Appendix A

HyperNet Builder Manual

This chapter acts as a manual that guides a HyperNet builder in creating a HyperNet package.

Pre-requisition: before you start to write a HyperNet be sure that you have the latest JDK (Java Develop Kit, not older than version 1.6) installed. JDK v1.6 is the tested version but other later versions should also work (have only tested ver.1.7). Besides the default Java libraries provided by the standard JDK, a HyperNet developer/builder should also use the HyperNet library, which can be downloaded from http://protocols.netlab.uky.edu/ shufeng/pvn/PVNLibrary/cillent-pvnlib.jar. Developer should include this HyperNet library into the HyperNet he/she is coding. In eclipse, this step can be achieved by right-click your Java project and choose the properties of your project. In "Java Build Path", choose "Libraries", and add the jar file (PVNlib.jar) you just downloaded as an external JAR. At the same time, you should make use of this library by adding

```
#import PVNlib.*;
```

into your code. Besides the HyperNet Library, a HyperNet developer typically will also need the library for communicating with the HyperNet hypervisor using XML RPC (xmlrpc-client-3.1.3.jar , xmlrpc-common-3.1.3.jar, xmlrpc-

server-3.1.3.jar) and for parsing xml files [1] (commons-logging-1.1.jar, ws-commons-util-1.0.2.jar, xercesimpl-2.8.0.jar). Please also include those "external" libraries in your HyperNet project in Eclipse, or include them in your java "CLASS-PATH" environment variable. Those Java Libraries can also be fetched from http://protocols.netlab.uky.edu/ shufeng/pvn/PVNLibrary .

A.1 Limitations

The current implementation of the HyperNet hypervisor supports GENI as the underlying virtual network infrastructure provider. More precisely, it supports reserving resources from any aggregates in ProtoGENI, InstaGENI and ExoGENI using the GENI AM API (via the geni omni toolkit [77]). However, for "findPR()" API call, it is only able to find virtual resources from the Kentucky Aggregate and Utah Aggregate in ProtoGENI, which provide 99 nodes and 680 nodes respectively. This is because we only reserved "representative" nodes from the above two aggregates in ProtoGENI. In our current developing stage, the number of resources that one can reserve from the hypervisor is sufficient for most of the experiments but we only have two physical locations to reserve nodes from. In other words, we do not have many options to "find gateway Programmable Routers". This is a huge limitation and we hope we could include more aggregates from ProtoGENI or other virtual network infrastructure providers (such as InstaGENI, ExoGENI and PlanetLab) in the future.

An other limitation is the "joining" of Internet participants. Currently the only way in our system to identify (locate) an Internet participant is via its IP address (yes we could use a routable host name of the Internet participant to find a nearby gateway PR with the minimum RTT, but in order to set up an IP tunnel between the participant and the chosen gateway, we still need to know the participant's IP address). Moreover, to set up a GRE tunnel between the participant and the selected

[1] ProtoGENI uses RSpec files to store network topology and network resources and RSpec files are displayed in the form of XML

gateway, the participant needs to have a public IP address. As a result, any Internet participant that sits behind a NAT box (which I guess applies to most of today's Internet users) can not "join" a HyperNet virtual network, unless we find another way of tunneling two Internet nodes through NAT box (and I am sure there are other ways, for example, overlay is an option, tun/tap interface is another option, with port forwarding properly set in the NAT box). Unfortunately the current implementation only supports GRE tunnel as the only way that an Internet participant joins a virtual network.

The third limitation is due to the HyperNet abstraction. A HyperNet network creator creates a virtual network via the Network Hypervisor. So that any problem happened on the VNIP layer is hidden underneath the hypervisor layer, and (hopefully) solved by the hypervisor. Unfortunately there are not many ways that the current hypervisor can "solve" a problem happened on GENI by its own. For example, in cases of "resource reservation out of limitation" or "the node is currently in use" errors returned by GENI, the hypervisor simply returns error message in the "buildTopo()" API call. The users have no clue of what went wrong. Moreover, since the hypervisor uses its own account on GENI to reserve GENI resources, there is no way for a network creator to log on to a reserved node and "debug" that node, unless the hypervisor shares its account (as well as its credentials) with the user. While a network creator does have the need to "debug" a virtual network, it is a necessary step for a HyperNet builder to debug its HyperNet while building the HyperNet. A possible solution is for the HyperNet builder to upload its GENI account on to the hypervisor so that the hypervisor can reserve resource on behave of the HyperNet builder. Although we have implemented this "debugging mode" in the hypervisor (the user informs the hypervisor about its credentials via configuring the "userCredential" field of the HyperNet config file, which points to the local directory that stores the GENI credential files), using this mode means exposing the user's GENI credential

to the hypervisor, which is not secure (the Network Hypervisor should not be able to know a user's GENI credential). One possible solution is to have the hypervisor to "delegate" its GENI credentials to each of its users so that whenever a user reserves resources from GENI through a HyperNet, GENI knows it is from a user of the Network Hypervisor. Currently the GENI community is working on implementing the support for delegating a GENI credential.

Finally, since the hypervisor is not a commercial product yet, it lacks a very important part: error handling. Currently, almost all the hypervisor API calls either returns true/false or returns a brief message stating what went wrong (relays what GENI AM API returned). There is no hint on what can be done to fix the error (in cases where an error is returned) except for contacting GENI aggregate administrators.

A.2 A HyperNet Package

Generally speaking, a complete HyperNet package should include two parts: a HyperNet executable tar-ball and a configuration file. The HyperNet executable tar-ball should include an executable java file (with links to all necessary pieces of libraries – you can either ask the user to download those jar files to their local PC and properly set up CLASSPATH, or export an executable jar which includes those libraries via Eclipse) and all necessary applications that you will need to use on each virtual nodes in your virtual network (ideally, those applications are expected to be available online such that your executable java file knows where to fetch them). The configuration file on the other hand, is the only interface for a network creator (an ordinary user who uses your HyperNet package) to manipulate the HyperNet. To prevent from a falsely formatted configuration file due to incomplete understanding from a network creator, it is highly recommended that the HyperNet builder creates an interface to limit what

a network creator can modify in the configuration file. However, a HyperNet builder itself should have complete understanding of the structure of the configuration file.

In the rest of this appendix, we first describe the Configuration file, followed by an introduction of the basic data types in creating a HyperNet package in section 3. A detailed list of Network Hypervisor APIs as well as HyperNet Library calls are presented in section 4. Last, we provide a sample HyperNet package in section 5.

A.3 Configuration File

A configuration file is the only interface that a HyperNet creator interacts to manipulate a HyperNetİt is basically a list of key-value pairs separated by line breakers. The key-value pair should be represented in the following format:

```
key: <value1>, <value2> ...
```

Currently the keywords for keys are: HyperNetName, participants, server, userCredential. Those keywords are not fixed. If needed, the HyperNet library can be extended to support more keywords in the future. In fact, a lot more keywords can already be added, such as HyperNet virtual network description, HyperNet virtual network type (public or private) and HyperNet virtual network requirements (states what additional end systems the participants should get), etc. A sample configuration file should be something like the following:

```
HyperNetName: myPrivateVideoMulticastNetwork
participants: 128.163.140.221, 128.163.140.223
server: 128.163.140.211
userCredential: youCanNeverGuess
```

"HyperNetName" states the name of the HyperNet virtual network. This name uniquely identifies a HyperNet virtual network within a Network Hypervisor server.

If the HyperNet virtual network is open to public, then this name is also the keyword for "HyperNet search engine" [2] to lookup existing HyperNets.

"participants" lists out the expected participants of the HyperNet virtual network. Each expected participant is either represented in the form of a domain name or an IP address to uniquely identify the network location of a participant [3].

"server" points to a Network Hypervisor server, in the form of either an IP address or a domain name. We expect the co-existence of multiple Network Hypervisor servers in the future. Those hypervisor servers provide a same set of APIs for HyperNets to use but they differ from each other by the different business relationships they established with different Virtual Network Infrastructure Providers (VNIPs). Another possibility is that a hypervisor service provider provides a hypervisor "cloud" to its users. The hypervisor "cloud" contains multiple hypervisor server instances within. Each user might use different domain names or IP addresses (each of them points to a hypervisor instance) to access to the cloud. This model can increase the robustness of the hypervisor service (e.g., the cloud could migrate a service request to a failed server instance to another nearby instance).

"userCredential" is a secret string that a participant uses when it requests to join a HyperNet virtual network. Unlike a participantID which is assigned to only one participant, this userCredential can be used by any participant who wants to join a HyperNet virtual network. In HyperNet networks where the participants are already known (e.g., a video conferencing network where all participants are family members or co-workers), the HyperNet network creator can make use of the "participants" keywords to define the secret that a legitimate participant should use. On the other hand, in HyperNet networks whose participants are not known ahead (e.g., a youtube-like public video broadcasting network whose participant can be anyone on

[2] A "HyperNet search engine" is an add-on provided by HyperNet server to support searching for existing HyperNets.

[3] In our current design, we do not deal with IP spoofing or other sorts of network attacks.

the Internet), the network creator can define a userCredential so that any requesting participant with a valid credential can be proofed to be legitimate.

A.4 Basic Data Types

There are 12 basic data types in HyperNet library: Node, Participant, Target, Restriction, Link, Address, Application, Protocol, Config, JoinRequest, TunnelInfo, Topology.

- Node

  Node is one of the key data types in the HyperNet library. It fully describes a Programmable Router in a HyperNet.

  Members:

```
//maximum number of interfaces
public static final int max_interface = 4;
public String virtual_id; //the virtual id of the node
public String virtualization_type; //e.g., emulab-vnode
public String node_type; //e.g., pc, d710
public String node_cm; //the component manager of the node
public String uuid; //uuid of the node
//tarfiles that will be loaded when the node starts up
public String tarfiles;
//the command that will be executed when the node starts up
public String startup_command;
public ArrayList<Interface> interfaces;//used interfaces
//available interfaces
public ArrayList<Interface> available_interfaces;
public int num_interfaces; //total number of interfaces
```

```
public String rspec; //the rspec related to this node
public String address; //the (IP) address of this node
public boolean available; //the availability of the node
public ArrayList<Node> neighbor; //the neighbors of the node
```

Constructors:

Node(): initializes a newly created Node object which has default 4 interfaces.

- Interface

  Interface class fully describes an interface on a node.

  Member:

```
public String id;       //the ID of the interface
public String address;  //the IP address of the Interface
public String netmask;  //the network mask of the interface
public String type;     //the type of the Interface (gre or vlan)
```

  Constructors:

  Interface(): initializes a newly created Interface object having all members set to empty.

  Interface(String myid): create a new Interface object setting its id to myid.

- Participant

  Participant fully describes a participant.

  Members:

```
public String URL = ""; //the URL of the participant
public String IPAddress = ""; //the IP address of the participant
```

Constructors:

Participant(): initializes a newly created Participant object with both URL and IPAddress set to empty;

Participant(String address): initalizes a newly created Participant object with IPAddress set to address if address is an IP address, otherwise URL is set to address.

- Target

  Target fully describes a target. Target has the same structure as Participant.

- Restriction

  Restriction the requirements applied when reserving/searching a programmable router.

  Members:

```
//the virtualization type of the PR
public String virtualization_type;
public String node_type; //the node type of the PR
public int num_interfaces; //the capability of the PR
```

  Constructors:

  Restriction(): initializes a newly created Restriction object that represents the minimum restriction: with num_interfaces set to 0, and the other two set to empty.

- Link

  Link fully describes a tunnel in a HyperNet network.

  Members:

```
public String virtual_id;   //virtual id of the link
//the nodes that traversed by the link
public ArrayList<Node> nodes;
public String rspec;    //the corresponding rspec
public int cost;        //the cost of the link
public float capacity;  //the capacity (Kbps) of the link
public float latency;   //the latency (ms) of the link
public float packetLoss;    //the lossrate (%) of the link
```

Constructors:

Link(): initializes a newly created Link object.

- Address

  Address fully represents an IP address (range).

  Members:

```
public String IPAddress;    //the IP Address
//the subnet mask if it is a address range
public String subnet_mask;
```

  Constructors:

  Address(): initializes a newly created Address object.

- Application

  Application fully describes an application.

  Members:

```
//the path to which the source of the application is stored
```

```
public String app_path;
public String run_command;
//the command to run the application
```

Constructors:

Application(): initializes a newly created Application object.

- Protocol

  Protocol fully represents a routing protocol.

  Members:

```
//the name of the routing protocol e.g., OSPF, IGRP, IS-IS
public String name;
```

  Constructors:

  Protocol(): initializes a newly created Protocol object.

- Config

  Config is the data structure that stores the configuration of a HyperNet virtual network.

  Members:

```
//the name of the HyperNet virtual network
public String HyperNetName;
public String credential;   //the credential of the HyperNet
//the address of the HyperNet hypervisor server
public String server;
//expected participants
```

```
public ArrayList<Participant> participantList;
//other HyperNet-specific keywords
public HashMap<String, String> keyword;
```

Constructors:

Config(): initializes a Config object, with all members allocated.

- JoinRequest

  JoinRequest is the data structure that stores a join request from a participant.

  Members:

```
public String fromIP; //the requesting participant's IP address
//the participant type (voluntary or involuntary)
public type participantType;
//the requesting participant's participant ID
public String participantID;
//the requesting participant's credential
public String credential;
public boolean solved;  //whether the request is solved or not
```

  Constructors:

  JoinRequest(): initializes solved to false, participantType to Voluntary, all other members to an empty string.

  JoinRequest(String ip, String id, String secret): initializes solved to false, participantType to Voluntary, all other members to the corresponding arguments.

  JoinRequest(String ip, String type, String id, String secret): initializes solved to false, participantType to type, all other members to the corresponding

arguments.

- TunnelInfo

  TunnelInfo stores the information about a tunnel that connects the participant to its assigned gateway PR.

  Members:

```
public String participantID;    //participant ID
//participant type, voluntary or involuntary
public JoinRequest.type participantType;
//requesting participant's IP address
public String participantAddr;
//requesting participant's assigned HyperNet address
public String participantHyperNetAddr;
public String GWIPAddress;  //assigned gateway's IP address
//assigned gateway's HyperNet address
public String GWHyperNetAddress;
//the command that the participant needs to run
//to configure the tunnel
public String cmd;
```

  Constructors:

  TunnelInfo(): initializes participantType to Voluntary, all other members to an empty string.

- Topology

  Topology stores the information needed about a network topology.

Members:

```
public ArrayList<Node> myNodes; //all the nodes in the topology
public ArrayList<Link> myLinks; //all the links in the topology
//the RSpec representation of the topology
public Document rspec;
```

Constructors:

Topology(): initializes and allocates all members.

A.5 Network Hypervisor API and HyperNet Library calls

There are 20 Network Hypervisor API calls provided by the current implementation of the Network Hypervisor:

- PVNLib()

  PVNLib() is the default constructor of the HyperNet library. It sets up the XML RPC client (connects to the XML RPC server) using the default server IP (127.0.0.1) and server Port (8891), and initializes internal members that stores the configuration information. Two other constructors are also provided if developer wants to name its own XML RPC server: PVNLib(String srvIP) and PVNLib(String srvIP, int srvPort). srvIP lets you specify the IP address of the XML RPC server and srvPort lets you specify the port number on which the XML RPC server is listening.

  Example:

  ```
  PVNLib myHyperNet = new PVNLib("128.163.104.211");
  ```

- boolean getConfig(String myFile)

  getConfig() will read the configuration from a configuration file specified by myFile. The configuration infomation will be stored internally in the PVNlib object. Upon success, Boolean.true will be returned.

  Example:

```
if(!myHyperNet.getConfig()) {return; }
```

- boolean registerHyperNet(String HyperNetName)

  registerHyperNet() will register the HyperNet with specified HyperNetName to the remote Network Hypervisor server (XML RPC server). If on the server side, there already exists a HyperNet with the same HyperNetName, Boolean.false will be returned, otherwise Boolean.true will be returned indicating a success registration.

  Example:

```
boolean retValue = myHyperNet.registerHyperNet(name);
if(retValue == false) {
    System.out.println("this HyperNet Name is already used!");
}
System.out.println("successfully registered "
    +myHyperNet.HyperNetName+" !");
```

- Node findPR(String HyperNetName, Participant p, Target t, Restriction r)

  findPR() will find the programmable router that is 1). close to Participant p;

2). close to Target t and 3). satisfying Restriction r. On success, it will return the selected Programmable Router in a Node object.

Example:

```
Node myNode;
myNode = myHyperNet.findPR("myHyperNet#1",
    new Participant("pc33.uky.emulab.net"),
    new Target(), new Restriction());
```

- Node addPR(String HyperNetName, Node pr)

  addPR() will reserve a Programmable Router specified by Node pr into the HyperNet. Upon success, it returns the Programmable Router reserved in a Node Object.

  Example:

```
Node myNode;
myNode = myHyperNet.findPR("myHyperNet#1",
    new Participant("pc33.uky.emulab.net"),
    new Target(), new Restriction());
if(myHyperNet.addPR("myHyperNet#1", myNode) == null) {
    System.out.println("addPR failure! Probably the node is
            not available or the server is down!");
}
```

- ArrayList<Node> addTunnel(String HyperNetName, Node pr1, Node pr2)

  addTunnel() will reserve a link (tunnel) which connects pr1 and pr2 into the HyperNet network. If Node pr1 and Node pr2 are from two different site (e.g.,

one on UKY emulab and the other on Utah emulab), a GRE tunnel will be created between the two nodes. Otherwise a etherlan will be created between the two nodes. Upon success, both pr1 and pr2 will be returned (in sequence) in an ArrayList.

Example:

```
Node myNode1 = myHyperNet.findPR("myHyperNet#1",
    new Participant("pc33.uky.emulab.net"),
    new Target(), new Restriction());
Node myNode2 = myHyperNet.findPR("myHyperNet#1",
    new Participant("pc388.emulab.net"),
    new Target(), new Restriction());
if(myHyperNet.addTunnel("myHyperNet#1", myNode1,
    myNode2) == null) {
    System.out.println("create tunnel failure!
        Probably the server is down!");
}
```

- Node removePR(String HyperNetName, Node pr)

  removePR() removes a programmable router which has the same virtual_id as Node pr from the HyperNet virtual network. Upon success, it returns the Programmable Router removed.

  Example:

```
Node myNode = new Node();
myNode.virtual_id = "an-existing-node";
if(myHyperNet.removePR("myHyperNet#1", myNode) == null) {
```

```
        System.out.println("removePR failure! Probably the node
            is not in the HyperNet virtual network yet or
            the server is down!");
    }
```

- Node updatePR(String HyperNetName, Node pr, String type)

  updatePR() updates a programmable router which has the same virtual_id as Node pr from the HyperNet virtual network. Upon success, it returns the Programmable Router updated.

  Example:

```
    Node myNode = new Node();
    myNode.virtual_id = "an-existing-node";
    myNode.address = "a new IP address";
    myNode.startup_command = "a new command";
    if(myHyperNet.updatePR("myHyperNet#1", myNode,
        "a new type") == null) {
        System.out.println("updatePR failure! Probably the node is
            not in the HyperNet virtual network yet or
            the server is down!");
    }
```

- ArrayList¡Node¿ removeTunnel(String HyperNetName, Node pr1, Node pr2)

  removeTunnel() will remove a link (tunnel) which connects pr1 and pr2 from the HyperNet network. Upon success, both pr1 and pr2 will be returned (in sequence) in an ArrayList.

  Example:

```
Node myNode1 = new Node();
myNode1.virtual_id = "existing_node1";
Node myNode2 = new Node();
myNode2.virtual_id = "existing_node2";
if(myHyperNet.removeTunnel("myHyperNet#1", myNode1,
    myNode2) == null) {
    System.out.println("remove tunnel failure! Probably
        the tunnel does not exist in the HyperNet virtual
        network or the server is down!");
}
```

- Link updateTunnel(String HyperNetName, Link tunnel)

  updateTunnel() will update a link (tunnel) that has the same virtual_id as the given tunnel. Upon success, the updated tunnel (type Link) will be returned.

  Example:

```
Link myTunnel = new Link();
myTunnel.virtual_id = "an-existing-virtual-id";
//a new capacity of 200Mbps for the tunnel
myTunnel.capacity = 200000;
if(myHyperNet.updateTunnel("myHyperNet#1",
    myTunnel) == null) {
    System.out.println("update tunnel failure! Probably the
        tunnel does not exist in the HyperNet virtual network
        or the server is down!");
}
```

- boolean updateTopo(String HyperNetName)

  updateTopo() will update a running HyperNet virtual network. More precisely, it will modify the generated manifest file and upload the modified manifest file to the VNIP and then let the VNIP enforce the modification on the running virtual network. Upon success, it returns true.

  Example:

```
//do remove, add, update to PRs and Tunnels, then:
boolean ret = myHyperNet.updateTopo(myHyperNet.HyperNetName);
if(ret == true) {
    System.out.println("successfully
        updated the HyperNet topo!");
} else {
    System.out.println("HyperNet updte topo failed!");
}
```

- boolean loadApp(String HyperNetName, Node pr, Application app)

  loadApp() loads an application "app" onto a specified programmable router "pr". This application gets executed when the virtual network is started and the programmable router is running.

  Example:

```
Application gatewayApp = new Application();
gatewayApp.app_path = "http://protocols.netlab.uky.edu
    /~shufeng/squid-3.1.19.tar";
gatewayApp.run_command = "sudo chown -R shuan3:uky-emulab-net
    /local/squid-3.1.19";
```

```
loadApp(myHyperNet.HyperNetName, myGWNode, gatewayApp);
```

- ArrayList<Node> getShortestPath(Node a, Node b)

  getShortestPath() fetches the shortest path from node a to node b and returns it in the form of a ordered list of nodes.

  Example:

```
Node myNode1 = myHyperNet.findPR("myHyperNet#1",
    new Participant("pc33.uky.emulab.net"),
    new Target(), new Restriction());
Node myNode2 = myHyperNet.findPR("myHyperNet#1",
    new Participant("pc388.emulab.net"),
    new Target(), new Restriction());
ArrayList<Node> myPath = getShortestPath(myNode1, myNode2);
```

- Node findCentralNode(ArrayList<Node> a)

  findCentralNode() finds the node that sits in the middle of a list of nodes. This "central node" is chosen as the most popular node appeared in the shortest paths between every pair of the given list of nodes. In cases where no node appears more than once in all shortest paths, this function call returns null. In this case, it is up to the caller to explore the underlying physical topology and decide a central node.

  Example:

```
Node a = new Node("pc33.uky.emulab.net");
Node b = new Node("pc43.uky.emulab.net");
Node c = new Node("pc35.uky.emulab.net");
```

```
Node d = new Node("pc77.uky.emulab.net");
Node e = new Node("pc20.uky.emulab.net");
Node f = new Node("pc12.uky.emulab.net");
Node g = new Node("pc8.uky.emulab.net");
ArrayList<Node> nodeList = new ArrayList<Node>();
nodeList.add(a);
nodeList.add(b);
nodeList.add(c);
nodeList.add(d);
nodeList.add(e);
nodeList.add(f);
nodeList.add(g);
Node centralNode = myHyperNet.findCentralNode(nodeList);
if (centralNode == null) {
    //customized code to find central node
}
```

- TunnelInfo join(Address myAddress, String HyperNetName, String participantID, String credential)

  join() is a function call used by the end system of a voluntary participant. It sends out a join request and waits for(returns) a message containing the information needed for setting up a tunnel between the voluntary participant and the selected gateway PR.

  Example:

```
TunnelInfo myGRETunnel = myHyperNet.join(myAddress,
    myHyperNet.HyperNetName, myID, myCredit);
```

- TunnelInfo joinOther(Address myAddress, Address, remoteAddress, String HyperNetName, String participantID, String credential)

  joinOther() is a function call used by the end system of a voluntary participant. It sends out a join request to "passively" join a third-party participant and waits for(returns) a message containing the information about the chosen gateway for the third-party participant so that proper routing tables and (local) domain name mappings could be updated for future packets targeted at the third-party participant.

  Example:

```
TunnelInfo remoteGW = myHyperNet.join(myAddress, remoteAddress,
    myHyperNet.HyperNetName, myID, myCredit);
```

- JoinRequest checkJoin(String HyperNetName)

  checkJoin() is used by the HyperNetś join request handling process which waits for any new coming join request and returns the join request if it is not solved yet.

  Example:

```
JoinRequest newRequest =
    myHyperNet.checkJoin(myHyperNet.HyperNetName);
```

- boolean addGW(String HyperNetName, TunnelInfo myGW)

  addGW() is used to configure a gateway to set up a tunnel with its assigned participant.

  Example:

```
myHyperNet.addGW(myConfig.HyperNetName, newGW);
```

- boolean buildTopo(String HyperNetName)

  buildTopo() will commit and deploy the HyperNet network. More precisely, it will generate a resource specification file (in the protoGENI case, a rspec file), upload it to the infrastructure provider (protoGENI), and deploy the HyperNet network using the APIs provided by the infrastructure provider. Upon success, it returns Boolean.true.

  Example:

```
boolean ret = myHyperNet.buildTopo(myHyperNet.HyperNetName);
if(ret == true) {
    System.out.println("successfully built the HyperNet topo!");
} else {
    System.out.println("HyperNet build topo failed!");
}
```

- (deprecated) Node getPR(String HyperNetName, Node pr)

  getPR() will fetch the detailed information about a programmable router from the manuscript that protoGENI generates. In protoGENI and many other network testbeds, it is common that these "infrastructure providers" allow their users to choose an available programmable router from a certain region (or a certain set of resources). An example of such a "region" could be an aggregate in protoGENI. The user specifies which region the programmable router should be chosen from. The infrastructure provider then picks an available programmable router via its own mapping algorithm from that region. As a result, the infrastructure provider gains better control of its managed resources, as opposed

to letting the user specify which programmable router to reserve. Since the infrastructure provider's mapping process can react faster towards resource failure and resource reservation conflicts, this also prevents reservation errors caused by the above two reasons. Thus, we believe future VNIPs may also use the same mechanism for their users to reserve resources.

After resources are chosen and reserved, a "manuscript" will be returned to the user stating which specific resources are picked. getPR() takes this manuscript as input and fetch the information (including IP address, hardware type, etc.) of a particular programmable route (defined by the virtual_id of the argument "pr"). (This function is deprecated since it is ProtoGENI-specific)

Example:

```
Node pr = new Node();
pr.virtual_id = "pc1";
pr = myHyperNet.getPR(myHyperNet.HyperNetName, pr);
```

- (deprecated) String getIPAddress(String hostName)

  getIPAddress() is the function that gets the IP address from a given domain name. The manuscript that protoGENI returns only contains the domain names of reserved programmable routers. To accomplish tasks such as configuring a GRE tunnel, we need to know the programmable router's IP address. (This function is deprecated since it can be implemented in the HyperNet Library)

  Example:

```
myIP = myHyperNet.getIPAddress(myHyperNet.HyperNetName);
```

To facilitate the creation of certain type of topologies (e.g., tree, ring, star, etc) and to easily setup static routing tables in a network, we also provide the following set of HyperNet Library calls:

- ArrayList <Node> setRoutes(String HyperNetName, Node pr, Node dstAddress, ArrayList<Node> path, boolean reversible)

  setRoutes() will set proper static routing entries into the routing table of source node, destination node and all nodes in the path so that source node can send packets to destination node. Node pr is the source node, Node dstAddress is the destination node, path is the list of nodes the packet should traverse, boolean reversible specifies whether the route in the reverse direction (from destination node to source node, via the reverse of path) should be enabled. Upon success, it returns an ArrayList containing (in sequence) the source node, the destination node, and the nodes in path.

  Example:

```
//initialize sourceNode, dstNode, myNode1 and myNode2**
ArrayList<Node> tempPath = new ArrayList<Node> ();
tempPath.add(myNode1);
tempPath.add(myNode2);
myHyperNet.setRoutes(myHyperNet.HyperNetName, sourceNode,
    dstNode, tempPath, true);
```

- Tree buildRPTree(String HyperNetName, ArrayList<Node> leaves)

  buildRPTree() creates a rendezvous point tree topology having all input "leaves" as leaf nodes. The central node of the input leaves is chosen as the rendezvous point.

  Example:

```
Tree myTree =
    myHyperNet.buildRPTree(myHyperNet.HyperNetName, gateways);
```

- Topology buildSPTree(String HyperNetName, ArrayList<Node> leaves)

  buildSPTree() builds a separate "sub-tree" for each client node (leaves in the parameter), and then combines all the "sub-trees" to form a complete SP Tree. The "sub-tree" for a client node is formed by the shortest paths from that client node to all the rest client nodes. As a result, this "sub-tree" is an optimum tree having the client node as root and the rest client nodes as leaves, with each route from the root to any leaf following the shortest path.

  Example:

```
Tree myTree =
    myHyperNet.buildSPTree(myHyperNet.HyperNetName, gateways);
```

- Topology buildRing(String HyperNetName, ArrayList<Node> nodes)

  buildRing() takes the input of a list of user-defined nodes and builds a "ring" topology by connecting each node in the list with its neighbors.

  Example:

```
Topology myRing =
    myHyperNet.buildRing(myHyperNet.HyperNetName, myRingNodes);
```

- Topology buildRandomRing(String HyperNetName, int nodeNum)

  buildRandomRing() takes the input of a number, and randomly select the number of nodes from the underlying infrastructure and build a ring topology connecting them.

Example:

```
Topology myRing =
    myHyperNet.buildRandomRing(myHyperNet.HyperNetName, 8);
```

- Topology buildStar(String HyperNetName, Node center, ArrayList<Node> nodes)

  buildStar() takes the input of a user-defined central node and a list of user-defined nodes and builds a "star" topology by connecting each node in the list with the central node. The central node is defined as a virtual machine in ProtoGENI to allow more then 4 interfaces allocated on it (since it may need to connect to a lot of starNodes).

  Example:

```
Topology myStar = myHyperNet.buildStar(myHyperNet.HyperNetName,
    centerNode, starNodes);
```

- Topology buildRandomStar(String HyperNetName, int nodeNum)

  buildRandomStar() takes the input of a number, and randomly select one node as the central node, and the specified number of nodes from the underlying infrastructure and build a star topology connecting them.

  Example:

```
Topology myStar =
    myHyperNet.buildRandomStar(myHyperNet.HyperNetName, 6);
```

- Topology buildMesh(String HyperNetName, ArrayList<Node> nodes)

  buildMesh() takes the input of a list of user-defined nodes and builds a "mesh" topology by connecting each node into a connected graph.

  Example:

```
Topology myMesh =
    myHyperNet.buildMesh(myHyperNet.HyperNetName, myMeshNodes);
```

- Topology buildRandomMesh(String HyperNetName, int nodeNum)

  buildRandomMesh() takes the input of a number, and randomly select the number of nodes from the underlying infrastructure and build a mesh topology connecting them.

  Example:

```
Topology myRandomMesh =
    myHyperNet.buildRandomMesh(myHyperNet.HyperNetName, 8);
```

A.6 A complete example

In the following, we present a complete example HyperNet package for a Network Creator to deploy a multicast HyperNet and for a HyperNet participant to join the multicast HyperNet.

A.6.1 A Multicast HyperNet Package

In the configuration file named myConfig.txt which is associated with the HyperNet, we have:

```
HyperNetName: myMulticastHyperNet
participants: 128.163.140.211, 128.163.140.224, 128.163.140.214, 128.163.140.210
server: bryce.netlab.uky.edu
userCredential: myPassword
```

The following is the multicast HyperNet code (RPCClient.java):

```
import pvnlib.*;
import applib.*;

public class RPCClient {
    private static PVNLib myHyperNet;
    private static Config myConfig;
    private static int greAddressStart = 254; //gre address assignment from 10.128.254.0
    private static HashMap<String, Node> gwMap = new HashMap<String, Node>();

    private static boolean regHyperNet(Config myConfig) {
        boolean retValue = myHyperNet.registerHyperNet(myConfig);
        if(retValue == false) {
            System.out.println("for some reason, register failed! maybe the name of your
        }
        else {
            System.out.println("successfully registered "+myHyperNet.HyperNetName+" !");
        }
        return retValue;
    }

    private static Node rsvNode(Node myNode) {
        Node node = myHyperNet.addPR(myConfig.HyperNetName, myNode, "pc");
        if(node != null) {
            System.out.println("successfully added PR "+myNode.node_cm);
        } else {
            System.out.println("reserve PR failure! Probably the server is down!");
        }
        return node;
    }

    private static ArrayList<Node> rsvLink(Node pr1, Node pr2) {
        ArrayList<Node> nodes = myHyperNet.addTunnel(myConfig.HyperNetName, pr1, pr2);
        if(nodes != null) {
            System.out.println("successfully created tunnel between "
                    +pr1.virtual_id + " and "+pr2.virtual_id);
        } else {
            System.out.println("create tunnel failure! Probably the server is down!");
        }
        return nodes;
    }

    private static void build() {
        boolean ret = myHyperNet.buildTopo(myConfig.HyperNetName);
        if(ret == true) {
            System.out.println("You DID IT! HyperNet network "+myConfig.HyperNetName+" c
            System.out.println("please goto http://protocols.netlab.uky.edu/~shufeng/gen
```

```
                "to see the rspec generated!");
        } else {
            System.out.println("build topo failed!");
        }
    }

    public static void main(String[] args) throws Exception {
        //read from config file
        myConfig = PVNLib.getConfig();
        myHyperNet = new PVNLib(myConfig.server);
        if(myConfig.HyperNetName.equals("")) {
            System.out.println("failed in getting configuration file!");
            return;
        }
        //register HyperNet
        if(!regHyperNet(myConfig)) {
            return;
        }
        ArrayList<Node> gateways = new ArrayList<Node>();

        int j = 0;
        // for each expected participant, assign a proper gateway
        System.out.println("Finding gateways for each participant in your list...");
        for(int i = 0; i < myConfig.participantList.size(); i++) {
            Node tmpNode = myHyperNet.findPR(myHyperNet.HyperNetName, myConfig.participa
            gwMap.put(myConfig.participantList.get(i).IPAddress, tmpNode);
            gateways.add(tmpNode);
            System.out.println("found "+tmpNode.address+" for "+myConfig.participantList
        }
        System.out.println("Finding Gateways...Done!");
        System.out.println("****************************************");

        //up to this point, a gateway is found for each participant
        System.out.println("Building Tree...");
        Tree myTree = myHyperNet.buildRPTree(myConfig.HyperNetName, gateways);
        System.out.println("Building Tree...Done!");

        //load pim sparse mode multicast protocol onto the tree
        System.out.println("Loading PIM Multicast Protocols onto Nodes...");
        MulticastApp myMApp = new MulticastApp(myHyperNet);
        myMApp.loadPIMSMProtocol(myTree);
        System.out.println("Loading PIM Multicast Protocols...Done!");

        //up to this point, a RP-tree is built
        //deploy the multicast tree
        System.out.println("Deploying Topology onto VNIP...");
        build();
```

```
        System.out.println("Deploying Topology onto VNIP...Done!\n");
        System.out.println("*****************************************");

        //up to this point, a tree topology is deployed

        int availableGW = 0;
        while(true) { //join request handling process
            //check join requests
            System.out.println("check Join Requests......");
            JoinRequest newJoin = myHyperNet.checkJoin(myConfig.HyperNetName);
            System.out.println("*****************************************");
            System.out.println("got new Join Request! from "+newJoin.fromIP);
            TunnelInfo newGW = new TunnelInfo();
            newGW.participantAddr = newJoin.fromIP;
            //assign gw
            newGW.GWIPAddress = gwMap.get(newJoin.fromIP).address;
            //for now the GER tunnel IP pair is always 10.128.240.254 and 10.128.240.253
            //assign HyperNet address
            newGW.participantHyperNetAddr = "10.128."+(greAddressStart-availableGW)+".1"
            newGW.GWHyperNetAddress = "10.128."+(greAddressStart-availableGW)+".2";
            //set up GRE tunnel with the gateway

            newGW.cmd = "sudo /sbin/ip tunnel del greX;";
            newGW.cmd += "sudo /sbin/ip tunnel add greX mode gre local "+newGW.participa
            newGW.cmd += "sudo /sbin/ip addr add " +newGW.participantHyperNetAddr+"/24 p
            newGW.cmd += "sudo /sbin/ip link set greX up multicast on;";
            newGW.cmd += "sudo /sbin/ip route add 224.0.0.0/4 dev greX;";
            newGW.cmd += "sudo sysctl -w net.ipv4.conf.all.rp_filter=0;";
            newGW.cmd += "sudo sysctl -w net.ipv4.conf.greX.rp_filter=0";

            //add gw
            myHyperNet.addGW(myConfig.HyperNetName, newGW);
            availableGW++;
            System.out.println("gateway for "+newJoin.fromIP+" is added!");

            System.out.println("*****************************************\n");
        } //while
    }
}
```

A.6.2 A Multicast HyperNet End System Package

A HyperNet participant who wants to join this Multicast HyperNet network also obtains a configuration file similar to the configuration file used to create the network:

```
HyperNetName: myMulticastHyperNet
participants: 128.163.140.211
server: bryce.netlab.uky.edu
```

```
userCredential: myPassword
```

In the "participants" field the user enters its own public IP address and "userCredential" should match with the userCredential set by the network creator.

The Multicast HyperNet End System code is the following: