, K. Chan, and K. McCloghrie, "Framework Policy Information Base," IETF, RFC 3318, March 2003.

[19] INTEC, Gent University, "RSVP-TE daemon for Diffserv over MPLS under Linux," http://dsmpls.atlantis.ugent.be/.

[20] R. Ahuja, T. Magnanti, and J. Orlin, *Network Flows: Theory, Algorithms and Applications*. Englewood Cliffs, NJ: Prentice-Hall, 1993.

[21] H. De Neve and P. Van Mieghem, "TAMCRA: A Tunable Accuracy Multiple Constraints Routing Algorithm," *Computer Communications*, vol. 23, pp. 667–679, 2000.

[22] D. Mills, "Network Time Protocol (Version 3) Specification, Implementation," IETF, RFC 1305, March 1992.