Lambda Station Version 1.0

Development Release

Abstract

Lambda Station provides forwarding and admission control service to interface production local networks to advanced and R&D wide-area networks. Lambda Station has three major interfaces offered as Web Services. It has an interface for applications and remote Lambda Stations to request provisioning of alternative path for selected flows, an interface to WAN provided where required, and an interface to configure local network infrastructure. A Lambda Station deals with the last-mile problem in local network. Selective forwarding of traffic is controlled dynamically at the demand of applications and at a granularity down to the single flow. This document describes the first development release of Lambda Station software version 1.0. (LS_v1.0). It includes the description of design principles, implementation details and results of using a Lambda Station service via several R&D high-bandwidth networks such as UltraScience Net and UltraLight.

Computing Division

Fermi National Accelerator Laboratory

Physics Department

California Institute of Technology

February 2006.

ACKNOWLEDGEMENTS

The Lambda Station project is ongoing joint effort of software developers, network analysts and engineers at Fermi National Accelerator Laboratory (Fermilab) and the California Institute of Technology (Caltech). The project is funded by the Mathematical, Informational, and Computational Sciences division of the U.S. Department of Energy's Office of Science.

The list of participants in this project:

Fermilab: Andrey Bobyshev, Matt Crawford, Phil Demar, Vyto Grigaliunas, Maxim Grigoriev, Don Petravick (PI), Ron Rechenmacher.
Caltech: Harvey Newman (PI), Julian Bunn, Frank Van Lingen, Dan Nae, Sylvain Ravot, Conrad Steenberg, Xun Su, Michael Thomas, Yang Xia.

There are four major directions of the project:

- exploiting a wide-area testbed infrastructure based on high-bandwidth R&D networks UltraScience Net, UltraLight, and others, with involvement of site production networks to be able to use Lambda Station's services
- developing Lambda Station software capable of dynamic configuring of site local area networks and interfacing with advanced WANs
- adapting of existing SciDAC applications to selectively exploit advanced research networks through Lambda Station services
- researching the behavior of network aware applications and their performance characteristics in a flow-based switching environment.

CONTACT INFORMATION.

Please use the following information to contact us:

E-Mail: lambda-station-technical@fnal.gov

WWW: http://www.lambdastation.org/

DOCUMENT STATUS AND LICENSE.

The Lambda Station project is ongoing project. The information contained in this document represents working status as of the date this document is published. It is not guaranteed that provided information is error-free and stable. The terms of following license is fully applied to this document itself and to any software described in this document.

INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Liabilities of the Government: This software is provided by URA, independent from its Prime Contract with the U.S. Department of Energy. URA is acting independently from the Government and in its own private capacity and is not acting on behalf of the U.S. Government, nor as its contractor nor its agent. Correspondingly, it is understood and agreed that the U.S. Government has no connection to this software and in no manner whatsoever shall be liable for nor assume any responsibility or obligation for any claim, cost, or damages arising out of or resulting from the use of the software available from this server.

Export Control: All documents and software available from this server are subject to U.S. export control laws. Anyone downloading information from this server is obligated to secure any necessary Government licenses before exporting documents or software obtained from this server.

Table of Contents

INTRODUCTION

This document describes the initial development (or experimental) release of a Lambda Station software version 1.0 (LSv1.0). The main goal of this release is to investigate design principles, test and evaluate proposed interfaces and to demonstrate a system that supports the full functional cycle of admission control and forwarding of application's traffic by request Lambda Station service, interaction between applications and Lambda Stations, dynamic configuring of local area networks on-demand of applications. A testbed built with LSv1.0 system involved the components of Fermilab and Caltech production networks and R&D UltraScience and UltraLight networks. In this documents we also include the results of using a Lambda Station service between test clusters at Fermilab and Caltech, and demonstration during SuperComputing 2005.

Lambda Station v1.0 is implemented in PERL. The services are accessible via SOAP protocol, however no great efforts were made yet for interoperability with other platforms.

TERMINOLOGY

ABBREVIATIONS

Below is the list of abbreviations used in this document.

- LS – a Lambda Station
- LSI – Lambda Station Interface
- LSC – a Lambda Station Controller
- PERLRE - Perl run-time environment
- SRM – Storage Resource Manager

CONCEPTS

- ***Flow*** - is a stream of IP packets with the same attributes such as source/destination IP addresses, protocols, ports and TOS (DSCP)
- ***Policy Based Routing (PBR)*** – is a technology to forward(route) IP packets to specified destination based on criteria other then destination IP address. A variety of criteria could be deployed, e.g. source IP address, protocol port ranges, DSCP and other or their combination.

- ***PBR-Client*** – one or more end systems that source and sink traffic flows that can be subject to policy based routing.
- ***Topology*** - is a network infrastructure that traffic will across to the given network destination
- ***Lambda Station*** - is a distributed control system to switch forwarding path of traffic on-demand of applications on per flow basis. A Lambda Station is dealing with the last-mile problem in local network. It is aimed to provide service for existing production use computing facilities
- ***Network (Lambda Station) Aware*** applications - application that can inquire and utilize certain knowledge of network characteristics and status to increase overall performance. Applications that can interact with a Lambda Stations is called a Lambda Station aware applications
- ***DSCP*** is Differentiated Services Code Point is an integer value encoded in the DS field of an IPv4 header. The DSCP is an example of traffic marking because its value corresponds with a preferred QoS as the packet traverses part of the network.
- ***DSCP tagging –*** a process adding of meaningful DSCP into IP headers of application's traffic either via native DSCP support by API or by some external tools (such as iptables in Linux).
- ***R&D Networks*** – experimental or special-purpose high-bandwidth networks that become available for many sites but that do not provide a commodity service.
- ***Requester*** – identity under which an application places a request for service.
- ***Ticket*** – Lambda Station's representation of a PBR-Client's request (or a set of connected requests).

LAMBDA STATION ARCHITECHTURE

Figure 1 shows a service based view on Lambda Station architecture. The central part of this architecture is LSI module which stands for Lambda Station Interface.

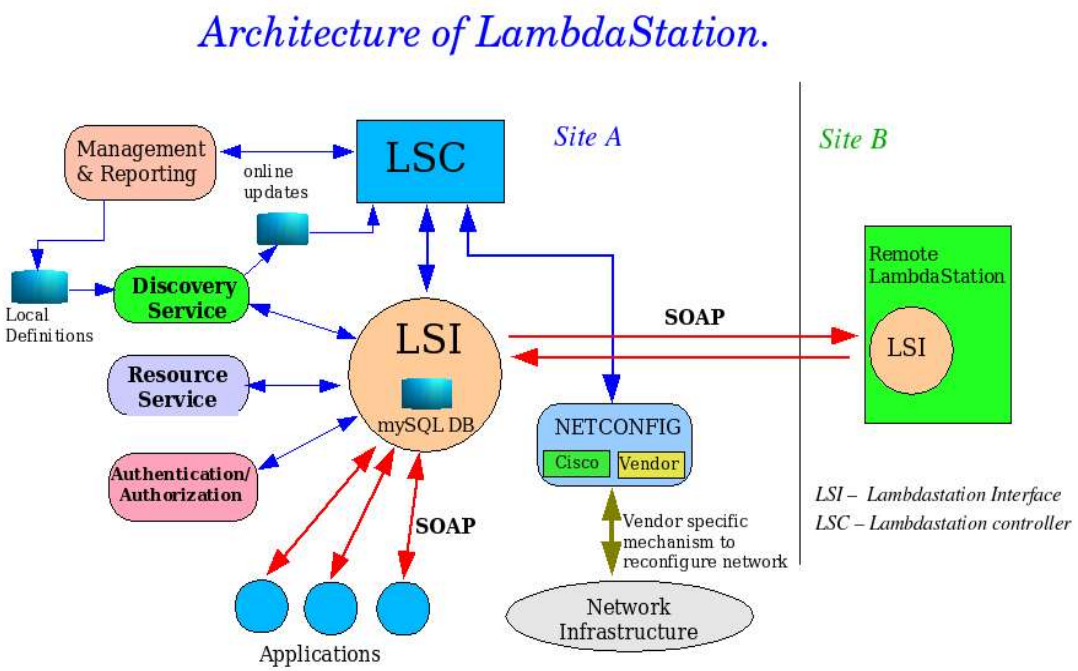


Figure 1

The LSI is a unified interface used by applications and remote Lambda Stations to request admission and forwarding service at the local site. The LSI assigns a unique ID to each request initiated by either a local application or a remote station acting on a request by a remote application. This ID is used to track the status and progress of the requested service or to cancel it. All requests go through an authentication and authorization procedure. From the LSI's perspective, requests of remote Lambda Stations are not different from requests submitted by local applications except that access to certain functions can be restricted to peer Lambda Stations. Lambda Stations have to keep knowledge about other remote Lambda Stations, their parameters and available PBR-Clients. Exchange and propagation of such knowledge is a function of the Discovery Service. To exchange information at least two Lambda Stations need to know about each to other.

A Lambda Station is not a bandwidth broker. At this time it does not support traffic policing and shaping. However, the goal of switching traffic into alternative paths is to get better performance for data transfers. That is why a Lambda Station needs to control and monitor its local resources and avoid making the effective performance of the high bandwidth path worse than the production path. The Resource Service has that responsibility. It should be mentioned that monitoring is based on requested bandwidth

and not on real-time traffic monitoring or bandwidth forecasting. Such capabilities may be added in future versions of Lambda Station. Finally, the Netconfig service is used to configure local networks dynamically for requested alternative paths.

A Lambda Station Model of a Complex Network

A Lambda Station ***deals with the last-mile problem in local networks***. It provides a means to adapt production network facilities to support access to advanced and/or research networks. Typically, a campus network can be presented by a hierarchical design with several logical layers. Such a hierarchical layout for a workgroup based approach to building campus networks is depicted in figure 2.

It consists of work group, core and border layers. Depending on the site's specific access to R&D networks for different groups of users may need to be configured at one or

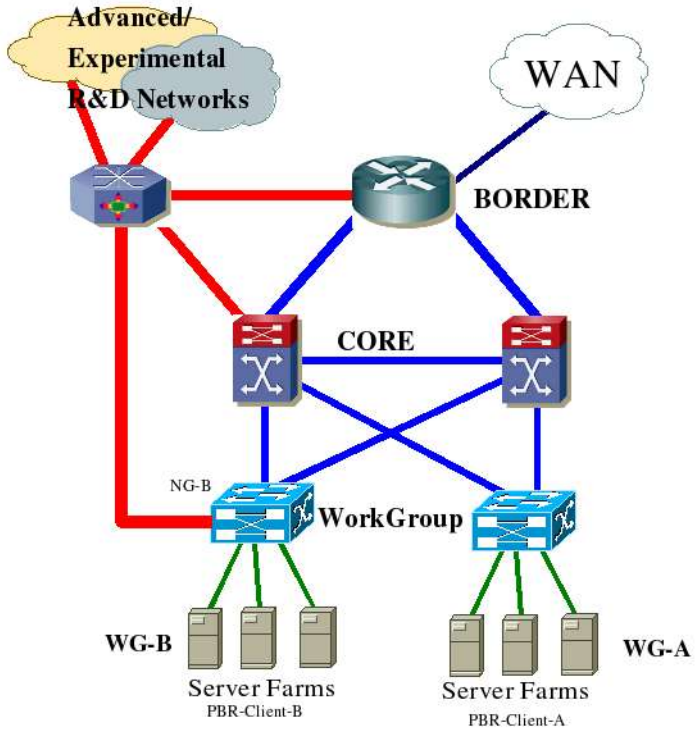


Figure 2

several layers. For the architecture in figure 2, outbound traffic of WG-B can be switched to R&D networks at the workgroup layer because it has direct connection to the R&D admission devices. In order to get incoming traffic from R&D networks forwarded by a symmetric path, the inbound rules for WG-B need to be configured at the R&D layer. The WG-A has no direct connection to R&D from its work group layer, hence

PBR rules need to be applied at the network core and R&D layer both, for inbound and outbound traffic. ***In general, work groups may require PBR rules to be applied on multiple layers of campus network for one or both directions of traffic.***

A logical model of a Lambda Station network is shown in figure 3. The main components of that model are PBR-Clients, groups of network devices and multiple external network connections. Let us assume that there are several alternative wide-area networks available to a site. In figure 3 the drawings in blue represent the regular production network topology. In green and red are alternative R&D Networks with perhaps higher bandwidth available but not intended for production or commodity use. The goal of Lambda Station is to forward traffic of PBR-Clients, designated down to per-flow granularity, toward the alternative networks, on demand from applications. In order to accomplish that goal Lambda Station will need to reconfigure one or several groups of devices with set of rules for one or both directions of traffic. Possibly different sets of rules will be applied to different groups of devices. How to group these devices depends on the site network design and involves taking into consideration physical topology of network and a need to minimize management efforts. For example, if a network administrator can reduce the number of rules or use the same set of rules for all work groups on several network layers it will certainly simplify management. As long as the same PBR rules are applied on several layers of hierarchical work group architecture Lambda Station network model can be represented by only one group of devices.

DSCP Tagging

It is desirable (but not strictly necessary) to know the criteria for selecting flows before a data transfer begins. Many applications use ephemeral transport ports that are not known before a connection is opened. They may also change dynamically during a session. It takes time to reconfigure the network, especially when two sites need to do so in synchrony. A DSCP is one of a few keys that can be specified in advance. ***A Lambda Station design does not necessarily rely on DSCP but can utilize it when available.*** While DSCP can help to solve the problem of flow matching prior to data transfer starts, it also introduces additional complexity. First, the passing DSCP is not guaranteed in the WAN. Second, in a dynamically configurable network DSCP tagging needs to be synchronized between sites and depends on the status of their networks. (See more in the Lambda Station awareness section). It is planned that two sites will be able to negotiate the best possible strategy to match flows with or without DSCP tagging. At this time a Lambda Station software does support two different modes to work with DSCP.

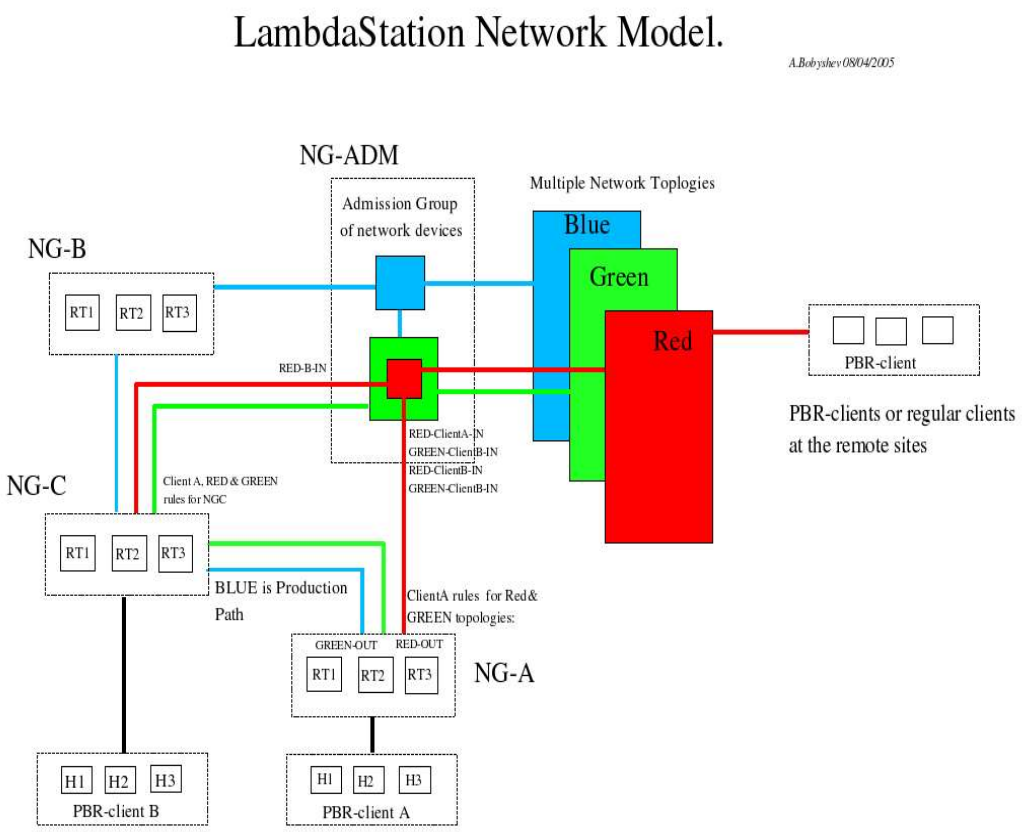


Figure 3

In the first mode, a site may choose to use fixed DSCP values to identify all traffic that will be switched by Lambda Station. Lambda Station then advises applications when to apply that DSCP value, and router configurations remain constant. This mode will typically used by sites that do not want their network devices dynamically reconfigured under Lambda Station's control.

In the second mode, a DSCP value is assigned on per ticket basis by the local Lambda Station. The same DSCP code can be used by multiple tickets as long as the source and/or destination IP addresses are used as additional flow selectors. The list of DSCP codes that can be used for assignments is configurable.

Lambda Station Aware Applications

There are several different scenarios for how a Lambda Station may control path selection for applications. Obviously, if an application has requested an alternative path it may also need to know the status and progress of its request. If the application uses

DSCP tagging to mark flows it also may need to synchronize tagging with the current status of an alternate path – especially if the site is using static network configuration. Also an application may need to monitor the status of a ticket because the remote site may decide to complete it or cancel for some reason and it may require corresponding actions at the local end of the session. Thus, close integration of Lambda Station enables applications to use the network in a more agile fashion. The Lambda Station API provides all the foregoing capabilities. It certainly introduces an additional level of complexity for applications and in many cases it cannot be fully exploited. That is why a Lambda Station does support other operational modes that are not required network awareness. Below we would like to consider a few typical use cases.

Use case: Application is capable of DSCP tagging

An application determines that it can exploit a high-bandwidth path for bulk data movement between two sites. It places a request to the local lambdastation. In the request the application can specify characteristics of the flows to be rerouted, start and end times, desired bandwidth, and other data. All requests go through authentication and authorization phases. If the request is accepted the ticket ID will be returned to the application. This ID can be used to monitor the status of the ticket. Using information from the created ticket, the Lambda Station controller contacts the LS at the remote site and tries to create a corresponding ticket. If successful, and both Lambda Stations agree on parameters of data movement and flow matching, it will receive the remoteID. The next step is configuring the network infrastructure for PBR. Each Lambda Station is responsible only for configuring its own site and monitoring status of the remote site. The application keeps checking status of ticket by localID. When the network is configured successfully it starts DSCP tagging its packets, which results in traffic being forwarded over in alternate path. If at any point there is an error, or the ticket is canceled for some reasons,the network will be reconfigured into its original state.

lsiperf – a sample Lambda Station aware application

As an example of Lambda Station aware application we wrote a wrapper for the well-known `iperf` network performance measurement tool. Lsiperf starts the usual iperf. In background it starts a Lambda Station client process which places a ticket request for an alternate path and watches its progress. If the path is established it starts DSCP marking of iperf's packets if DSCP tagging was requested. It also performs some other actions corresponding to ticket's status. For example, if the ticket is canceled it stops tagging.

Authentication and Authorization

A Lambda Station relies on the authentication schemes of its operating environment and the framework used for integration of its components. Lambda Station's SOAP Server is built on the Apache 2.55 HTTP. All authentication schemes supported in Apache 2.55 can be used with LS software. Lambda Station v1.0 uses basic (password) authentication over SSL or X.509 client certificates.

Authorization rules control access to certain functions based on the identity of the requester.

Currently there are two defined privileges:

- new ticket operations (alias `new`) which allows the requester to create, complete, cancel and modify tickets
- join privilege (alias `join`) allows joining a new request to an opened ticket. Ticket is opened if it is in one of the following states: `bookedLocal`, `booked`, `active` (see section Ticket's States).

A requester could be authorized to create and/or join tickets based on srcSite, dstSite and a list of PBR clients at each site. Authorization is configured in two files, lambda.deny and lambda.allow. See lsDefs.pm for exact locations of these files. The comparison can be exact or partial. The deny list is checked first and overrides the allow list. The format of authorization files is the following:

requester:{`exact`|`match`}:*privilege*:*SiteID1*(*PBR-Client1*,...):*SiteID2*(*PBR-Client1*,...)

The keyword ***ANY*** can be used to match any patterns.

Example:

```
# Allow access to lambdastation objects
#requester:exact|match:privileges:srcSite1(client1,client2);
srcSite2(Client2...):dstSite(client1..)
#netadm@lambda.fnal.gov:exact:new,join:Fermilab(any):Caltech(CMS)
fnal.gov:match:new,join:Fermilab(any):any(any)
/DC=org/DC=doegrids:match:new,join:Fermilab(any):any(any)
```

Resource Monitoring

A Lambda Station is not a bandwidth broker. At this time it does not support traffic policing and shaping. However, the goal of switching traffic into an alternative path is to get better performance for data transfer. Hence a Lambda Station needs to control allocation of local resources to avoid creating traffic congestion. However, it is difficult

or impossible for complex applications to predict their actual consumption of bandwidth. Real-world systems experience contention for disk access, memory, and CPU, even if their ideal network behavior is well understood. The resources Lambda Station is allocating are WAN paths connected to the site network.

Definitions and description of resources.

A Lambda Station resource definition is presented in table below.

Name	***Type***	***Description***	***Example***
name	string	Name of link	StarLight10G
slotLength	integer	time quantum (in seconds) for path allocations	900 for 15min interval
futureSlots	integer	how many slots to reserve for advance BW reservations	2000
pastSlots	integer	how many slots of history to keep	33600
BWout	string	BW of outbound link, units K-Kbps, M-Mbps, G-Gbps	10G, 622M, 10000000
BWin	string	BW of inbound link, units K-Kbps, M-Mbps, G-Gbps	10G, 622M, 10000000
maxBWout	integer	percentage allowed over subscription for Bwout on per slot basis	25
maxBWin	integer	the same as above but for inbound traffic	25
slotsOverMax	integer	percentage allowed oversubscribed slots per request	20

A resource's status is described by three tables for each traffic direction (BWin, BWout):

- reserved – reserved BW by all granted requests
- requested – all requested BW if all requests are granted, links with infinite BW
- actual – real-time monitoring and forecasting ***(not yet in use)***

Specification of API for resource monitoring service.

- allocate

- deallocate
- updateResource
- showResource
- reserved2sql

allocate/deallocate – Methods to request BW allocation or release

Input parameters (named):

Name	*Type*	*Description*
BWout	string	requested outbound BW
BWin	string	request inbound BW
startTime	integer	unix time
endTime	integer	unix time

Output:

True if granted or successfully released

False if no resource available

reserved2sql – store operational table in SQL database

showResource – print operation tables to STDOUT

Flow Descriptions

The term ***flow*** is often used in context of this document. Its definition was given in the Terminology section. Flow information needs to be moved among different components of Lambda Station software and exchanged with a remote Lambda Station. Two different formats for representing flows are in use by Lambda Station software. The first format is used to define PBR-Clients, to exchange flow information between applications and the local Lambda Station, and between Lambda Stations. In that format flows are defined by a structure of several named strings describing source and destination IP blocks, transport protocol and ports, and DSCP codes:

srcIP => “SRC_CIDR_BLOCK1, SRC_CIDR_BLOCK2,...”,

srcPort => “protocol1 [operator1 srcPorts1], protocol2 [operator2 srcPorts2],... ”

dstIP => “DST_CIDR_BLOCK1, DST_CIDR_BLOCK2,...”,

dstPort => “protocol1 operator1 dstPorts1, protocol2 operator2 dstPorts2,... ”,

dscpOut => code,

dscpIn => code.

Example:

srcIP => [“131.225.2.1/24, 131.225.252.0/25”],

srcPort => ['tcp range 5000-6000','udp range 3000-3020', 'tcp eq 35'],

dstIP => [“131.215.2.1/23”],

dstPort => ['tcp range 5000-7000', 'udp range 3000-3020', 'tcp eq 35', 'udp le 2000'],

dscpOut => 1,

dscpOut => 0x16

For the purpose of network configuration and comparison of flows it is more convenient to represent it in an extended format which is a set of records like below:

Name	***Description***	***Example***
protocol	protocol	tcp,udp
srcBase	source network	131.225.2.0
srcWild	source wildcard, or reverse netmask	0.0.0.255
srcOperator	operator applied to source ports	range, le, lt, ge, gt, eq
srcPort	source ports if defined for selected protocol, range of port is described as minPort-maxPort	2000 - 3000
dstBase	destination base network	131.215.207.0
dstWild	destination reverse netmask	0.0.0.127
dstOperator	operator applied to destination ports if applicable	range, le, lt, ge, gt, eq
dstPort	destination ports if defined for selected protocol, range of port is described as minPort-maxPort	5000 - 6000
dscpOut	DSCP code in outbound traffic	4
dscpIn	DSCP code in inbound traffic	6

Thus, when converted, one record in the first format will result in multiple records with consecutive indexes in the second format.

LAMBDA STATION INTERFACE SPECIFICATION

This section provides functional specification of the Lambda Station interface. It does not mean to be a programming language specific that is why we need to agree on definition of some basic data types. An XML Schema data types specification will be taken as basis. Also we will be using built-in primitive or derived data types and leave constraining facets as implementation details. Main data types used in description of this specification are:

- integer
- boolean
- string
- enumeration
- named pattern
- token
- union
- lists

Functional calls to LSUI are divided into several groups: information, service, and internal.

Information Methods

- whoami – return identity of lambdastation.
- sayHello – a request message to test reachability of remote lambdastation
- ip2client – convert IP address into ID-string of PBR client
- checkIP4client – check whether IP belongs to specified PBR client
- getPBRid (current name getCPRGId) – return association of IP address with PBR clients IDs
- getKnownLambdas – get list of known lambdastations
- getStationParameters – get parameters of specified lambdastation
- getKnownClients – get known PBR clients at specified site
- getClientInfo – get client's parameters
- NetConfigMode – return mode for configuring of network (dynamic, static)

Here and below **$soap** is a reference to object created by SOAP::Lite to communication via SOAP protocol:

```
my $soap = SOAP::Lite
   -> uri($myStations->{$station}->{uri})
   -> proxy($myStations->{$station}->{proxy}, timeout => $HTTPTIMEOUT)
   -> on_fault(
         sub { my ($soap,$res) = @_;
          $SOAP_ERROR = ( ref $res)? $res->faultstring : $soap->transport->status . "\n";
          LogIt('SOAPFaults@lambda',"$SOAP_ERROR\n",$logopt);
      });
```

A parameter named by **$lsIntf** is reference to object create to communicate with Lambda Station Interface $lsIntf = new lsui ();

whoami – *Return lambdastation identity*

Input : NONE

Output : two strings

the first string is name of site (lambdastation)

the second is IP address

Errors:

Example:

```
$arrayRef = $soap->whoami()->result;
$mylambdaName = $arrayRef->[0];
$ipaddr = $arrayRef->[1];
```

sayHello – *Request/response message to test reachability of remote site.*

Input:

name - any string

Output:

string – Hello, ***{name}.*** Where name is input name or ANONYMOUS if not defined.

Errors: NONE returned by method itself. All errors conditions are returned by SOAP transport layer

Example:

```
$arrayRef = $soap->whoami()->result;

$mylambdaName = $arrayRef->[0];

$reply =  $soap->sayHello( $mylambdaName)->result;
```

ip2client – *Determine PBR client's identity for specified IP address.*

Input:

ipaddr - a named string with IP address

Output:

a string of CSV: Site1,Client1,Site2,Client2....

Errors: If string is empty – no PBR clients are found for that address.

Example:

```
my $arrayRef = $soap->ip2client( ipaddr => $Ipaddr)->result;
```

checkIP4client – *Check whether specified IP address belongs to specified PBR client.*

Input:

ipaddr – a named string with IP address

Site - a named string with Site ID

Client – a named string with Client ID

Output:

1 (true) if specified IP address belongs to specified client at

0 (false) – if not

Errors:

NONE returned by this method itself.

Example:

```
my $res = $lsIntf->checkIP4client(
                        ipaddr => $ip,
                        Site   => $MySiteName,
                        Client => $MyClientName);
```

Service group of methods.

The service group consists of the following calls:

- openSvcTicket2
- openSvcTicket
- cancelTicket
- completeTicket
- getTicket
- getTicketStatus
- getMyRemoteID
- getFlowsSpec
- updateFlowsSpec
- updateDscp

openSvcTicket2 – *Request alternate network path for flows*

Input Parameters (all input parameters are ***named).*** Parameters of **data type** '***string'*** have multiple constrain facets: length, pattern, enumeration. See explanation below regarding required and optional parameters):

Name	***Type***	***Description***	***Example***
srcSite	string	Source site ID	Fermilab
srcClient	string	Source PBR client ID	CMS
srcIP	string	CSV-list of source IP addresses in CIDR format	"131.225.207.0/25, 131.225.207.133, 131.225.207.134"

Name	*Type*	*Description*	*Example*
srcPort	string	CSV-list of source ports in the format {**protocol operator ports**}Known operators are as defined in Cisco's named ACLs	"tcp eq 25, tcp range 5000 – 6000, udp le 32000".
dsSite	string	Destination site ID	Caltec
dstClient	string	Destination PBR Client ID	CMS-SRM
dstIP	string	CSV-list of destination IP addresses in CIDR format	"131.215.207.2, 131.225.207.3, 131.207.128/25"
dstPort	string	CSV list of destination ports (the same format as srcPort)	"tcp eq 25, tcp range 5000 – 6000, udp le 32000".
localPath	string	ID of local path	StarLight10G
remotePath	string	ID of remote path	MyFastPathToTheWorld
BWout	string	requested outbound bandwidth to reserve	5G – 5 Gigabit per second, 500M – 500 Mbps
BWin	string	requested inbound bandwidth to reserve	same as above
dscpReqOut	string	request for outbound DSCP tagging. Possible options are YES,NO, DESIRABLE	
dscpReqIn	string	request for inbound DSCP tagging. Possible options are YES,NO, DESIRABLE	
boardTime	unsigned integer	Boarding time a number of seconds since the EPOCH, a time when lambdastation will start provisioning of the requested alternative path	
startTime	unsigned integer	Start time when an alternative path needs to be ready. Application may anticipate connectivity problems between boardTime and startTime. However, in practice startTime = boardTime. Due to the discrete nature of network configuring startTime can not be guaranteed very precisely anyway	

Name	*Type*	*Description*	*Example*
endTime	unsigned integer	Time (a number of seconds since EPOCH) when ticket needs to be expired.	

Output:

Type	*Description*	*Example*
integer	The localID of ticket. If ticket with same flow is existing already it will returned its localID (join operation) otherwise new ticket will be created	
integer	corresponding ID at remote site or 0 if not yet assigned. At remote site it will be localID of corresponding ticket, and localID above will be its remoteID	
integer	actual end time. Two modes are considered: returning endTime of existing ticket or extend endTime as specified in the latest request.	

Errors: Many error conditions can occur. Errors are reported via object reference object->{error}

Comments:

It is not necessary to specify all parameters in request. Most of them can be determined automatically based on definition of PBR clients or taken as default values.

openSvcTicket – *Request alternate network path for specific flows.*

This method is an earlier form of openSvcTicket2 and can still be used. It gets all same input parameters but returns only localID assigned to the request by site lambdastation. It does not search already opened tickets for the duplicate flows.

Input Parameters (all input parameters are ***named strings***. See explanation below regarding required and optional parameters):

Name	*Description*	*Example*
srcSite	Source site ID	Fermilab

Name	*Description*	*Example*
srcClient	Source PBR client ID	CMS
srcIP	CSV-list of source IP addresses in CIDR format	“131.225.207.0/25, 131.225.207.133, 131.225.207.134”
srcPort	CSV-list of source ports in the format {**protocol operator ports**}Known operators are as defined in Cisco's named ACLs	“tcp eq 25, tcp range 5000 – 6000, udp le 32000”.
dstSite	Destination site ID	SiteA
dstClient	Destination PBR client ID	CMS-SRM
dstIP	CSV-list of destination IP addresses in CIDR format	“131.215.207.2, 131.225.207.3, 131.207.128/25”
dstPort	CSV list of destination ports (the same format as srcPort)	“tcp eq 25, tcp range 5000 – 6000, udp le 32000”.
localPath	ID of local path	StarLight10G
remotePath	ID of remote path	MyFastPathToTheWorld
BWout	requested outbound bandwidth to reserve	5G – 5 Gigabit per second, 500M – 500 Mbps
BWin	requested inbound bandwidth to reserve	same as above
dscpReqOut	request for outbound DSCP tagging. Possible options are YES,NO, DESIRABLE	
dscpReqIn	request for inbound DSCP tagging. Possible options are YES,NO, DESIRABLE	
boardTime	Boarding time a number of seconds since the EPOCH, a time when lambdastation will start provisioning of the requested alternative path	
startTime	Start time when an alternative path needs to be ready. Application may anticipate connectivity problems between boardTime and startTime. However, in practice startTime = boardTime. Due to the discrete nature of network configuring startTime can not be guaranteed very precisely anyway	
endTime	Time (a number of seconds since EPOCH) when ticket needs to be expired.	

Output:

	Description	***Example***
integer	The localID of ticket.	

The ***openSvcTicket2*** and ***openSvcTicket*** methods are most complex and important methods of the LSI. Let us follow for the steps implemented in openSvcTicket2 method for a simplest form of its call. Suppose application places request and provides only source and destination IP addresses.

1. The LSUI will try to determine PBR-Client definitions for both addresses because PBR could be applied only to these which are defined by network administrators and what there is required network infrastructure. If no PBR-Clients are found it is error .
2. It will convert flow into internal canonical form that could be easily converted into form understandable by routers. At this time we deal with Cisco routers only. It is known that basically the same form can be used for Force10Networks router.
3. It will search requests database for already opened tickets with exactly the same flow or subset. If found openSvcTicket2 method will return ID of already opened ticket. An opened ticket is a ticket in any of following states: bookedLocally, bookedRemotely, booked, active
4. If no opened ticket is found it will create a new one and returns assigned localID.

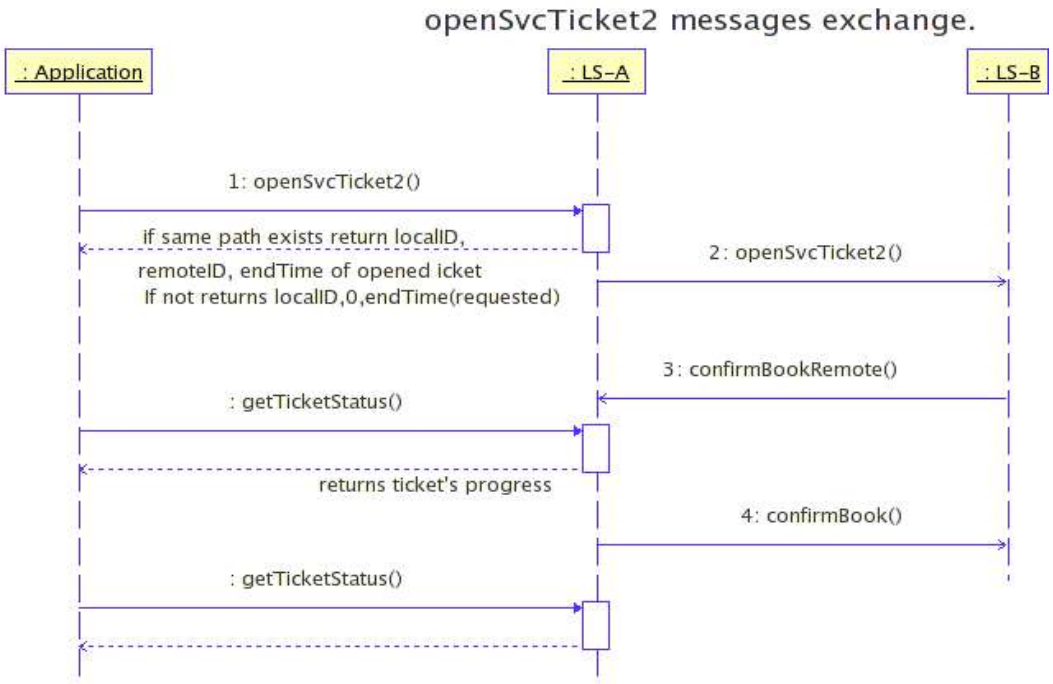


Figure 4

cancelTicket – *Method to cancel specified ticket.*

Description: *This method is updating local status of ticket to '**cancelLocal**'*

Input *:*

Name	***Type***	***Description***	***Example***
localID	integer	The local ID assigned to ticket	

Output:

1 (true) if successful

0(false) if errors

Errors:

If called via SOAP the errors will be returned via faultcode and faultstring. If called in perl run-time environments faultcode and faultstrings are accessible via object reference $self->{cancelTicket}{faultcode} and $self->{cancelTicket}{faultstring}

faultcode	***faultstring***	***Comments***
parameter.error	The localID needs to be specified for cancelTicket request	
db.error	Cannot connect to DB by $DSN	

completeTicket – *Method to complete specified ticket gracefully.*

Description: *This method is updating local status of ticket to '**activeExpire'.** From functional point of view it is very similar with cancelTicket. Lambdastation goes through same steps as in case of cancelTicket while complete it. The main reason to have two different methods is only to distinguish causes and keep track of it.*

Input :

Name	***Type***	***Description***	***Example***
localID	integer	local ID assigned to ticket	

Output:

1 (true) if successful

0(false) if errors

Errors:

If called via SOAP errors will be returned via faultcode and faultstring. If called in perl run-time environments faultcode and faultstrings are accessible via object reference $self->{cancelTicket}{faultcode} and $self->{cancelTicket}{faultstring}

faultcode	***faultstring***	***Comments***
parameter.error	localID needs to be specified for completeTicket request	
db.error	Cannot connect to DB by $DSN	

getTicket – *Method to select tickets based on various selection criterion and return only specified portion of ticket's information.*

Description: *This method selects tickets based on m*ultiple selection criteria combined by logical AND operation. At least one criterion needs to be specified. Tickets are described by multiple fields. By optional parameter header only certain fields can be selected.

Input (all input parameters are named):

Name	***Type***	***Descriptions***	***Default/Comments***
header	list of strings	Select data fields to be returned	if not specified then the full ticket's header will be taken. The full header is @TICKET_HEADER in lsDefs.pm
localID	integer	The **localID** of ticket to return information about.	If specified all other selection criterion are ignored. No Default.
localStatus	list of strings	select tickets that are in any of listed status.	All statuses are defined by @TICKET_STATUS in lsDefs.pm
remoteStatus	list of strings	similar as localStatus but for remoteStatus.	All statuses are defined by @TICKET_STATUS in lsDefs.pm
dstSite	list of strings	select tickets for any site specified in the list	
from	unsigned integer	select tickets with boardTime (unixTime) between '**from**' and '**to**'	none
to	unsigned integer	If any of these parameters is not specified the current clock is taken	none
endTimeFrom	unsigned integer	similar with '**from**' and '**to**' but applied to endTime ticket's parameter	
endTimeTo	unsigned integer		

Name	*Type*	*Descriptions*	*Default/Comments*
age	unsigned integer	Select tickets with '***boardTime***' created less then '***age***' seconds ago. If any of '***from*** ***and*** ***' to'*** is specified then '***age'*** is ignored.	none

Output: reference to array of arrays of values.

The first array (index 0) will be the list of fields returned for every tickets. It duplicates input ***header*** parameter, except it may add mandatory **localID** field. All following arrays will return values in order specified by actual header of all selected tickets. If no tickets are selected the reference will be to empty array.

Errors:

For SOAP calls errors are reported via ***faultcode*** and ***faultsring*** mechanism. If lsui.pm interface is used directly in PERLRE these errors are reported via object reference $self->{getTicket}{faultcode} and $self->{getTicket}{faultstring}

Example:

```
my $refAofA = $lsIntf->getTicket(
                        dstSite     => [$remoteStation],
                        localStatus => ['bookedLocal'],
                       from => time() - 300 );
print "There is(are) ", $#{@$refAofA}, " ticket(s) for remoteSite $remoteStation\n"
                                              if ($options{debug});
next if ($#{@$refAofA} <= 0);
```

getTicketStatus – *Method to return status of ticket specified by its localID.*

Description: *This method is a shortcut for getTicket method described above to return only status information for specified by localID ticket.*

Input (named parameters):

Name	*Type*	*Description*
localID	integer	The localID of ticket.

Output:

A string with localStatus. The list of known statuses is specified by @TICKET_STATUS in lsDefs.pm.

Errors:

For SOAP calls errors are reported via ***faultcode*** and ***faultsring*** mechanism. If lsui.pm interface is used directly in PERLRE these errors are reported via object reference $self->{getTicketStatus}{faultcode} and $self->{getTicketStatus}{faultstring}

faultcode	*faultstring*	*Comments*
parameter.error	The localID needs to be specified for getTicketStatus method	
parameter.error	Unknown status of ticket	
db.error *	Cannot connect to DB via specified DSN	

- - not yet implemented.

getMyRemoteID – *Method to return remoteID associated with local ticket.*

Description: This method is a shortcut for getTicket method to return ID of ticket at remote site associated with ticket's local ID.

Input (named parameters):

Name	*Type*	*Description*
localID	integer	The localID of ticket.

Output:

Parameter	***Type***	***Comments***
remoteID	integer	Returns remoteID. 0 – valid remoteID which means it is not yet assigned by remote Lambda Station

Errors:

For SOAP calls errors are reported via ***faultcode*** and ***faultsring*** mechanism. If lsui.pm interface is used directly in PERLRE these errors are reported via object reference $self->{getMyRemoteID}{faultcode} and $self->{getMyRemoteID}{faultstring}

faultcode	***faultstring***	***Comments***
parameter.error	The localID needs to be specified for getMyRemoteID method	
parameter.error	Unknown status of ticket	
db.error *	Cannot connect to DB via specified DSN	

getFlowsSpec – *Method to return specification of flows associated with ticket.*

Description: This method returns specification of flows associated with ticket in the format used by applications.

Input:

Name	***Type***	***Descriptions***	***Default/Comments***
header	list of strings	Select data fields to be returned	if not specified then the full flows header will be taken. The full header is @FLOWSSPEC_HEADER in lsDefs.pm
localID	integer	The **localID** of ticket to return information about.	If specified all other selection criterion are ignored. No Default.

Output: list of lists of strings

Parameter	***Type***	***Comments***
actualHeader	list of strings	Returns the list of fields that were selected, usually it is a duplication of input parameter ***header***.
FlowsSpec	list of strings	Values for selected fields of flows specification

The fields that can be selected are localID, srcIP, dstIP, srcPort, dstPort, dscpOut, dscpIn. They are described in openSvcTicket2 method.

Errors:

For SOAP calls errors are reported via ***faultcode*** and ***faultsring*** mechanism. If lsui.pm interface is used directly in PERLRE these errors are reported via object reference $self->{getFlowsSpec}{faultcode} and $self->{getFlowsSpec}{faultstring}

faultcode	***faultstring***	***Comments***
parameter.error	The localID needs to be specified for getFlowsSpec method	
db.error *	Cannot connect to DB via specified DSN	

updateFlowsSpec – *Method to modify flows specification of ticket.*

Input Parameters (all input parameters are ***named).*** Parameters of datatype '***string'*** may have multiple constraing facets, length, pattern, enumeration. See explanation below regarding required and optional parameters):

Name	***Type***	***Description***	***Example***
localID	integer	The localID of ticket	
srcIP	string	CSV-list of source IP addresses in CIDR format	"131.225.207.0/25, 131.225.207.133, 131.225.207.134"

Name	*Type*	*Description*	*Example*
srcPort	string	CSV-list of source ports in the format {**protocol operator ports**}Known operators are as defined in Cisco's named ACLs	"tcp eq 25, tcp range 5000 – 6000, udp le 32000".
dstIP	string	CSV-list of destination IP addresses in CIDR format	"131.215.207.2, 131.225.207.3, 131.207.128/25"
dstPort	string	CSV list of destination ports (the same format as srcPort)	"tcp eq 25, tcp range 5000 – 6000, udp le 32000".
dscpOut	integer	DSCP code assigned for outbound traffic	
dscpIn	integer	DSCP assigned for inbound traffic. This is actually DSCP used at remote site and assigned by remote lambdastation. In campus network with complex topology we may need to know it for correct PBR of inbound traffic.	

Output:

Type	*Description*	*Example*
boolean	The result of update. True if successful otherwise is false. Detailed errors returned as described below.	

Errors:

For SOAP calls errors are reported via ***faultcode*** and ***faultsring*** mechanism. If lsui.pm interface is used directly in PERLRE these errors are reported via object reference $self->{updateFlowsSpec}{faultcode} and $self->{updateFlowsSpec}{faultstring}

faultcode	*faultstring*	*Comments*
parameter.error	The localID needs to be specified for getFlowsSpec method	
db.error *	Cannot connect to DB via specified DSN	

Example:

getDscp – *Method to return DSCPout associated with ticket.*

Description: This method returns DSCP (outbound) associated with ticket in the format used by applications.

Input:

Name	***Type***	***Descriptions***	***Default/Comments***
localID	integer	The **localID** of ticket to return information about.	If specified all other selection criterion are ignored. No Default.

Output:

Type	***Description***	***Example***
integer	DSCP for outbound traffic	

Errors:

For SOAP calls errors are reported via ***faultcode*** and ***faultsring*** mechanism. If lsui.pm interface is used directly in PERLRE these errors are reported via object reference $self->{getDscp}{faultcode} and $self->{getDscp}{faultstring}

faultcode	***faultstring***	***Comments***
parameter.error	The localID needs to be specified for getDscp method	
db.error *	Cannot connect to DB via specified DSN	

updateDscp – *Method to update either DSCPout or DSCPin or both of them for flows associated with ticket.*

Input Parameters (all input parameters are ***named).*** Parameters of datatype '***string'*** may have multiple constraing facets, length, pattern, enumeration. See explanation below regarding required and optional parameters):

Name	*Type*	*Description*	*Example*
localID	integer	The localID of ticket	
dscpOut	integer	DSCP for outbound traffic	
dscpIn	integer	DSCP assigned for inbound traffic. This is actually DSCP used at remote site and assigned by remote lambdastation. In campus network with complex topology we may need to know it for correct PBR of inbound traffic.	

Output:

Type	*Description*	*Example*
boolean	The result of update. True if successful otherwise is false. Detailed errors returned as described below.	

Errors:

For SOAP calls errors are reported via ***faultcode*** and ***faultsring*** mechanism. If lsui.pm interface is used directly in PERLRE these errors are reported via object reference $self->{updateFlowsSpec}{faultcode} and $self->{updateFlowsSpec}{faultstring}

faultcode	*faultstring*	*Comments*
parameter.error	The localID needs to be specified for updateDSCP method	
db.error *	Cannot connect to DB via specified DSN	

DIAGRAM OF TICKET's STATES.

Each request for service is associated with created ticket. In fact, a request for service

typically results (if no error conditions are detected) in creation of two tickets, one at the local site and another at the remote site. Some actions on configuring local networks neeed to be synchronized with certain level of accuracy. A LSC module is responsible for updating local information about the status of the corresponding ticket at remote site. Its activity diagram is depicted in figure

- bookedLocal – application placed request to local Lambda Station and got assigned localID.
- bookedRemote – similar to bookedLocal but request was placed by remote Lambda Station
- booked – both sites are agree on parameters of sessions, all local assignments were completed successful (e.g. DSCP) and exchange between sites
- active – local area network was successfully configured (for dynamic mode)
- canceledLocal/canceledRemote – ticket was canceled (e.g. CTRL/C signal to application) by local application or remote Lambda Station in the response to its application
- activeExpired – ticket has expired but local network is still configured
- revokedLocal – local site was configured but remote was not successful to do so within waiting idle timeout, local site needs to back off
- revoked, canceled, completed – the final ticket's state which results from revoking, canceling and graceful completion of ticket correspondently. From prospective of the final network configuration they are the same but we need to keep track of reasons causing final state.

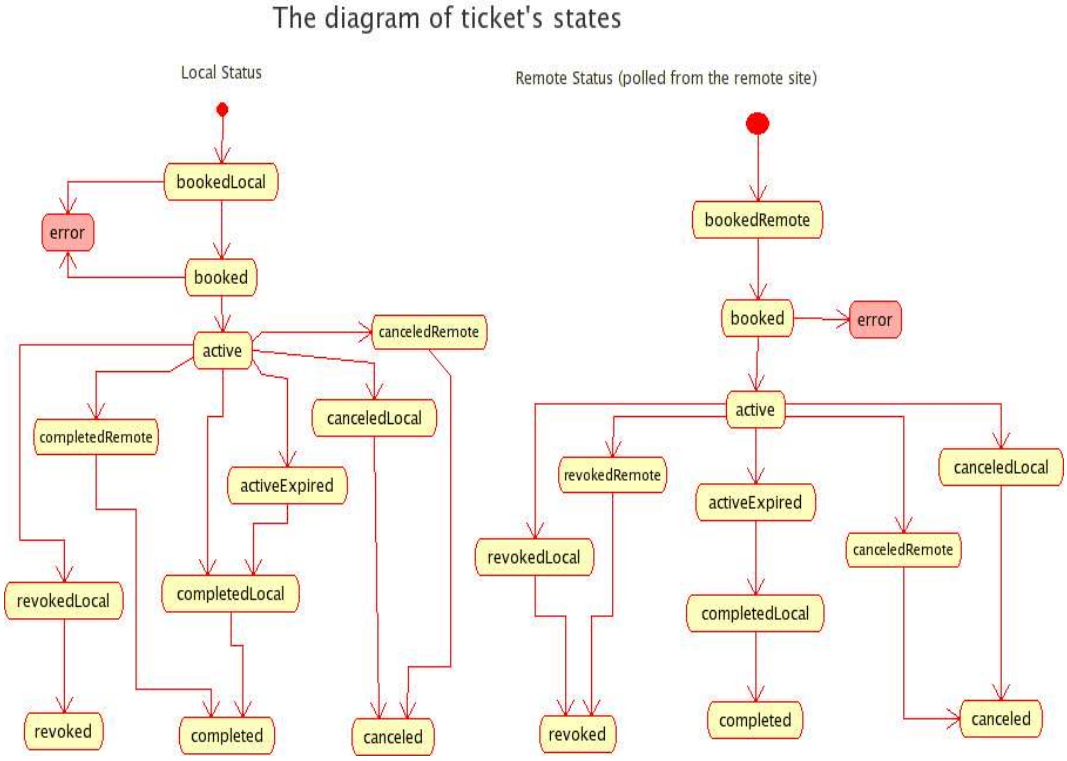

The diagram of ticket's states
Local Status
bookedLocal
error
booked
active
canceledRemote
completedRemote
canceledLocal
activeExpired
revokedLocal
completedLocal
revoked
completed
canceled
Remote Status (polled from the remote site)
bookedRemote
booked
error
active
revokedRemote
activeExpired
canceledLocal
revokedLocal
canceledRemote
completedLocal
revoked
completed
canceled

Figure 5

The activity diagram of Lambda Station controller is shown figure 6 below.

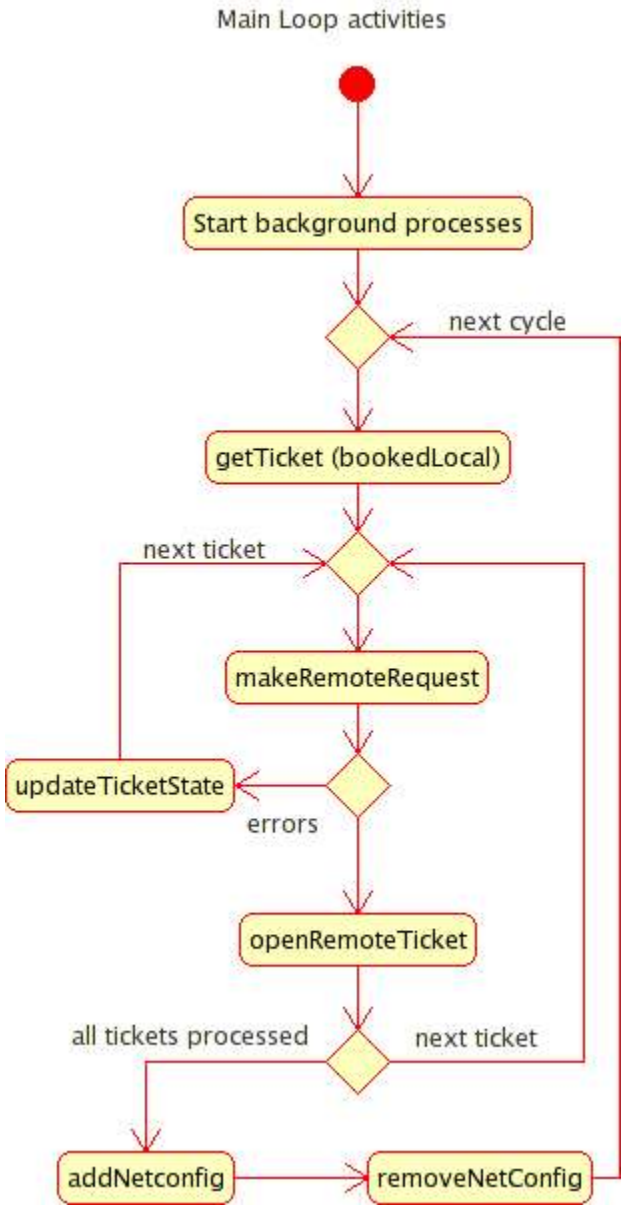


Figure 6

A dedicated process watches status of tickets and make actions corresponding to its status:

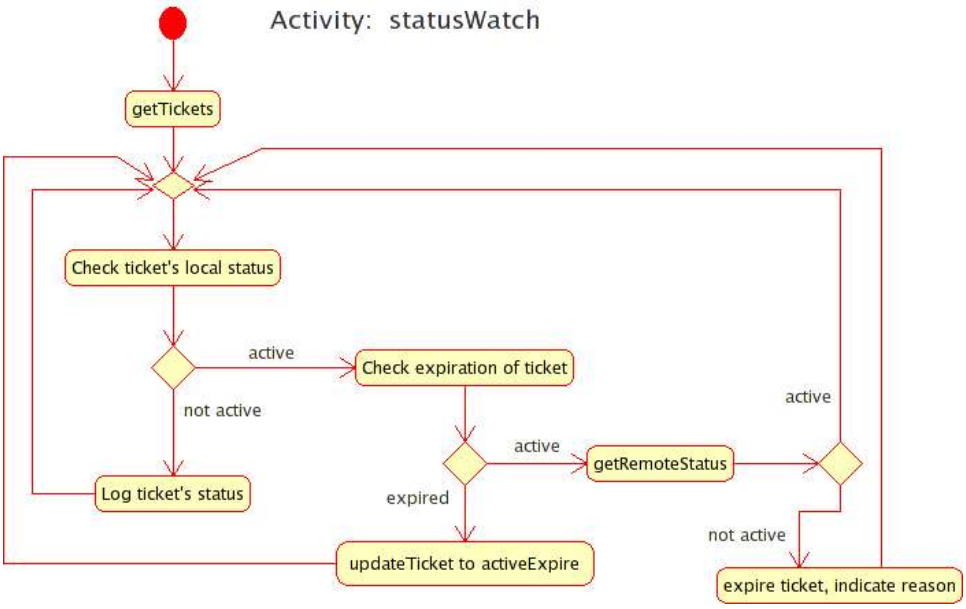


Figure 7

NETWORK CONFIGURATION MODULE.

The NetConfig module is used to dynamically modify the configuration of local network devices. This module has a vendor dependent components. At this time we support only Cisco routers with IOS version supporting sequencing type of ACLs. A policy based routing is used as technology for selective flows forwarding. There are several tasks that need to be completed to configure PBR in Cisco devices:

- interface on which PBR is applied needs to be configured with ***"ip policy route-map"*** statement
- route map needs to be configured as ordered list of match/action statements
- match criteria need to be associated with ACLs

Currently NetConfig module implements the least disruptive approach (from our point of view) to modify PBR dynamically by updating ACLs associated with match criteria. It means that basic PBR configuration needs to be prepared by network administrators based on site's specific and needs. Thus actual PBR configuration needs to be described in LS configuration file ($LAMBDA/src/lsKnownPBRCLients). Devices with same policy rules are grouped together to simplify management. A work to implement the feature to auto discover site's PBR configuration is in progress but not yet available in

the current version of NetConfig module.

External methods:

- new
- addFlows
- defFlows

Internal methods (only brief description is provided):

- showFlows – print in a canonical format current flows associated with object
- delByIndex – delete flows from canonical table by specifying index in the internal table. It is used when addFlows failed for some routers in group
- makeACL – assembly ACL from frames based on static definitions and generated from templates
- buildTemplate – generate an ACL frame from template
- TemplateError – Processing of errors in template
- cmpFlow2all - verify whether new flows are already existing
- getAllSameFlows - find all the same flows
- preserve – save current status of lsNetConfig module
- restore – restore previously saved status of lsNetConfig module.
- SaveNetConfig – save routers's configurations on TFTP server

new – Method to define flow object

Input parameters:

Name	***Type***	***Description***
routerGroup	string	ID of device group (from PBR Client definition)
routerList	string	CSV list of IP addresses of routers that need to be configured. If empty configure all routers defined by **routerGroup**
policyOut	string	CSV list of outbound policies
policyIn	string	CSV list of inbound policies
dataHeader	string	CSV list of flow field to be presented in data, If not defined then all fields

Name	*Type*	*Description*
data	integer	reference to buffer with data
snmpRW	string	SNMP-RW community string
snmpTimeout	integer	SNMP timeout
snmpRetries	integer	a count of SNMP retries before consider request as failed
netconfigState	string	a file name to store status of objects maintained yb NetConfig module
tftpserver	string	An IP address of TFTP server used to update routers
tftpdir	string	A relative location on TFTP server
reload	boolean	restore status of lsNetConfig module from file defined by netconfigState
uploadConfig	boolean	send snmpSET request to all routers that need to be configured to download their configuration updates (Cisco specific, Force10Networks should also work)

Output: True if successful, False if any errors

Errors: Errors are reported via object reference object->{error}

addFlows – Method to add flows from PBR configuration

Name	*Type*	*Description*
routerList	string	CSV list of IP addresses of routers that need to be configured. If not specified take from object definition by method new
dataHeader	string	CSV list of flow field to be presented in data, If not defined then take it from object definition
data	integer	reference to data
policyOut	string	CSV list of outbound policies. If not defined then as defined for whole object by method new
policyIn	string	CSV list of inbound policies. If not defined then as defined for whole object by method new

Output:

Errors:

delFlows – Method to delete flows from PBR configuration

Same as for addFlows

A sequencing ACL model.

The current version of Lambda Station software does support only Cisco IOS sequencing type of ACLs. The reconstruction of ACLs from active routers's configurations is not yet available. That is why the structure of ACLs used in PBR configuration needs to be defined. At large sites network administrators typically organize ACLs in some logical way to simplify their management. To address this issue rather then to use simple numbering of entries we consider an ACL as consisting of several frames. The frames can be either static or dynamic and they are stored in separate files under subdirectory $CFGDIR/Template/Name_of_ACL. In static frames ACL statements are explicitly configured. Dynamic frames are defined by templates. Administrators can defined several modes to assembly complete ACL by specifying the sequence in which frames need to put taken. Figure 8 below illustrates it.

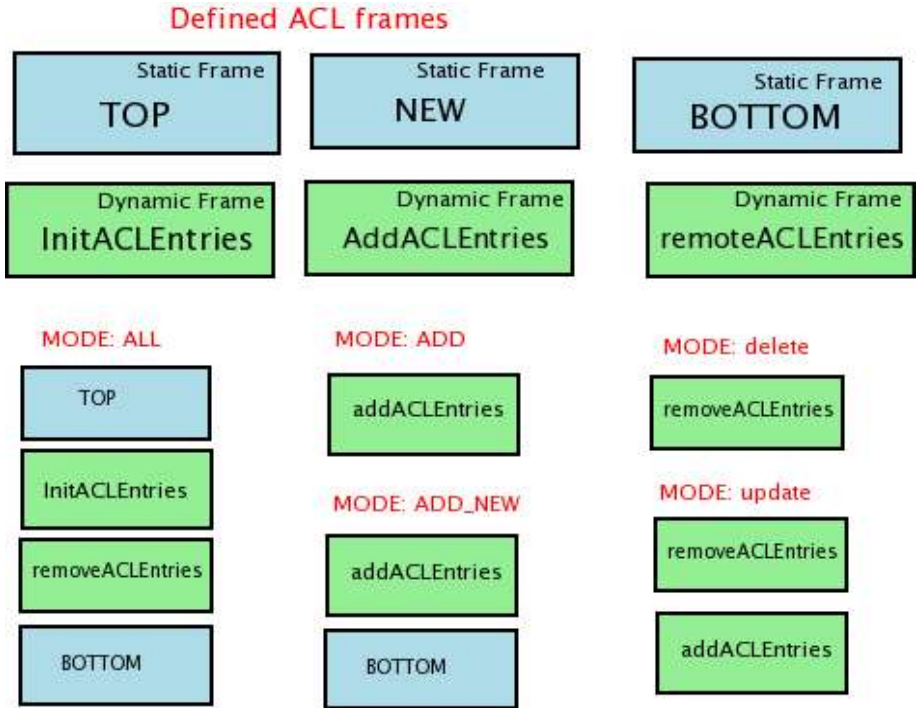


Figure 8

IMPLEMENTATION DETAILS.

The diagram in figure 9 shows implementation details of Lambda Station software version 1.0.

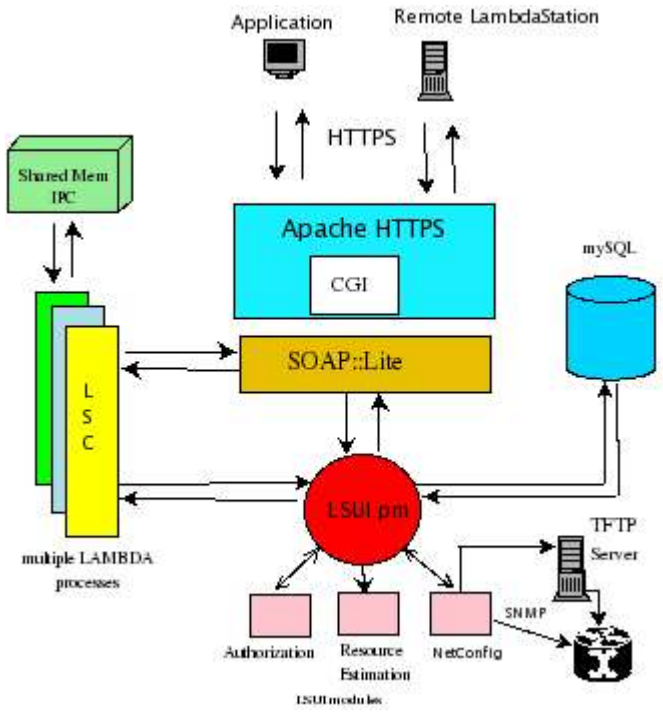


Figure 9

The current Lambda Station software runs on Linux 2.4.x platform. The shared memory support needs to be enabled if not enabled already by default. A Lambda Station consists of SOAP Server, HTTP Server, mySQL server, TFTP server, Lambda Station controller LSC (Perl program called LAMBDA which starts multiple processes to perform their specific tasks) and Lambda Station interface LSUI. Authorization, resource monitoring, network configuring and other functions are implemented in separate modules. The SOAP Server is based on Apache HTTP 2.0.55 software, SOAP::Lite PERL module and CGI script lambdaSrv.cgi. See SOAP::Lite docs for information on how to setup SOAP Server with SOAP::Lite and existing HTTP Server. We used SOAP::Lite 0.60 module although there are already newer version.

A mySQL database server is used to hold ticket's queue, maintain information about

parameters of flows and state of individual tickets. The database server does not need to be network accessible if it runs on the same host as SOAP Server. In this case all queries between SOAP server and mySQL database server will be local to host.

A TFTP server is used by NetConfig module to upload configuration changes in routers. This way of configuring is probably Cisco specific. We did not research equipment of other manufacturers but Force10Networks routers should work as well. The configuration updates for routers are generated by NetConfig module and stored on TFTP server. Then SNMP-set requests with information about location and names of updates are sent to all routers required changes. Routers will initiate a process to download configuration changes from TFTP server. In average it takes 20-40 secs to update multiple Cisco routers because IOS does not allow to do TFTP updates too often.

INSTALLATION NOTES.

Because the current version of Lambda Station software is still in development only the brief installation notes are available.

In first you have to install and configure the following software required by LSv01:

1. Linux 2.4.x, enable SysV support of shared memory if not enabled by default in your distribution. If enabled you should see device /dev/shm when type command **"df".**
2. PERL 5.8.7 (there is no a strong dependency on the version of Perl but it is a good idea to use reasonable recent one. In our setup it is 5.8.7).
3. ls_bundle_YYYYMMDDD.pm is bundle of all CPAN modules currently installed in our setup. You can use CPAN utility to add it in your host.
4. Apache HTTP Server 2.55. If you intend to use X.509 digital certificates for authentication of access you may need to apply patch for ssl_engine.c file. It seems that there is a long staying bug in Apache HTTP 2.x software that prevents method POST to work correctly with mutual authentication based on digital certificates. In some Apache's ChangeLog file a few releases back it was mentioned that this bug has been fixed. However, seems it is still there and in our case patch solved the problem. The patch is included in LSv1.0 distribution tree under Apache subdirectory.
5. Install and configure mySQL database server and create tables ***Requests*** and ***Flows.*** You can use script $LAMBDA/sql/lsReqTablesInnoDB.sql

   The names of database and tables are in lsDefs.pm. The default name of database

is ***lsdb***, table for requests is ***Requests***, and table for flows is ***Flows***.

mysql < $LAMBDA/sql/lsReqTablesInnoDB.sql

You also may want to create a table for each resource you are going to monitor. The script $LAMBDA/sql/lsResourceInnoDB.sql is a sample of such a script. The name of tables for resource monitoring is the same as in the resource definition file lsTopology.pm .

6. Authentication of access to mySQL Database Server, HTTP Server and SNMP-RW access to routers

To access mySQL Database Server, Apache HTTP Server, SOAP Server and routers with SNMP-RW it is necessary to provide userid/password information. That information is kept encrypted in file defined by $LSPASSWORDS (modify lsLocations.pm if needed). The format of line when decrypted is **selector:userid:password**. The CPAN module Crypt::Simple was used to implement it. The script $LAMBDA/src/lsEncrypt (see lsEncrypt –help) can help to maintain and view this information. You also will need to have the same password phrase for all programs that will be using same authentication data. A password phrase is stored in file ***lsSecret***. It does not matter what garbage you will put in that file, just share same one for all programs. By default, the statement: ***use Crypt::Simple passfile => "$ENV{HOME}/etc/lambda/lsSecret"*** is included in all LSv1.0 programs that required authentication data**.** You can modify it according to your preference. Make sure both files **lsSecret** and ***$LSPASSWORDS*** have as restrictive access permission as possible. In general it should be accessible only for user which runs Lambda Station software. Create userid/password for the following selectors: **htaccess, db, SNMPRW**. As mentioned above name of database and tables are defined in lsDefs.pm. In some most recent pieces of code we started to use DSN to access mySQL server defined in the XML configuration file $CFGDIR/lsMyLambdaConfig.xml (see section <Database> </Database>). Make sure these settings are consistent with these which are in lsDefs.pm).

7. Configure SOAP Server. To build up a SOAP Server we used SOAP::Lite and Apache HTTP Server 2.55. See SOAP::Lite instructions how to setup SOAP Server by using existing HTTP server. Copy $LAMBDA/cgi/*.cgi files in location for CGI scripts according to configuration of your HTTP server.

EXAMPLE OF A LAMBDA STATION SETUP.

In this section we will guide you through the major steps to setup Lambda Stations for two sites and start communication between them. Here is what we will try to setup:

SiteA, PBR Clients : CMS, WORKER. An alternative network topology is starlight10G

SiteB, PBR Clients: STORAGE, SRM. An alternative topology is highway.

We will assume that mySQL, Apache 2.55, Perl, all required CPAN modules are installed and configured. Lambda Station software is extracted in some directory defined by environment variable $LAMBDA.

In this example location for CGI scripts on the Lambda Station of SiteA will be ***~netadmin/public_html/ls*** and ***~netadmin/secure_html/lssl*** on Lambda Station of SiteB.

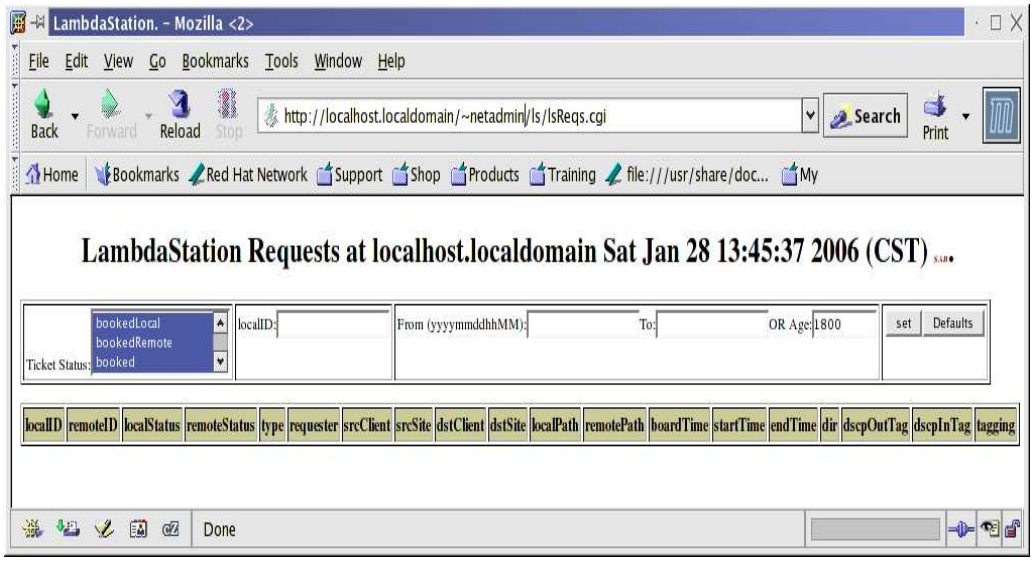
The userid ***dbadmin*** will be used to access SQL tables in ***lsdb*** database.

```
mysql> show GRANTS for 'dbadmin'@'localhost.localdomain';
+-----------------------------------------------------------------------------------------------------------------------+
| Grants for dbadmin@localhost.localdomain                                                                              |
+-----------------------------------------------------------------------------------------------------------------------+
GRANT ALL PRIVILEGES ON `lsdb`.* TO 'dbadmin'@'localhost.localdomain'
|GRANT USAGE ON *.* TO 'dbadmin'@'localhost.localdomain' IDENTIFIED BY PASSWORD
'*2E2918FDHTW71O4EY2Y3Y231' |
```

In the LS_V1.0 directory $LAMBDA/cfg there are several examples of configuration files with extension **SiteA** and **SiteB** (e.g. myLambdaConfig.xml.SiteA and myLambdaConfig.xml.SiteB). Rename it to $LAMBDA/cfg/myLambdaConfig.xml

1. In first, inspect $LAMBDA/src/lsLocations.pm whether you need to modify any parameters. Most locations are defined relatively to $LAMBDA but you may want to change some of them or modify names for password and authorization files.
2. Setup SOAP Server (see SOAP::Lite docs for more details) and placed CGI script located in $LAMBDA/cgi/lambdaSrv.cgi under ~netadmin/public_html/ls directory. You may need to edit this script for actual path as defined by $LAMBDA environment variable.
3. Copy CGI script $LAMBDA/cgi/lsReqs.cgi in the ~netadmin/public_html/ls . By using script $LAMBDA/sql/lsReqTablesInnoDB.sql create tables for requests and

flows specification. The selector **db** (in lower case**)** is used to select userid/password from encrypted file for accessing mySQL tables as described in the previous section. Use script $LAMBDA/src/lsEncrypt –verbose to see, add and modify if needed access parameters. Although there are no yet any requests in the database you should be able to use script ***lsReqs.cgi*** to access Request and Flows tables via URL http://lambdaStation_at_siteB/~netadmin/ls/lsReqs.cgi



4. Create definitions for at least two Lambda Stations in file $LAMBDA/src/lsKnownLambdaStations.pm. Use existing file as an example. At each site you will have to define two stations, local one and at least one remote station. These stations will need to know how to reach each other. At each station you can add information about other stations not known to all other remote stations. Once communication channel between two stations is established they will exchange information about all known other stations and PBR clients.
5. Create file with definitions of PBR clients. The existing file $LAMBDA/src/lsKnownPBRClients.pm provides example of such definition. Keep in mind that if you are going to switch path of selected flows generated by applications on these cluster the definitions should reflect topology of the real network. Network infrastructure should be capable of doing policy based routing for that cluster. In first try to generate dynamic updates for network without actual uploading it in routers. It is safer if you do not have yet correct snmpRW in encrypted password file to prevent that changes are placed in routers accidentally. Certainly if uploaded ACLs

are not involved into active configuration they should make no harm.

6. Start $LAMBDA/src/lambda –debug –verbose and check HTTP Server's log files for any errors. You may see something like that:

```
$VAR1 = {
        'sc05' => {
                    'active' => 0,
                    'name' => 'sc05',
                    'proxy' => 'https://a33.302.sc05.org/~netadmin/ls/lambdaSrv.cgi',
                    'uri' => 'lsui',
                    'ipaddr' => '140.221.183.33'
                  },
        'SiteA' => {
                     'active' => 0,
                     'name' => 'SiteA',
                     'proxy' => 'http://10.225.247.162/~netadmin/ls/lambdaSrv.cgi',
                     'ipaddr' => '10.225.247.162',
                     'uri' => 'lsui'
                   },
```

As you can see SiteA does not appear to be active which typically indicates some access permission problems. A periodical gathering of information about lambda stations and their parameters goes via SOAP calls. Check $LAMBDA/logs for additional messages in LS log files.

Now try to place request by $LAMBDA/tapi/lsreq test program. See available options lsreq –help. Rather then specify multiple input parameters you may want to modify defaults in Perl code directly. Open ***lsReqs.cgi*** in web browser to see information about requests. At this point you might expect errors because of conflicts in definitions or **remote.Ticket** error because there is no yet a second Lambda Station to communicate to. Additionally you may want to use other test scripts located at $LAMBDA/tapi such as lsGetKnownPBRClients,lsGetKnownLambdas, lsGetKnownLinks.

To set up a second lambda station follow the same steps as above. Once two Lambda Stations are configured and know how to communicate each to other (lsKnownLambdaStations.pm) you should see periodical exchange of information about known PBR clients and Lambda Stations (if configured more then two) known at each site.

1138507794 01/28/2006 22.09:54|updateClients@SiteA|Known clients at the Fermilab are DCN,CMS,CDF.

1138507801 01/28/2006 22.10:01|updateClients@SiteA|Known clients at the Caltech are CMS.

1138507802 01/28/2006 22.10:02|statusWatch|Count of concurrent synRemStatus processes is 0.

1138507806 01/28/2006 22.10:06|updateClients@SiteA|Skipping inactive station sc05

1138507807 01/28/2006 22.10:07|updateClients@SiteA|Known clients at the SiteA are WORKER,CMS.

1138507811 01/28/2006 22.10:11|updateClients@SiteA|Known clients at the SiteB are SRM,STORAGE.

1138507815 01/28/2006 22.10:15|updateClients@SiteA|Skipping inactive station mylaptop

1138507815 01/28/2006 22.10:15|main|There is(are) 0 ticket(s) for remoteSite Fermilab

1138507816 01/28/2006 22.10:16|main|There is(are) 0 ticket(s) for remoteSite Caltech

1138507816 01/28/2006 22.10:16|main|There is(are) 0 ticket(s) for remoteSite SiteB

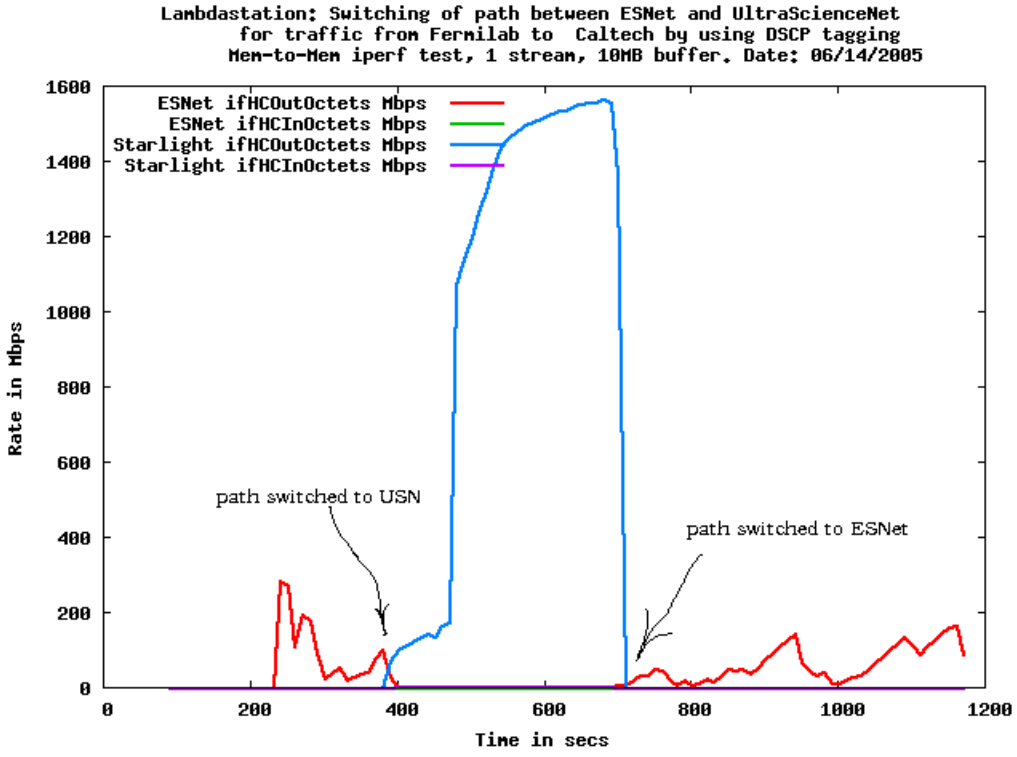
7. Place another request with $LAMBDA/tapi/lsreq script. If status of ticket goes through bookedLocal, booked states then you get communication between stations. The next step is to generate and update network configuration. If ***NetConfigMode*** is ***dynamic*** in $LAMBDA/cfg/myLambdaConfig.xml then Lambda Station will generate ACLs for PBR as defined in lsKnownPBRClients.pm. Make sure that in $LAMBDA/cfg/Template there are templates named the same as ***policy*** in PBR Client definitions. The names are case sensitive. Also if ***uploadConfig*** is not '**yes**' or not defined then Lambda Station will skip actual uploading of configuration changes in routers assuming that it is successfully completed ***(dry run mode)***. Configurations are stored under $TFTPDIR, in subdirectory called by name of device groups defined in lsKnownPBRClients. ***The subdirectories are NOT created automatically***. You need to create it. If ***NetConfigMode*** is ***static*** Lambda Station will skip generating of network configuration updates. In this case network need to be statically configured to switch flows based on DSCP tagging initiated on host site. Applications control whole switching process of flows.
8. Now you can try to test program that communicate with Lambda Station via SOAP protocol, e.g. $LAMBDA/tapi/lsSoapReq . Check parameters of communication and modify them according to your settings.

EXPERIMENTING WITH LAMBDA STATION.

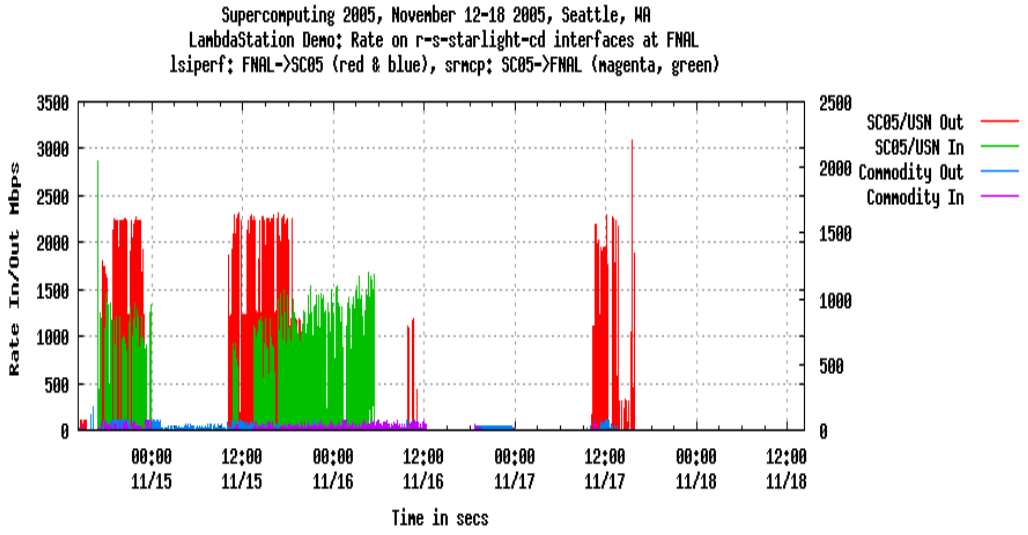
The current LS software was used to built a Lambda Station testbed to use network aware applications between Fermilab and Caltech. The ESNet was our production path

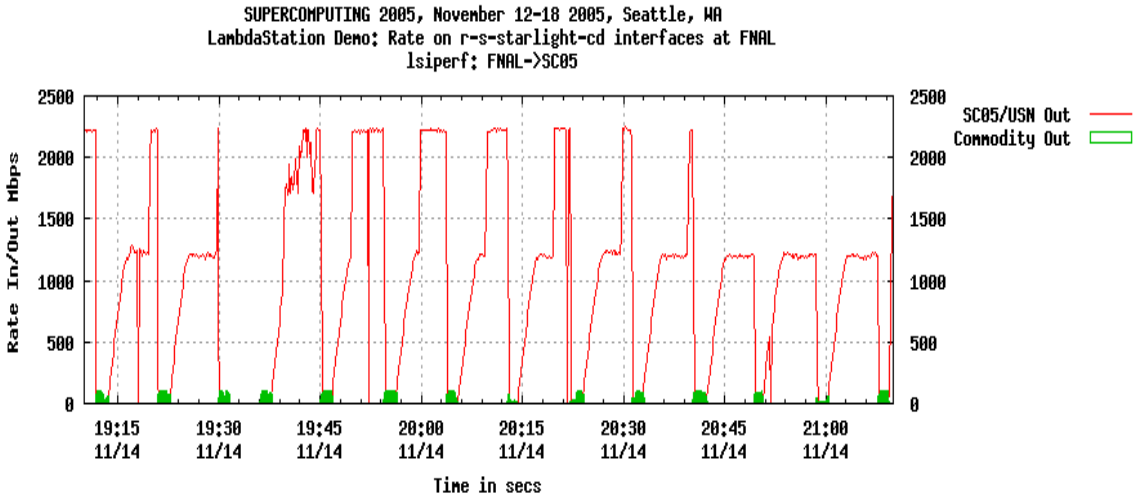
and two high bandwidth networks, UltraScienceNet and UltraLight were used as alternative network topologies. This software was also used during SuperComputing 2005 to demonstrate flow based switching between SciNet and Fermilab. Two applications were used in both experiments, lsiperf which is an example of Lambda Station aware application, and SRM with modified version of gridFTP supporting DSCP tagging.

The graph below represents the results of throughput measurements for selective switching of flows between two alternative paths. The tests were conducted between Fermilab and Caltech for memory to memory data transferring by using ***lsiperf*** tool, 1 stream, 10MB buffer size. Path switching is based on policy based routing configured at both sites for pairs of source/destination addresses marked by DSCP. A path switching is initiated on host site by turning DSCP tagging on or off for specific flows. In our tests we used ***iptables*** to do actual tagging for traffic associated with certain UID (user identifier) or PID (process identifier). The test was started via ESNet path with five streams. Throughput has reached about 350Mbps. To avoid saturation of the OC12 link at Fermilab the number of streams was reduced to only one. In about 3 mins hosts on both ends started DSCP tagging of traffic to switch forwarding via USN path, and then, in about 5 mins tagging was switched off to forward packets via ESNet path again.



The graphs below demonstrates Lambda Station control of forwarding path for two applications, lsiperf and SRM during SuperComputing 2005, November 2005, Seattle, WA. A path to Fermilab was dynamically switched between commodity Internet and a dedicated 10G link from SC05 booth in Seattle to Fermilab.





REFERENCES

1. A Lambda Station Project Web site http://www.lambdastation.org/
2. Donald L.Petravic, Fermilab, LambdaStation: Exploring Advanced Networks in Data Intensive High Energy Physics Applications, Project Proposal, *http://www.lambdastation.org/omnibus-text.pdf*
3. Phil DeMar, Donald L.Petravic. LambdaStation: A forwarding and admission control service to interface production network facilities with advanced research network paths, CHEP2004 International Conference on Super Computing, Interlaken, Switzerland, 27th September - 1st October 2004.