A Guide to User Space Routing – VLSP – The Very Lightweight Network & Service Platform

Stuart Clayman

Jan 2018

Abstract

This document gives an overview of the *VLSP* framework. We have designed a testbed called the *Very Lightweight Network & Service Platform* based on this framework. It uses a set of Virtual Routers and Virtual Network Connections to create a test environment which can easily accommodate (i) fast setup and teardown of a Virtual Router, and (ii) fast setup and teardown of a Virtual Connection. Each Virtual Router can run small programs and service elements.

The document describes the platform itself and the components in more detail. The usage and startup of the platform is presented together with the configuration options.

1 Introduction

This document gives an overview of the *VLSP* framework - sometimes called *User Space Routing* as it does everything in user-space. We have designed a testbed called the *Very Lightweight Network & Service Platform* based on this framework. It uses a set of Virtual Routers and Virtual Network Connections to create a test environment which can easily accommodate (i) fast setup and teardown of a Virtual Router, and (ii) fast setup and teardown of a Virtual Connection. Each Virtual Router can run small programs and service elements.

The platform consists of a number of virtual routers running as Java virtual machines (JVMs) across a number of physical machines. The routers are logically independent software entities which, as with real routers, communicate with each other via network interfaces. The network traffic is made up of datagrams, which are send to and from each router. The datagrams are not real UDP datagrams, but are our own virtual datagrams (called USR datagrams) which are used.

The platform has three components, the major one of which is the "Router" itself. They are complemented by a lightweight "Local Controller" which is similar to a hypervisor and has the role of sending instructions to start up or shutdown routers on the local machine and for routers to setup or tear-down connections with other virtual routers. The whole testbed is supervised by a "Global Controller". This software entity is like a combined Virtual Infrastructure Management (VIM) / Orchestrator, and has the role of a co-ordinator. That is to say, it informs Local Controllers when to start up and shut down routers and when to connect them to each other or disconnect them from each other.

Since its inception, VLSP has been used in many experimental situations that have benefited from its flexibility, adaptability, lightweightness, and scalability. VLSP achieves better simplicity and non-functional characteristics over using a hypervisor running a standard virtual machine and standard OS (i.e. improved scalability; lower resource utilization; quicker startup speed; reduced heaviness; eliminate the issue where most of the router functionality is not needed; and more networking flexibility). VLSP is fully distributed and modular. This has been achieved, as the management components as well as the host controllers and the virtual entities themselves can be deployed across any number of physical hosts interacting via REST calls.
From the experimental usage of VLSP we have determined:

What is VLSP good for?

- Testing and evaluating alternative SDN / NFV scenarios
- Mid scale tests of network software written in Java (100s and likely 1000s of virtual routers).
- Testing software robustness to "unexpected" network conditions (sudden "rude" start up/shut down exposes software deficiencies).
- Testing software robustness to unreliable networks.
- A compromise between simulation (realism questionable) and large testbed (requires many physical machines).
- Comparing simulation with testbed results

What is VLSP not *yet good for?*

- It is not optimized for forwarding performance, compared to a real router it routes packets at a slower speed and uses more overhead (i.e. due to the focus on the support of new network management and control features).
- It is difficult to support facilities and protocols that rely on maximum bandwidth calculations, e.g. traffic engineering algorithms estimating the link bandwidth.

- The direct interaction with or the driving of hardware interfaces is out not currently addressed.

1.1 Motivation

There were many motivations for designing and building the *User Space Routing* framework. These were accumulated from experience on various research projects, including RESERVOIR which investigated running services in virtual machines, and the AutoI project, which investigated the virtualization of network elements. It was found that the use of a hypervisor and the associated virtual machines did work as expected and as required, however, there were some issues that hindered various experimental situations.

We found that using a hypervisor and virtual machines added only 5% to 10% overhead to operations, compared to running the same operations in the physical machine, which is most cases was entirely acceptable. The small loss of efficiency was easily overcome by the flexibility of having virtual machines. In terms of experimental and research issues, some were general issues and others were specific to the domain. We found the following general issues:

- the number of virtual machines that can run on a physical host is limited. This can be due to the actual resources of the physical machine that need to be shared (such as the number of cores and the amount of memory available), together with the switching capabilities of the hypervisor.
- the speed of startup of a virtual machine can be quite slow. Although virtual machines boot up in the same order of magnitude as a physical host, there are extra layers and inefficiencies that slow them down. Also, if many virtual machines are started concurrently, then we observe that the physical machine and the hypervisor thrash trying to resolve resource utilization.
- the size of a virtual machine image is quite large. A virtual machine has to have a disc image which contains a full operating system and the applications needed for the relevant tasks. To start a virtual machine, the operating system needs to be booted and then the applications started. So every virtualized application needs the overhead of a full OS.

and we found the following issues that were more domain specific:

- in terms of virtual networks, and virtualized routers in particular, we observered that 95% of the router functionality we never utilized in any of the experiments that were run. Although software routers such a XORP and Quagga allow anyone to play and evaluated soft networks, the overhead of a virtual machine, with a full OS, and an application where only 2% is used, seems to be an ineffective approach for many situations.

- when trying to configure the IP networking of virtual machines and virtual routers, there are some serious hurdles. The virtual machines do not talk directly to the network, but go via the hypervisor. The hypervisor has various schemes for connecting virtual machines to the underlying network, each of which has different behaviour. In most situations where experimentation of virtual routers is required, there needs to be a large range of IP addresses available. However, this is often hard to come by. We found that the limits of addressing, the IP networking configuration, and virtual machine to virtual machine interoperability a hindrance to network topology and network flexibility.

It was felt that to make more progress in the area of dynamic and virtual networking experimentation and research, we needed to design and build a testbed that did not have these limits, but still retain virtual machine technology.

The main goals of the testbed over using a hypervisor running a standard virtual machine and standard OS are to have:

- better scalability
- lower resource utilization
- quicker startup speed
- reduced heaviness
- eliminate the issue where 95% of the router functionality not needed
- more networking flexibility

The choice was made to write our own simple router with simple service capabilities, in Java, that could run in a Java Virtual Machine (the JVM).

1.2 Benefits

The benefits of a lightweight VM that includes a simple router and the basic capabilities of a service component are:

- it is possible to run many more routers on a host
- it is easier to test scalability and stability
- it is possible do enhanced monitoring and management evaluations

- it provides a different way to do virtual networks: we can create arbitrary topologies using virtual routers
- it is possible to do more evaluations of network management and orchestration functions by not using complete routers and full services

In general, it is a more effective platform for experimenting with many aspects of virtual networks, management of virtual elements, and orchestration experiments.

The VLSP integrates and unifies management of networks, compute, and services into a single platform. The whole system, including a lightweight virtual router, was designed and built from scratch to enable flexible experimentation of virtual infrastructures. VLSP has the following advantages:

Lightweight virtual router implementation: to allow evaluation of diverse scenarios, including the testing of various management lifecycle schemes. It allows the testing of software robustness to "unexpected" network/node conditions (such as sudden start up/shut down events exposing software deficiencies), where a virtual element can disappear at run-time under management control. VLSP is a positioned between simulation (realism questionable) and large testbed (requires many servers). Lightweight routers being deployed and managed in a distributed environment, suitable for reliability and scalability tests.

Distributed infrastructure implementing logically-centralized management and control: By employing the SDN paradigm, while avoiding overflowing centralized components, to support optimizations, based on centralized decisions using a global system picture derived from an integrated monitoring infrastructure based on Lattice [**?**]. It also supports flexible and adaptable service provisioning, plus aspects beyond traffic engineering, (e.g. adaptable service deployment / operation, optimized distributed node placement etc).

Experimentation on management: to focus on the experimentation of distributed management and control components of software-defined infrastructures rather than data plane performance. Along these lines, VLSP brings the following benefits: (i) more routers can be deployed on each host, making it easier to test scalability and stability; (ii) enhanced distributed management, control and monitoring evaluations can be carried out, which is difficult to achieve in a running environment using a number of deployed data centers; (iii) arbitrary topologies can be created using virtual routers, providing a more general way to form virtual networks; (iii) suitable for both experimentation with virtual networks and lightweight virtual servers; and (iv) supports experiments with dynamic topologies (such as migratable virtual routers) and mobile extensions of the network infrastructure).

Support of integrated SDN and NFV environments: to uncover the potential of virtualized SDN solutions and NFV deployments, in a naturally integrated way, with common management and control facilities.

JVM based Runtime Environment: The evaluation of network topologies allowing hundreds or thousands of virtual nodes (routing or compute), using software written in Java. Each node resides in its own virtual machine, and so is independent of and secure from all other virtual nodes. The software execution environment, available on all nodes, allows the deployment of diverse network control components. Existing Java code can be quickly ported to run on our software router (our experience is from two or three hours, to a few days for relatively complicated software packages). Deployment of virtualized devices over heterogeneous hardware and software environments due to the very wide support of the JVM technology.

2 The Platform

In this section there is a more detailed overview of the platform itself. VLSP provides a complete environment from the *protocol stack* up to the *service management* level, including a tailor-made monitoring facility. Consequently, we are able to experiment with new complete 5G network management and control facilities over virtual networks based on our lightweight virtual entity resembling both servers and routers.

VLSP is a distributed management infrastructure that has centralized functionality and is responsible for the setup, configuration, optimization, and shutdown of network entities. It has been implemented for the purpose of testing and evaluating various aspects of managing highly dynamic virtual environments, in particular the 6 aspects of:

(i) efficient service function,
(ii) optimized service function availability,
(iii) service and VNF lifecycle automation,
(iv) service placement automation,
(v) efficient resource utilization, and
(vi) dynamic resource up/down scaling (elasticity).

VLSP is a testbed that consists of a large number of virtualized entities which execute on a number of physical machines. The entities are logically independent software entities that communicate with each other via network interfaces. The testbed has been validated for some of these aspects in previous work we have undertaken on virtualized and highly dynamic networks.

The VLSP set up has various components described in this section and include:

(i) a supervisor and experimental controller realizing the basic functionalities of an Orchestration Layer / Virtual Infrastructure Management (VIM) called the *Global Controller*;

(ii) per-host *Host Controllers*; and

(iii) the *virtualized entities*, which run inside a JVM.

VLSP is configured by the *Global Controller* running on one physical server, and the *Host Controllers* running on physical machines that host virtual entities. Under control of the *Global Controller*, the individual *Host Controllers* start or interact with virtual entities when needed. The choice of *Host Controller* is decided by the *Placement Engine*, which uses an algorithm to determine the *best* physical machine to deploy the new virtual entity. The *Global Controller* also sends requests, via a *Host Controller*, to connect virtual routers together via virtual links.

In order to manage the challenging and dynamic infrastructures of virtual networks there needs to be a monitoring system which collects data and reports on the behavior of both the physical resources (e.g. CPU usage, memory usage) and the virtual resources (e.g. utilization level of the virtual links). The monitoring data items are sent to the *Global Controller* components. The monitoring information is used to take decisions regarding network strategies and enforces these decisions. The monitoring system *Lattice* is used as has been proven to be ideal for the task of collecting monitoring data for various types of dynamic network environments. Each virtual entity has at least one probe that can generate data. Monitoring data is also collected from each *Host Controller*. All of this monitoring data is send to the *Global Controller*, and it is the data that is used by the *Placement Engine* for determining where a new virtual entity is placed.

2.1 Main Function Elements

The main elements of the Platform are the Orchestration Layer / Virtual Infrastructure Management (VIM) called the *GlobalController*, the per-host *Host Controllers*, called the *LocalControllers*, and the *Routers*. These are each explained in further detail below.

The main elements can be found in the following table.

Name	**Description**

Host Controllers	The host controllers execute on every physical machine and manipulate and configure virtual routers, links and virtual router applications.
Monitoring System	The monitoring system, comprising probes that are tiny configurable applications probing the software or hardware for monitoring data, as well as the data consumers.
Event Management	It is responsible for the runtime operation, including support for event-based notifications and time scheduling.
Virtual Router Protocol Stack	The lightweight network protocol stack of the VRs.
Virtual Router Application Environment	The application environment that hosts VR applications.
Virtual Link Functionality	The functionality of the virtual links, including link weighting and other configuration options.
Virtual Machine for Virtual Router and Application Functionality	A virtual machine with the virtual router and the relevant applications functionality.

There is one *VIM* for the platform, and it has the following functions:

- it starts and stops the LocalControllers
- it acts as a control point for the platform by sending out commands,
- it acts as a management element for the platform by collecting monitoring data and enabling reactive behaviour

The Virtual Infrastructure Management (VIM) can run on the same host as a LocalController, or in a large setup it can run on a separate host. The VIM component is responsible for the management and lifecycle of the virtualized elements that will be used within a network, particularly virtual network elements. As the virtual elements are not physical themselves, but exist on top of physical elements, their lifecycle and their management needs to be approached carefully to ensure continued operation and consistency.

The virtual network elements, which exist on top of physical networks, can be setup with arbitrary topologies and with an arbitrary number of end-points. The virtual links in a virtual topology are eventually mapped onto physical links in the underlying network. A virtual link may span multiple physical links, and cross many physical routers, or it may span a single physical network link. New virtual links can be added or can be removed from a virtual network dynamically at run-time.

The virtual networks are very flexible and adaptable, and generally have few limitations, except that a virtual link cannot support more traffic and higher-data rates than the underlying physical links. Furthermore, a whole virtual network can be shutdown as needed, if the applications that use it no longer need the network. Such a shutdown frees resources from the underlying physical network.

The VIM component has a seemingly simple task, but in reality the management requires continual monitoring, analysis, and adaption of the virtual elements to the physical elements. As all of these virtual elements are distributed, the management is a complex task. The VIM interacts with the virtual network elements that will be present in a running virtual network. All of the elements of the VIM component constitute a fully distributed system, whereby an element or node can reside on any host. A full virtual network can be instantiated on a single machine, for demonstration or testing purposes, or instantiated across multiple servers, in a full deployment situation.

The VIM directly controls the lifecycle of each virtual element, by collecting knowledge on the status of physical resources in order to determine where a virtual element can be created. The virtual network element will be created, managed, and shutdown by lifecycle phase of this component.

Due to the dynamic nature of virtual elements and because they can be disassociated from the physical elements they are mapped to, it is possible to do a live adaption of a virtual element from one physical host to another physical one, at run-time.

The VIM controller acts as a control point for managing the virtual elements. This block accepts all its input via the VIM REST interface from other management applications / network services. The monitoring engine acts a collection point for the monitoring data needed to keep the management functions running. Control commands are sent to the VIM and they are either acted upon immediately or are passed to the corresponding Host Controllers.

The main VIM elements can be found in the following table.

Name	**Description**
VIM Controller	It is the heart of the component, providing the central control of the VIM operations.
Scripting Engine	VIM can be configured using Closure scripting.

Monitoring Engine	It is the main monitoring component of the infrastructure, i.e., collecting & manipulating measurements from the monitoring probes residing at the virtual entities.
Virtual Entities / Topology Configurators	These functions are responsible for the configuration of virtual routers, links and topologies, supporting different levels of abstraction.
Configuration Actuators	The Virtual Entities / Topology configurators communicate with the configuration actuators which in turn enforce the configuration changes through the LNH's host controllers.

There is one *LocalController* for each physical host that needs to execute virtual routers. A LocalController has the following functions:

- it starts and stops virtual routers
- it tells routers to create and remove virtaul network connections
- to get or set attributes on routers or links

A LocalController is similar to a hypervisor in a normal virtualization environment, as it has a role of stopping and stopping virtaul machines.

Within the platform, many *Routers* can be created. These virtaul routers behave like a real hardware router, with the caveat that they have much simpler functionality.

In figure 1, the relationship between these elements is shown. There is the VIM / GlobalController, shown in brown, which can take various configurations in order to setup and control a run. These configurations can be static configurations, where there is a fixed topology, or dynamic configurations, where the topology of the network and the number of links changes on-the-fly under the control of the GlobalController.

The GlobalController interacts with various LocalControllers. This interaction path is shown as a dotted line. A LocalController, shown in purple, executes on each physical host that participates in the platform. Each LocalController takes requests from the GlobalController and takes the required action. This can be to start or stop a virtual router, to create or remove a virtual link, to get or set attributes on routers or links.

To start a new Router, shown in blue, the LocalController on the relevant host will start a new Java Virtual Machine, shown in white, executing the specific Router code. The router has various elements, which will be discussed later, however for this discussion the

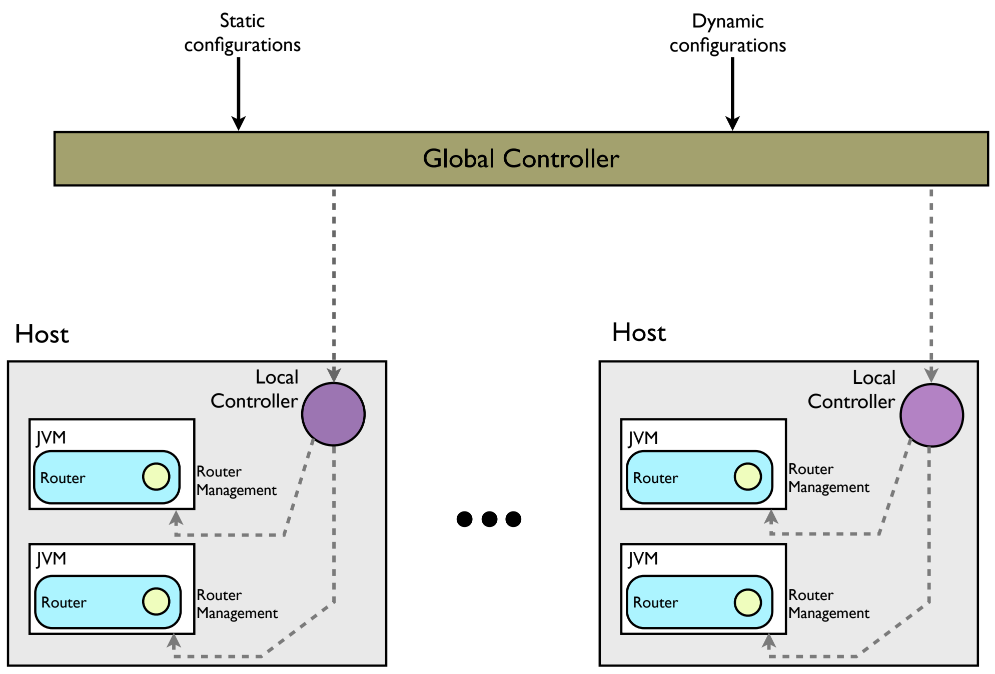

Figure 1: General Architecture

most important one is the Router Management element, shown in yellow. Once a Router is up and running, the LocalController interacts with it via the Router Management interface. It is using this interface that commands and requests for the Router are sent by the LocalController. This path is also shown via a dotted line.

Although there is a considerable amoount of infrastructure in the platform dealing with control, the main aim of the platform is to create a topology of virtual routers. These routers execute on a set of hosts, with virtual links between the virtual routers. In figure 2, we see how the topology of virtual routers and virtual links manifests itself across multiple hosts, three in this case.

An alternative view of the platform where the hosts are not shown, but there is the control path and the virtual routers and virtual links is shown in figure 3. Control propogates from the GlobalController, via the LocalControllers, to the Routers. The topology of Routers is connected by separate virtual links.

2.2 Main Platform Functions

The main functions of the Very Lightweight Network & Service Platform are outlined in this subsection. The platform uses a set of Virtual Routers and Virtual Network Connections to create a test environment which can easily accommodate:

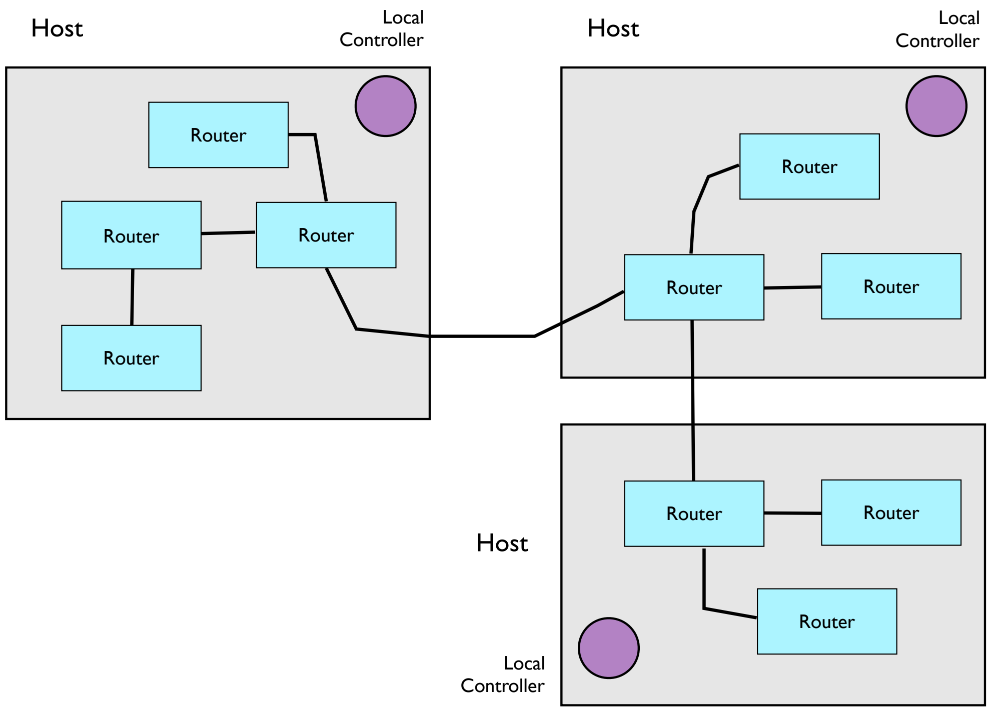

Figure 2: Hosts with Virtual Routers and Virtual Links

- fast setup / teardown of a Virtual Router
- fast setup / teardown of a Virtual Connection
- each Virtual Router can run small programs / service elements

These are explained in more detail in the following sections.

2.3 Routers

The software virtual router is implemented in Java and is a relatively complex software entity. The routers hold network selections to the other virtual routers they are aware of and exchange routing tables to determine the shortest path to each other router. Datagrams are sent between routers and queued at input and output. A system of virtual ports (like the current transport layer ports) are exposed with an interface very similar to standard “sockets”. Virtual services can be run on the virtual routers and listen and send on the virtual sockets. The datagrams themselves, have headers with a source address, destination address, protocol, source port, destination port, length, checksum and time to live – many of the features of real UDP packets are replicated.

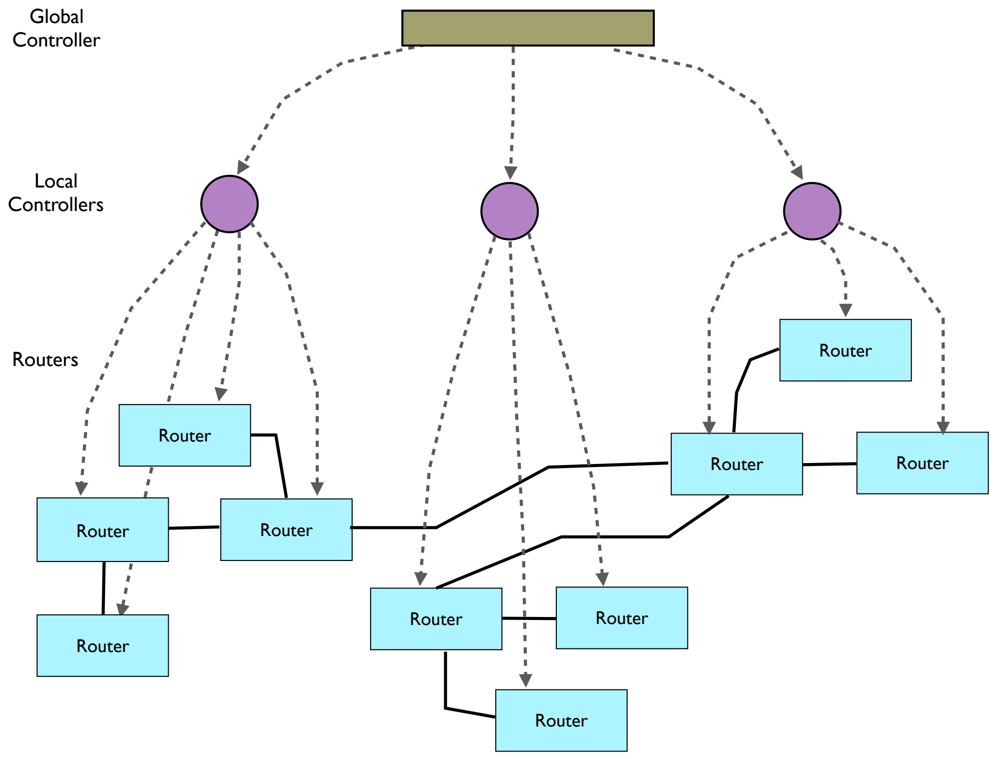

Figure 3: Full Connectivity with Control Path

2.4 Routing and packet transmission

Routing in these virtual routers is distance-vector based. To prevent routing storms minimum times between table transmissions are set. In addition, because the experiment here demands a certain "churn" of virtual routers then addresses which disappear permanently must be dealt with. In distance vector routing it is well-known that dead addresses can leave routing loops. This is dealt with in the current system by implementing time to live (TTL) in packets and also implementing a maximum routing distance beyond which a router is assumed unreachable and removed from routing tables.

Virtual services can run on the routers. Naturally the only important virtual services for the purposes of our evaluations are networked services. The services tested include simple network test protocols such as ping and traceroute and ftp style send and receive services. The major software used, however, is the Lattice monitoring framework developed for RESERVOIR. The virtual services can listen on virtual ports and send datagrams to any address and virtual port.

Datagrams are queued both inbound and outbound, the outbound queue is blocking in order that transmitting services can slow their sending rate. The inbound queue is tail-drop so that when too much traffic is sent drops will occur somewhere. TTL is decreased at each hop and, on expiry, a ”TTL expired” packet is returned – this allows the virtual router system to implement traceroute as a virtual service. Virtual routers in the system send all traffic including routing tables and other control messages via network sockets. Control messages (routing tables, echo, echo reply, TTL expired and so on) are routed in the same way as data packets on a hop by hop basis using the routing tables. Datagram transmission is UDP-like in this iteration of the system, that is, delivery is not guaranteed and a failure to deliver will not be reported to the service (although if the router on which a virtual service runs has no route to the host this can be reported to the service).

2.5 Start up and tear down

Start up and tear down for routers is scheduled by the Global Controller and directly performed by the Local Controller which resides on each physical machine as the virtual router. The virtual routers would operate perfectly well without such a controller, for example, if they were set up and connected manually or by some other distributed control system. The Local Controller is necessary to spawn off new routers on a machine. In addition, the Local Controller is used here to pass on Global Controller commands to shut down or connect virtual routers.

2.6 Virtual Services / Applications

Each virtual router has a socket interface by which services and applications can be written. All networking interactions are done using our USR Datagram layer, which is very similar to the standard Java socket and datagram interface. Each service can open a socket on a given address and port, and can send traffic to an address and a port.

By having our own virtual datagram layer we could control all the networking within the virtual router. However, by making the socket and datagram interface very similar to the standard Java one, we were able to utilize existing Java software which has a standard UDP mechanism.

As an example of easy re-used of existing software, consider that it took just a few hours to port the Lattice monitoring framework onto this new platform. We wrote an implementation of the data plane using USR instead of UDP. In terms of the source code, there are few differences between each implementation.

2.7 Monitoring

The monitoring software used in VLSP is called Lattice and has been used for monitoring virtualised services in federated cloud environments, for monitoring virtual networks, and as the monitoring system for an information management platform that aggregates, filters, and collects data in a scalable manner within virtual networks. Lattice, which is also an open-source software, has been proven to be ideal for the task of collecting monitoring data for various types of dynamic network environments.

Each virtual router has a set of probes which generate data for virtual link usage, cpu / memory usage per thread, and virtual applications resource consumption. This data is sent to the Monitoring Engine of the Global Controller which processes it. This data is used by the Placement Engine for determining where a new virtual router is placed.

2.8 REST API for Dynamic Programmable Control

A dynamic programmable control environment allows the definition of scenarios, resources, or other software entity parameters using appropriate functions via a REST API. In our case, we use a Java client or one written in the Clojure language for the dynamic configurations. This allows us to perform fully Software-Defined Operations using a functional language that has an expressive representation of the configuration settings, while being very brief.

2.9 Event Engine for Probabilistic Experiment Control

Many experiments that are undertaken in the networking domain are based on arrival rates that are taken from a probability distribution. Within VLSP we have an *event engine* that can generate events to create a router, destroy a router, create a link, and delete a link, all based on probability distributions.

For any particular run we can specify these distributions together with their associated parameters in an XML configuration file. The options for the distribution are set in a Type field as one of: Uniform, Exponential, LogNormal, or PoissonPlus, matching well-known topology models. Fur- thermore, we are providing pre-defined configuration files with realistic topologies, e.g. for data centers.

2.10 VLSP Visualization

One of the management tools we have built for VLSP is a Network Visualization tool. It takes a logical graph view of the virtual topology, including the routers and the links, and presents a visualization graph by using the graphviz tool. From data held by the Orchestration Controller, a version of the network is generated in the dot language that graphviz uses. As dot is very flexibile, it is easy to create extensions to the visualizations which include the virtual applications that execute on the routers. Furthermore, we are able to present key nodes in different shapes and colours in order to highlight the different features of a topology. As an example, consider the topology on the cover which represents a partial snapshot from one of our experiments. Those with a similar colour are logically connected for managing the same data streams, and those nodes represented as a diamond shape are running a Data Proxy application. It allows us to see how data management applications are deployed across the whole topology.

3 Usage

In this section the usage of the platform is described. We show how to start up the platform and how to configure different runs.

Remember that there are 3 main components of the platform:

- the Global Controller
- Local Controllers
- Routers

Within a run, there will be one Global Controller, which controls the run, a Local Controller on each physical machine that participates, and a set of Routers that run on the physical machines.

To start the whole VLSP system, we execute the *Global Controller* and pass in an XML file which has the configuration options for each particular run, specifying a Java class called: `usr.globalcontroller.GlobalController`. The options include information such as the ports to listen on; the hosts that the *Host Controller* is to execute on and its associated ports; its monitoring elements; the placement engine to use; and other virtual entity options. When the *Global Controller* starts it attempts to setup all of the monitoring elements and start all of the *Host Controllers* on the specified hosts, this is a Java class called: `usr.localcontroller.LocalController`. Once this has been achieved the VLSP system is ready for experiments.

The interface between all the VLSP components uses HTTP. All communicated information is transmitted using REST and JSON descriptions. To maintain the lightness of the system we use the `monoid Resty` jar library, which is only 120K. This compares to the commonly used, but fully featured, Jersey library which is 1.5Mb.

The start up and shutdown of virtual routers is managed by the *Global Controller* but is performed by the *Host Controller* which resides on each host after receiving REST calls. The *Host Controller* is also used to control the connection of virtual routers with virtual links. The *Host Controller* behaves in the same way a hypervisor does in other virtualised environments, and it also passes on *Global Controller* commands to Routers. A virtual entity will be started on the same physical machine as the *Host Controller*.

3.1 Pre-Configuration

If you do not have VLSP yet, you can find the source on *github* at

```
https://github.com/stuartclayman/VLSP .
```

As the platform is written in Java, we need to set the JAVA_HOME and the CLASSPATH environment variables.

```
% export JAVA_HOME=/usr/java/jdk1.8/
```

To start the run of the platform, either set the current directory to be where the platform is installed and the relevant environment variables need to be set before everything is started:

```
% cd /install_place
```

```
% export CLASSPATH=.:./libs/*:
```

or set the install_place in the classpath:

```
% export CLASSPATH=/install_place:/install_place/libs/*:
```

3.2 Starting

The Global Controller is started and passed an XML configuration control file – the path to the configuration file can be relative or absolute.

```
% java usr.globalcontroller.GlobalController control-config.xml
```

or:

```
% java usr.vim.Vim control-config.xml
```

This will start the Global Controller on the local host. The class `usr.vim.Vim` is a subclass of the GlobalController, with identical functionality.

It with automatically start the Local Controllers and the Routers in the relevant places. The links between the Routers will be created, under the control of the Global Controller, based on the configuration file.

3.3 Configuration for Controlled Setups

The Controlled Setups start with only a GlobalController and LocalControllers, and they wait for external control via the REST API to configure the virtual topology. The following is a configuration, which has 4 main sections:

- GlobalController – with configuration options for the GlobalController itself
- LocalController – with configuration details for LocalControllers
- EventEngine – setup for an EventEngine
- RouterOptions – with configuration for the virtual Routers

This configuration starts the Global Controller on port 8888 of the local host, with a placement engine of class `usr.globalcontroller.LeastUsedLoadBalancer`. The Lattice monitoring is configured with 5 different data consumers. The Global Controller starts Local Controllers on 3 other hosts: host1, host2, and host3. The Local Controllers will listen on port 10000 of each host.

The configuration for the routers is held in the file `scripts/routeroptions.xml`. All of the configuration options can be found in section 6.1.

Listing 1: control-wait-config.xml

```
<SimOptions>
  <GlobalController>
    <Port>8888</Port>
    <StartLocalControllers>true</StartLocalControllers>

    <PlacementEngineClass>usr.globalcontroller.LeastUsedLoadBalancer</PlacementEngineClass>
    <Monitoring>
      <LatticeMonitoring>true</LatticeMonitoring>
      <MonitoringPort>7799</MonitoringPort>

      <Consumer>
        <Name>usr.globalcontroller.HostInfoReporter</Name>
      </Consumer>
      <Consumer>
        <Name>usr.globalcontroller.NetIFStatsReporter</Name>
      </Consumer>
      <Consumer>
        <Name>usr.globalcontroller.RouterAppsReporter</Name>
      </Consumer>
      <Consumer>
        <Name>usr.globalcontroller.ThreadGroupListReporter</Name>
      </Consumer>
      <Consumer>
        <Name>usr.globalcontroller.ThreadListReporter</Name>
      </Consumer>
    </Monitoring>
  </GlobalController>

  <LocalController>
    <Name>host1</Name>
    <Port>10000</Port>
    <LowPort>11001</LowPort>
    <HighPort>12000</HighPort>
    <MaxRouters>100</MaxRouters>
  </LocalController>

  <LocalController>
    <Name>host2</Name>
    <Port>10000</Port>
    <LowPort>12001</LowPort>
    <HighPort>13000</HighPort>
    <MaxRouters>100</MaxRouters>
  </LocalController>

  <LocalController>
    <Name>host3</Name>
    <Port>10000</Port>
    <LowPort>13001</LowPort>
    <HighPort>14000</HighPort>
    <MaxRouters>100</MaxRouters>
  </LocalController>

  <EventEngine>
    <Name>Empty</Name>
    <EndTime>86400</EndTime>
  </EventEngine>

  <RouterOptions>
     scripts/routeroptions.xml
  </RouterOptions>

</SimOptions>
```

In the above configuration for `GlobalController` the following options have been set:

Option	**Value**	**Description**

Port	8888	The GlobalController will listen for REST API commands on port 8888.
StartLocalControllers	true	The GlobalController should start LocalControllers if they are not already running.
PlacementEngineClass	usr.globalcontroller. LeastUsedLoadBalancer	The name of the class for the Placement Engine.
Monitoring	More config options	If required, the details for Monitoring.
→ LatticeMonitoring	true	Should the GlobalController start the Lattice Monitoring sub-system.
→ MonitoringPort	7799	The Lattice Monitoring sub-system will listen on port 7799 for monitoring data.
→ Consumer	usr.globalcontroller. HostInfoReporter	A monitoring consumer that collects data from LocalControllers.
→ Consumer	...	Other monitoring consumers.

Now we present the configuration for `LocalController` blocks. A GlobalController needs one or more LocalController definitions. In the above example, there are 3 LocalController definitions, with the following options having been set:

Option	**Value**	**Description**
Name	host1	The name of the host that the LocalController should be started on.
Port	10000	The LocalController will listen on port 10000 for commands from the GlobalController.
LowPort	11001	The LocalController will start a virtual router, with the router management port in a range, with this the low end.
Highort	12000	The LocalController will start a virtual router, with the router management port in a range, with this the high end.
MaxRouters	100	The maximum number of virtual routers to start in this host.

Now we present the configuration for `EventEngine` block. The setup for externally controlled configurations has the following options set:

Option	**Value**	**Description**

Name	Empty	The EventEngine to execute. Empty means there is no events being generated.
EndTime	86400	The amount of time the system will run – even if there are no events being created. In this case it is *86400 seconds*, which is *1440 minutes*, or *24 hours*.

Now we present the configuration for `RouterOptions` block. This always has the name of another configuration file which contains the actual router options. In this case it is the file: `scripts/routeroptions.xml`. We will now look at the router configuration file: routeroptions.xml.

Listing 2: routeroptions.xml

```
<RouterOptions>
   <Monitoring>
     <LatticeMonitoring>true</LatticeMonitoring>
     <Probe>
       <Name>usr.router.NetIFStatsProbe</Name>
       <Rate>1000</Rate>
     </Probe>
     <Probe>
       <Name>usr.router.ThreadListProbe</Name>
       <Rate>1000</Rate>
     </Probe>
     <Probe>
       <Name>usr.router.ThreadGroupListProbe</Name>
       <Rate>5000</Rate>
     </Probe>
     <Probe>
       <Name>usr.router.AppListProbe</Name>
       <Rate>5000</Rate>
     </Probe>
   </Monitoring>

   <APManager>
       <Name>None</Name>
   </APManager>

</RouterOptions>
```

In the above configuration for `RouterOptions` the following options have been set:

Option	**Value**	**Description**
Monitoring	More config options	If required, the details for Monitoring.

→ LatticeMonitoring	true	Should the Router start the Lattice monitoring and send data from the defined probes.
→ Probe	usr.router. NetIFStatsProbe	A monitoring probe that collects *Network Interface* packets data from a Router .
→ Probe	...	Other monitoring probes.
APManager	More config options	If required, the details for an Aggregation Point Manager.
→ Name	None	This setup does not use Aggregation Points.

3.4 Configuration for Probabilistic Setups

This setup is configured to use a single host, with the GlobalController and the LocalController both residing on the localhost. The following is a minimal configuration, which has 4 main sections:

- GlobalController which defines values for the Global Controller
- LocalController which defines values for Local Controllers
- EventEngine which defines values for an event engine that drives the creation of routers and links
- RouterOptions which defines values for the routers.

This configuration starts the Global Controller on port 8888 of the local host, and that it should start a Local Controller. The Local Controller will be on port 10000 of the local host. It also tells the LocalController to allocate Routers which listen on a range of ports, between port 11001 and 12000, and that there should be a maximum of 50 routers on localhost.

The EventEngine section configures the Probabilistic engine to generate new Router and Connection requests. The engine will execute for 600 seconds, and will get the probability distribution information from the file `probdists.xml`.

Finally, the configuration for the routers is held in the file `routeroptions.xml`.

Listing 3: control-probabalistic-config.xml

```
<SimOptions>

  <GlobalController>
    <Port>8888</Port>
    <StartLocalControllers>true</StartLocalControllers>
    <ConnectedNetwork>true</ConnectedNetwork>
  </GlobalController>

  <LocalController>
    <Name>localhost</Name>
    <Port>10000</Port>
    <LowPort>11001</LowPort>
    <HighPort>12000</HighPort>
    <MaxRouters>50</MaxRouters>
  </LocalController>

  <EventEngine>
    <Name>Probabilistic</Name>
    <EndTime>600</EndTime>
    <Parameters>probdists.xml</Parameters>
  </EventEngine>

  <RouterOptions>
      routeroptions.xml
  </RouterOptions>

</SimOptions>
```

The options set for the configuration of the `GlobalController` and the `LocalController` are as described earlier. But we present the configuration for `EventEngine` block, as this is different here.

Option	**Value**	**Description**
Name	Probabilistic	The EventEngine is the Probabilistic event engine which generates events in a probabilistic manner.
EndTime	600	The amount of time the system will run – even if there are no events being created. In this case it is *600 seconds*, which is *10 minutes.*
Parameters	probdists.xml	The name of the file to get the probability functions and coefficients for the probabilistic event engine.

The probabilistic event engine is defined in the class `usr.engine.ProbabilisticEventEngine`. It expects extra probability distribution configuration to be defined. For this example the following is defined:

Listing 4: probdists.xml

```
<ProbabilisticEngine>
     <NodeBirthDist>
          <ProbElement>
               <Type>Exponential</Type>
               <Weight>1.0</Weight>
               <Parameter>3.0</Parameter>
          </ProbElement>
     </NodeBirthDist>

     <NodeDeathDist>
          <ProbElement>
               <Type>Exponential</Type>
               <Weight>0.7</Weight>
               <Parameter>60</Parameter>
          </ProbElement>
          <ProbElement>
               <Type>LogNormal</Type>
               <Weight>1.0</Weight>
               <Parameter>7.0</Parameter>
               <Parameter>1.5</Parameter>
          </ProbElement>
     </NodeDeathDist>

     <LinkCreateDist>
          <ProbElement>
               <Type>PoissonPlus</Type>
               <Weight>1.0</Weight>
               <Parameter>1.5</Parameter>
               <Parameter>1.0</Parameter>
          </ProbElement>
     </LinkCreateDist>

     <Parameters>
        <PreferentialAttachment>true</PreferentialAttachment>
     </Parameters>
</ProbabilisticEngine>
```

Here we present the configuration for `ProbabilisticEngine` block.

Option	**Value**	**Description**
NodeBirthDist		Options for the probabilistic creation of new virtual routers.
NodeDeathDist		Options for the probabilistic lifetime of a virtual router.
LinkCreateDist		Options for the probabilistic creation of new virtual links.

→ ProbElement		The specification for this probability distribution.
→ Type	Exponential	Use the Exponential distribution.
→ Weight	1.0	A weight for thr distribution.
→ Parameter	3.0	A coefficient parameter for thr distribution.
Parameters		Optional extra parameters.
→ PreferentialAttachment	true	Use a preferential attachment strategy when adding new links.

The values for the *Type* of probability distribution are: `Exponential`, `LogNormal`, `PoissonPlus`, and `Uniform`.

We will now look at the router configuration.

Listing 5: routeroptions-with-ap.xml

```
<RouterOptions>

   <APManager>
      <Name>Pressure</Name>
      <MaxAPs>100</MaxAPs>
      <MinAPs>1</MinAPs>
      <MaxAPWeight>5</MaxAPWeight>
      <MinPropAP>0.1</MinPropAP>
   </APManager>

</RouterOptions>
```

Here we present the configuration for `RouterOptions` block.

Option	**Value**	**Description**
APManager	More config options	If required, the details for an Aggregation Point Manager.
→ Name	Pressure	This setup uses the Pressure Aggregation Point algorithm.
→ MaxAPs	100	The maximum number of APs in the network.
→ MinAPs	1	The minimum number of APs in the network.
→ MaxAPWeight	5	
→ MinPropAP	0.1	

The values for the *Name* of the Aggregation Point Manager strategy are: `Pressure`, `Random`, `HotSpot`, and `None`.

3.5 Stopping

The Global Controller will keep running for the number of seconds defined in *EventEngine* $\rightarrow$ *EndTime* specification. In the Configuration for Controlled Setup example it was set to 86400 seconds, and in the Probabilistic Setup example it was set to 600 seconds. Once the specified time has elapsed the GlobalController will go into its shutdown phase and will automatically stop all app, remove all virtual links, terminate all virtual routers, and stop all LocalControllers.

It is possible to stop the GlobalController under software control by sending a special command via the REST API. This call acts the same way as the time elapsing and will go into the shutdown phase.

```
GET http://host:8888/command/SHUT_DOWN
```

4 Internal Design of Elements

This section describes the internal design of the elements of the platform. The GlobalController, the LocalControllers, and the Routers are shown in more detail.

4.1 The Global Controller

The Global Controller is the main control point for the whole of VLSP. It provides the Virtual Infrastructure Management (VIM) and Orchestration Layer required to setup and manage virtual infrastructures.

The main elements of the Global Controller are:

Event Management: This is responsible for the runtime operation, including support for event-based notifications and time scheduling of events. All setups that use the Probabilistic Experiment Control will use an *EventEngine* that generates events for router creation, link creation, and router deletion. These events are queued for future activation in an *EventScheduler*. The configuration files specify which probability distributions are used for these operations.

For setups using the REST API for Dynamic Programmable Control, the call to the Management Console is converted into an event that executes immediately.

Management Console - REST API: This is an HTTP listener that supports the REST API into the Global Controller, and essentially the whole of VLSP.

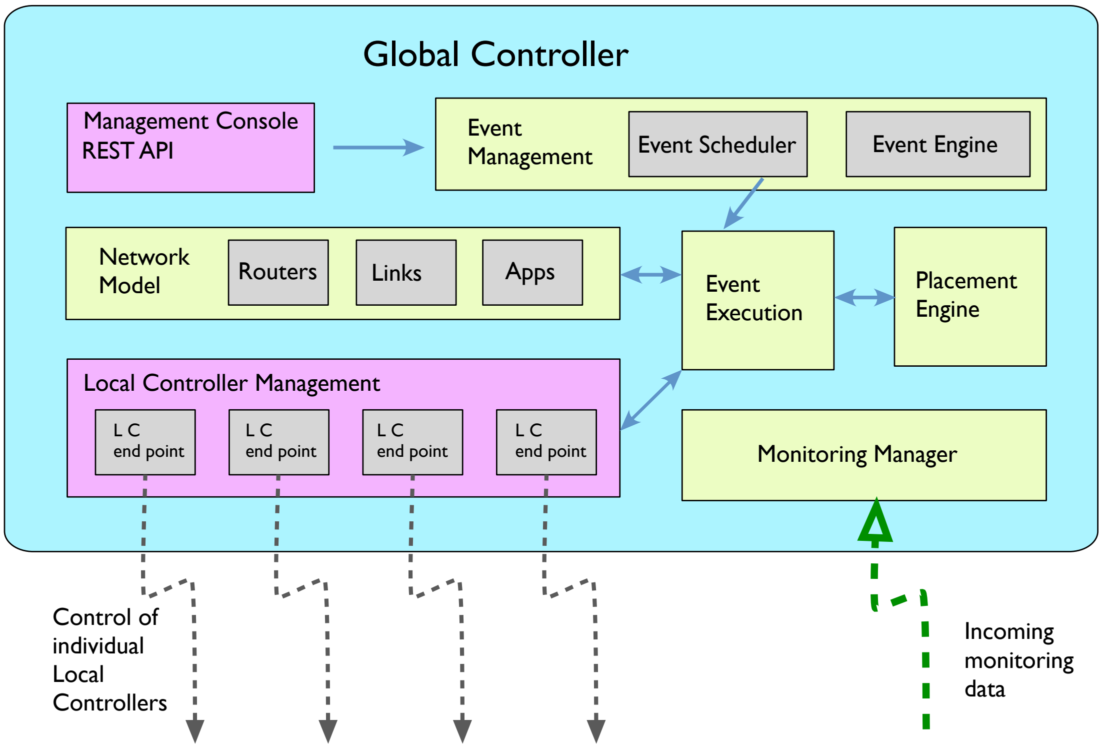

Figure 4: Global Controller Architecture

Event Execution: This executes the actions for each of the events that the *EventScheduler* chooses as the next action. Each Event has a time, an event type, and some event parameters. During event execution there is a handler for each type of event, which causes the relevant actions to be performed.

Placement Engine: This decides where to place a new router when one is created. The Placement Engine uses data about the current state, context, and configuration of the system to choose a location. The Placement Engine is a flexible part of VLSP and can be changed at run-time under software control.

Network Model: This keeps information on the topology of the virtual network that has been created and is being managed, including all of the *Routers* and *Links*. It also contains information on the *Apps* that are running on eahc virtual router. The Model is used by many components in the Global Controller for decision making, outputs via the REST API, as well as visualization.

Monitoring Manager: This manages the monitoring sub-system of VLSP. It starts any configured probes on the virtual routers, and starts any configured data consumers and data reporters. The monitoring data is used by many components inside the Global Controller.

Local Controller Management: This is responsible for starting Local Controllers on specified hosts, interacting directly with those Local Controllers, and managing their state. This element has an *end − point* for each Local Controller that has been defined, as well as a direct network connection to that Local Controller for sendign it commands.

All operations to start and stop routers, and commands for routers are directed through the relevant Local Controller.

4.1.1 Management Console

The Management Console supports the REST API into the Global Controller. All of the REST commands are presented in Appendix 7.

4.1.2 Placement Engine

We have experimented with various placement algorithms, more details of which can be found in our other papers. The *Placement Engine* of VLSP is the management component in charge of performing the actual placement of the virtual entities and the application nodes they host, according to the initial topology and the resource usage of the virtual network elements. This is an important feature because, when we configure a network or a service, given initial context information, some of these parameters may change during the course of the systems operation and a reconfiguration may be required to maintain optimized behaviour. Consequently, our approach has a mechanism to achieve adaptation in a flexible manner.

The decision of the *Placement Engine*, which can be changed at run-time under software control, is encoded in an algorithm which can be either rather simple, such as counting the number of virtual routers on a host, or it can be complex, based on a set of constraints and policies that represent the network properties. Therefore, to determine the placement for the next virtual router, the GlobalController uses the defined Placement Engine.

The name of the Placement Engine class is set in the `<PlacementEngineClass>` field. The default class name is: `usr.globalcontroller.LeastUsedLoadBalancer`. This is a list of the current implementations:

```
usr.globalcontroller.LeastBusyPlacement
usr.globalcontroller.LeastUsedLoadBalancer
usr.globalcontroller.NupPlacement
```

`usr.globalcontroller.EnergyEfficientPlacement`

The user can write their own Placement Engine, and set it in the config Such a Placement Engine can take into account any attributes that are needed to make a decision which chooses a particular Local Controller.

4.1.3 Monitoring Manager

The Monitoring Manager is responsible for starting, stopping, and processing the monitoring processes within VLSP. Probes can be configured to start in the Routers which send various kinds of measurment data related to different router processes and tasks.

Router Probes

The current set of probes that have been defined are:

Name	**Description**

`usr.router.AppListProbe`	This probe sends a list of all the executing applications in a router.
	The probe sends the `RouterName` plus a table of `Data` with 1 row per app.
	
	ProbeAttributeType.STRING
	`RouterName`
	TableProbeAttribute
	`Data`
	
	`"AID", ProbeAttributeType.INTEGER`
	`"StartTime", ProbeAttributeType.LONG`
	`"ElapsedTime", ProbeAttributeType.LONG`
	`"RunTime", ProbeAttributeType.LONG`
	`"UserTime", ProbeAttributeType.LONG`
	`"SysTime", ProbeAttributeType.LONG`
	`"State", ProbeAttributeType.STRING`
	`"ClassName", ProbeAttributeType.STRING`
	`"Args", ProbeAttributeType.STRING`
	`"Name", ProbeAttributeType.STRING`
	`"RuntimeKeys", ProbeAttributeType.LIST`
	`"RuntimeValues", ProbeAttributeType.LIST`

`usr.router.NetIFStatsProbe`	This probe sends a list of stats for each of the network interfaces of the Router.
	The probe sends the `RouterName` plus a table of `Data` with 1 row per app.
	
	ProbeAttributeType.STRING
	`RouterName`
	TableProbeAttribute
	`Data`
	
	`"name", ProbeAttributeType.STRING`
	`"InBytes", ProbeAttributeType.INTEGER`
	`"InPackets", ProbeAttributeType.INTEGER`
	`"InErrors", ProbeAttributeType.INTEGER`
	`"InDropped", ProbeAttributeType.INTEGER`
	`"InDataBytes", ProbeAttributeType.INTEGER`
	`"InDataPackets", ProbeAttributeType.INTEGER`
	`"OutBytes", ProbeAttributeType.INTEGER`
	`"OutPackets", ProbeAttributeType.INTEGER`
	`"OutErrors", ProbeAttributeType.INTEGER`
	`"OutDropped", ProbeAttributeType.INTEGER`
	`"OutDataBytes", ProbeAttributeType.INTEGER`
	`"OutDataPackets", ProbeAttributeType.INTEGER`
	`"InQueue", ProbeAttributeType.INTEGER`
	`"BiggestInQueue", ProbeAttributeType.INTEGER`
	`"OutQueue", ProbeAttributeType.INTEGER`
	`"BiggestOutQueue", ProbeAttributeType.INTEGER`

`usr.router.ThreadGroupListProbe`	This probe sends data for each Thread Group in the Router.
	The probe sends the `RouterName` plus a table of `Data` with 1 row per app.
	
	ProbeAttributeType.STRING
	`RouterName`
	TableProbeAttribute
	`Data`
	
	`"Name", ProbeAttributeType.STRING`
	`"StartTime", ProbeAttributeType.LONG`
	`"ElapsedTime", ProbeAttributeType.LONG`
	`"RunTime", ProbeAttributeType.LONG`
	`"UserTime", ProbeAttributeType.LONG`
	`"SysTime", ProbeAttributeType.LONG`
	`"Mem", ProbeAttributeType.LONG`

`usr.router.ThreadListProbe`	This probe sends data for each Thread Group in the Router.
	The probe sends the `RouterName` plus a table of `Data` with 1 row per app.
	
	ProbeAttributeType.STRING
	`RouterName`
	TableProbeAttribute
	`Data`
	
	`"Name", ProbeAttributeType.STRING`
	`"StartTime", ProbeAttributeType.LONG`
	`"ElapsedTime", ProbeAttributeType.LONG`
	`"RunTime", ProbeAttributeType.LONG`
	`"UserTime", ProbeAttributeType.LONG`
	`"SysTime", ProbeAttributeType.LONG`
	`"Mem", ProbeAttributeType.LONG`
	`"ThreadGroup", ProbeAttributeType.STRING`

Local Controller Probe

There is currently one probe residing in the Local Controller. It is turned on automatically if monitoring is configured to be on. This probe is designed to send host based info back to the Global Controller.

Name	Description

`usr.localcontroller.` `HostInfoProbe`	This probe sends data about the physical host that the LocalController is managing.
	The probe sends the `Name` of the host, cpu usage, memory usage, plus a table of network statistics, with 1 row per network interface of the physical host.
	
	ProbeAttributeType.STRING
	`Name`
	ProbeAttributeType.FLOAT
	`cpu-user`
	ProbeAttributeType.FLOAT
	`cpu-sys`
	ProbeAttributeType.FLOAT
	`cpu-idle`
	ProbeAttributeType.FLOAT
	`load-average`
	ProbeAttributeType.INTEGER
	`mem-used`
	ProbeAttributeType.INTEGER
	`mem-free`
	ProbeAttributeType.INTEGER
	`mem-total`
	TableProbeAttribute
	`net-stats`
	
	`"if-name", ProbeAttributeType.STRING`
	`"in-packets", ProbeAttributeType.LONG`
	`"in-bytes", ProbeAttributeType.LONG`
	`"out-packets", ProbeAttributeType.LONG`
	`"out-bytes", ProbeAttributeType.LONG`

Global Controller Reporters

The reporters which can be configured in the Global Controller are designed to collect measurements which are sent by the probes in other parts of the system.

The current set of probes that have been defined are:

Name	**Description**
`usr.globalcontroller.` `HostInfoReporter`	This reporter collects measurements from the `HostInfoProbe` embedded in the Local Controller. Additionally data is sent to /tmp/gc-channel13.out.
`usr.globalcontroller.` `RouterAppsReporter`	This reporter collects measurements from the `AppListProbe` embedded in the router, if it has been configured. Additionally data is sent to /tmp/gc-channel10.out.
`usr.globalcontroller.` `NetIFStatsReporter`	This reporter collects measurements from the `NetIFStatsProbe` embedded in the router, if it has been configured. Additionally data is sent to /tmp/gc-channel7.out.
`usr.globalcontroller.` `ThreadGroupListReporter`	This reporter collects measurements from the `ThreadGroupListProbe` embedded in the router, if it has been configured. Additionally data is sent to /tmp/gc-channel14.out.
`usr.globalcontroller.` `ThreadListReporter`	This reporter collects measurements from the `ThreadListProbe` embedded in the router, if it has been configured. Additionally data is sent to /tmp/gc-channel15.out.

4.2 The Local Controller

The Local Controller is a per host Host Controller that accepts commands that are specifically for that host. The Local Controller acts as a management point for starting and stopping Routers, for creating and deleting links, and also for monitoring the host it runs on.

4.3 Routers

The Routers themselves have multiple network interfaces, one interface for each connection to another router, as well as 6 internal main functional blocks.

The Router functions include:

- *Management Console* – which provides a control interface from the outside world

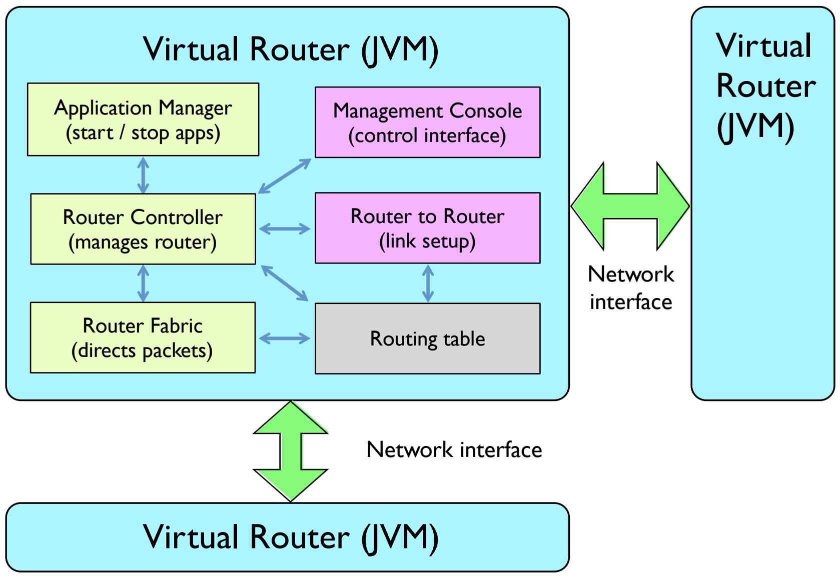
Virtual Router (JVM)
Application Manager (start / stop apps)
Management Console (control interface)
Router Controller (manages router)
Router to Router (link setup)
Router Fabric (directs packets)
Routing table
Virtual Router (JVM)
Network interface
Network interface
Virtual Router (JVM)

Figure 5: Router Top Level View with Main Functional Blocks

- *Router to Router* – which provides the mechanism to setup new connections to other routers
- *Router Controller* – which controls the internal operation of the router
- *Router Fabric* – which connects the network interfaces to the router and provides the forwarding mechanism
- *Routing Table* – which holds information about how to route datagrams
- *Application Manager* – which starts up and shuts down applications that might run on the router

Each of these is shown in figure 5.

Each of the functional blocks is manifested into an design element within the interal architecture of the router.

These functions include:

- *MC* – Management Consolewhich provides a control interface from the outside world
- *R2R* – Router to Router which provides the mechanism to setup new connections to other routers
- *RC* – Router Controller which controls the internal operation of the router
- *RF* – Router Fabric which connects the network interfaces to the router and provides the forwarding mechanism

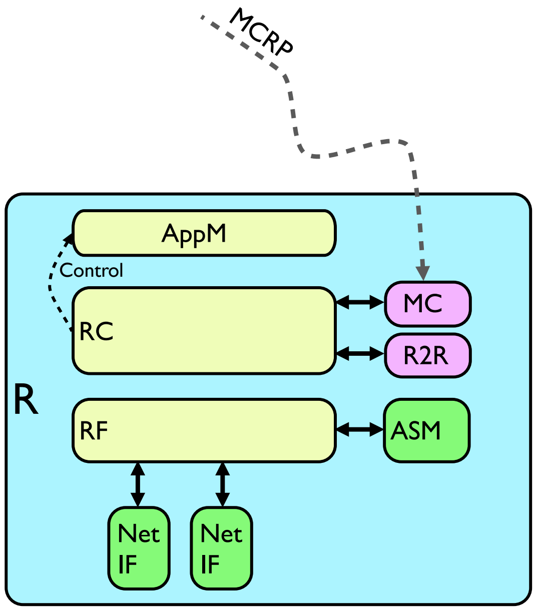

Figure 6: Router Internal Architecture

- *AppM* – Application Manager which starts up and shuts down applications that might run on the router
- *NetIF* – a Network Interface to another virtual Router.
- *ASM* – the Application Socket Mux - which is the Network Interface to the local router. All *Sockets* for local apps are multiplexed through this interface.

Each of these is shown in figure 6.

4.3.1 Connecting Routers

In this section the mechanism by whch one Router connects to another Router in order to create a new virtual network connection is shown.

Each virtual router is connected directly to another virtual router via a Network Interface (NetIF). The connection can be configured to be a TCP connection or a UDP connection.

The TCP connection gives an underlying transport for the traffic that is reliable.

The UDP connection gives an underlying transport for the traffic that is unreliable.

In figure 7, two apps are started on a Router, each with it's own Socket. We see how the Socket is connected to the Router Fabric via the ASM.

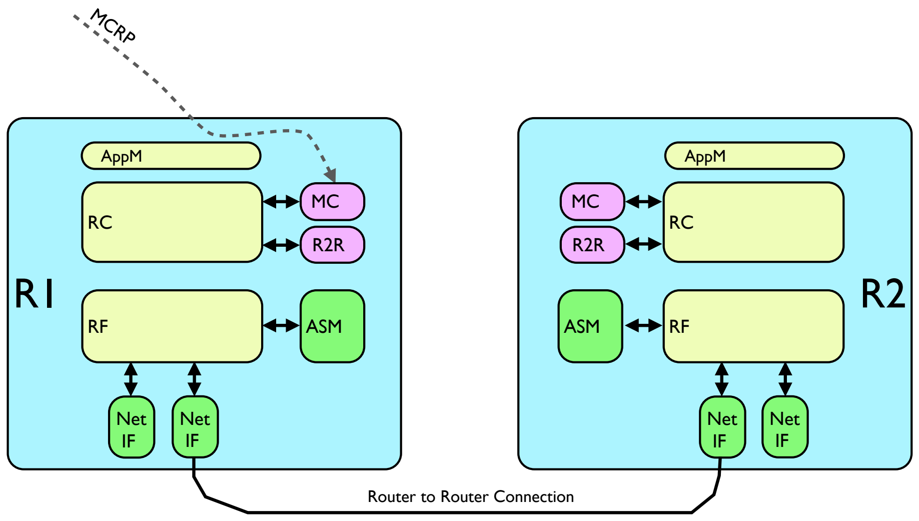

Figure 7: Two Routers

4.3.2 Applications and Application Lifecycle

Applications and services on the platform are run as applications on a router.

Each router has a local network interface that provides a Socket level API for these applications. In appendix 9, there is are full details of this API.

This Socket level API presents a UDP-like socket that can be used for writing networking applications that run directly on the router. These applications are started at run-time by making a call to the Global Controller, passing in the name of the class and the arguments it needs to start up.

5 System Programmability

5.1 Writing Applications and Services

All applications are started under control of the Application Manager and must implement the Java interface `usr.applications.Application`.

```
/**
 * An Application has is a Runnable object that has a managed lifecycle.
 */
public interface Application extends Runnable {
    /**
     * Initialize with some args
     */
```

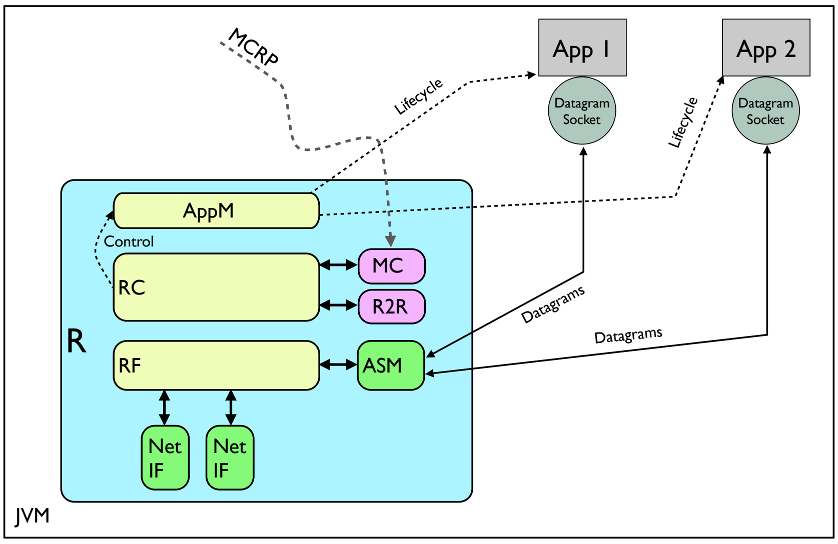

Figure 8: Application Manager and Application Lifecycle

```
    public ApplicationResponse init(String[] args);

    /**
     * Start an application. This is called before run().
     */
    public ApplicationResponse start();

    /**
     * Stop an application.
     * This is called to implement graceful shut down and cause run() to end.
     */
    public ApplicationResponse stop();
}
```

Notice that `Application` extends `Runnable`, so this means that applications must have a `run()` method. This is where the *main* body of the app resides.

The ApplicationResponse object has 2 fields: (i) a *boolean* value to indicate if the app is progressing well, where *true* means success and $false$ mean failure, (ii) a message that may be passed on.

If any of these methods returns an ApplicationResponse with a $false$ value then the system will stop what it is doing, and clean up the application context.

5.1.1 An Example App – Sending Data

Here is an app that opens a Socket and sends datagrams. When it is started it takes 3 arguments: an address to send to, a port to send to, and a count of the number of datagrams.

```
public class Send implements Application {
    Address addr = null;
    int port = 0;
    int count = 0;

    boolean running = false;
    DatagramSocket socket = null;
```

The `init()` method is defined to process all the arguments that are passed in.

```
    public ApplicationResponse init(String[] args) {
        if (args.length == 3) {
            // try address
            try {
                addr = AddressFactory.newAddress(args[0]);

            } catch (Exception e) {
                return new ApplicationResponse(false, "UnknownHost " + args[0]);
            }

            // try port
            Scanner scanner = new Scanner(args[1]);

            if (scanner.hasNextInt()) {
                port = scanner.nextInt();
                scanner.close();
            } else {
                scanner.close();
                return new ApplicationResponse(false, "Bad port " + args[1]);
            }

            // try count
            scanner = new Scanner(args[2]);

            if (scanner.hasNextInt()) {
                count = scanner.nextInt();
                scanner.close();
            } else {
                scanner.close();
                return new ApplicationResponse(false, "Bad count " + args[2]);
            }

            // we got here ok so return true
            return new ApplicationResponse(true, "");

        } else {
            return new ApplicationResponse(false, "Usage: Send address port count");
        }
    }
```

If the application returns an ApplicationResponse with a *true* value then the system will progress to the `start()` method.

In this case the application creates a new socket and connects to the remote router. If the connect fails then the ApplicationResponse is *false* and the application will stop, otherwise it returns an ApplicationResponse with *true* and it progresses to the `run()` method.

```
public ApplicationResponse start() {
    try {
        // set up socket
        socket = new DatagramSocket();

        socket.connect(addr, port);

    } catch (Exception e) {
        return new ApplicationResponse(false, "Cannot open socket " + e.getMessage());
    }

    running = true;

    return new ApplicationResponse(true, "");
}
```

The `run()` method creates a new datagram and sends it onwards.

```
public void run() {
    Datagram datagram = null;

```

```
        // a buffer for data
        byte [] buffer;

        // loop around
        for (int i = 0; i < count; i++) {
            // Code to Fill the Buffer

            datagram = DatagramFactory.newDatagram(buffer);

            try {
                socket.send(datagram);

            } catch (Exception e) {
                if (socket.isClosed()) {
                    break;
                } else {
                    // maybe try again
                }
            }
        }
    }
```

Finally the `stop()` method is shown. This method is called by the Application Manager when the Application is terminated in some way, either by explicitly being stopped by a REST API call or by shutting down the whole system.

```
    public ApplicationResponse stop() {
        running = false;

        if (socket != null) {
            socket.close();
        }

        return new ApplicationResponse(true, "");
    }
```

5.1.2 An Example App – Receiving Data

Here is an app that opens a Socket and receives datagrams. When it is started it takes 1 argument, whic is the port to listen on.

```
public class Recv implements Application {
    int port = 0;

    int count = 0;

    boolean running = false;
    DatagramSocket socket = null;
```

The `init()` method is defined to process the arguments that are passed in.

```
    public ApplicationResponse init(String[] args) {
        if (args.length == 1) {
            // try port
            Scanner scanner = new Scanner(args[0]);

            if (scanner.hasNextInt()) {
                port = scanner.nextInt();
                scanner.close();
            } else {
                scanner.close();
                return new ApplicationResponse(false, "Bad port " + args[1]);
            }

            return new ApplicationResponse(true, "");

        } else {
            return new ApplicationResponse(false, "Usage: Recv port");
        }
    }
```

Again, if the application returns an ApplicationResponse with a *true* value then the system will progress to the `start()` method.

In this case the application creates a new socket and *binds* to the specified port. If this fails then the ApplicationResponse is $false$ and the application will stop, otherwise it returns an ApplicationResponse with *true* and it progresses to the `run()` method.

```
    public ApplicationResponse start() {
        try {
            // set up socket
            socket = new DatagramSocket();

            socket.bind(port);

        } catch (Exception e) {
            return new ApplicationResponse(false, "Cannot open socket " + e.getMessage());
        }

        running = true;

        return new ApplicationResponse(true, "");
    }
```

The `run()` method receives each datagram prints out some meta-data.

```
    public void run() {
        Datagram datagram;

        try {
            while ((datagram = socket.receive()) != null) {

                System.out.print(count + ". ");
                System.out.print("HdrLen: " + datagram.getHeaderLength() +
                                 " Len: " + datagram.getTotalLength() +
                                 " Time: " + (System.currentTimeMillis() -
                                                  datagram.getTimestamp()) +
                                 " From: " + datagram.getSrcAddress() +
                                 " To: " + datagram.getDstAddress() +
                                 ". ");

                byte[] payload = datagram.getPayload();

                if (payload == null) {
                    System.out.print("No payload");
                } else {
                    System.out.print(new String(payload));
                }
                System.out.println("");

                count++;
            }
        } catch (SocketException se) {
            System.out.println(se.getMessage());
        }
    }
```

Finally the `stop()` method is shown. This method is called by the Application Manager when the Application is terminated in some way, either by explicitly being stopped by a REST API call or by shutting down the whole system.

```
    public ApplicationResponse stop() {
        running = false;

        if (socket != null) {
            socket.close();
        }

        return new ApplicationResponse(true, "");
    }
```

5.2 Writing Topology and Management Applications

This section presents the ways of writing topology focused applications and management applications for VLSP. Such applications can include scenarios such as: (i) experiment setup, control, and execution; (ii) orchestration strategies; (iii) dashboards; etc.

5.2.1 VIM Client

The Vim Client, `usr.vim.VimClient` is a Java wrapper for the REST API, detailed in appendix 7. The VimClient methods are shown in detail in appendix 8, It is possible to write a Vim Client in any language, and we have versions of them written in `clojure`, `Julia` and also `python`.

If you do not have VLSP Client yet, you can find the source on *gitlab* at

```
https://gitlab.com/sclayman/vlsp-client
```

In the following example we show how to use the VimClient to create a virtual network. First, a new VimClient is created with a connection to the *GlobalController*. The name of the host running the GlobalController is specified, as well as the port it is listening on.

```
    VimClient conn = new VimClient(gc_host, 8888);
```

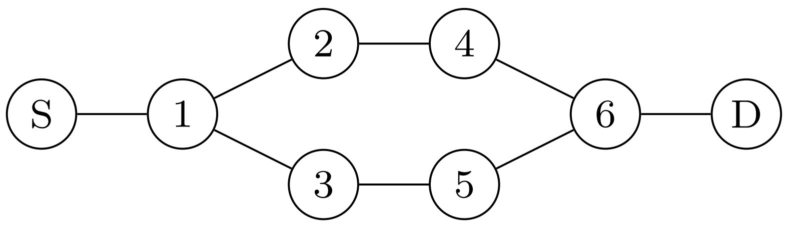

Figure 9: Network - 8 Nodes

Then the virtual routers are created. Each call to `createRouter()` will create a new router, and return a JSON value. The return values are described in Appendix 7 - the REST API Component Interaction.

```
    JSONObject r1 = conn.createRouter();
    int router1 = (Integer)r1.get("routerID");

    JSONObject r2 = conn.createRouter();
    int router2 = (Integer)r2.get("routerID");

    JSONObject r3 = conn.createRouter();
    int router3 = (Integer)r3.get("routerID");

    JSONObject r4 = conn.createRouter();
    int router4 = (Integer)r4.get("routerID");

    JSONObject r5 = conn.createRouter();
    int router5 = (Integer)r5.get("routerID");

    JSONObject r6 = conn.createRouter();
    int router6 = (Integer)r6.get("routerID");

    JSONObject rS = conn.createRouter();
    int routerS = (Integer)rS.get("routerID");

    JSONObject rD = conn.createRouter();
    int routerD = (Integer)rD.get("routerID");
```

Then the links are created between the virtual routers.

```
    JSONObject l1 = conn.createLink(router1, router2, 10);
    int link1 = (Integer)l1.get("linkID");
```

```
  JSONObject l2 = conn.createLink(router1, router3, 10);
  int link2 = (Integer)l2.get("linkID");

  JSONObject l3 = conn.createLink(router2, router4, 10);
  int link3 = (Integer)l3.get("linkID");

  JSONObject l4 = conn.createLink(router3, router5, 10);
  int link4 = (Integer)l4.get("linkID");

  JSONObject l5 = conn.createLink(router4, router6, 10);
  int link5 = (Integer)l5.get("linkID");

  JSONObject l6 = conn.createLink(router5, router6, 10);
  int link6 = (Integer)l6.get("linkID");

  JSONObject lSto1 = conn.createLink(routerS, router1, 10);
  int linkSto1 = (Integer)lSto1.get("linkID");

  JSONObject lDto6 = conn.createLink(routerD, router6, 10);
  int linkDtoS = (Integer)lDto6.get("linkID");
```

5.3 Extending The VIM

The Virtual Infrastructure Manager (VIM) is the Global Controller in VLSP. It has been designed to be adaptable and extendtable for various needs. It implements a class called: *Lifecycle*.

```
public class MyVim extends GlobalController {
  ....
}
```

There are 3 main lifecycle methods:
Init the VIM

▷ `public boolean init();`

Start the VIM

▷ `public boolean start();`

Stop the VIM

▷ `public boolean stop();`

5.3.1 GlobalController Hooks

Also there are a set of hooks / callback functions that have been embedded in the Global Controller that can be called in subclasses to allow flexible operation on top of the normal behaviour of the Global Controller.
This is called at then end of init phase.

▷ `public boolean postInitHook();`

This is called at then beginning of the start phase.

▷ `public boolean preStartHook();`

This is called at then end of the start phase.

▷ `public boolean postStartHook();`

This is called at then beginning of the stop phase.

▷ `public boolean preStopHook();`

This is called at then end of the stop phase.

▷ `public boolean postStopHook();`

This is called at regular intervals during normal execution.

▷ `public boolean runLoopHook();`

This is called at then beginning of the shutting down the system.

▷ `public boolean preShutdownHook();`

This is called at then end of the shutting down the system. It is nearly the last operation.

▷ `public boolean postShutdownHook();`

This is called just after a router is started.

▷ `public boolean routerStartedHook(int id);`

This is called just after a router is stopped.

▷ `public boolean routerEndedHook(int id);`

This is called at then just before a LocalController is started.

▷ `public boolean preLocalControllerStartHook(LocalControllerInfo lcInfo);`

This is called at then just after a LocalController is started.

▷ `public boolean postLocalControllerStartHook(LocalControllerInfo lcInfo);`

This is called at then just before a LocalController is stopped.

▷ `public boolean preLocalControllerStopHook(LocalControllerInfo lcInfo);`

This is called at then just after a LocalController is stopped.

▷ `public boolean postLocalControllerStopHook(LocalControllerInfo lcInfo);`

5.3.2 Writing A Placement Engine

As stated earlier, the *Placement Engine* of VLSP is the management component in charge of performing the actual placement of the virtual entities and the application nodes they host, according to the initial topology and the resource usage of the virtual network elements. The system already comes with the following Placement Engines:

`usr.globalcontroller.LeastBusyPlacement`
`usr.globalcontroller.LeastUsedLoadBalancer`
`usr.globalcontroller.NupPlacement`
`usr.globalcontroller.EnergyEfficientPlacement`

All Placement Engines classes need to implement this Java interface: `usr.globalcontroller.PlacementEngine` which shown here:

```
 /**
 * A PlacementEngine is repsonsible for determining the placement
 * of a Router across the active resources.
 */
public interface PlacementEngine {

   /**
    * Get the relevant LocalControllerInfo for a placement of a router with
    * a specified name and address.
    */
   public LocalControllerInfo routerPlacement(String name, String address);

   /**
    * Get the relevant LocalControllerInfo for a placement of a router with
    * a specified name and address, but also supplying extra parameters. It
    * is used for prediction of future load.
    */
   public LocalControllerInfo routerPlacement(String name, String address,
                                              String parameters);

   /**
    * Get all the possible placement destinations
```

```
     */
    public Set<LocalControllerInfo> getPlacementDestinations();
}
```

Any time a new Router is created the router creation mechanism will eventually call into the GlobalController calling either this method:

```
    /**
     * Do some placement calculation with extra parameter passing
     */
    public LocalControllerInfo placementForRouter(String name, String address) {
        return placementEngine.routerPlacement(name, address);
    }
```

or this method

```
    /**
     * Do some placement calculation with extra parameter passing
     */
    public LocalControllerInfo placementForRouter(String name, String address,
                                                  String parameters) {
        return placementEngine.routerPlacement(name, address, parameters);
    }
```

depending on whether *parameters* were passed in from the outside.

Both of these methods call directly into the PlacementEngine with a name and address the system has chosen, plus any parameters passed in from the outside, if that option was called. The value returned holds information about the nominated LocalController for the actual placement. Using this returned value, the GlobalController will actually start the Router on that host.

6 Appendix - Configuration Options

The following table presents the configuration options for the Global Controller.

6.1 Global Controller Configuration Options

Table 13: Global Controller Configuration Options

Configuration Options	
Field	**Description**
`<SimOptions>` (required)	The top level node for all configuration options to the Global Controller.
`<GlobalController>` (optional)	The node for specific options for the Global Controller. Only one such section is allowed.
`<GlobalController>` → `<Simulation> (optional)`	If true, the Global Controller simulates routers rather than starting them on local controllers. Boolean, default is `false`.
`<GlobalController>` → `<Port> (optional)`	The port that the Global Controller should listen on. Integer value – default is `8888`.
`<GlobalController>` → `<StartLocalControllers>` `(optional)`	Should the the Global Controller start any localcontrollers, if necessary. Boolean, default is `true`. If false GC assumes local controllers started by hand.
`<GlobalController>` → `<ConnectedNetwork>` `(optional)`	If true the GC will add a link to connect the network if it becomes disconnected. Boolean, default is `false`.
`<GlobalController>` → `<AllowIsolatedNodes>` `(optional)`	If false the GC will reconnect a node which has no links. Boolean, default is `true`.

Table 13: Global Controller Configuration Options

Configuration Options	
Field	**Description**
`<GlobalController>` → `<PlacementEngineClass>` `(optional)`	The name of the class of the placement engine in the Global Controller. String, default is `usr.globalcontroller.LeastUsedLoadBalancer`.
`<GlobalController>` → `<VisualizationClass>` `(optional)`	The name of the class for Visualization in the Global Controller. String, there is no default.
`<GlobalController>` → `<Monitoring>` `(optional)`	Should the Global Controller start the Lattice monitoring framework on the routers. Boolean, default is `false`.
`<GlobalController>` → `<RemoteLoginUser>` `(optional)`	Provide a user name to create ssh tunnel when starting local controllers (can be overridden per controller) String, no default.
`<GlobalController>` → `<LowPort>` (optional)	Lowest port to be used for listening by local controllers (can be overridden by individual controllers) Integer, default is `10000`.
`<GlobalController>` → `<HighPort>` `(optional)`	Highest port to be used for listening by local controllers (can be overridden by individual controllers) Integer, default is `20000`.
`<LocalController>` `(optional)`	The node for specific options for the Local Controller. One such section should be present for every physical machine in the test bed. (For simulation there should be none.)

Table 13: Global Controller Configuration Options

Configuration Options	
Field	**Description**
`<LocalController> → <Name> (compulsory)`	The name of the host that the Local Controller should run on. String (no default)
`<LocalController> → <Port> (compulsory)`	The port that the Local Controller should listen on. Integer (no default)
`<LocalController> → <LowPort> (optional)`	The lowest port number that the Local Controller should start a Router listening on. Integer (default inherited from GlobalController)
`<LocalController> → <HighPort> (optional)`	The highest port number that the Local Controller should start a Router listening on. Integer (default inherited from GlobalController)
`<LocalController> → <MaxRouters>`	The maximum number of Routers that a Local Controller should start on the host. Integer (default 100)
`<LocalController> → <RemoteLoginUser>`	The username used to login to this controller via ssh to start LocalController String (no default – use current user as default) but can be inherited from GC
`<LocalController> → <RemoteStartController>`	Command string used to start LocalController on machine String (no default but see below) if not set use `java -cp [Classpath from environment] usr.localcontroller.LocalController`

Table 13: Global Controller Configuration Options

Configuration Options	
Field	**Description**
`<EventEngine>`	The node for specific options for the Event engine.
`<EventEngine>` → `<Name>`	The name of the probability distribution function.
`<RouterOptions>`	The node for specific options for the routers.
`<Output>`	The node for specific options for generating output from a run.

The following table presents the configuration options for the router configuration.

6.2 Router Configuration Options

Table 14: Router Configuration Options

Configuration Options	
Field	**Description**
`<RouterOptions>`	The top level node for all configuration options to the Router.
`<Router>` (optional)	The node for specifying the actual Router class.
`<Router>` → `<RouterClass>`	The class name of the virtual router to start. String: The default is `usr.router.Router`.
`<RoutingParameters>`	The node for specific options for the routing.

Table 14: Router Configuration Options

Configuration Options	
Field	**Description**
`<RoutingParameters>` → `<LinkType>`	What kind of transport is used for the virtual link: either UDP or TCP. Default option is `TCP`.
`<RoutingParameters>` → `<MaxCheckTime>`	How many millis to wait between checks of routing table. Example option is `60000`.
`<RoutingParameters>` → `<MinNetIFUpdateTime>`	Shortest interval between routing updates down given NetIF. Example option is `5000`.
`<RoutingParameters>` → `<MaxNetIFUpdateTime>`	Longest interval between routing updates down given NetIF. Example option is `30000`.
`<RoutingParameters>` → `<DatagramType>`	String: Value is class name which will be datagram class used by network. E.g. usr.net.GIDDatagram. String must point to valid class implementing Datagram interface.
`<APManager>`	The node for specific options for the AP-Manager.
`<APManager>` → `<Name>`	The name of the Aggregation Point selection algorithm. Options are `None`, `Pressure`, `Random`, or `HotSpot`
`<APManager>` → `<MaxAPs>`	The maximum number of APs allowed.
`<APManager>` → `<MinAPs>`	The minimum number of APs allowed.
`<APManager>` → `<RouterConsiderTime>`	Time router reconsiders APs.
`<APManager>` → `<ControllerConsiderTime>`	Time controller reconsiders APs.

Table 14: Router Configuration Options

Configuration Options	
Field	**Description**
`<APManager> → <MaxAPWeight>`	Maximum link weight an AP can be away.
`<APManager> → <MinPropAP>`	Minimum proportion of APs in the network.
`<APManager> → <MonitorType>`	Maximum proportion of APs in the network.
`<Output>`	The node for specific options for generating output from a run.

The following table presents the configuration options for generating using the probability distributions.

6.3 Probabilistic Generation Configuration Options

Table 15: Probabilistic Generation Configuration Options

Configuration Options	
Field	**Description**
`<ProbabilisticEngine>`	The top level node for all configuration options to the Event engine.
`<NodeBirthDist>`	The node for specific options for Router node birth
`<NodeBirthDist> → <ProbElement>`	A probability distribution specification for a NodeBirthDist.
`<NodeBirthDist> → <ProbElement> → <Type>`	The type of the probability distribution. Options are `Uniform`, `Exponential`, `LogNormal`, or `PoissonPlus`

Table 15: Probabilistic Generation Configuration Options

Configuration Options	
Field	**Description**
`<NodeDeathDist>`	The node for specific options for Router node death .
`<NodeDeathDist>` → `<ProbElement>`	.
`<NodeDeathDist>` → `<ProbElement>` → `<Type>`	The type of the probability distribution. Options are `Uniform`, `Exponential`, `LogNormal`, or `PoissonPlus`
`<LinkCreateDist>`	The node for specific options for creation of new links .
`<LinkCreateDist>` → `<ProbElement>`	.
`<LinkCreateDist>` → `<ProbElement>` → `<Type>`	The type of the probability distribution. Options are `Uniform`, `Exponential`, `LogNormal`, or `PoissonPlus`

7 Appendix - REST API Component Interaction

In this Appendix there is a description of the REST API Component Interactions, with explaining each request and response. It describes the interface and lists the resources that are exposed by the Global Controller / Virtual Infrastructure Manager. It is followed by description of all of the REST calls for each of the resources that VLSP can interact with.

The Global Controller is the element that has access from other components. The Local Controller is not publically accesssible, and is there to support the Virtual Infrastructure Manager in its role of managing the lifecycle of Virtual Routers and Virtual Compute nodes. As such, we will only document the interface of the Virtual Infrastructure Manager, as it is the only element that can be addressed directly.

The resources that are addressible via the Virtual Infrastructure Manager are:

- Routers
- Links
- Applications

from any of the API calls.

API Calls

The interface is grouped into API calls for:

- Routers
- Links
- Links on a Router
- Applications on a Router
- Router Link Stats
- Shutdown for Routers

with information on the HTTP *Call*, the arguments, the elements in the JSON *Response*, and an *Example* of a response.

7.1 Router

This section has the REST calls for Router resources

7.1.1 Create Router

This call creates a router with the specified args

Call

```
POST http://host:8888/router/?args
```

Args

```
[name]
[address]
[parameters]
```

Response

```
routerID
name
address
mgmtPort
r2rPort
time
```

Example

```
{"routerID":1, "address":"1", "name":"Router-1", "mgmtPort":11001,
    "r2rPort":11002, "time":1362054061000}
```

7.1.2 Delete Router

This deletes a specified router

Call

```
DELETE http://host:8888/router/2
```

Args

```
Router id as final path element.
No args defined yet.
```

Response

```
routerID
status
```

Example

```
{"routerID": 10, "status": "deleted"}
```

7.1.3 List All Routers

This will list all routers and return all router ids.

Call

```
GET http://host:8888/router/
```

Args

```
[detail] = one of: id | all

[name]
[address]
```

Response

```
type
list
```

Example

```
{ "type": "router", "list": [1, 2, 4, 7]}
```

7.1.4 Get Router Info

This will get info on a specified router

Call

```
GET http://host:8888/router/2
```

Args

```
Router id as final path element.
No args defined yet.
```

Response

```
routerID
name
address
links
mgmtPort
r2rPort
time
```

Example

```
{"routerID":2, "address":"2", "name":"Router-2", "links":[1], "mgmtPort":11003,
    "r2rPort":11004, "time":1361817254727}
```

7.1.5 Get The Router Count

This will get the current number of routers in the system

Call

```
GET http://host:8888/router/count
```

Args

```
count is final path element.
No args defined yet.
```

Response

```
value
```

Example

```
{ "value": 5 }
```

7.1.6 Get Maximum ID of a Router

This will get the maximum ID of a router in the system

Call

```
GET http://host:8888/router/maxid
```

Args

```
maxid is final path element.
No args defined yet.
```

Response

```
value
```

Example

```
{ "value": 42 }
```

7.2 Link

This section has the REST calls for Link resources. It deals with all links in a network managed by the NEM.

7.2.1 Create Link

This will create a link between specified Routers. The link will be added to the network.

Call

```
POST http://host:8888/link/?args
```

Args

```
router1
router2
[weight]
[linkName]
```

Response

```
linkID
linkName
weight
nodes
time
```

Example

```
{"linkID":196612, "linkName":"Router-1.Connection-0", "nodes":[1,2],
    "time":1362079103160,"weight":10}
```

7.2.2 Delete Link

This will remove a link from the network

Call

```
DELETE http://host:8888/link/7
```

Args

```
Link id as final path element.
No args defined yet.
```

Response

```
linkID
status
```

Example

```
{"linkID": 196612, "status": "deleted"}
```

7.2.3 List All Links

This will list all links and return all link ids.

Call

```
GET http://host:8888/link/
```

Args

```
[detail] - one of: id | all
```

Response

```
type
list
[detail]
```

Example

```
{"type": "link", "list": [1, 2, 5]}


Where 'detail=all'

{"type":"link", "list":[2293796,196612,1835044,1179673,655376,262153],
  "detail":[{"id":2293796,"name":"Router-5.Connection-0","nodes":[5,6],
             "time":1361989254649,"weight":10},
             {"id":196612,"name":"Router-1.Connection-0","nodes":[1,2],
             "time":1361989254649,"weight":10},
             {"id":1835044,"name":"Router-4.Connection-0","nodes":[4,6],
             "time":1361989254649,"weight":10},
             {"id":1179673,"name":"Router-3.Connection-0","nodes":[3,5],
             "time":1361989254649,"weight":10},
             {"id":655376,"name":"Router-2.Connection-0","nodes":[2,4],
             "time":1361989254649,"weight":10},
             {"id":262153,"name":"Router-1.Connection-1","nodes":[1,3],
             "time":1361989254649,"weight":10}}],
}
```

7.2.4 Get Link Info

This will get info on a specified link

Call

`GET http://host:8888/link/8`

Args

```
Link id as final path element.
No args defined yet.
```

Response

```
linkID
linkName
weight
nodes
time
```

Example

```
{"linkID":8, "linkName":"Router-5.Connection-0", "weight":10, "nodes":[5,6],
    "time":1362079709109}
```

7.2.5 Set Link Weight

This will set a new weight on a specified link

Call

```
PUT http://host:8888/link/8?weight=30
```

Args

```
Link id as final path element.
weight
```

Response

```
linkID
linkName
weight
nodes
time
```

Example

```
{"linkID":8, "linkName":"Router-5.Connection-0", "weight":30, "nodes":[5,6],
    "time":1362079709109}
```

7.2.6 Get The Link Count

This will get the current number of links in the system

Call

```
GET http://host:8888/link/count
```

Args

```
count is final path element.
No args defined yet.
```

Response

```
value
```

Example

```
{ "value": 17 }
```

7.3 Links on a Router

This section has the REST calls for Link resources attached to a specified Router. It only deals with Links for the specified Router.

7.3.1 List Router Links

This call lists all links on a specified router

Call

```
GET http://host:8888/router/9/link/
```

Args

```
[attr]  - one of:  id | name | weight | connected
```

Response

```
routerID
type
list
```

Example

```
{"routerID": 9,"type":" link", "list":[9830481,7078009,6488164]}
```

7.3.2 Get Router Link Info

This will get info on a specified link on a specific router

Call

```
GET http://host:8888/router/9/link/12
```

Args

```
Link id as final path element.
No args defined yet.
```

Response

```
linkID
linkName
weight
nodes
time
```

Example

```
{"linkID":12, "linkName":"Router-9.Connection-0" ,"weight":10, "nodes":[9,10],
    "time":1362080281512}
```

7.4 Applications on a Router

This section has the REST calls for App resources

7.4.1 Create App

This call creates an app on a specified router

Call

```
POST http://host:8888/router/7/app/?args
```

Args

```
Router id and 'app'.
className
args
```

Response

```
appID
appName
classname
args
routerID
starttime
runtime
```

Example

```
{"appID":2, "appName":"/Router-7/App/usr.applications.Send/1",
    "classname":"usr.applications.Send", "args":"[8, 4000, 2500000, -s, 1024]",
    "routerID":7, "starttime":1362090555686,"runtime":0}
```

7.4.2 List Apps

This call lists all apps on a specified router

Call

```
GET http://host:8888/router/7/app/
```

Args

```
Router id and 'app'.
None.
```

Response

```
type
app
```

Example

```
{"type": "app", "list": [2, 3]}
```

7.4.3 Get App Info

This will get info on a specified app on a specific router

Call

```
GET http://host:8888/router/7/app/2
```

Args

```
Router id and 'app'.
App id as final path element.
```

Response

```
appID
appName
classname
args
routerID
starttime
runtime
```

Example

```
{"appID":2, "appName":"/Router-7/App/usr.applications.Send/1",
    "classname":"usr.applications.Send", "args":"[8, 4000, 2500000, -s, 1024]",
    "routerID":7, "starttime":1362090555686, "runtime": 42854}
```

7.5 Router Link Stats

This section has the REST calls for getting the link stats for Link resources attached to a specified Router. It only deals with Links for the specified Router.

7.5.1 Get Router Link Stats All

This call returns link stats for links on a specified router

Call

`GET http://host:8888/router/9/link_stats/`

Args

`Router id and 'link_stats'.`
`None.`

Response

`value`

Example

```
{"type": "link\_stats", "links": [12 30 104], "link_stats":[
    ["Router-12 Router-7.Connection-2" 358050 1714 0 21 337824 1656 228431
        1078 0 0 205428 1007 0 1 0 1],
    ["Router-30 Router-7.Connection-7" 155976 715 0 5 136884 671 73858 302 0 0
        53040 260 0 1 0 2],
    ["Router-104 Router-7.Connection-19" 50203 232 0 0 45900 225 2067 3 0 0 0
        0 0 1 0 1] ],
  "routerID": 7}
```

7.5.2 Get Router Link Stats

This call returns link stats for links on a specified router to a specified router

Call

`GET http://host:8888/router/9/link_stats/12`

Args

`Router id and 'link_stats' and dst router id`
`None.`

Response

`value`

Example

```
  {"type": "link\_stats", "links": [12], "link_stats": [
      ["Router-12 Router-7.Connection-2" 358050 1714 0 21 337824 1656 228431
          1078 0 0 205428 1007 0 1 0 1]],
    "routerID": 7}
```

7.6 Shutdown for Routers

This section has the REST calls for getting info about Routers that have shut down.

7.6.1 List All Shutdown Routers

This will list all the routers that have been shutdown and removed from the system

Call

`GET http://host:8888/removed/`

Args

`None.`

Response

`type`
`list`

Example

```
{"type":"shutdown", "list": [{"routerID":1,"time":1362133396696},
    {"routerID":2,"time":1362133396696}, {"routerID":3,"time":1362133396696}]}
```

7.7 Local Controllers

This section has the REST calls for getting info about Local Controllers.

7.7.1 List Local Controllers

This will list all LocalControllers and return all LocalController ids. A LocalController id is not an interger, but a hostname:port pair.

Call

```
GET http://host:8888/localcontroller/
```

Args

```
Optional: [detail] = all
```

Response

```
type
list
```

Example

```
{"list":["clayone:10000"],"type":"localcontroller"}
```

7.7.2 Get LocalController Info

This will get info on a specific LocalController.

Call

```
GET http://host:8888/localcontroller/localhost:10000
```

Args

`LocalController id as final path element.`

Response

`detail`

Example

```
{"detail":[{"IP":"localhost/127.0.0.1", "maxRouters":100,
    "name":"localhost:10000", "noRouters":0, "port":10000, "routers":[],
    "status":"ONLINE", "usage":0}], "list":["localhost:10000"],
    "type":"localcontroller"}
```

7.8 Summary Table

Task	**HTTP Method**	**URI**	**Args**	**Response**
Create Router	POST	`http://host:8888/router/?args`	[name] [address] [parameters]	routerID name address mgmtPort r2rPort time
Delete Router	DELETE	`http://host:8888/router/2`	router ID	routerID status
List All Routers	GET	`http://host:8888/router/`	[detail] = id | all	type: "router" list: [1, 2, 4, 7]
Get Router Info	GET	`http://host:8888/router/2`	router ID	routerID name address links mgmtPort r2rPort time
Get No Of Routers	GET	`http://host:8888/router/count`		value

Task	**HTTP Method**	**URI**	**Args**	**Response**
Get Max Router ID	GET	`http://host:8888/router/maxid`		value
Create Link	POST	`http://host:8888/link/?args`	router1 router2 [weight] [linkName]	linkID linkName weight nodes time
Delete Link	DELETE	`http://host:8888/link/7`		linkID status
List All Links	GET	`http://host:8888/link/`	[detail] = id | all	type: ”link” list: [8, 12, 14, 27]
Get Link Info	GET	`http://host:8888/link/8`	link ID	linkID linkName weight nodes time
Set Link Weight	PUT	`http://host:8888/link/8?args`	weight	inkID linkName weight nodes time
Get No Of Links	GET	`http://host:8888/link/count`		value
List Router Links	GET	`http://host:8888/router/9/link/`	[attr] = id | name | weight | connected	routerID: 9 type: ”link” list: [12, 14]
Get Router Link Info	GET	`http://host:8888/router/9/link/12`	router ID	linkID linkName weight nodes time

Task	**HTTP Method**	**URI**	**Args**	**Response**
Create App	POST	`http://host:8888/router/9/app/?args`	routerID className args	appID appName classname args routerID starttime runtime
List Apps	GET	`http://host:8888/router/9/app/`		type: ”app” list: [2, 3]
Get App Info	GET	`http://host:8888/router/9/app/2`	app ID	appID appName classname args routerID starttime runtime
Get Router Link Stats	GET	`http://host:8888/router/9/link_stats`		routerID: 9 type: ”link_stats” links: link_stats:
Get Router Link Stats	GET	`http://host:8888/router/9/link_stats/12`	router ID	routerID: 9 type: ”link_stats” links: link_stats:
List All Shut-down Routers	GET	`http://host:8888/removed/`		type: ”shutdown” list: []
List Local Con-trollers	GET	`http://host:8888/localcontroller/`	[detail] = all	type: ”local-controller” list: [”clay-one:10000”]

Task	**HTTP Method**	**URI**	**Args**	**Response**
Get Local Con-troller Info	GET	`http://host:8888/localcontroller/clayone:10000`	local controller ID	detail

8 Appendix - Vim Client Methods

The following table shows the Java methods for the VimClient. There is a method for each of the REST API calls. This allows for full operability using the Java VIM Client.

The returned values are of type `JSONObject` and they contain the *Response* values presented in Appendix 7.

Create a new router.

▷ `JSONObject createRouter()`

Create a new router, passing in some parameters which are given to the PlacementEngine.

▷ `JSONObject createRouter(String parameters)`

Create a new router, passing in its name and address.

▷ `JSONObject createRouter(String name, String address)`

Create a new router, passing in its name and address, and some parameters which are given to the PlacementEngine.

▷ `JSONObject createRouter(String name, String address, String parameters)`

Create a new router, passing in its name.

▷ `JSONObject createRouterWithName(String name)`

Create a new router, passing in an address.

▷ `JSONObject createRouterWithAddress(String address)`

Delete a router, given its *id*.

▷ `JSONObject deleteRouter(int routerID)`

Get a list of all the routers.

▷ `JSONObject listRouters()`

Get a list of all the routers, passing in *detail*.

▷ `JSONObject listRouters(String detail)`

Get a list of all the routers which have been removed from the system.

▷ `JSONObject listRemovedRouters()`

Gets info on a router specified by *id*.

▷ `JSONObject getRouterInfo(int id)`

Gets information on all the links of a router specified by *id*.

▷ `JSONObject getRouterLinkStats(int id)`

Gets information on the link of a router specified by *id*, going to another router specified by *dstID*.

▷ `JSONObject getRouterLinkStats(int id, int dstID)`

This will get the current number of routers in the system.

▷ `JSONObject getRouterCount()`

This will get the maximum ID of a router in the system.

▷ `JSONObject getMaxRouterID()`

Create a link between 2 routers, specifying the *ids* for both routers.

▷ `JSONObject createLink(int routerID1, int routerID2)`

Create a link between 2 routers, specifying the *ids* for both routers and a link weight.

▷ `JSONObject createLink(int routerID1, int routerID2, int weight)`

Create a link between 2 routers, specifying the *ids* for both routers, a link weight, and a link name.

▷ `JSONObject createLink(int routerID1, int routerID2, int weight, String linkName)`

Delete a link, given its *id*.

▷ `JSONObject deleteLink(int linkID)`

Get a list of all the links.

▷ `JSONObject listLinks()`

Get a list of all the links, passing in *detail*.

▷ `JSONObject listLinks(String detail)`

Gets info on a link specified by *id*.

▷ `JSONObject getLinkInfo(int id)`

Set the weight of a link specified by *id*.

▷ `JSONObject setLinkWeight(int linkID, int weight)`

This will get the current number of links in the system.

▷ `JSONObject getLinkCount()`

Get a list of all the links on a router specified by *rid*.

▷ `JSONObject listRouterLinks(int rid)`

Gets info on all links on router specified by *rid* and *attr*. Values for attr are one of: [id | name | weight | connected].

▷ `JSONObject listRouterLinks(int rid, String attr)`

Gets info on a specific link on a router specified by *rid* and *linkID*.

▷ `JSONObject getRouterLinkInfo(int rid, int linkID)`

Create a new Application running on router, specified by *rid*. The class name of the app is *className* and the args for the app are specified in *args*.

▷ `JSONObject createApp(int rid, String className, String args)`

Stop an app, specified by *appID*, on a router specified by *rid*.

▷ `JSONObject stopApp(int rid, int appID)`

List all apps on router *rid*.

▷ `JSONObject listApps(int rid)`

Get info for an app, specified by *appID*, on a router specified by *rid*.

▷ `JSONObject getAppInfo(int rid, int appID)`

List all aggregation points.

▷ `JSONObject listAggPoints()`

Get info on an aggregation point specified by *id*.

▷ `JSONObject getAggPointInfo(int id)`

Tell router *rid* that it's aggregation point is router specified as *apID*.

▷ `JSONObject setAggPoint(int apID, int rid)`

Get a list of all the LocalControllers.

▷ `JSONObject listLocalControllers()`

Get information on a LocalController given a *name* which looks like host:port.

▷ `JSONObject getLocalControllerInfo(String name)`

Change the status of a LocalController given a *name* which looks like host:port, to a new status. Values for status are: "online" or "offline".

▷ `JSONObject setLocalControllerStatus(String name, String status)`

Get info on a LocalController, specified by *name*.

▷ `JSONObject getLocalControllerInfo(String name)`

9 Appendix - Socket Interface Methods

The following table shows the standard Java interface for DatagramSockets. It shows the method, what the method does, and whether it is supported by the User Space Routing framework.

Table 17: DatagramSocket Method Support

Method Support		
Method	**Description**	**In USR**
`DatagramSocket()`	Constructs a datagram socket and binds it to any available port on the local host machine.	YES
`DatagramSocket(int port)`	Constructs a datagram socket and binds it to the specified port on the local host machine.	YES
`DatagramSocket(int port, InetAddress addr)`	Creates a datagram socket, bound to the specified local address.	REPL
Replacement ⇒	`DatagramSocket(Address addr, int port)`	
`DatagramSocket(SocketAddress bindaddr)`	Creates a datagram socket, bound to the specified local socket address.	NO
`void bind(SocketAddress addr)`	Binds this DatagramSocket to a specific address & port.	REPL
Replacement ⇒	`void bind(int port)`	
`void close()`	Closes this datagram socket.	YES
`void connect(InetAddress addr, int port)`	Connects the socket to a remote address for this socket.	REPL
Replacement ⇒	`void connect(Address address, int port)`	
`void connect(SocketAddress addr)`	Connects this socket to a remote socket address (IP address + port number).	YES

Table 17: DatagramSocket Method Support

Method Support		
Method	**Description**	**In USR**
`void disconnect()`	Disconnects the socket.	YES
`InetAddress getInetAddress()`	Returns the address to which this socket is connected.	REPL
Replacement ⇒	`Address getRemoteAddress()`	
`InetAddress getLocalAddress()`	Gets the local address to which the socket is bound.	YES
Replacement ⇒	`Address getLocalAddress()`	
`int getLocalPort()`	Returns the port number on the local host to which this socket is bound.	YES
`SocketAddress getLocalSocketAddress()`	Returns the address of the endpoint this socket is bound to, or null if it is not bound yet.	NO
`int getPort()`	Returns the port for this socket.	YES
`SocketAddress getRemoteSocketAddress()`	Returns the address of the endpoint this socket is connected to, or null if it is unconnected.	NO
`boolean isBound()`	Returns the binding state of the socket.	YES
`boolean isClosed()`	Returns whether the socket is closed or not.	YES
`boolean isConnected()`	Returns the connection state of the socket.	YES
`void receive(DatagramPacket p)`	Receives a datagram packet from this socket.	REPL
Replacement ⇒	`Datagram receive()`	

Table 17: DatagramSocket Method Support

Method Support		
Method	**Description**	**In USR**
`void send(DatagramPacket p)`	Sends a datagram packet from this socket.	REPL
Replacement ⇒	`send(Datagram dg)`	

The DatagramSocket of USR does not support any Socket Options or Traffic Class functions, and so none of these methods are supported.		
`DatagramChannel getChannel()`	Returns the unique DatagramChannel object associated with this datagram socket, if any.	NO
`boolean getBroadcast()`	Tests if SO_BROADCAST is enabled.	NO
`void setBroadcast(boolean on)`	Enable/disable SO_BROADCAST.	NO
`boolean getReuseAddress()`	Tests if SO_REUSEADDR is enabled.	NO
`int getSendBufferSize()`	Get value of the SO_SNDBUF option for this DatagramSocket, that is the buffer size used by the platform for output on this DatagramSocket.	NO
`int getSoTimeout()`	Retrive setting for SO_TIMEOUT.	NO
`int getReceiveBufferSize()`	Get value of the SO_RCVBUF option for this DatagramSocket, that is the buffer size used by the platform for input on this DatagramSocket.	NO
`void setReceiveBufferSize(int size)`	Sets the SO_RCVBUF option to the specified value for this DatagramSocket.	NO

Table 17: DatagramSocket Method Support

Method Support		
Method	**Description**	**In USR**
`void setReuseAddress(boolean on)`	Enable/disable the SO_REUSEADDR socket option.	NO
`void setSendBufferSize(int size)`	Sets the SO_SNDBUF option to the specified value for this DatagramSocket.	NO
`void setSoTimeout(int timeout)`	Enable/disable SO_TIMEOUT with the specified timeout, in milliseconds.	NO
`int getTrafficClass()`	Gets traffic class or type-of-service in the IP datagram header for packets sent from this DatagramSocket.	NO
`void setTrafficClass(int tc)`	Sets traffic class or type-of-service octet in the IP datagram header for datagrams sent from this DatagramSocket.	NO

10 Appendix - Datagram

In this appendix there is a description of the datagrams that are sent by the routers. First there is a datagram outline, then the datagram elements are described in more detail.

USRD	hdr len	total len	flags	ttl	proto col	src addr	dst addr	*pad*	src port	dst port

↪

time stamp	flow id	pay load	check sum

The following table 18 presents each of these elements in more detail, by showing the number of bytes each element takes, and giving a description of the element.

Table 18: Datagram Elements

Datagram Elements			
Element	**Bytes**	**Position**	**Description**
USRD	4	0	The literal String "USRD".
Hdr Len	1	4	The size, in bytes, of the header. This currently set to 36.
Total Len	2	5	The size, in bytes, of the whole Datagram.
Flags	1	7	Some optional flags. This gives 8 bits to play with.
TTL	1	8	The TTL of a Datagram.
Protocol	1	9	The protocol specified for a particular Datagram. Currently there is a CONTROL protocol and a DATA protocol.
Src Addr	4	10	The source address of the Datagram.
Dst Addr	4	14	The destination address of the Datagram.
Pad	2	18	Some spare bytes, currently padded as 2 spaces ▽▽.
Src Port	4	20	The source port of the Datagram.
Dst Port	4	22	The destination port of the Datagram.
Timestamp	8	24	The time the Datagram was created.

Table 18: Datagram Elements

Datagram Elements			
Element	**Bytes**	**Position**	**Description**
Flow ID	4	32	A flow ID.
Payload	N	36	The payload itself. It's size, N, is Total Len − Hdr Len − 4.
Checksum	4	$36 + N$	The checksum.

The following table presents the methods that can be used to manipulate a Datagram object.

TODO

11 Appendix - Log Files

VLSP produces some log files as it executes. These are presented here.

File	Description
/tmp/gc-channel7.out	This file shows a regualar output of the network interface statistics for each router. It is modelled on *netstat − i*.
/tmp/gc-channel8.out	This file shows the management commands for the Aggregation Point, if it is configured.
/tmp/gc-channel9.out	This file shows the main commands for the virtual network.
/tmp/gc-channel10.out	The file has a list of the current running apps on each Router. It is similar to *ps*.
/tmp/gc-channel11.out	This file has an overview of monitoring data arriving. It shows the type of the measurement, the probe ID of the sending probe, plus the sequence number.
/tmp/gc-channel12.out	This file shows the decisions made by the Placement Engine.
/tmp/gc-channel13.out	This file shows the data from the HostInfo probe, if it is configured.
/tmp/gc-channel14.out	This file has the data for each Router, showing CPU usage for each Thread Group, if it is configured.
/tmp/gc-channel15.out	This file has the data for each Router, showing CPU usage for each Thread, if it is configured.
/tmp/Router-*id*-channel6.out	

channel7

```
00:53:31 Router-1 localnet | 624 | 3 | 0 | 0 | 624 | 3 | 1040 | 5 | 0 | 0 | 1040 | 5 | 0 | 1 | 0 |
    0 |
00:53:31 Router-2 Router-1.Connection-0 | 623 | 5 | 0 | 0 | 416 | 2 | 207 | 3 | 0 | 0 | 0 | 0 | 0 |
    1 | 0 | 1 |
00:53:31 Router-3 Router-1.Connection-1 | 146 | 2 | 0 | 0 | 0 | 0 | 146 | 2 | 0 | 0 | 0 | 0 | 0 | 1
    | 0 | 1 |
00:53:31 Router-4 Router-1.Connection-2 | 138 | 2 | 0 | 0 | 0 | 0 | 77 | 1 | 0 | 0 | 0 | 0 | 0 | 1
    | 0 | 1 |
```

channel8

```
00:21:05 ROUTER 1 BECOME AP
00:21:05 ROUTER 2 SET AP 1
00:30:00 ROUTER 3 SET AP 1
00:50:00 ROUTER 4 SET AP 1
01:00:00 ROUTER 5 SET AP 1
01:10:00 ROUTER 6 SET AP 1
```

```
01:20:00 ROUTER 7 SET AP 1
01:30:87 ROUTER 8 SET AP 1
01:40:60 ROUTER 2 BECOME AP
01:40:60 ROUTER 3 SET AP 2
01:40:60 ROUTER 7 SET AP 2
01:40:60 ROUTER 8 SET AP 2
01:40:60 ROUTER 9 SET AP 2
```

channel9

```
00:12:18 START ROUTER 1
00:20:88 START ROUTER 2
00:21:05 CREATE LINK 1 TO 2
00:29:15 START ROUTER 3
00:29:31 CREATE LINK 2 TO 3
00:29:44 CREATE LINK 1 TO 3
00:42:93 START ROUTER 4
00:43:06 CREATE LINK 1 TO 4
00:43:19 CREATE LINK 2 TO 4
00:43:34 CREATE LINK 3 TO 4
00:54:49 START ROUTER 5
00:54:65 CREATE LINK 4 TO 5
00:54:78 CREATE LINK 2 TO 5
00:54:90 CREATE LINK 1 TO 5
00:55:02 CREATE LINK 3 TO 5
01:02:92 REMOVE LINK 4 TO 5
01:02:92 REMOVE LINK 4 TO 3
01:02:92 REMOVE LINK 4 TO 2
01:02:92 REMOVE LINK 4 TO 1
01:02:92 STOP ROUTER 4
01:07:23 START ROUTER 6
```

channel10

```
Router-1:
00:21:22 AID | StartTime | ElapsedTime | RunTime | UserTime | SysTime | State | ClassName | Args |
     Name | RuntimeKeys | RuntimeValues |
00:21:22 2 | 1519134637453 | 6 | 223000 | 103000 | 120000 | RUNNING |
     plugins_usr.aggregator.appl.InfoSource | [-o, 1/3000, -t, 1, -d, 30, -p, rt, -n,
     info-source-1] | /Router-1/App/plugins_usr.aggregator.appl.InfoSource/2 | [] | [] |
00:21:22 1 | 1519134637318 | 141 | 254000 | 152000 | 102000 | RUNNING |
     plugins_usr.aggregator.appl.AggPoint | [-i, 0/3000, -t, 5, -a, average, -n, agg-point-1-1] |
     /Router-1/App/plugins_usr.aggregator.appl.AggPoint/1 | [] | [] |
```

channel11

```
< HostInfo.033bd4e6-ffc5-4adc-bc93-ea985b932520.0
< HostInfo.033bd4e6-ffc5-4adc-bc93-ea985b932520.1
< NetIFStats.b3435bfc-2620-4da9-a096-7996cfd8345e.0
< ThreadGroupList.89fcc524-a7cd-4828-984d-9c1d2b214b9e.0
< ThreadList.eedf60ee-84d2-4909-9a38-030cb760e891.0
< NetIFStats.b3435bfc-2620-4da9-a096-7996cfd8345e.1
< ThreadList.eedf60ee-84d2-4909-9a38-030cb760e891.1
< NetIFStats.b3435bfc-2620-4da9-a096-7996cfd8345e.2
< ThreadList.eedf60ee-84d2-4909-9a38-030cb760e891.2
< NetIFStats.b3435bfc-2620-4da9-a096-7996cfd8345e.3
< ThreadList.eedf60ee-84d2-4909-9a38-030cb760e891.3
```

channel12

```
00:10:96 LeastUsedLoadBalancer: choose localhost:10000 minUse: 0.0 for Router-1/1
00:19:69 LeastUsedLoadBalancer: choose localhost:10000 minUse: 0.0125 for Router-2/2
00:27:95 LeastUsedLoadBalancer: choose localhost:10000 minUse: 0.025 for Router-3/3
00:41:74 LeastUsedLoadBalancer: choose localhost:10000 minUse: 0.0375 for Router-4/4
00:53:31 LeastUsedLoadBalancer: choose localhost:10000 minUse: 0.05 for Router-5/5
01:06:04 LeastUsedLoadBalancer: choose localhost:10000 minUse: 0.05 for Router-6/6
01:14:90 LeastUsedLoadBalancer: choose localhost:10000 minUse: 0.05 for Router-7/7
01:26:84 LeastUsedLoadBalancer: choose localhost:10000 minUse: 0.0625 for Router-8/8
01:39:14 LeastUsedLoadBalancer: choose localhost:10000 minUse: 0.05 for Router-9/9
01:47:78 LeastUsedLoadBalancer: choose localhost:10000 minUse: 0.0625 for Router-10/10
```

channel13

```
Name: localhost:10000 cpu-user: 2.33 cpu-sys: 11.67 cpu-idle: 85.99 load-average: 2.37 mem-used:
    4224 mem-free: 684 mem-total: 4908 net-stats: [<0: if-name: STRING>, <1: in-packets: LONG>,
    <2: in-bytes: LONG>, <3: out-packets: LONG>, <4: out-bytes: LONG>]
[(STRING en0), (LONG 10243227), (LONG 6705175607), (LONG 11143166), (LONG 5038734889)]
[(STRING awdl0), (LONG 2), (LONG 170), (LONG 827), (LONG 327579)]

Name: localhost:10000 cpu-user: 6.52 cpu-sys: 13.4 cpu-idle: 80.43 load-average: 2.5 mem-used: 4227
    mem-free: 606 mem-total: 4833 net-stats: [<0: if-name: STRING>, <1: in-packets: LONG>, <2:
    in-bytes: LONG>, <3: out-packets: LONG>, <4: out-bytes: LONG>]
[(STRING en0), (LONG 10243255), (LONG 6705182119), (LONG 11143180), (LONG 5038736649)]
[(STRING awdl0), (LONG 2), (LONG 170), (LONG 827), (LONG 327579)]
```

channel14

```
Router-1 -- Router-1 -- starttime: [2018/02/20 13:50:27.362] elapsed: [51.126] cpu: [0.701674]
    user: [0.611783] system: [0.089891] mem: 25412536
Router-1 -- localnet -- starttime: [2018/02/20 13:50:27.491] elapsed: [50.997] cpu: [0.001551]
    user: [0.001147] system: [0.000404] mem: 3320
```

```
Router-1 -- TCPNetIF-11003 -- starttime: [2018/02/20 13:50:37.172] elapsed: [41.316] cpu:
    [0.009690] user: [0.007773] system: [0.001917] mem: 137464
Router-1 -- Application-0 -- starttime: [2018/02/20 13:50:37.390] elapsed: [41.098] cpu: [0.000254]
    user: [0.000152] system: [0.000102] mem: 1368
Router-1 -- Application-1 -- starttime: [2018/02/20 13:50:37.467] elapsed: [41.021] cpu: [0.000530]
    user: [0.000330] system: [0.000200] mem: 2032
Router-1 -- TCPNetIF-11005 -- starttime: [2018/02/20 13:50:45.571] elapsed: [32.917] cpu:
    [0.008427] user: [0.006964] system: [0.001463] mem: 109632
Router-1 -- TCPNetIF-11007 -- starttime: [2018/02/20 13:50:59.190] elapsed: [19.298] cpu:
    [0.002350] user: [0.001821] system: [0.000529] mem: 91800
Router-1 -- TCPNetIF-11009 -- starttime: [2018/02/20 13:51:11.036] elapsed: [7.452] cpu: [0.001648]
    user: [0.001196] system: [0.000452] mem: 63528
Router-1 -- TOTALS -- starttime: [2018/02/20 13:50:27.362] elapsed: [51.126] cpu: [0.726124] user:
    [0.631166] system: [0.094958] mem: 25818
```

channel15

```
Router-1 -- /Router-1/RoutingTableTransmitter/587938662 - Router-1 -- starttime: [2018/02/20
    13:50:27.434] elapsed: [44.205] cpu: [0.009569] user: [0.008248] system: [0.001321] mem: 209976
Router-1 -- /Router-1/RouterController/940146249 - Router-1 -- starttime: [2018/02/20 13:50:27.434]
    elapsed: [44.205] cpu: [0.000072] user: [0.000028] system: [0.000044] mem: 0
Router-1 -- /Router-1/RouterConnections/779731315 - Router-1 -- starttime: [2018/02/20
    13:50:27.452] elapsed: [44.187] cpu: [0.005425] user: [0.004781] system: [0.000644] mem: 134176
Router-1 -- FileManager-0 - Router-1 -- starttime: [0.000] elapsed: [0.000] cpu: [0.000090] user:
    [0.000037] system: [0.000053] mem: 0
Router-1 -- ActionDistributor-0 - Router-1 -- starttime: [0.000] elapsed: [0.000] cpu: [0.004524]
    user: [0.003782] system: [0.000742] mem: 12256
Router-1 -- LeaseCleaner-0 - Router-1 -- starttime: [0.000] elapsed: [0.000] cpu: [0.000486] user:
    [0.000376] system: [0.000110] mem: 5440
Router-1 -- ActionDistributor-1 - Router-1 -- starttime: [0.000] elapsed: [0.000] cpu: [0.000748]
    user: [0.000495] system: [0.000253] mem: 728
Router-1 -- ActionDistributor-2 - Router-1 -- starttime: [0.000] elapsed: [0.000] cpu: [0.061469]
    user: [0.058378] system: [0.003091] mem: 1813424
Router-1 -- Notifier-0 - Router-1 -- starttime: [0.000] elapsed: [0.000] cpu: [0.016647] user:
    [0.015788] system: [0.000859] mem: 992136
Router-1 -- Reader-0 - Router-1 -- starttime: [0.000] elapsed: [0.000] cpu: [0.012417] user:
    [0.011085] system: [0.001332] mem: 530160
Router-1 -- Dispatcher-0 - Router-1 -- starttime: [0.000] elapsed: [0.000] cpu: [0.026552] user:
    [0.023705] system: [0.002847] mem: 1135248
Router-1 -- Reader-1 - Router-1 -- starttime: [0.000] elapsed: [0.000] cpu: [0.000711] user:
    [0.000540] system: [0.000171] mem: 9240
Router-1 -- Reader-2 - Router-1 -- starttime: [0.000] elapsed: [0.000] cpu: [0.003765] user:
    [0.003381] system: [0.000384] mem: 26464
Router-1 -- Dispatcher-1 - Router-1 -- starttime: [0.000] elapsed: [0.000] cpu: [0.057997] user:
    [0.052258] system: [0.005739] mem: 1559864
Router-1 -- Reader-3 - Router-1 -- starttime: [0.000] elapsed: [0.000] cpu: [0.000514] user:
    [0.000301] system: [0.000213] mem: 4760
Router-1 -- Reader-4 - Router-1 -- starttime: [0.000] elapsed: [0.000] cpu: [0.001097] user:
    [0.000875] system: [0.000222] mem: 13664
```

```
Router-1 -- Dispatcher-2 - Router-1 -- starttime: [0.000] elapsed: [0.000] cpu: [0.000828] user:
     [0.000721] system: [0.000107] mem: 40160
Router-1 -- Reader-5 - Router-1 -- starttime: [0.000] elapsed: [0.000] cpu: [0.000466] user:
     [0.000274] system: [0.000192] mem: 4744
Router-1 -- Reader-6 - Router-1 -- starttime: [0.000] elapsed: [0.000] cpu: [0.001153] user:
     [0.000909] system: [0.000244] mem: 13592
Router-1 -- Dispatcher-3 - Router-1 -- starttime: [0.000] elapsed: [0.000] cpu: [0.001898] user:
     [0.001748] system: [0.000150] mem: 87792
Router-1 -- Reader-7 - Router-1 -- starttime: [0.000] elapsed: [0.000] cpu: [0.000645] user:
     [0.000493] system: [0.000152] mem: 7408
Router-1 -- Dispatcher-4 - Router-1 -- starttime: [0.000] elapsed: [0.000] cpu: [0.013667] user:
     [0.011445] system: [0.002222] mem: 310992
Router-1 -- Router-1.probe.NetIFStats - Router-1 -- starttime: [0.000] elapsed: [0.000] cpu:
     [0.020871] user: [0.017778] system: [0.003093] mem: 404288
Router-1 -- Router-1.probe.ThreadGroupList - Router-1 -- starttime: [0.000] elapsed: [0.000] cpu:
     [0.015644] user: [0.012141] system: [0.003503] mem: 500088
Router-1 -- Router-1.probe.AppList - Router-1 -- starttime: [0.000] elapsed: [0.000] cpu:
     [0.016773] user: [0.013642] system: [0.003131] mem: 297576
Router-1 -- Router-1.probe.ThreadList - Router-1 -- starttime: [0.000] elapsed: [0.000] cpu:
     [0.066873] user: [0.051732] system: [0.015141] mem: 3126560
Router-1 -- Router-1.dataSource - Router-1 -- starttime: [0.000] elapsed: [0.000] cpu: [0.034326]
     user: [0.027926] system: [0.006400] mem: 1829280
Router-1 -- Notifier-1 - Router-1 -- starttime: [0.000] elapsed: [0.000] cpu: [0.000176] user:
     [0.000102] system: [0.000074] mem: 4256
Router-1 -- Dispatcher-5 - Router-1 -- starttime: [0.000] elapsed: [0.000] cpu: [0.036126] user:
     [0.032171] system: [0.003955] mem: 1054496
Router-1 -- Dispatcher-6 - Router-1 -- starttime: [0.000] elapsed: [0.000] cpu: [0.002260] user:
     [0.001896] system: [0.000364] mem: 54152
Router-1 -- AppStart-Router-1 - Router-1 -- starttime: [2018/02/20 13:50:37.306] elapsed: [34.334]
     cpu: [0.100819] user: [0.091999] system: [0.008820] mem: 5801664
Router-1 -- DataConsumer - Router-1 -- starttime: [0.000] elapsed: [0.000] cpu: [0.016829] user:
     [0.015671] system: [0.001158] mem: 459440
Router-1 -- agg-point-1-1-collected-1514513915-IOThread - Router-1 -- starttime: [0.000] elapsed:
     [0.000] cpu: [0.006752] user: [0.006046] system: [0.000706] mem: 5128
Router-1 -- forwarder_probe - Router-1 -- starttime: [0.000] elapsed: [0.000] cpu: [0.000159] user:
     [0.000077] system: [0.000082] mem: 544
Router-1 -- Thread-10 - Router-1 -- starttime: [0.000] elapsed: [0.000] cpu: [0.010957] user:
     [0.009405] system: [0.001552] mem: 456976
Router-1 -- Thread-11 - Router-1 -- starttime: [0.000] elapsed: [0.000] cpu: [0.004437] user:
     [0.003999] system: [0.000438] mem: 134576
Router-1 -- agg-point-1-1-forwarded-1434775500-IOThread - Router-1 -- starttime: [0.000] elapsed:
     [0.000] cpu: [0.000386] user: [0.000169] system: [0.000217] mem: 0
Router-1 -- agg-point-1-1-filtered-727995600-IOThread - Router-1 -- starttime: [0.000] elapsed:
     [0.000] cpu: [0.000419] user: [0.000197] system: [0.000222] mem: 0
Router-1 -- localhostForwarderDataSource - Router-1 -- starttime: [0.000] elapsed: [0.000] cpu:
     [0.000087] user: [0.000037] system: [0.000050] mem: 32
Router-1 -- info-source-1-log-421031011-IOThread - Router-1 -- starttime: [0.000] elapsed: [0.000]
     cpu: [0.001692] user: [0.001186] system: [0.000506] mem: 4784
Router-1 -- info-source-1 - Router-1 -- starttime: [0.000] elapsed: [0.000] cpu: [0.002089] user:
     [0.001774] system: [0.000315] mem: 19712
```

```
Router-1 -- info-sourceelapsedTime - Router-1 -- starttime: [0.000] elapsed: [0.000] cpu:
     [0.022038] user: [0.020098] system: [0.001940] mem: 510104
Router-1 -- Notifier-2 - Router-1 -- starttime: [0.000] elapsed: [0.000] cpu: [0.000180] user:
     [0.000098] system: [0.000082] mem: 4256
Router-1 -- Dispatcher-7 - Router-1 -- starttime: [0.000] elapsed: [0.000] cpu: [0.008818] user:
     [0.007468] system: [0.001350] mem: 590728
Router-1 -- Notifier-3 - Router-1 -- starttime: [0.000] elapsed: [0.000] cpu: [0.000211] user:
     [0.000128] system: [0.000083] mem: 4256
Router-1 -- Notifier-4 - Router-1 -- starttime: [0.000] elapsed: [0.000] cpu: [0.000170] user:
     [0.000104] system: [0.000066] mem: 4256
Router-1 -- /localnet/InQueue - localnet -- starttime: [2018/02/20 13:50:27.493] elapsed: [44.147]
     cpu: [0.000895] user: [0.000644] system: [0.000251] mem: 1544
Router-1 -- /Router-1.Connection-0/InQueue - TCPNetIF-11003 -- starttime: [2018/02/20 13:50:37.172]
     elapsed: [34.468] cpu: [0.003084] user: [0.002511] system: [0.000573] mem: 74968
Router-1 -- /Router-1.Connection-0/OutQueue - TCPNetIF-11003 -- starttime: [2018/02/20
     13:50:37.173] elapsed: [34.467] cpu: [0.000932] user: [0.000619] system: [0.000313] mem: 5552
Router-1 -- /Router-1.Connection-0/TCPNetIF-11003 - TCPNetIF-11003 -- starttime: [2018/02/20
     13:50:37.173] elapsed: [34.467] cpu: [0.002448] user: [0.002052] system: [0.000396] mem: 18144
Router-1 -- /Router-1/plugins_usr.aggregator.appl.AggPoint/0 - Application-0 -- starttime:
     [2018/02/20 13:50:37.390] elapsed: [34.250] cpu: [0.000254] user: [0.000152] system:
     [0.000102] mem: 1368
Router-1 -- /Router-1/plugins_usr.aggregator.appl.InfoSource/1 - Application-1 -- starttime:
     [2018/02/20 13:50:37.467] elapsed: [34.173] cpu: [0.000530] user: [0.000330] system:
     [0.000200] mem: 2032
Router-1 -- /Router-1.Connection-1/InQueue - TCPNetIF-11005 -- starttime: [2018/02/20 13:50:45.571]
     elapsed: [26.069] cpu: [0.005617] user: [0.004831] system: [0.000786] mem: 72344
Router-1 -- /Router-1.Connection-1/OutQueue - TCPNetIF-11005 -- starttime: [2018/02/20
     13:50:45.571] elapsed: [26.069] cpu: [0.000536] user: [0.000336] system: [0.000200] mem: 5240
Router-1 -- /Router-1.Connection-1/TCPNetIF-11005 - TCPNetIF-11005 -- starttime: [2018/02/20
     13:50:45.571] elapsed: [26.069] cpu: [0.000640] user: [0.000487] system: [0.000153] mem: 5856
Router-1 -- /Router-1.Connection-2/InQueue - TCPNetIF-11007 -- starttime: [2018/02/20 13:50:59.190]
     elapsed: [12.450] cpu: [0.001533] user: [0.001252] system: [0.000281] mem: 64024
Router-1 -- /Router-1.Connection-2/OutQueue - TCPNetIF-11007 -- starttime: [2018/02/20
     13:50:59.190] elapsed: [12.450] cpu: [0.000467] user: [0.000274] system: [0.000193] mem: 4928
Router-1 -- /Router-1.Connection-2/TCPNetIF-11007 - TCPNetIF-11007 -- starttime: [2018/02/20
     13:50:59.190] elapsed: [12.450] cpu: [0.000652] user: [0.000492] system: [0.000160] mem: 5848
Router-1 -- /Router-1.Connection-3/InQueue - TCPNetIF-11009 -- starttime: [2018/02/20 13:51:11.036]
     elapsed: [0.604] cpu: [0.000549] user: [0.000424] system: [0.000125] mem: 30664
Router-1 -- /Router-1.Connection-3/OutQueue - TCPNetIF-11009 -- starttime: [2018/02/20
     13:51:11.036] elapsed: [0.604] cpu: [0.000294] user: [0.000162] system: [0.000132] mem: 4616
Router-1 -- /Router-1.Connection-3/TCPNetIF-11009 - TCPNetIF-11009 -- starttime: [2018/02/20
     13:51:11.037] elapsed: [0.603] cpu: [0.000321] user: [0.000215] system: [0.000106] mem: 4920
```

12 Appendix - Java Packages

The following table shows the Java packages in the User Space Routing framework.

Table 20: Java Packages

Java Packages	
Package	**Description**
`usr.APcontroller`	This package has classes to control allocation of aggregation points.
`usr.applications`	A package for user applications
`usr.common`	This package has utility classes.
`usr.console`	This package provides classes that deal with processing network connections and requests to a ManagementConsole of a component of the system.
`usr.dcap`	This package provides classes that do Datagram capture directly from the network interface.
`usr.engine`	This package provides classes that add events into the system.
`usr.engine.linkpicker`	This package provides classes that chooses links based upon finding the node with some lifetime.
`usr.events`	This package has the generic functions for the Event the system.
`usr.events.globalcontroller`	This package has the functions for the Event the system within the GlobalController.
`usr.events.vim`	This package has the generic interfaces for the Event the system.
`usr.events.vimfunctions`	This package has the functions for the Event subsystem run externally.
`usr.globalcontroller`	This package provides classes that are part of a GlobalController.

Table 20: Java Packages

Java Packages	
Package	**Description**
`usr.globalcontroller.command`	This package provides classes that implement the commands of a GlobalController.
`usr.globalcontroller.visualization`	This package provides classes that will generate a visualization of the current network topology.
`usr.interactor`	This package provides classes that act as a client to a ManagementConsole of a component of the system.
`usr.localcontroller`	This package provides classes that are part of a LocalController.
`usr.localcontroller.command`	This package provides classes that implement the commands of a LocalController.
`usr.logging`	This package provides classes that are used for logging.
`usr.model.abstractnetwork`	This package provides classes that present an abstract network representation.
`usr.model.lifeEstimate`	This package provides classes that produces estimates of life spans given information about node births and deaths.
`usr.net`	This package provides classes that implement networking.
`usr.output`	output
`usr.protocol`	This package provides classes that define the protocols that are used between the components.
`usr.router`	This package provides classes that are part of a Router.
`usr.router.command`	This package provides classes that implement the commands of a Router.
`usr.vim`	This package provides classes that are part of the VIM.

13 Appendix - Router Commands

In this Appendix there is a description of the MCRP protocol, with explaining each request and response.

In this Appendix there is a description of the MCRP protocol, with explaining each request and response.

The commands that can be sent to a Router through its Management Console are shown here. These commands affect the router in various ways, where some are quite lightweight (such as getting the Router's name), and others are more heavyweight (such as connecting the Router to another Router).

Other components connect to the router via the Management Console, acting in essence as a client, to interact with the Router. In most instances, a permanent connection will be made by the Local Controller on the machine that runs the Router. The Router also accepts multiple connections from any number of elements. It is usually other Routers thay connect to Routers in order to create new virtual network connections. Care needs to be taken if multiple elements send control commands.

The following table shows each of these commands.

Table 21: Router Commands

Router Commands	
Command	**Description**
GetName	Gets the name of the Router.
SetName	Sets the name of the Router.
GetRouterAddress	Gets the address of the Router.
SetRouterAddress	Sets the address of the Router.
GetConnectionPort	Gets the port that the Router listens on in order to accept new Router to Router virtual network connections.
ListConnections	Lists all of the virtual network connections that the Router has.
IncomingConnection	Tells the Router that a new virtual network connection *has* already come in from another specified Router. This command comes from another Router, and is the second phase of Router to Router connections.

Table 21: Router Commands

Router Commands	
Command	**Description**
CreateConnection	Tells the Router that a new virtual network connection is coming in from another specified Router. This command comes from another Router, and is the first phase of Router to Router connections.
SetLinkWeight	Set the weight of a specified virtual network connection / link between two Routers.
EndLink	End a specified virtual network connection / link between two Routers.
ListRoutingTable	Lists the whole routing table.
GetPortName	Gets the name of a specified port.
GetPortAddress	Get the address associated with a specified port.
SetPortAddress	Set the address associated with a specified port.
GetPortWeight	Get the weight associated with a specified port.
SetPortWeight	Set the weight associated with a specified port.
GetPortRemoteRouter	Gets the name of the remote router, at the other end of a virtual network connection, on a specified port.
GetPortRemoteAddress	Gets the address of the remote router, at the other end of a virtual network connection, on a specified port.
AppStart	Start an application on the Router.
AppStop	Stop an application.
AppList	List all the applications running on the Router.
GetNetIFStats	Get the statistics for all the $networkinterfaces$ on the Router.
GetSocketStats	Get the statistics for all the *sockets* on the Router.
MonitoringStart	Start the monitoring sub-system and all the configured probes on the Router.

Table 21: Router Commands

Router Commands	
Command	**Description**
MonitoringStop	Stop the monitoring sub-system and all the configured probes on the Router.
SetAP	Set the Aggregation Point config for a Router.
ReadOptionsFile	Read the configuration options for routers from a file, e.g. routeroptions.xml.
ReadOptionsString	Read the configuration options for routers, passed in a a big string.
ShutDown	Tells the Router to shutdown. All applications will be stopped, all network connections to other Routers will be closed, and every component of the router will clean up.
RouterOK	Asks the router if it is OK.

14 Acknowledgements

I would like to thank the following people for their direct contributions to VLSP.

Firstly, to Richard Clegg for writing all of the Event Engine, the Event Scheduler, the Probabilistic Experimental Contol sub-system, as well as all of the routing parts and the router to router routing table exchange mechanism. He also spent time with me fighting the wonders of Java's thread and synchronization mechanisms on the way to creating virtual networks of 700 routers across 12 servers.

Second, to Lefteris Mamatas who also used the system for much work, for using the system so heavily, for pushing the system to its limits, and for extending the system for external experimentation to support his IKMS. Also to Francesco Tusa and Dario Valocchi for their contributions and insights when using VLSP for research and experimentation.

Contents