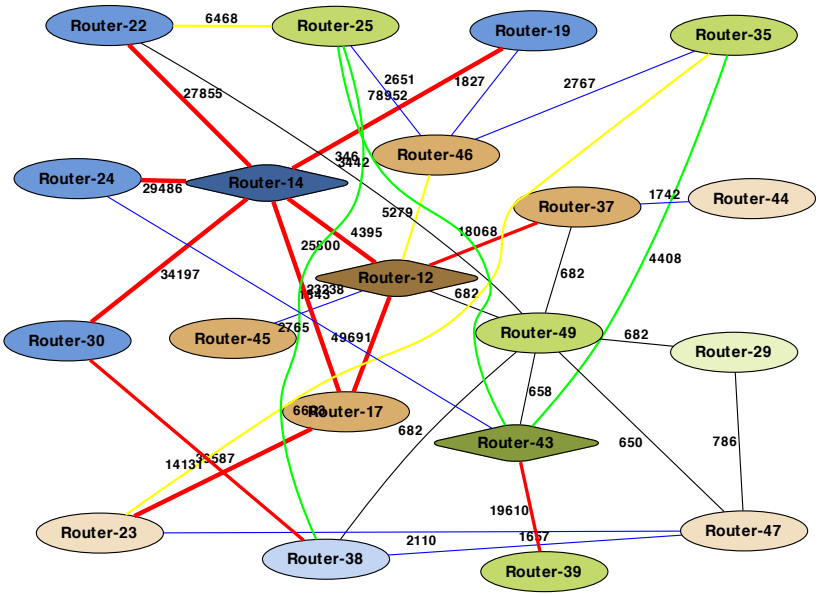
A Guide to User Space Routing: The Very Lightweight Network and Service Platform

Stuart Clayman Richard Clegg

April 2011



Abstract

This document gives an overview of the *User Space Routing* 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.

Contents

CHAPTER 1

Introduction

This document gives an overview of the *User Space Routing* 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 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 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 does not exist on a real running virtual router system but, has the role of experiment 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. In a real system these decisions might be made by the management system.

1.1 Motivation

There were many motivations for designing and building the *User Space Routing* framework. These were accumulated from experience on the RESERVOIR project, 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 98% 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 98% 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 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.

CHAPTER 2

The Platform

In this section there is a more detailed overview of the platform itself.

2.1 Main Function Elements

The main elements of the Platform are the the *GlobalController*, the *LocalControllers*, and the *Routers*. These are each explained in further detail below.

There is one *GlobalController* 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 GlobalController can run on the same host as a LocalController, or in a large setup it can run on a separate host.

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.

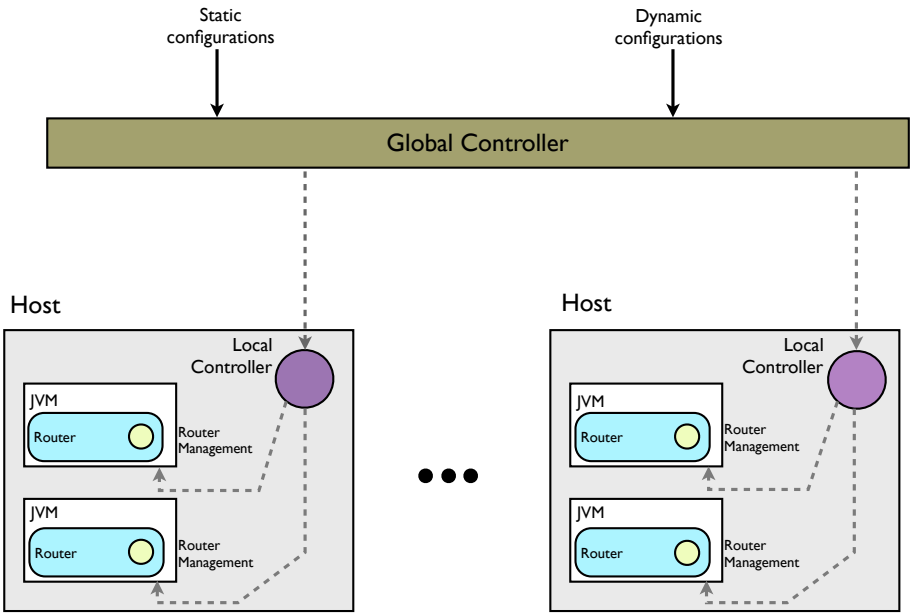


Figure 2.1: General Architecture

In figure 2.1, the relationship between these elements is shown. There is the 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 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 2.2, we see how the topology of virtual routers and virtual links manifests

itself across multiple hosts, three in this case.

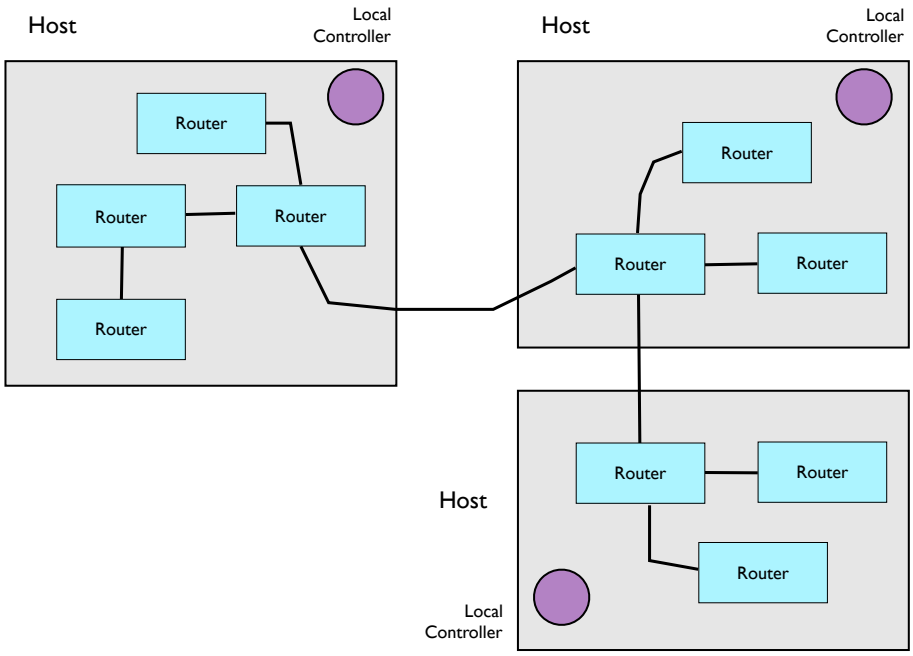


Figure 2.2: Hosts with Virtual Routers and Virtual Links

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 2.3. 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:

- 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.

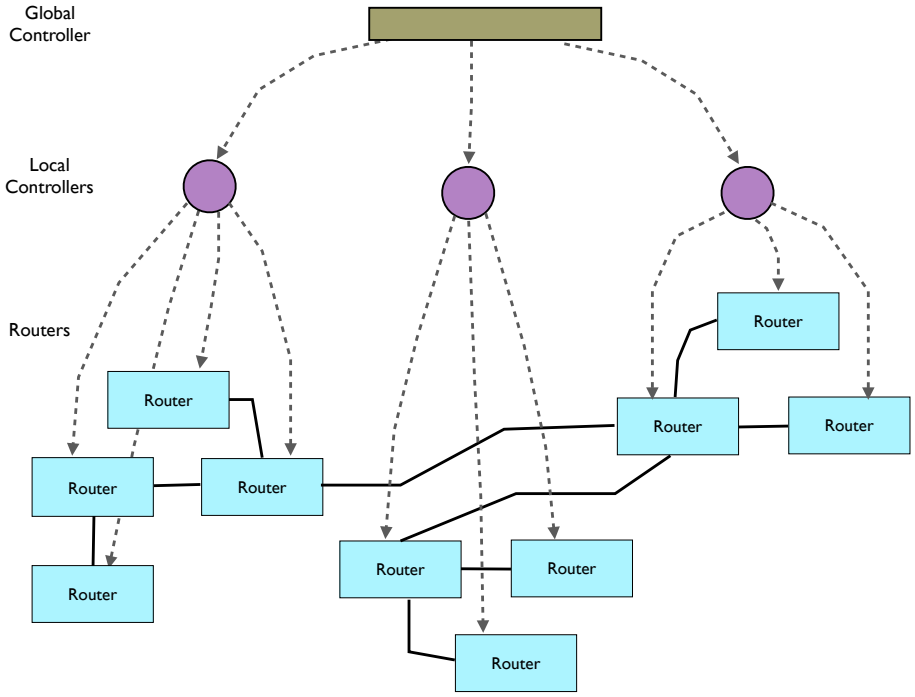


Figure 2.3: Full Connectivity with Control Path

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.

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

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 existing RESERVOIR 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.

CHAPTER 3

Usage

In this subsection 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.

3.1 Setup

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

```
% cd install_place
```

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

```
% export JAVA_HOME=/usr/java/jdk1.6.0_04/
```

```
% export CLASSPATH=.:./libs/monitoring-0.6.7.jar:./libs/timeindex-20101020.jar:
./libs/aggregator-0.3.jar:
```

3.2 Starting

The Global Controller is started and passed an XML configuration control file.

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

This will start the Global Controller on the local host, and 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

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 3.1: control-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>
```

We will now look at the router configuration.

Listing 3.2: routeroptions.xml

```
<RouterOptions>

    <RoutingParameters>
        <MaxCheckTime>60000</MaxCheckTime>
        <MinNetIFUpdateTime>5000</MinNetIFUpdateTime>
        <MaxNetIFUpdateTime>30000</MaxNetIFUpdateTime>
    </RoutingParameters>

    <APManager>
        <Name>Pressure</Name>   <!-- None / Pressure / Random / HotSpot -->
        <MaxAPs>100</MaxAPs>
        <MinAPs>1</MinAPs>
        <RouterConsiderTime>10000</RouterConsiderTime>
        <ControllerConsiderTime>10000</ControllerConsiderTime>
        <MaxAPWeight>5</MaxAPWeight>
        <MinPropAP>0.1</MinPropAP>
        <MonitorType>traffic</MonitorType>
    </APManager>

</RouterOptions>
```

and the probability distribution configuration:

Listing 3.3: 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>
```

CHAPTER 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 Routers

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

- *Management Console* – which provides a control interface from the outside world
- *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 4.1.

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

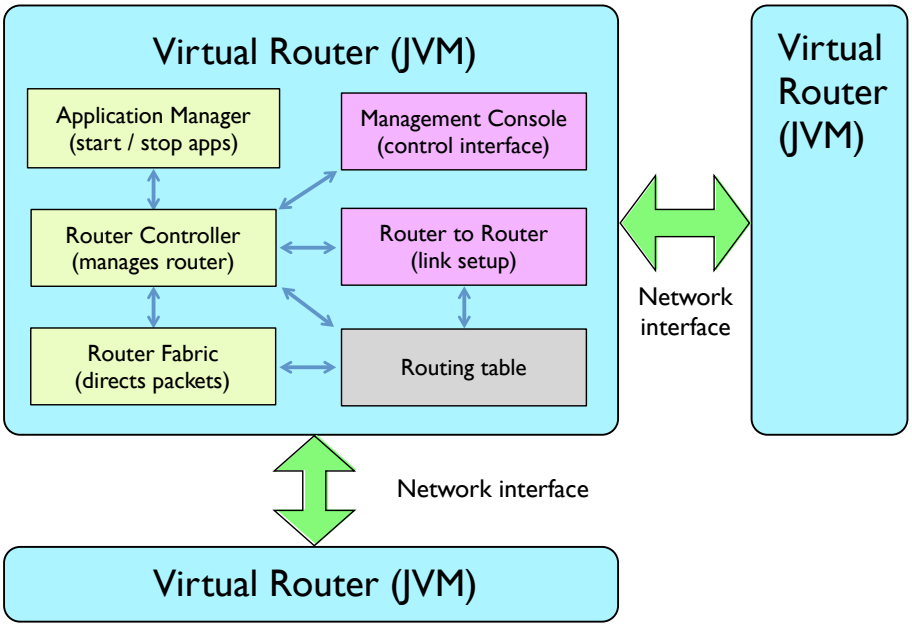


Figure 4.1: Router Top Level View with Main Functional Blocks

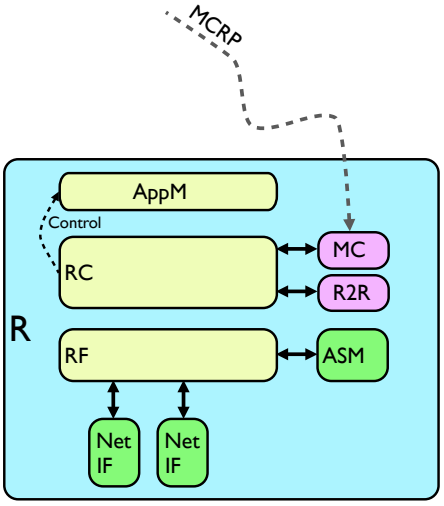


Figure 4.2: Router Internal Architecture

4.2 Router Commands

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 4.1: Router Commands

Router Commands	
Command	**Description**
Quit	Tells the Router that the client will quit the connection to the Management Console. The connection is terminated by the Router.
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.
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.
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.
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.
EndLink	End a specified virtual network connection between two Routers.
ListRoutingTable	Lists the whole routing table.

Table 4.1: Router Commands

Router Commands	
Command	**Description**
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	
GetSocketStats	
MonitoringStart	
MonitoringStop	
ReadOptionsFile	
ReadOptionsString	
SetAP	
Ping	
Echo	

4.3 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.

4.4 Applications and Application Lifecycle

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 C, there is are full details of this API.

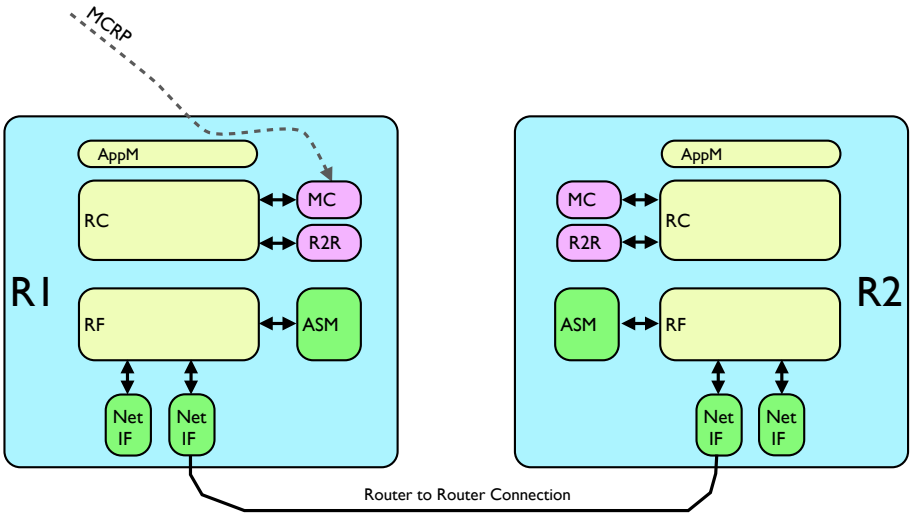


Figure 4.3: Two Routers

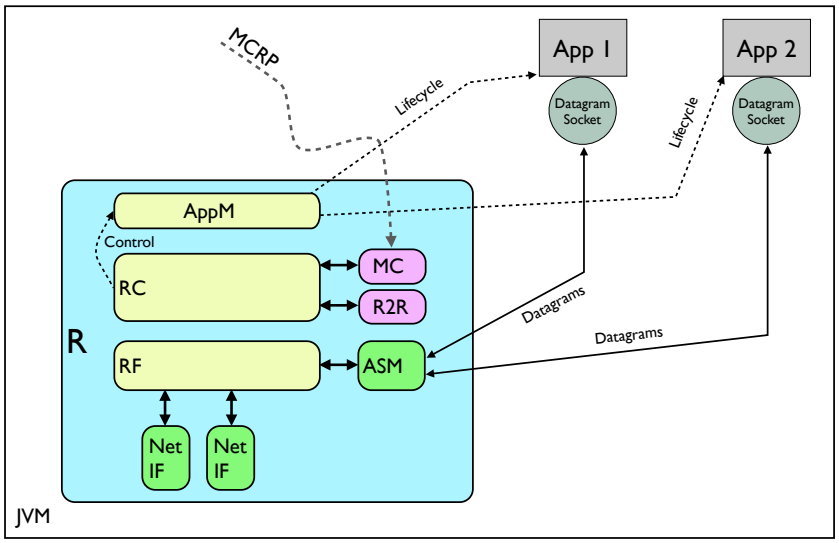


Figure 4.4: Application Manager and Application Lifecycle

CHAPTER 5

Summary

The benefits of the lightweight Integrated Virtual Platform which has a virtual router and the basic capabilities of a service component are:

- it provides a different way to do virtual networks: we can create arbitrary topologies using virtual routers
- uniform manageability of all types of virtual machines can be undertaken in the same environment
- we can do more evaluations of service management functions by not using full VEEs and full services. For example, scalability can be evaluated more easily, and it can cover both the dimension of virtual services as well as the dimension of virtual networks.

APPENDIX A

Appendix - Configuration Options

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

Table A.1: 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.

Table A.1: Configuration Options

Configuration Options	
Field	**Description**
`<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`.
`<GlobalController>` → `<LatticeMonitoring>` `(optional)`	Should the 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.)
`<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)

Table A.1: Configuration Options

Configuration Options	
Field	**Description**
`<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`
`<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.

Table A.2: Configuration Options

Configuration Options	
Field	**Description**
<RouterOptions>	The top level node for all configuration options to the Router.
<RoutingParameters>	The node for specific options for the routing.
<RoutingParameters> → <MaxCheckTime>	Example option is 60000.
<RoutingParameters> → <MinNetIFUpdateTime>	Example option is 5000.
<RoutingParameters> → <MaxNetIFUpdateTime>	Example option is 30000.
<RoutingParameters> → <DatagramType>	String: Value is class name which will be datagram class used by network. E.g. usr.net.IPV4Datagram. String must point to valid class implementing Datagram interface.
<APManager>	The node for specific options for the APManager.
<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>	.
<APManager> → <ControllerConsiderTime>	.
<APManager> → <MaxAPWeight>	.
<APManager> → <MinPropAP>	.
<APManager> → <MonitorType>	.

Table A.2: Configuration Options

Configuration Options	
Field	**Description**
<Output>	The node for specific options for generating output from a run.

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

Table A.3: 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`
<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`

APPENDIX B

Appendix - Java Packages

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

Table B.1: 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.engine`	This package provides classes that add events into the simulation
`usr.globalcontroller`	This package provides classes that are part of a GlobalController.
`usr.globalcontroller.command`	This package provides classes that implement the commands of a GlobalController.
`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.

Table B.1: Java Packages

Java Packages	
Package	**Description**
`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.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.

APPENDIX C

Appendix - Socket Interface

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 C.1: 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)`	

Table C.1: Method Support

Method Support		
Method	**Description**	**In USR**
`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
`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

Table C.1: Method Support

Method Support		
Method	**Description**	**In USR**
`void receive(DatagramPacket p)`	Receives a datagram packet from this socket.	REPL
Replacement ▷	`Datagram receive()`	
`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 C.1: 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

APPENDIX D

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	pay load	check sum

The following table D.1 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 D.1: 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 24.
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.

Table D.1: Datagram Elements

Datagram Elements			
Element	**Bytes**	**Position**	**Description**
Dst Addr	4	14	The destination address of the Datagram.
Pad	2	18	Some spare bytes, currently padded as 2 spaces $\bigtriangledown\bigtriangledown$.
Src Port	4	20	The source port of the Datagram.
Dst Port	4	22	The destination port of the Datagram.
Payload	N	24	The payload itself. It's size, N, is Total Len − Hdr Len − 4.
Checksum	4	$24 + N$	The checksum.

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

TODO

APPENDIX E

Appendix - MCRP Protocol

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

Table E.1: Generic MCRP Protocol Values

Generic MCRP Protocol Values		
Request	**Success Code**	**Error Code**
`QUIT`	200	400
	Quit talking to the Component. It closes a connection to the Management Console of the Component.	
Request: `QUIT` *Response:* `200 BYE`		
`SHUT_DOWN`	201	401
	Shut down the Component.	
`ON_ROUTER`	202	401
	Send an MCRP request for the specified Router. `ON_ROUTER router_id className args`	
Request: `ON_ROUTER 1 usr.applications.Ping 2`		

Table E.2: Router Specific MCRP Protocol Values

Router Specific MCRP Protocol Values		
Request	**Success Code**	**Error Code**
`GET_NAME`	221	400
	Get the name of a router.	
Request: `GET_NAME` *Response:* `221 Router-17`		

List of Figures

List of Tables

Listings