The Design of PRIME SSFNet: A Tutorial for Developers of the Network Simulator

Jason Liu

Latest Update: *September 11, 2007*

Contents

1 Introduction

PRIME stands for Parallel Real-time Immersive network Modeling Environment and it's a simulator/emulator of computer networks, potentially at a large scale with tens of thousands up to millions of network entities. PRIME consists of two components: PRIME SSF, which is the simulation kernel supporting parallel simulation and real-time simulation and emulation, and PRIME SSFNet, which is the network simulator built on top of PRIME SSF. This document is on how to develop network protocols, services, and applications for the PRIME SSFNet simulator. Please refer to the *PRIME SSF User's Manual* for questions on PRIME SSF (which is distributed with the source code under the `ssf/doc` directory). By the way, SSF stands for Scalable Simulation Framework, which is a recently developed standard or API for discrete-event simulation running on parallel platforms.

This document is accompanied by the *PRIME SSFNet Reference Manual* (which is also distributed with the source code and the HTML version can be generated by typing `make` in the `ssfnet/doc` directory). The reference manual documents the source code of SSFNet, including the definition of all classes and methods therein. In contrast, this document focuses on the design aspects of PRIME SSFNet. That is, it will not include all the implementation details, such as the

exact definition of the classes and the prototype of the functions. The readers are recommended to refer to the accompanying reference manual for such details.

It is important to mention that PRIME SSF and PRIME SSFNet are both based on previous projects. In particular, PRIME SSF is based on DaSSF (which later became iSSF) and PRIME SSFNet is based on DaSSFNet (which later became iSSFNet and RINSE). There exists a somewhat parallel effort more focused on cyber-security aspects of global networks in Professor David M. Nicol's research group at the University of Illinois at Urbana-Champaign. Professor Nicol originally led the development of both DaSSF and DaSSFNet. PRIME SSFNet is implemented in C/C++. However, there is a Java version of the simulator, which can be located at `www.ssfnet.org`. Although the Java development has been discontinued for a few years, a good collection of documents still exist at the web site, which one could use as reference.

The rest of this document is organized as follows. In section 2, we begin with a discussion on using the Domain Modeling Language (DML) for configuring network models. In section 3, we describe the basic building blocks and facilities of the simulator, including definitions of networks, hosts, routers, links, interfaces, protocols, packets, and so on. In section 4, we focus on basic services of the simulator that one uses to glue together the above network entities in a simulation experiment. These services also provide the network models with essential run-time information of the simulation environment, such as the mapping from network addresses to simulation objects. In section 5, we follow the steps for building and running a network simulation model. Existing protocol implementations are discussed in section 6, and existing applications are discussed in section 7. Support for emulation and real-time simulation is described in section 8. Section 9 contains advanced models implemented in PRIME SSFNet. In section 10, we summarize some miscellaneous but important issues one should know to develop a network model.

2 DML for Network Configurations

2.1 The Plain DML

Despite the cryptic nature of its name, DML is a very simple language. PRIME SSFNet uses DML for configuring network models, that is, for describing the network topology, the network protocols, and the network traffic. We call the DML files used for configuring the network collectively as *model DML*. PRIME SSFNet pre-processes the model DML, which includes collecting important information, assigning IP addresses to network interfaces, and possibly calculating routing information for the routers. The results of this preprocessing will be stored in one or several DML files (which we call the *intermediate DML*).

Informally speaking, DML is simply a list of attributes. Each attribute is a key-value pair. The key is the name of the attribute (i.e., an identifier). The value of the attribute can either be a number (i.e., an integer or a floating point value), a character string, or recursively another list of attributes. That is, DML allows nested attributes: the value of an attribute is a list of attributes, which are enclosed in brackets. The following is an example. Note that in DML comment starts with "#" and ends at the end of the line.

```
planet "Earth"          # name of the planet (a string)
weight 6.5e21           # in tons (a floating point number)
population [
```

```
    number 6242324585   # actual population (an integer)
    date "08/08/2002"   # on this date (a string)
    time "15:00:29 GMT" # at this time (a string)
]
```

Note that, if the attribute value is a string that does not have white space or special characters, such as "[", one does not need to put the string in quotes, though we recommend that one always use the double quotes to avoid confusion.

Conceptually, DML corresponds to a tree. The root and the internal nodes of the tree each represent a list of attribute. The leaves correspond to the individual attributes (including both the attribute keys and values). In this way, any attribute can be refernced using a search path, which is quite similar to files and directories in a file system, except that, instead of using "/" to separate files in a search path as in Unix, DML uses "." to separate the names. For example, we can refer to the population number in the previous example with `".population.number"`. The first "." is used to represent that the path starts from the root. Of course, one can use a relative path without the starting ".", in which case the DML parser knows that the search starts from the current attribute list. It is important to know that attribute keys in DML are case-insensitive. That is, "planet" and "PLANET" would refer to emthe same attribute in the above example.

DML has three special attribute keys: `_find`, `_extends`, and `_schema`. One can ignore `_schema`, since it is not implemented in PRIME SSFNet. The attribute key `_find` is used to point to another attribute that DML is going to replace the current one with. For example, the following two DMLs function exactly the same:

```
today [
    date "08/08/2002"
    time "15:00:29 GMT"
]
planet "Earth"
weight 6.5e21
population [
    number 6242324585
    _find .today.date
    _find .today.time
]
```

```
today [
    date "08/08/2002"
    time "15:00:29 GMT"
]
planet "Earth"
weight 6.5e21
population [
    number 6242324585
    date "08/08/2002"
    time "15:00:29 GMT"
]
```

The attribute key `_extends` is used to replace the current attribute with the list of attributes pointed to by the attribute value. When the same list of attributes is used many times in a DML, using `_extends` can tremendously reduce the DML file size. The following shows another way of writing the same DML as the previous example:

```
today [
    date "08/08/2002"
    time "15:00:29 GMT"
]
planet "Earth"
weight 6.5e21
```

```
population [
    number 6242324585
    _extends .today
]
```

2.2 Describe Network Models Using DML

Although seemingly simple, DML can be quite powerful in terms of expressiveness. SSFNet uses DML to describe network models, including network topology, protocols, and application traffic. Here, we give a quick tour of the model DML. We start from a simple network with only two hosts, each with a network interface. They connect through a link:

```
Net [
   host [id 1 interface [id 1]]
   host [id 2 interface [id 1]]
   link [attach 1(1) attach 2(1)]
]
```

The network model is described as a list of attributes defined within a `net` attribute. Each host is identified by an `id`. Within each host, there is an `interface` attribute representing the NIC within the host. Each NIC also has its own `id`. The hosts can take any non-negative integer value to be their ids as long as they differ within the same network. Particularly, the ids do not need to be consecutive. The same rule applies to the ids of the interfaces within a host. The link attribute is used to describe the connection between the two hosts. It defines two attachment points. One attached to the interface named `"1(1)"`, the other `"2(1)"`. Here, we use the so-called NHI addresses to refer to hosts, routers, and network interfaces. For example, `"2(1)"` means that it's the network interface of id 1 within the host of id 2. By the way, NHI stands for network, host, and interface.

Now, let's put a router and two more hosts in this network. This time the hosts are connected using an Ethernet rather than a point-to-point link. The added router contains two network interfaces: one connects to the Ethernet and the other is reserved to be connected to the outside of the LAN. The DML is shown as follows:

```
Net [
   host [id 1 interface [id 1]]
   host [id 2 interface [id 1]]
   host [id 3 interface [id 1]]
   host [id 4 interface [id 1]]
   router [id 5
       interface [id 1]
       interface [id 2]
   ]
   link [
       attach 1(1) attach 2(1) attach 3(1)
       attach 4(1) attach 5(1)
   ]
]
```

Sometimes it's inconvenient to describe all hosts one at a time if they only differ in their ids. In this case, one can use `id_range` instead of `id` to describe a list of hosts or routers with consecutive ids. The same rule also applies to interfaces inside a host or router. The four hosts in the above example can be described alternatively with the following line:

```
host [id_range [from 1 to 4] interface [id 1]]
```

The above DML shows only one local area network with four hosts and one router. If we decide to expand the model and create a network of two such LANs, all we need to do is to define a network with two instances of the above network:

```
Net [
    Net [id 1
       host [id_range [from 1 to 4] interface [id 1]]
       router [id 5
           interface [id 1]
           interface [id 2]
       ]
       link [
           attach 1(1) attach 2(1) attach 3(1)
           attach 4(1) attach 5(1)
       ]
    ]
    Net [id 2
       host [id_range [from 1 to 4] interface [id 1]]
       router [id 5
           interface [id 1]
           interface [id 2]
       ]
       link [
           attach 1(1) attach 2(1) attach 3(1)
           attach 4(1) attach 5(1)
       ]
    ]
    link [attach 1:5(2) attach 2:5(2)]
]
```

Similar to hosts, routers, and network interfaces, all networks, except the outermost one, must be assigned with an id. In the above example, we assign a unique network id to each of the sub-networks. Actually, since the two sub-networks are all the same except their ids, one can use `id_range` for the sub-networks as well:

```
Net [
    Net [
       id_range[from 1 to 2]
       host [id_range [from 1 to 4] interface [id 1]]
       router [id 5
```

```
            interface [id 1]
            interface [id 2]
        ]
        link [
            attach 1(1) attach 2(1) attach 3(1)
            attach 4(1) attach 5(1)
        ]
    ]
    link [attach 1:5(2) attach 2:5(2)]
]
```

In both case, a connection is made between these two sub-networks. We have a link with two attachment points. The NHI address `"1:5(2)"`, for example, means that it's the interface of id 2 in the router of id 5 within the network of id 1.

One can image how a large network model is constructed in this recursive fashion. The NHI addressing scheme allows a network defined previously to become a subnet of a larger network model. The entire network can thus be defined in a hierarchical way (which conceptually is the same as how the Internet is built).

In the above example, we use DML only for defining hosts, routers, network interfaces, and their interconnections. Without proper protocols running within these hosts and routers, however, the network doesn't do anything. In SSFNet, protocols running inside each host or routers are organized as a protocol stack (also called protocol graph). A protocol stack consists of several protocol sessions[1]. In the following example, we make the four hosts in one sub-network all run five protocol sessions: IP, TCP, socket, a TCP server application, and a TCP client application:

```
host [
    id_range [from 1 to 4] interface [id 1]
    graph [
        ProtocolSession [name "TCPserver"
            use "SSF.OS.TCP.test.blockingTCPServer"]
        ProtocolSession [name "TCPclient"
            use "SSF.OS.TCP.test.blockingTCPClient"]
        ProtocolSession [name "socket"
            use "SSF.OS.Socket.blockingSocketMaster"]
        ProtocolSession [name "tcp"
           use "SSF.OS.TCP.tcpSessionMaster"]
        ProtocolSession [name "ip" use "SSF.OS.IP"]
    ]
]
```

The `name` attribute specifies the name of the protocol session. Each protocol session within a protocol graph must have a unique name. The `use` attribute identifies the class that implements the protocol. SSFNet allows a protocol to have several different implementations and

[1] In SSFNet, protocols do not necessarily refer only to network protocols. For example, socket is treated as a protocol in SSFNet. More precisely, protocols in SSFNet are modules that provide communication-related services on hosts and routers.

the user can choose which implementation to use for a particular network model. For example, `SSF.OS.Socket.blockingSocketMaster` is a socket interface implemented with blocking system calls. We will describe these protocols and services in section 6 and section 7. Note that the order in which these protocol sessions are specified in the protocol graph in DML is not important. One can list the protocol sessions in any order. In PRIME SSFNet, the IP-layer protocol is the only protocol required in a protocol graph.

A router can also define the protocol sessions in the same fashion. Following the previous example, rather than running the application protocol sessions (i.e., TCP client and server applications), we need to install routing protocols, such as BGP and OSPF instead:

```
router [
    id 5 interface [id_range [from 1 to 2]]
    graph [
        ProtocolSession [ name "bgp"
            use "SSF.OS.BGP4.BGPSession" ]
        ProtocolSession [ name "ospf"
            use "SSF.OS.OSPF.sOSPF" ]
        ProtocolSession [name "socket"
            use "SSF.OS.Socket.blockingSocketMaster"]
        ProtocolSession [name "tcp"
           use "SSF.OS.TCP.tcpSessionMaster"]
        ProtocolSession [name "ip" use "SSF.OS.IP"]
    ]
]
```

The current version of PRIME SSFNet does not include routing protocols. Routing is done using shortest-path calculation at the pre-processing step and the result (static) forwarding tables are loaded upon simulation execution. This situation will change in later improvements.

Each protocol can be further supplied with various parameters. For example, the TCP client application must specify when it should starts sending a request to the server, and what is the size of the request message to be sent to the server, and what is the file size to be transferred from the server to the client. The TCP server application also must know the port number on which it is receiving requests from the clients. These parameters are specified within the `protocolsession` attribute of the corresponding protocol:

```
graph [
    ProtocolSession [name "TCPserver"
        use "SSF.OS.TCP.test.blockingTCPServer"
        port 10  # port number
        request_size 4  # request msg size in bytes
    ]
    ProtocolSession [name "TCPclient"
        use "SSF.OS.TCP.test.blockingTCPClient"
        start_time 1.0 # start time to send request
        start_window 1.0 # window for random send time
        request_size 4  # request msg size (in bytes)
```

```
            file_size 10000000 # request download size (in bytes)
        ]
        # other protocol sessions ...
    ]
```

The last step of network model configuration is to specify the application traffic. This is done by using the `traffic` attribute at the top-level network. The `traffic` attribute contains a list of traffic patterns, each describing a client or a set of clients initiating connections to a set of servers. To continue the previous example, we have two sub-networks, each with four hosts running both TCP servers and clients. We make the clients in one sub-network to connect to the servers in the other sub-network, and vice versa.

```
    Net [
        traffic [
            pattern [client 1
                servers [port 10
                    nhi_range [from 2:1(1) to 2:4(1)]]
            ]
            pattern [client 2
                servers [port 10
                    nhi_range [from 1:1(1) to 1:4(1)]]
            ]
        ]
        # the rest define the two sub-networks ...
    ]
```

The first traffic pattern says that all clients defined in network 1 will connect to one of the servers with IP addresses in the range between `2:1(1)` and `2:4(1)`. That is, the server will be chosen from one of four hosts: `2:1(1)`, `2:2(1)`, `2:3(1)` and `2:4(1)`, all in network 2. The server applications are all listening to port 10 for incoming connections.

2.3 The DML Library

DML is included as an open source library (`libdml`) in the PRIME distribution. The library provides an API for parsing, loading, and tretrieving the DML attributes. Each DML attribute, whether it's a single attribute or a compound attribute (with an attribute value as a list of attributes), is represented as an `prime::dml::Configuration` object[2].

The `prime::dml::Configuration` class has three important methods. The `findSingle()` method is used to obtain the DML attribute value of a given key. For example, using a previous example, `findSingle(".population.time")` will return the character string `"15:00:29 GMT"`. Similarly, `findSingle(".today")` will return the configuration object that represents the list of attributes including date and time. If the attribute with the given key does not exist, a

[2]In PRIME, we use namespace to protect against name collisions. All SSF classes and functions are defined in the `prime::ssf` namespace. All SSFNet classes and functions are defined in the `prime::ssfnet` namespace. Two other namespaces are `prime::dml` for the DML library and `prime::rng` for the random number generator (RNG) library.

NULL pointer will be returned. Otherwise, the user can use the `isConf()` method to determine whether the the returned value is a configuration object (i.e., another list of attributes) or a single attribute value (i.e., a character string). The returned value should then be cast to the appropriate data type accordingly. Note that if one wants to store the character string returned by this method in case of a single attribute value, the string should be copied explicitly. This is because the method returns only a pointer to an internal buffer, which could be later overwritten. The last method of concern is `find()`, which returns a list of attribute values of the same key (it is possible to have multiple attributes with the same key in DML). The user can enumerate through the list of attribute values. Note that one can also use a key with wild-card characters (i.e., "*" and "?" using their normal interpretations) as the argument to `findSingle()` and `find()`. Detail about the DML library and its API is in the *PRIME SSF Reference Manual*.

Note that the API of the current DML library (version 2) is somewhat cumbersome at listing and retrieving the attributes, even with the support of wild-cards. It also does not have a full support for type checking. Further improvement of the API seems necessary.

Also note that an easy way to learn how to use the DML library is through examples. In SSFNet, you can learn the use of the library from existing protocol implementations. For example, the `config()` method of `prime::ssfnet::IPSession` shows how one can retrieve DML attributes to configure the network protocol.

3 Basic Building Blocks

3.1 Virtual Time and Time Management

In this section, we discuss the virtual time type used by the simulator. We also discuss timers and time queues.

Virtual Time. The virtual time (which is also called the simulation time) in PRIME SSFNet is encapsulated by the `prime::ssfnet::VirtualTime` class. It provides the necessary time unit conversions between seconds, milliseconds, microseconds, and nanoseconds. Internally, the simulation time used by the underlying SSF simulator is of `prime::ssf::time_t` type, which is defined to be a long long integer. The internal `ltime_t` value is used to account the number of clock ticks. By default, a clock tick is set to be a nanosecond[3]. The `prime::ssfnet::VirtualTime` class has a set of constructors to create a simulation time from different primitive data types. The constructors all have an optional second argument, which can be used to specify the unit fo time (it's nanosecond by default). For example, the following create a time delay of 1.5 seconds:

```
VirtualTime delay(1.5, VirtualTime::SECOND);
```

The `second()`, `millisecond()`, `microsecond()`, and `nanosecond()` methods are used to return the virtual time in different time units. The return value in this case is a double precision floating point number. It is important to note that casting a `prime::ssfnet::VirtualTime` value to a primitive type results in the number of simulation clock ticks. Also defined in the

[3]Although it was originally suggested one should not assume that one tick is always one nanosecond, it is actually quite safe to treat it as such in our implementation.

`prime::ssfnet::VirtualTime` class are common arithmetic, comparison, and assignment operators so that a virtual time can be used together with most primitive types. In the following example we add another 150 milliseconds to the delay and then multiply it by a factor of 10. The result delay in microsecond is assigned to a variable. The delay is also used in the `waitFor()` function (defined in PRIME SSF) to suspend the simulation process for the prescribed period of time. There's an implicit type cast from `prime::ssfnet::VirtualTime` to `prime::ssf::ltime_t` when calling `waitFor()`, where the number of nanosecond clock ticks will be used.

```
delay += VirtualTime(150, VirtualTime::MILLISECOND);
delay *= 2;
double var_us = delay.microsecond();
waitFor(delay); // implicit cast to ltime_t
```

Timers. Timers are sometimes needed by protocols to regulate time advancement. For example, a timer is needed by a protocol that supports retransmissions. The `prime::ssfnet::Timer` class is designed particularly for this purpose. The class provides a `callback()` method, which the user can override in the derived class. The `callback()` method will be invoked when the timer expires. The `prime::ssfnet::Timer` class is an abstract class. That is, it cannot be instantiated directly. One has create a subclass, in which the `callback()` method is expected to be overloaded to specify the actions that need to be taken upon timeout.

Normally each timer is permanently associated with a protocol session. That is, a timer is typically owned by a protocol session and the ownership is not to be changed once it is set. To ensure this, the protocol session is passed as an argument to the constructor. Subsequently, one can call the `getSession()` method to return a pointer to the owner protocol session. A timer may also be independent of any protocol session, in which case the constructor of the timer class is called with a pointer to the current community as argument (we will discuss community in section 4). The `getSession()` method will return NULL in this case.

Each timer is also associated with a delay. Once the `set()` method is called, the timer starts running. And the `callback()` method will be called after the given delay (i.e., when the timer expires). One can set the delay when creating the timer object (passing the delay as an argument to the constructor), calling the `set()` method, or the `setDealy()` method. In any case, the delay value can be obtained through the `getDelay()` method. After a timer starts to run, one can stop it by calling the `cancel()` method. Cancelling a timer that has not started bears no effect. The `isRunning()` method and the `isCancelled()` method are designed for one to query the state of the timer; the former returns true if the timer is running while the latter returns true if the timer either hasn't started or is cancelled. If the timer is running, the `firetime()` method will return the simulation time at which the timer expires. If the timer is not running, the method will return 0 instead.

Timer Queues. In case a protocol session needs to manage many instances of the same timer, i.e., the same type of operation is to be scheduled at multiple time instances, for the sake of efficiency, one can use the `prime::ssfnet::TimerQueue` class. This class represents a queue of timers, each associated with a distinct user-defined data, which will be processed using the same callback function. To improve efficiency, the system will insert the timestamped data into a queue and only schedule one timer that corresponds to the earliest timestamp. When the timer expires,

the callback function will be invoked to process the corresponding data with the earliest timestamp. If more data are present in the timer queue, another timer will be scheduled subsequently. Thus, viewed from the simulation kernel, there is at most one timer actually in use for the entire queue of timers at any time. This ensures only one event is placed onto the simulator's event list and better efficiency is expected.

The constructor of the `prime::ssfnet::TimerQueue` class has two parameters: the first is the pointer to the protocol session that owns the timer queue, and the second is an optional boolean type argument. By default (when the second parameter is not provided), the user can insert data with timestamps of any particular order into the queue. One the other hand, if the user can be sure the data are inserted with non-decreasing order, the second parameter can be set to be true when constructing the timer queue, in which case a more efficient data structure will be used to deal with the special case of non-decreasing timestamps.

The user can invoke the `insert()` method and the `append()` method to insert a timestamped data into the queue. The latter requires that the timestamp of the data is larger than those already inserted into the queue. Internally, all data are sorted according to their timestamps and are stored in non-decreasing order in the queue. The timestamped data is represented by the `prime::ssfnet::TimerQueueData` class or its subclass. The base class contains nothing other than the timestamp (which can be obtained using the `time()` method). The user is expected to extend the `prime::ssfnet::TimerQueueData` class in order to include any additional information necessary to identify the timer event.

Like the `prime::ssfnet::Timer` class, the `prime::ssfnet::TimerQueue` class is an abstract class. The user needs to create a subclass from this class and provide a `callback()` method to handle the timeout. When a timer expires, the `callback()` method will be invoked with an argument being a pointer to the `prime::ssfnet::TimerQueueData` object.

One can use the `remove()` method to remove a timestamped data from the timer queue. The `clearQueue()` method can also be called to clear all data in the queue. The method has an optional argument: if one wants to reclaim the data when they are being cleared out from the queue, the argument should be set to be true.

3.2 DML Objects: Networks, Hosts, Routers, Links, and Interfaces

A DML object (of the abstract class `prime::ssfnet::DmlObject`) is a network entity that can be identified in DML. This includes networks, hosts, routers, links, and network interfaces. A DML object can be identified in simulation by its NHI address. Each DML object has three public attributes: `id`, which is an integer identifier, `nhi`, which is a `prime::ssfnet::Nhi` object, and `myParent`, which is a pointer to the parent DML object containing this object. For example, a network interface is contained in a host or router, which is in a network, which may be contained in a larger network.

Networks. A network or a sub-network in SSFNet is a `prime::ssfnet::Net` object. Conceptually, a network is a collection of hosts, routers, and sub-networks. The hosts and routers are connected through their network interfaces by links. An entire network model can thus be defined recursively. The outermost network is called the top-level network. Given a network model defined in DML, the simulator first creates the top-level network and then invokes its `config()` method with an argument of the DML configuration (i.e., a pointer to the `prime::dml::Configuration`

object) that represents the list of attributes within the outermost `net` in DML. The `config()` method creates the hosts, routers, and other networks defined within the network and calls their `config()` methods subsequently. The connections between hosts and routers are then made via links defined within the network. Upon returning from the `config()` method of the top-level network, the entire network model (in particular, the portion that assigned to the current processor) is thus created and configured. Note that we refer to host and router interchangeably. The functional difference between them is negligible at this point. We recommend, however, one should consistently use host to represent a machine running as the end host and router for a machine that provides routing and switching capabilities.

The list of hosts (and routers), sub-networks, and links defined in a network can be retrieved using the `getHosts()`, `getNets()`, and `getLinks()` methods, respectively. The return value of these methods is either a map or a vector. One can also translate from an NHI address to the corresponding DML object using the `nhi2obj()` method as long as it belongs to this network (or its sub-networks). Furthermore, one can get the IP prefix of this network using the `getPrefix()` method. Each network object also includes, as protected attributes, a list of predefined DML attributes (stored in the `prime::ssfnet::NetAccessory` object). This is important for protocols, such as OSPF and BGP, that need to mark the network as an autonomous system or an OSPF area. The `control()` method of the network class can be used to query for these attributes. In particular, an application protocol implementation can query for the application traffic model using this method. We describe the traffic specification in a later section.

The NetAccessory class is organized as a map from an integer (byte) id to a boolean, an integer, a double floating point number, or a pointer to an obscure object. That is, the attributes and their value types must be predefined. Further improvement of the API seems necessary.

Hosts and Routers. Both hosts and routers are represented by the `prime::ssfnet::Host` class. This class is derived both from the `prime::ssfnet::DmlObject` class, which is discussed previously, and the `prime::ssfnet::ProtocolGraph` class, which represents a protocol stack (i.e., a list of protocol sessions) on the host. We defer the discussion of protocol graphs and protocol sessions to a later section. The `inNet()` method returns the immediate network that contains this host or router. The `isRouter()` method returning true indicates the host object is representing a router rather than an end host. Each host or router may contain one or more network interfaces. The list of network interfaces can be obtained through the `getInterfaces()` method. One can get a particular network interface through its id (using the `getInterface()` method) or its IP address (using the `getInterfacebyIP()` method). Each host or router has a default IP address, which can be obtained using the `getDefaultIP()` method.

Conceptually each host or router also maintains its own clock. One can use `getNow()` to retrieve the current (simulation) time. Each host object also maintains a random number generator, which is seeded as a function of the host's default IP address. The user has the option to set the RNG level in DML (using the `rng_level` attribute). If the RNG level is set to be "protocol" (which is the default), each protocol session running on the host will have its own random number stream. Otherwise, if the RNG level is set to be "host", all protocol sessions on the host will share the same random stream.

Although not yet implemented, the clocks within hosts and routers should not be perfectly synchronized. There should be some offset and possibly clock drifts. Currently getNow() will simply return the simulation clock value. Possible improvements include adding a jitter start time

and adding another function that returns a time period that takes clock drifts into account.

Links. A `prime::ssfnet::Link` object is used to represent a shared-medium connecting two or more network interfaces. There is a delay associated with each link object and it can be configured in DML (zero by default). One can get the link delay using the `getDelay()` method. The `getInterfaces()` method returns the list of network interfaces attached to the link. Similarly, the `getIfaceNhis()` method returns a list of NHI addresses of all network interfaces attached to the link. Like network, a link also has a network IP prefix, which is one can get by using the `getPrefix()` method.

The link object also provides a way of associating an IP address with its corresponding network interface object. The `ip2iface()` method returns the network interface object of the given IP address, assuming the network interface object is instantiated on the same processor. (It is possible for a link to span across multiple processors when parallel simulation is engaged). Note that a packet sent from a network interface is expected to be delivered to all network interfaces on the same link except the sender. This function is actually accomplished by the network interface.

Currently PRIME SSFNet does not include any detailed model below layer 3. For example, there's no Ethernet model. Also, there's no ARP. The hardware address is assumed to be the IP address when packets are delivered.

Network Interfaces. The `prime::ssfnet::Interface` class is used to represent the network interfaces. Similar to host, this class is derived from both the `prime::ssfnet::DmlObject` class and the `prime::ssfnet::ProtocolGraph` class. The reason for deriving it from the protocol graph is that a network interface can implement separate protocols—in particular, the MAC-layer and the PHY-layer protocols—which can all be specified in DML. If in DML the protocols are not specified, the network interface will instantiate two default protocol sessions: one is `prime::ssfnet::SimpleMac` and the other is `prime::ssfnet::simplePhy`, both functioning merely as placeholders.

There are two related methods: the `getHighestProtocolSession()` method returns the highest protocol session on the protocol stack implemented by the network interface (e.g., the MAC-layer protocol); the `getLowestProtocolSession()` method returns the lowest protocol session on the protocol graph within the network interface (e.g., the PHY-layer protocol). A network interface is always owned by a host. The `getHost()` method returns a pointer to the host that owns the interface.

Each network interface is assigned an IP address, which can be obtained using the `getIP()` method. In SSFNet, the hardware address of a network interface is the same as its IP address, since we do not have layer-2 models and we do not have ARP (or the Address Resolution Protocol). So the `getMacAddr()` method simply returns IP address as well. The `getLink()` method returns the link object this network interface is attached to. The `getPeerInterfaceByIP()` returns a pointer to the peer network interface specified by the IP address, which is only valid if the peer network interface is located on the same processor.

Network interfaces are responsible for sending and receiving packets. The `sendPacket()` method is used to send a packet (a `prime::ssfnet::Packet` object) to its desired destination. As a matter of fact, this method will deliver the packet to all other network interfaces connected with the sender interface (i.e., attached to the same link). If the network interface is located on another processor in a parallel simulation scenario, an event will be written out to the out-channel.

The SSF simulation kernel will be responsible for delivering the event to the corresponding simulation process and finally to the target network interface. If the network interface is located on the same processor, a timer will be scheduled to expire after some delay and the packet will be given to the target network interface when the timer expires. The packet is delivered by SSFNet by invoking the `receivePacket()` method of the target network interface. The sender's MAC address (which is the same as the sender's IP address) is also provided as an argument to the method.

3.3 Protocol Graphs and Protocol Sessions

Each host or router must specify one or more network protocols. These network protocols are organized as a protocol graph (also called a protocol stack with regard to the traditional layered approach). A protocol graph is represented by the abstract `prime::ssfnet::ProtocolGraph` class. As mentioned earlier, both the host class and the network interface classe are subclasses of the protocol graph class. The protocol graph maintains a list of protocol sessions, each representing a protocol running either on a host or router, or within a network interface of a host or router. Regardless whether the protocol session is in a host or in a network interface of a host, the `getHost()` method can be used to obtain a pointer to the host object.

All protocol sessions are configured by DML. Specifically, within each host or router, one can define the list of protocol sessions using the `graph` attribute. Also, for protocols running in a network interface, one can provide the list of protocol sessions inside the `interface` attribute. Each protocol session is defined by a `protocolsession` attribute. The ordering of the protocol sessions is determined by the protocols themselves and is not dependent on the ordering of the `protocolsession` attributes in DML. For example, a TCP protocol session would assume there is an IP protocol session defined below it on the protocol stack. The following example shows a host with one network interface, In the host, we define a socket layer, which is run on top of TCP and UDP over IP. Within the network interface, we define a MAC and a PHY-layer protocol.

```
host [id 10
  graph [
    ProtocolSession [name "socket"
       use "SSF.OS.Socket.blockingSocketMaster"]
    ProtocolSession [name "TCP"
       use "SSF.OS.TCP.tcpSessionMaster"]
    ProtocolSession [name "UDP"
       use "SSF.OS.UDP.udpSessionMaster"]
    ProtocolSession [name "IP" use "SSF.OS.IP"]
  ]
  interface [id 0
    ProtocolSession [ name "MAC" using "SSF.OS.SimpleMAC"]
    ProtocolSession [ name "PHY" using "SSF.OS.SimplePHY"
      bitrate 1e8 latency 0 jitter_range 0.2
    ]
  ]
]
```

Each protocol graph must define a network protocol session (which has a name being either "NET" or "IP"). If it is undefined in DML, the system will create a default one nonetheless. The `getNetworkLayerProtocol()` method returns a pointer to the network protocol session. If emulation is enabled, PRIME SSFNet will also require each host to have an emulation session. Again, if it's undefined in DML, the system will create a default an emulation session. The only difference is that the automatically generated emulation sessin will have its *intractability* flag disabled and therefore will not accept external connections (refer to section 8 for more details).

A pointer to each protocol session object defined in a protocol graph can be retrieved under its name using the `sessionForName()` method. Presumably, each protocol session must have a unique name, which is case insensitive. (An exception to the uniqueness is discussed below). A list of names for standard protocol sessions is defined in `src/os/ssfnet.h`, including PHY, MAC, NET, IP[4], TCP, UDP, SOCKET, EMU[5], ICMP, and HTTP. More names will be defined as more protocol sessions are added. Each protocol session is also associated with a unique number, which we call the protocol type. For protocols corresponding to real network protocols, the protocol type is the same as the demux key used in the IP packet. For example, ICMP is 1, TCP is 6, and UDP is 17. For protocols that are used solely in the simulation, the protocol type is an integer uniquely defined in the simulator. They are listed in the `src/os/ssfnet.h` file. The `sessionForNumber()` method returns the pointer to the protocol session of the given protocol type.

The `prime::ssfnet::ProtocolSession` class represents a protocol on the protocol stack. It's actually the base class for a protocol session. It provides almost all necessary support for implementing a protocol in the simulator, including passing information between protocol layers. Each protocol session has a name, a character string that is stored in the `name` attribute. A protocol can have more than one implementation. The `use` attribute stores the name of the protocol class from which this protocol session object is to be instantiated. The `version` attribute stores the version of the protocol, the use of which is not yet implemented, however.

Since C++ does not support self-reflection, each protocol implementation, represented by a subclass of the `prime::ssfnet::ProtocolSession` class, must declare using a macro named `SSFNET_REGISTER_PROTOCOL`, defined in the `prime::ssfnet::Protocols` class. The declaration is neceesary so that the system can associate the name of the protocol session (which is case insensitive) with the protocol session class itself when parsing the DML configuration file. The macro is supposed to be declared together with the class definition (in the source file) and it has two parameters: the name of the protocol session class and a string that identifies the protocol in DML. For example, the default IP protocol implementation is declared in `src/os/ipv4/ip_session.cc` using:

```
SSFNET_REGISTER_PROTOCOL(IPSession, "SSF.OS.IP");
```

This creates a static mapping from the string name `"SSF.OS.IP"` to the factory method of the `prime::ssfnet::IPSession` class. When DML is parsed and if a protocol session is specified such as the following:

[4]The network-layer protocol has two names, NET and IP, that can be used interchangeably.

[5]The emulation and the socket protocol sessions are not network protocols. The former is defined to support emulation by importing and exporting network traffic, and the latter is defined for supplying socket programming interface to applications.

```
graph [
   ...
   ProtocolSession [name "IP" use "SSF.OS.IP"]
 ]
```

The corresponding factory method will be invoked and as a result an instance of the IP session class will be created accordingly. After an instance is created, the system will invoke the `config()` method with the DML configuration of the protocol session passed as argument. After that, the `init()` method will be called to initialize the protocol session. In a similar fashion, the `wrapup()` method will be called when the simulation finishes. The `config()`, `init()`, and `wrapup()` methods in the `prime::ssfnet::ProtocolSession` class provide only necessary functions for configuring, initializing, and wrapping up the protocol session instance. It is expected that the ones in the subclasses will override these methods. It is important to note that the `config()`, `init()`, and `wrapup()` methods defined in the subclasses must first invoke the corresponding methods in the base class so that the basic functions are executed. For example, the methods defined in the `prime::ssfnet::IPSession` class should start as follows:

```
void IPSession::config(prime::dml::Configuration *cfg) {
    ProtocolSession::config(cfg);
    ...
}
void IPSession::init() {
    ProtocolSession::init();
    ...
}
void IPSession::wrapup() {
    ProtocolSession::wrapup();
    ...
}
```

Failing to invoke the methods in the base class would cause unpredictable errors. For example, the `inited()` method, which returns true if the protocol session has been initialized, relies on the `init()` method at the base class being called. The `inited()` method is useful for sequencing calls to the `init()` methods of protocol sessions. For example, if protocol session A requires protocol session B to be initialized before A can be initialized, A's `init()` method can call B's `init()` method as long as B's `inited()` returns false. This arrangement is necessary since the simulator does not enforce ordering in its calls to the `init()` method of all protocol sessions defined in the DML configuration.

The communication among protocol sessions within the same protocol graph is carried out primarily through three methods. The `pushdown()` method is used by the protocol session above to send (or "push") a protocol message down the protocol stack. This method actually calls the `push()` method, which the protocol session is expected to override to deal specifically with protocol messages from protocol sessions above. On the reverse path, the `popup()` method used by the protocol session below to "pop" a protocol message that is received up the protocol stack. Similarly, the `popup()` method calls the `pop()` method, which the protocol session is expected

to override. We will elaborate more on the `push()` and `pop()` methods when we discuss the protocol messages in the next section.

The third method is `control()`, which is used by the protocol session to receive control messages (or information queries) from other protocol sessions both above and below on the protocol stack. It is expected that the protocol class overrides this method to handle different control messages, which are distinguished by their types. The type, which is an integer, is passed as the first argument to the method, with its semantics defined by each protocol session. We recommend the user to define control message types using values beyond 100 in order to avoid conflicts with reserved types whose values are kept within 100. The second argument to the method is the control message itself, which is simply a generic pointer. The actual semantics of this control message depend on its type and the message should be cast to the appropriate type accordingly upon use. The protocol session that invokes the `control()` method also needs to provide a reference to itself as a third argument. It is important to know that the caller of the `control()` method is responsible to reclaim the control message once the method returns. The method returns an integer, indicating whether the control message has been successfully processed. Unless specified otherwise, zero means success. Similar to the `config()`, `init()`, and `wrapup()` methods, the derived class should invoke the method in the base class when encountering a control type that it does not know to to process. The following snippet is an example that shows how this method is used in the `prime::ssfnet::IPSession` class (the control types shown are actually integers defined elsewhere):

```
int IPSession::control(int ctrltyp, void* ctrlmsg,
                       ProtocolSession* sess) {
  switch(ctrltyp) {
    case IP_CTRL_VERIFY_LOCAL_IP_ADDRESS: {
      // ctrl msg is a pointer to a 32-bit address
      return verify_local_ip_address(*((uint32*)ctrlmsg));
    }
    case IP_CTRL_GET_FORWARDING_TABLE: {
      // ctrl msg points to  a forwarding table
      *((ForwardingTable**)ctrlmsg) = forwarding_table;
      return 0;
    }
    ...
    default:
      return ProtocolSession::control
               (ctrltyp, ctrlmsg, sess);
  }
}
```

There are also methods that can be used to significantly help the implementation of a protocol session. The `inGraph()` method returns the protocol graph in which this protocol session resides. The `inHost()` method returns a pointer to the host (or the router) that owns this protocol session, regardless whether the protocol session is defined directly in the host's protocol graph or is defined as a protocol session in the network interface. The `getNow()` method returns the current time of the host or router. The `getRandom()` method returns a pointer to the random number

generator assigned to the protocol session. As mentioned earlier, if the RNG level of the host or router is set to be "protocol" (which is the default), the protocol session will implicitly create its own random stream. If the RNG level is set to be "host", all protocol sessions on the host will share the same random stream.

In a typical situation, each protocol session must use a unique protocol number to represent the type of the protocol. This number will be used to retrieve the protocol session object using the protocol graph's `sessionForNumber()` method. A protocol session subclass (derived from `prime::ssfnet::ProtocolSession`) must override the `getProtocolNumber()` method to return the corresponding protocol number. It is also used to demultiplex the protocol messages (we discuss the protocol messages in the next section). In most cases, there is only one instance of a protocol session in a host or a router. Having multiple instances of a protocol session is possible, however, especially if it is an application-layer protocol. For example, there can be multiple TCP or UDP client applications running on each host. The `instantiation_type()` method specifies how protocol instances may coexist with one another on a protocol stack. The method returns `PROT_UNIQUE_INSTANCE`, by default, meaning the protocol allows only one instance of the protocol session on a protocol stack. Protocol session subclasses can override this method to behave differently. The method should return `PROT_MULTIPLE_INSTANCES` to allow multiple instances of the same implementation of the protocol session on a protocol stack, or `PROT_MULTIPLE_IMPLEMENTATIONS` to allow multiple instances from potentially different implementations of the same protocol to exist on the protocol stack. In the latter two cases, the `sessionForName()` method and the `sessionForNumber` method may return just one instance of the protocol session.

3.4 Protocol Messages and Packets

Protocol sessions on the same protocol stack communicate with one another using protocol messages. Messages to be sent out from upper-layer protocols are pushed down the protocol stack to lower-layer protocols. The corresponding packet headers are added for encapsulation. The `prime::ssfnet::ProtocolMessage` class is the base class representing a packet header. Specific protocols should have their own protocol message subclasses to represent different packet headers. For example, the `prime::ssfnet::IPMessage` class represents the packet header used by IP. Similarly, the `prime::ssfnet::ICMPMessage` class represents the packet header for the ICMP protocol, the `prime::ssfnet::TCPMessage` class represents TCP headers, and the `prime::ssfnet::UDPMessage` class represents UDP headers. They are all derived from the `prime::ssfnet::ProtocolMessage` class. A protocol message is organized in simulation as a (single) linked list of objects instantiated from the subclasses of the `prime::ssfnet::ProtocolMessage` class. The payload of a protocol message may very well be the packet header of an upper-layer protocol on the protocol stack. For example, since TCP is to be encapsulated in IP, the protocol message starts with a `prime::ssfnet::IPMessage` object, followed a `prime::ssfnet::TCPMessage` object. The IP protocol message treats the TCP protocol message as its payload. The reason why a single linked list is chosen rather than a double linked list in the implmentation is that it's easier to support optimizations made by individual protocol message classes (e.g., using memory reference counters to reduce the memory consumption).

The `prime::ssfnet::ProtocolMessage` class provides methods for creating, copy-

ing, and reclaiming protocol message instances, attaching and detaching packet headers, determining message types, and serializing and deserializing protocols messages to and from a byte stream. Serialization is necessary for the simulator to deliver packets across memory boundaries (for distributed simulation). A protocol message can be packed into a byte array before transport. Conversely, a byte array can be unpacked into a list of protocol messages. To support serialization, a derived protocol message class must register with the simulator using the `SSFNET_REGISTER_MESSAGE` macro, defined with the `prime::ssfnet::Messages` class. The macro has two parameters: the name of the protocol message class and an integer identifier of the type of the protocol message, which is the same as the type of the protocol. As mentioned earlier, for protocols corresponding to real network protocols, the protocol type is the same as the demux key used in the IP packet. For protocols that are used solely in the simulation, the protocol type is an integer uniquely defined in the simulator. They are listed in the `src/os/ssfnet.h` file. (The `type()` method should be overridden to return the type id). When packing the protocol message, the type id will be embedded in the byte stream. When unpacking, the type id will allow the simulator to invoke the constructor of the corresponding protocol message class.

It is important to note that a derived protocol message class must supply a default constructor without parameters. This is necessary to support serialization, so that the simulator will be able to re-create the protocol message object from the given message type id. It is also required that each derived class provides a copy constructor. Note that C++ does not implicitly call the copy constructor of the base class from that of the derived class. It must be made to do so explicitly. Among many things, the copy constructor of the base class is expected to clone the payload of the protocol message. It is also important that the base class's copy constructor is called by the copy constructor of the derived class. Failing this, the payload of the message will not be copied correctly. Suppose that `MyMessage` is a derived protocol message class, the following example shows how to write its copy constructor:

```
MyMessage::MyMessage(const MyMessage& msg) :
    ProtocolMessage(msg) {
        myfield_1 = msg.myfield_1;
        myfield_2 = msg.myfield_2;
        ...
}
```

Another required method in the derived protocol message class is `clone()`, which should be defined in the derived class as something like:

```
ProtocolMessage* MyMessage::clone()
    { return new MyMessage(*this); }
```

To reclaim a protocol message, one should not directly use the `delete()` operator, which will invoke the destructor of the protocol message. Instead, one should call the `erase()` method. There are two reasons: one is that we should reclaim the memory of the current protocol message as well as its subsequent payload; the other is to support possible memory reference counter schemes implemented by different protocol message classes. For better protection, the destructor of the protocol message (and its extended classes) should be declared as a protected method.

The `carryPayload()` method can be used to append another protocol message treating it as the payload. Both `payload()` and `dropPayload()` methods return the next packet

header. The difference is that the second method will detach the payload from the linked list. The `getMessageByType()` method goes through the protocol message chain and returns the first protocol message of the given protocol type.

The following methods are used for serialization. The `totalPackingSize()` method returns the total number of bytes needed to serialize the whole chain of protocol messages. The method actually invokes the `packingSize()` method of each protocol message on the linked list, which returns the number of bytes needed to pack the packet header. The `totalPackingSize()` method the simply adds it up. It is expected that the derived protocol message class override the `packingSize()` method. It is important that the method in the base class be called by the corresponding method of the derived class. The byte count returned by the method in the base class should be included in the calculation[6]. Similarly, the `serialize()` method (or the `deserialize()` method) is expected to be overridden in the derived class. The method is used to serialize (deserialize) the protocol message into (from) the given buffer starting at a given offset. It is also important that the method in the base class is called first by the method in the derived class so that the base class gets the opportunity to insert (or retrieve) necessary information supporting serialization. Failing to do that will prevent serialization from working properly. In the following example, suppose we have a protocol message with two data fields: one 32-bit integer (a) and one single precision floating pointer number (b). The `serialize()` and `deserialize()` methods defined in the `prime::ssf::ssf_compact` class can be used for (endian format) translation between variables in host order and those in network order. To allow protocol sessions running on different machine platforms to be able to correctly communicate with one another (via the serialized byte streams), the information packed into the byte streams should be in network order.

```
int MyMessage::packingSize() {
  return ProtocolMessage::packingSize()+
      sizeof(int32)+sizeof(float);
}

void MyMessage::serialize(byte* buf, int& offset) {
  ProtocolMessage::serialize(buf, offset);
  prime::ssf::ssf_compact::serialize(a, &buf[offset]);
  offset += sizeof(int32);
  prime::ssf::ssf_compact::serialize(b, &buf[offset]);
  offset += sizeof(float);
}

void MyMessage::deserialize(byte* buf, int& offset) {
  ProtocolMessage::deserialize(buf, offset);
  prime::ssf::ssf_compact::deserialize(a, &buf[offset]);
  offset += sizeof(int32);
  prime::ssf::ssf_compact::deserialize(b, &buf[offset]);
  offset += sizeof(float);
```

[6]In the current implementation, the base class will contribute an integer value (i.e., the message type) to identify the protocol message in the byte stream.

```
}
```

The `prime::ssfnet::ProtocolMessage` class also provides methods for supporting emulation. In particular, there are methods for translating from the protocol messages used in simulation to the corresponding packets that are sent over physical networks, and vice versa. We delay the discussion of emulation support until Section 8.

3.5 IP Addresses, IP Prefixes, and Forwarding Tables

IP Addresses. Each network interface in the network model is statically assigned an IP address[7]. An IP address is a 32-bit unsigned integer[8], type defined as `prime::ssfnet::IPADDR`. Along with the type definition, there are several macros that one can use at convenience. `IPADDR_LENGTH` is the number of bits that we use to represent an IP address, i.e., 32. `IPADDR_INVALID` represents an invalid IP address, which is defined to be 0. `IPADDR_ANYDEST` is the broadcast IP address, which is `255.255.255.255`. `IPADDR_LOOPBACK` represents the IP address of the local host, which is `128.0.0.1`.

IP Prefixes. The `prime::ssfnet::IPPrefix` class represents the IP prefix that we use to describe a set of IP addresses for classless inter-domain routing (CIDR). An IP prefix consists of a network address and a length indicating the number of significant bits to be used as the network mask, e.g., `10.10.0.0/16`. The class contains static utility functions that one can use to convert an IP address between the integer representation and the dot notation. The `ip2txt()` method returns the character string containing the dot notation of the IP address. Conversely, the `txt2ip()` method turns the dot notation to a 32-bit integer. Note that there are two incarnations to the `ip2txt()` method, one with an additional argument, a pointer to a buffer where the result dot notation of the IP address is expected to be stored. If the second argument is not provided, the method will use an internal buffer to store the result and return its location. In this case, the user should not reclaim the buffer since it belongs to the system. Also, since the system uses only one such internal buffer to store the result, the user is advised that successive calls to the method will cause the result to be overwritten. The following example shows a correct way to display multiple IP addresses:

```
IPADDR src_ip, dest_ip;
...
/* the wrong way to print out the two IP addresses */
printf("src_ip=%s, dest_ip=%s\n",
       IPPrefix::ip2txt(src_ip),
       IPPrefix::ip2txt(dest_ip));
*/
/* the correct way should be: */
char src_buf[20], dest_buf[20];
printf("src_ip=%s, dest_ip=%s\n",
```

[7] It can also be addressed using the NHI address. The simulator has a global name service providing translations between the NHI addresses and the IP addresses. We describe the global name service in more detail in section 4.

[8] IPv6 is not implemented.

```
        IPPrefix::ip2txt(src_ip, src_buf),
        IPPrefix::ip2txt(dest_ip, dest_buf));
```

Forwarding Tables. Each host or router must have a forwarding table so that packets can be forwarded to their destinations accordingly. The forwarding table is maintained by the IP protocol session in the protocol graph (an IP protocol session is required to be present in all routers and hosts). Forwarding tables are implemented by the `prime::ssfnet::ForwardingTable` class. The class is inherited from the based class named `prime::ssfnet::Trie`. Trie is a data structure for fast longest prefix matching. Conceptually, it is a mapping from a bit string to a user-defined data. The bit string is represented as a 32-bit integer. To insert the bit string into the trie, a length field must be specified to identify the number of significant bits of the bit string that will be used for indexing. Both the bit string and the length field will be used together as the key in trie for the user-defined data inserted in the data structure. The user-defined data is represented by the `prime::ssfnet::TrieData` class and its subclasses. The base class is simply a place holder.

In a forwarding table, the trie key will be an IP prefix and the user-defined data will be a route entry, which is represented by the `prime::ssfnet::RouteInfo` class, a subclass of the `prime::ssfnet::TrieData` class. Each route entry contains a destination IP prefix (which has an IP address and a length field), an IP address of the next hop, a pointer to the network interface, an integer cost, and an identifier of the protocol that manages the route entry. There are two ways to specify a route entry in DML: one is within the host or router's DML configuration and the other is to use the `FORWARDING_TABLE` attribute that can be placed separately from the main network configuration. The latter is particularly useful if one pre-calculates the routes during the preprocessing of a network DML model. We discuss the pre-processing of DML model in the next section. The following example shows both cases; the forwarding tables for routers and hosts in net 1 are specified in place, while those in net 2 are specified separately.

```
Net [
  Net [id 1
    router [id 0 graph [...]
      interface [id 0 ...]
      interface [id 1 ...]
      interface [id 2 ...]
      nhi_route [dest "1(0)" interface 1]
      nhi_route [dest "2(0)" interface 2]
      nhi_route [dest "default" interface 0]
    ]
    host [id 1
      graph [...]
      interface [id 0 ...]
      nhi_route [dest "default" interface 0]
    ]
    host [id 2
      graph [...]
      interface [id 0 ...]
```

```
      nhi_route [dest "default" interface 0]
    ]
    link [attach 0(1) attach 1(0)]
    link [attach 0(2) attach 2(0)]
  ]
  Net [id 2
    router [id 0
      graph [...]
      interface [id 0 ...]
      interface [id 1 ...]
      interface [id 2 ...]
    ]
    host [id 1
      graph [...]
      interface [id 0 ...]
    ]
    host [id 2
      graph [...]
      interface [id 0 ...]
    ]
    link [attach 0(1) attach 1(0)]
    link [attach 0(2) attach 2(0)]
  ]
  link [attach 1:0(0) attach 2:0(0)]
]
forwarding_table [node_nhi "2:0"
  nhi_route [dest "1(0)" interface 1]
  nhi_route [dest "2(0)" interface 2]
  nhi_route [dest "default" interface 0]
]
forwarding_table [node_nhi "2:1"
  nhi_route [dest "default" interface 0]
]
forwarding_table [node_nhi "2:2"
  nhi_route [dest "default" interface 0]
]
```

Whether we define the route entries within routers and hosts or using separate `forwarding_table` attributes, we can use either the `route` attribute or the `nhi_route` attribute to specify them. A `route` attribute may contain five sub-attributes: `dest_ip` specifies the IP prefix of the destination, `interface` specifies the id of the network interface through which the destination can be reached, `next_hop` specifies the IP address of the next hop, `cost` is the cost (an integer) of the route entry, and `protocol` identifies the protocol that maintains the route entry. We may use "default" for `dest_ip` if we want to define a default route entry. The protocol must be selected from those predefined in the `prime::ssfnet::RouteInfo` class: "STATIC" means the route

entry is installed statically (i.e., it's pre-calculated and specified in DML); "IGP" means the route entry is maintained by an interior gateway protocol (such as OSPF), "EGP" means it's by an exterior gateway protocol (such as BGP). The `next_hop` attribute does not have to be specified when the route is through a point-to-point connection (i.e., the corresponding network interface is connecting to only another network interface). Otherwise, the next hop must be specified. If the cost of the route is unspecified, it is one by default[9]. If the protocol is unspecified, it is assumed to be "STATIC".

As shown in the above example, the route entries can also be specified using the `nhi_route` attribute. The only difference from the `route` attribute described above is that we use NHI addresses for destination and next hop. Also, rather than `dest_ip`, we use `dest` to specify the destination's NHI address. When the route entries are loaded into the system, these NHI addresses will be converted to IP addresses. Note that the system first assumes that the NHI addresses provided in `dest` and `next_hop` are relative NHI addresses. That is, they refer to DML objects within the same network of the host or router where the forwarding table is defined. If it is not, the system then treats the NHI addresses as absolute NHI addresses. Alternatively, to avoid confusion, one can specify an absolute NHI address by starting it with a colon, such as ":1:1(0)".

Currently using nhi_route, we can only specify the NHI address of a network interface or use "default" for the destination. In particular, one cannot use the NHI address of a network. This situation can be improved if we have a global mapping from the network NHI addresses to the IP prefixes. We will add it in a later version.

The `prime::ssfnet::ForwardingTable` class is used to represent the forwarding table, maintaining entries from routing calculations by one or more routing protocols running on each host or router. Specifically, the `addRoute()` method can be used to add a route entry; the `removeRoute()` method can be used to delete a route entry; the `getRoute()` method returns a route to a given IP prefix; the `getDefaultRoute()` method fetches the default route; and the `invalidateAllRoutes()` method removes all routes in the forwarding table of a particular protocol.

The forwarding table is maintained by the network layer protocol. It can be accessed in a protocol session by sending a control message to the IP protocol session: one can load forwarding table entries from a DML configuration (using a control message of type `IP_CTRL_LOAD_FORWARDING_TABLE`), insert route entries directly to the forwarding table (using a control message of type `IP_CTRL_INSERT_ROUTE` or `IP_CTRL_INSERT_DMLROUTE`), or retrieve a pointer to the forwarding table object itself (using a control message of type `IP_CTRL_GET_FORWARDING_TABLE`). The following example shows how to insert a new route entry from a routing protocol:

```
// prepare the new route
RouteInfo* newroute = new RouteInfo();
...
newroute->resolve(inHost());

// access to the ip session
ProtocolSession* ipsess =
    inHost()->getNetworkLayerProtocol();
```

[9]The cost of a default route entry is set to be the maximum by default.

```
// directly insert into table obtained from control
ForwardingTable* ft;
ipsess->control(IP_CTRL_GET_FORWARDING_TABLE, &ft, this);
ft->addRoute(newroute->destination, newroute);

// alternatively, use control to insert the route
ipsess->control(IP_CTRL_INSERT_ROUTE, newroute, this);
```

3.6 Application Traffic Specification

As mentioned earlier, application network traffic can be specified using the `traffic` attribute at the top-level network in DML. Traffic is arranged as a list of traffic patterns, each describing a set of servers that a client or a set of clients will be able to connect to. In the system, the application network traffic is represented by the `prime::ssfnet::Traffic` class, which provides the methods for mapping clients to servers (with which they can establish connections). There is one such traffic object for each simulation process. Application-layer client protocols are able to access this traffic information and then connect to the chosen servers accordingly. The following example shows that one can obtain access to the traffic object from a protocol session by sending a control message to a network object:

```
Traffic* traffic = 0;
Host* host = inHost();
host->inNet()->control
    (NET_CTRL_GET_TRAFFIC, (void*)&traffic);
if(traffic == NULL) ... // no global traffic specified in DML
```

One can then obtain the list of traffic patterns using the `getServers()` method. This method is meant to be called from a client protocol. It will check whether the host running the client protocol matches the client specified in the traffic patterns and return only those that match. More specifically, the client NHI address specified in each traffic pattern is an NHI address of a network or a host. In the former case, all hosts belong to the network (as well as its sub-networks) can be qualified as a match. Information of the corresponding servers of all matched traffic patterns will be returned in the form of a vector of pointers to `TrafficServerData` objects. Each object contains a server's NHI address, the port number used by the server for incoming connections, the number of simultaneous connections allowed to be made from the client to the server, and a character string indicating the type of service. One can get servers only with a particular type of service by providing the corresponding character string as the third argument to the `getServers()` method. There is no standard way to define the type of services used by various application client protocols. For example, the TCP client applications implemented in the `ssfnet/os/tcp/test` directory use "forTCP" as the type of service. For new applications, one can use any string for the service type as long as it does not collide with existing applications.

The following example shows an application network traffic specified in DML. All client applications in network 1 will connect to server applications in network 2. More specifically, TCP client applications in network 1 may contact the TCP server application running in host `2:1(1)` waiting on port 10. UDP client applications in network 1 may contact any one of UDP server

applications running in hosts `2:1(1)` through `2:4(1)`, all waiting on port 20. Similarly, client applications in network 2 will connect to server applications in network 1.

```
Net [ # top-level network
  traffic [
    pattern [client 1
      servers [port 10 list "forTCP" nhi 2:1(1)]
      servers [port 20 list "forUDP"
        nhi_range [from 2:1(1) to 2:4(1)]]
    ]
    pattern [client 2
      servers [port 10 list "forTCP" nhi 1:1(1)]
      servers [port 20 list "forUDP"
        nhi_range [from 1:1(1) to 1:4(1)]]
    ]
  ]
  ...
]
```

A client application protocol running on host, say `1:3:23:14(0)`, may try to obtain the list of servers offering "forTCP" service, as follows:

```
SSFNET_VECTOR(TrafficServerData*) servers;
if(traffic->(inHost(), servers, "forTCP")) {
   // if there are servers returned
   for(SSFNET_VECTOR(TrafficServerData*)::iterator
       iter = servers.begin();
       iter != servers.end(); iter++) {
      TrafficServerData* server_data = (*iter);
      ...
   }
}
```

In this example, the result `servers` vector will contain only one element that represents the server on host `2:1(1)`.

4 Basic Services

4.1 Structure of a Network Model

PRIME SSFNet support parallel and distributed network simulation. That is, the simulation may run on multiple processors. On each physical processor that one uses to run the simulation, the simulator instantiates a `prime::ssfnet::AlighmentService` object. The PRIME SSFNet simulator does not care for whether the processors share the same memory or run on a distributed platform. The underlying SSF simulation kernel takes care of communication and synchronization in both cases.

In DML, each network, router, or host can be given a different alignment name using the `ALIGNMENT` attribute. Networks, routers, and hosts sharing the same alignment name are assigned to the same timeline. Each timeline potentially can be assigned to a different processor for parallelism. In the simulator, the timeline is also called community. The following example is a network model with three timelines (or communities): "alignment1" contains routers `1:0` and `1:1`, and hosts `1:2` to `1:11`; "alignment2" contains router `2:0` and hosts `2:1` to `2:10`, "default" contains routers `0:0` and `0:1`, and hosts `1:12` to `1:21`. That is, if the alignment name of a host, router, or subnet is not specified, it shares the same timeline as its parent network. Ultimately, if the top net's alignment is not specified, a "default" timeline will be used.

```
Net [ # "default" alignment even though it's not specified
  Net [net 0
    router [id 0 ...]
    router [id 1 ...]
    link [attach 0(1) attach 1(1)]
  ]
  Net [id 1 alignment "alignment1"
    router [id 0 ...]
    router [id 1 ...]
    host [id_range [from 2 to 11] ...]
    host [id_range [from 12 to 21] alignment "default" ...]
    link [attach 0(2) attach 2(0) ... attach 11(0)]
    link [attach 1(1) attach 12(0) ... attach 21(0)]
    link [attach 0(1) attach 1(0)]
  ]
  Net [net 2 alignment "alignment1"
    router [id 0 ...]
    host [id_range [from 1 to 10] ...]
    link [attach 0(1) attach 1(0) ... attach 10(0)]
  ]
  link [attach 0(0) 1:0(0)]
  link [attach 1(0) 2:0(0)]
]
```

Before the simulation is run, a pre-processor program will be used to partition the network model among the available processors. As a result, the timelines will be assigned to processors to maintain load balance. When the simulation starts, an alignment service object (of the `prime::ssfnet::AlignmentService` class) will be instantiated on each processor and the class provides the necessary alignment information so that the simulator will be able to locate the processor to which the timeline is assigned. The alignment service also creates and manages the timelines that assigned to the corresponding processor.

Along with the alignment service, on each processor, the simulator instantiates an object of the `prime::ssfnet::NameService` class for global name resolution service. This class contains global information so that we can, among other things, translate between host names, IP addresses, and NHI addresses.

Each timeline is represented by an instance of the `prime::ssfnet::Community` class. A community is implemented as a self-aligned SSF entity. Thus, keeping a pointer to the community object is important for one who wants to invoke simulation methods of the SSF entity. Each community contains a list of all hosts and routers that belong to the timeline.

Each community in simulation is further associated with a run-time environment, which is represented by the `prime::ssfnet::Milieu` class. The class provides communication support between communities (that possibly running on different processors), so that packets can be delivered correctly across processor and memory boundaries. The `prime::ssfnet::Milieu` class also provides translations between host names, IP address, NHI addresses, and their corresponding network objects instantiated in simulation.

The above simulation objects are all accessible from each protocol session. Recall that from a protocol session one can get the host (or router) using the `getHost()` method. In turn, the community object can be accessed from the host or router, using the `getCommunity()` method of the `prime::ssfnet::Host` class. From community, one can access the alignment service using the `getAlignmentService()` method, the name service using the `getNameService()` method, and its milieu using the `getMilieu()` method.

4.2 Alignment and Name Services

As mentioned in the previous section, the simulator creates an alignment service and a name service object on each processor that participates the simulation. The alignment service (of the `prime::ssfnet::AlignmentService` class) is used to manage the communities (or timelines) that have been assigned to the corresponding processor. The `getCommunity()` method returns a pointer to the community object of the given alignment name (as specified in DML), provided that the community belongs to the current processors. Otherwise, the method return NULL. The `nicnhi2obj()` method can be used to find the corresponding interface object from its NHI address. Similarly, the `hostnhi2obj()` method returns the corresponding host object from its NHI address. It is advised that both methods could be expensive given that they are designed to linearly scan through all communities on the processor to find the corresponding object.

The name service (of the `prime::ssfnet::NameService` class) is a data structure that stores the global names, including alignment names, host names, IP addresses, and NHI addresses, and provides necessary mapping between these names. The `hostname2align()` method converts the name of a host[10] to the name of the alignment (of the community) it belongs to. The `nhi2align()` method converts the NHI address of a host or a network interface to the name of the alignment it belongs to. The `ip2align` method converts an IP address to the name of the alignment.

The `hostname2ip()` method returns the default IP address of a host from its name. Similarly, an NHI address of a network interface can be converted to the corresponding IP address using the `nicnhi2ip()` method. Conversely, an IP address can be converted to the NHI address using the `ip2nicnhi()` method. Furthermore, an NHI address of a host can be converted to the name of the host using the `nhi2hostname()` method, provided it is defined in DML. One can also convert a network's NHI address to an IP prefix using the `netnhi2prefix` method.

[10] A string name can be given to a host or router using the `NAME` attribute in DML.

Although these methods are incomplete to cover all possible name conversions, they can be combined together to provide more functions.

4.3 Community and Milieu

A community is the set of hosts and routers that belong to the same timeline sharing the same alignment name. Each community is implemented as an SSF entity that is aligned to itself only. The `getAlignmentName()` method returns the name of the alignment as specified in DML. The number of hosts and routers defined in the community can be obtained using the `getTotalHost()` method. The list can be obtained via the `getHostBegin()` and `getHostEnd()` methods, both returning the iterators that one can use to scan through the list. In the following example, suppose `comm` is a pointer to the community object:

```
for(Community::HostVectorIterator iter = comm->getHostBegin();
    iter != comm->getHostEnd(); iter++) {
  Host* host = (*iter);
  ...
}
```

The `prime::ssfnet::Milieu` class establishes communications between communities. It is implemented as an SSF process. That is, the `prime::ssfnet::Milieu` class is a subclass of the `prime::ssf::process` class and the `action()` method is overloaded to receive packets sent from network interfaces that belong to hosts in remote communities. The method is responsible for distributing the packets arrived at the input channel to the corresponding local network interfaces.

The `prime::ssfnet::Milieu` class also provides translation services, finding the network objects from the host names, the IP addresses, or NHI addresses. The `nicnhi2obj()` method and the `nicip2obj()` method return the corresponding network interface object form its NHI address and IP address. The `hostname2obj()` method and the `hostnhi2obj()` method return the corresponding host object form its name and its NHI address.

5 Put It Altogether

5.1 Install SSFNet

After PRIME is downloaded and uncompressed, it creates a directory for its own, say `prime`. We call it the PRIME root directory. In addition to SSFNet, the distribution contains SSF (the parallel discrete-event simulator) and other supporting libraries, such as RNG (the random number generator), DML (the domain modeling language parser), and METIS (the graph partitioner for partitioning network models). SSFNet requires all these other packages.

To install SSFNet, one first needs to configure it. This could be done by running the `configure` script under the PRIME root directory. PRIME code uses GNU automake and autoconf tools to customize the code to various platforms. If the PRIME code is obtained directly from the code repository (rather than a public distribution where the `configure` script has already been prepared), one may need to first bootstrap the code. This is achieved by running the `bootstrap`

script under the PRIME root directory. The script requires both automake and autoconf tools be installed properly. They must also be fairly recent versions. If problem occurs, one can try using the other script called `bootify.sh` instead, which uses a static and working copy of the configuration scripts.

There are options to the `configure` script. The configuration at the PRIME root directory will pass on these options to the configuration scripts for DML, RNG, SSF, and SSFNet. By default, all these packages will be included in the installation. A detailed description on the configuration of the SSF simulator can be found in its user's manual (included in the `ssf/doc` directory). For example, one can turn on the debugging support for SSF using the following command:

```
CXXFLAGS="-Wall -g" CFLAGS="-Wall -g" \
    ./configure --enable-ssf-debug
```

By default, PRIME uses the gcc compiler. The above example sets the C/C++ compiler flags to show the warnings against all possible offenses in the code. The `-g` option asks the compiler to produce debugging information within the binary executable so that debugger can work with this debugging information. By turning on debug in SSF (using the `--enable-ssf-debug` option, the simulator will allow run-time debugging information to be printed out on demand. In addition, this option turns on the run-time sanity checking (for example, the `assert` macro will be functional to provide the needed diagnostic tests in the code).

Currently, PRIME SSFNet has only three configuration options: `--enable-ssfnet-debug` allows debugging information for various SSFNet components; `--enable-ssfnet-emulation` enables emulation support; and `--enable-ssfnet-openvpn` enables the simulation gateway using OpenVPN. The last two options are used for emulation, which is described in more detail in section 8.

Once the package is configured properly, one only needs to run `make` to build everything. Separately, if one wants to create the documents, one needs to go to `ssf/doc` and `ssfnet/doc` directories and run `make`. The documents include both PDF user manuals and HTML reference manuals.

In later versions, one should be able to run tests using "make test" to make sure the installation is successful. This is only partially supported in the current version.

5.2 Build and Run a Network Model

We use an example included `ssfnet/test/ping` to describe the entire process of building and running a network model. The first step is to create a model DML (as in `oneping.dml`). The model contains three network nodes connected in a line: one host (alpha), one router (beta), and another host (gamma). All nodes are running ICMP and IP. Host alpha runs an additional application protocol session, called `ping`, that will ping host gamma ten times[11] The `show_drop` option in the IP session is enabled so that every time a packet is dropped in the IP layer, the simulator will printed out a message to inform the user. The additional `start_drop` and `end_drop` attributes in host gamma are used to create an interesting case by intentionally dropping some of the IP packets (which are numbered from 2 to 5) that were to sent out from this host.

[11] There is another way to describe the traffic to be generated by the ping application using the top-level `traffic` attribute.

Next, we need to pre-process the model DML in order to generate necessary information for the simulator to build the model. In particular, we need to assign IP addresses to all network interfaces in the model. This is achieved using the utility program `dmlenv`, which is in the PRIME root directory. The program takes the model DML as input and generates another DML (via the standard output) that includes all necessary information for the simulator to build the model. Optionally, one can also set a network prefix to the IP addresses to be assigned to the network interfaces using the `-b` option. If it is not provided, the `dmlenv` program will use `10.0.0.0` as the network prefix. The following example creates the intermediate DML (`oneping-env.dml`) with IP addresses assigned for the network at Mines:

```
./dmlenv -b 138.67.0.0 oneping.dml > oneping-env.dml
```

The network model does not have any routing protocol. In order for the ICMP packets to get to their destinations, one would need to provide correct forwarding table entries for the IP protocol sessions running on the hosts and routers. SSFNet provides support for calculating static routes during the pre-processing step. Again we use the `dmlenv` program but with an `-r` option. The option is also used to supply a list of NHI addresses of the destinations. The following example creates the forwarding tables with entries to reach hosts with NHI addresses `1` and `2:2`.

```
./dmlenv -r 1,2:2 oneping.dml > oneping-rt.dml
```

Instead of providing the list of NHI addresses as route destinations, one can also use `all` to represent all hosts and routers.

Currently, the static routes are calculated using the shortest-path algorithm. And the cost of the links is the delays. The forwarding table for each host or router can be quite large if the network is large. To manage the size of the forwarding tables, one can limit the number of routable destinations in the list provided using `-r`. *Also one can set a cut-off distance using the* `-d` *option so that hosts with a distance beyond this threshold will not be considered as reachable. Both schemes are not ideal. One alternative would be to use OSPF areas. In addition, the routing should be carried out as hierarchical. Another improvement will be to use a cost other than simply the delay of the link. We also need to consider inter-domain routing that based on policy rather than shortest paths.*

Now we are all set to run the network model. The executable `ssfnet` accepts at least two command-line arguments: the simulation time (in seconds) and a list of DML file names (including those generated above during the pre-processing step).

6 Existing Protocol Implementations

6.1 Simple PHY, Simple MAC, and Network Queues

Not yet finished.

6.2 Internet Protocol (IP)

Not yet finished.

6.3 Internet Control Message Protocol (ICMP)

Not yet finished.

6.4 User Datagram Protocol (UDP)

Not yet finished.

6.5 Transmission Control Protocol (TCP)

Not yet finished.

6.6 Simple Network Management Protocol (SNMP)

Not yet finished.

7 Existing Applications

7.1 Sockets

Not yet finished.

7.2 TCP Applications

Not yet finished.

7.3 UDP Applications

Not yet finished.

7.4 Hypertext Transfer Protocol (HTTP)

Not yet finished.

8 Emulation Support

8.1 The Emulation Infrastructure

The emulation system consists of the following major components:

1. Emulation Session (`prime::ssfnet::EmuSession`) which intercepts virtual packets and exports them to the I/O threads
2. I/O threads (`prime::ssfnet:IOThreads`), which handle connections between PRIME and simulation gateway(s),

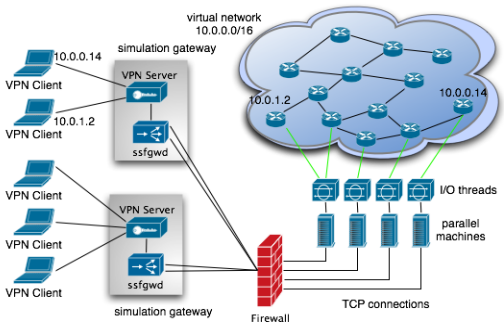

Figure 1: The Emulation Infrastructure

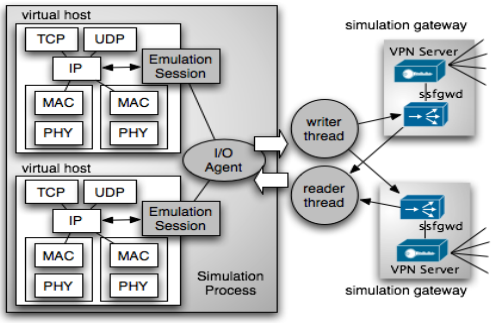

Figure 2: Connecting the Simulator and the Simulation Gateway

3. Simulation gateway
    i. `ssfgwd`, which forwards traffic between the I/O threads and the OpenVPN server,
    ii. OpenVPN Server which manages clients and forwards client traffic to `ssfgwd`
4. OpenVPN Client which captures and injects traffic at the machine hosting an application.

Fig. 1 illustrates how PRIME, the simulation gateways and the OpenVPN clients are connected. The connection between the simulation gateway and PRIME is depicted in greater detail in Fig. 2.

If emulation support has been compiled into PRIME, a network simulation may use emulation by:

1. specifying one or more simulation gateways with the DML file(s) and
2. adding an emulation session protocol to each simulated host's protocol stack.

`vpnscript` is a utility program used to automatically generate the configuration files needed by the OpenVPN servers and clients. The DML file, along with the DML files generated by dmlenv, are used by `vpnscript` to generate the OpenVPN configuration files and scripts to start the clients and server.

The `vpnscript` first creates a certificate authority used by the OpenVPN servers to validate the encrypted incoming client connections. The script then generates a pair of public and private

keys for each OpenVPN server or client. Finally, `vpnscript` creates a compressed archive file that includes all information to start an OpenVPN server or client. For each simulation gateway, the archive includes the private key of the OpenVPN server and the public keys of all OpenVPN clients managed by this simulation gateway (each corresponding to an emulated network interface). Also included in the archive is a mapping from public keys to IP addresses. The mapping is used by the OpenVPN server to statically assign IP addresses to incoming client connections, which are distinguished by the clients' public keys. For each OpenVPN client, the archive includes the private key of the client and the public keys of all designated simulation gateways. For convenience, a script is included in these archives to help start the OpenVPN servers and clients automatically.

8.1.1 Emulation Sessions

Each emulated host in the simulation contains an emulation protocol session that intercepts packets at the IP layer. The `prime::ssfnet::EmuSession` class contains three important properties: *host_mirrored*, *intractability* and *ip_forwarding*.

1. *host_mirrored* is a property which indicates whether traffic should be intercepted for a particular emulated host. *host mirrored* can be accessed via `set_host_mirrored()`.

2. *ip_forwarding* is a property which indicates if the emulation session should intercept packets that the virtual host is forwarding. If *ip_forwarding* is set to false, only the traffic that is destined for the emulated host will be intercepted. If the property is set to true, all traffic that passes through the host will be intercepted. *ip_forwarding* can only be set during protocol initialization.

3. *intractability* defines whether a particular emulation session can interact with the outside world. *intractability* can be accessed via `setInteractability()` and `getInteractability()`.

When a virtual packet arrives at a virtual host it is sent to the IP protocol which then called `pop()` on the associated emulation session protocol. If the emulation session protocol has both the *host_mirrored* and `getInteractability()` set to true, then the emulation protocol will intercept the packet if: 1) the destination address is this virtual host or 2) *ip_forwarding* is set to true. If the packet is not intercepted, it is returned to the IP protocol which forwards the packet to correct upper layer protocol. If the virtual packet is intercepted, it is first translated into a real packet by calling `createEmulationPacket()` on the virtual packet. The real packet is then handed to the *writer* by calling `handle_departure()` on I/O threads associated with timeline the current virtual host is a member of.

Packets from the real host are received by the emulation session when the I/O threads call `external_arrival` on the host for which the packet is destined. As described later in section 8.1.4 special messages are sent from the OpenVPN server to signal the connection of disconnection of a real client. `external_arrival` intercepts those messages and toggles *host_mirrored* accordingly. If the packet is not a special control message, the packet is pushed down to the ip protocol as if it originated higher in the stage.

8.1.2 I/O Threads

I/O threads are composed of a thread for reading in real packets and a thread for writing out virtual packets. The *reader* thread and a `writer` thread are contained in the `prime::ssfnet:IOThreads` class. *reader* threads are responsible for taking real packets from the simulation gateway, converting the real packets into virtual packets, and inserting the resulting virtual packets into the into the correct stack of a virtual host in the simulation. *writer* threads are responsible for taking virtual packets from a emulation session, converting the virtual packets into real packets, and handing the real packets to the correct simulation gateway. *reader* threads, in addition to injecting traffic into the simulation, listen for clients to connect and disconnect from simulation gateways. When a client connects to a simulation gateway, a specific message is sent to the reader thread to notify the emulation session that the corresponding host is connection and traffic should be exported. Likewise, a message a sent when clients disconnect so traffic is not exported. These messages allow the *writer* threads to know which `ssfgwd` to forward specific client traffic to.

During the initialization of each `prime::ssfnet:IOThreads` instance, *reader* thread is started by creating a thread running the `read_message()` function within the class. First, the *reader* thread initiates a TCP connection with each known simulation gateway's `ssfgwd` process. After each connection, a one byte identifier to sent to the `ssfgwd` to identify the connection as a *reader*. After the identifier is sent, a list of ip addresses is sent to the gateway. The list of addresses, sent by the function `send_ip_addresses()`, tells the gateway that this *reader* is responsible for the addresses in the list. After each connection has been established and setup by the *reader* thread, `read_message()` goes into a servicing loop where it waits for real packets from the gateway. When a real packet is received, the *reader* thread first reads the leading ip address, which is the address of the virtual host the following packet will be inserted into. This address is used to specify which virtual host to inject the packet into (see section 8.1.7). After the leading address is read, the rest of the packet is read into a buffer. The packet is then turned in a `prime::ssfnet::EmuEvent` and inserted in the *readers* local channel. The local channel is serviced by the function `handle_arrival()` (in the I/O threads class), which converts the `prime::ssfnet::EmuEvent` into a `prime::ssfnet::EmuMessage` by passing the event into the `prime::ssfnet::EmuMessage` as an argument to the constructor. `handle_arrival()` then injects the virtual packet(`prime::ssfnet::EmuMessage`) into the appropriate virtual host's stack by calling `external_arrival()` on the appropriate virtual host's emulation session. The `external_arrival()` function is where the special control messages from the OpenVPN server are caught and processed causing the *host_mirrored* flag to be toggled appropriately.

Likewise, the *writer* thread is started running the `write_message()` function with the `prime::ssfnet:IOThreads` class. First, the *reader* thread initiates a TCP connection with each known simulation gateway's `ssfgwd` process. After each connection, a one byte identifier to sent to the `ssfgwd` to identify the connection as a *writer*. After the identifier is sent, the *writer* goes into a servicing loop where it continually gets outbound packets (actually `prime::ssfnet::EmuEvent` instances) from its local channel. The *writer* thread determines which TCP connection to send the packet on. Before sending the packet, the *writer* first sends the ip address from which the packet originated. The address is used by `ssfgwd` to route the packet to correct client (see sec-

tion 8.1.7). Packets are inserted into the *writer*'s local channel when a emulation session by the `handle_departure` function.

In addition to everything going on above, code to measure the delay between the `ssfgwd` and the I/O threads has been inserted. Currently this code is not used, and such has not been described here.

8.1.3 `ssfgwd`

The ssfnet simulation gateway daemon (`ssfgwd`) is a process, run separate from PRIME, most often run as a daemon process (hence the name). After the simulation is started each *reader* and *writer* thread (a pair for each timeline within the simulation) will connect to each simulation gateway. These connections are maintained by `ssfgwd`.

As described in 8.1.2, *reader threads* send a list of IP address of which the thread is responsible. The simulation gateway records this list of IP addresses and creates a mapping from the IP addresses to the corresponding *reader* thread connection. This mapping is later used to forward traffic from the client machines (via the OpenVPN server) to the correct *reader* thread.

Traffic received from *writer* threads is forwarded to the OpenVPN server which then forwards the packets along to the correct client. Traffic from OpenVPN is sent to the correct reader thread using the previously described map.

Packets from the simulator are expected to pre-pended with the IP address of the host interface that intercepted the virtual packet. Packets from the OpenVPN server are expected to have the address of the source client pre-pended. These pre-pended address are used for routing the packets to and from the correct clients and virutal hosts. See section 8.1.7 for more details.

There are many command line options for `ssfgwd`. The daemon has functionality to help test and debug the connection to PRIME and the connection to OpenVPN. Additionally, `ssfgwd` is capable of using standard IP tunnel interfaces instead of pipes to communicate with OpenVPN (or a similar process).

8.1.4 OpenVPN

OpenVPN was chosen to allow applications to dynamically connect to the simulation gateway and to emulate network interfaces on the client machines. On each simulation gateway a modified OpenVPN server. The modified server is configured at run–time using an OpenVPN server configuration file automatically generated from the virtual network specification (DML). Each OpenVPN client connects to a chosen simulation gateway using another client configuration file, also generated automatically beforehand. Each client will have an IP address assigned to the VPN interface that matches the IP address of the virtual host's interface in the simulation so that client machines can assume the proper identity as applications running on this machine can transparently forward traffic to the simulated network. See section 8 for information on the generation of the configuration files and IP addresses.

8.1.5 OpenVPN Server

The OpenVPN server source code was modified in four ways. First, we modified the server to spawn the `ssfgwd` process and listen for packets from `ssfgwd` over a standard pipe. Second, a plug–in was added to send notification messages when clients connect and disconnect. These notifications are described in 8.1.1. Third, we modified the server to pre–pend all packets from its clients with the IP address of the client where the packet originated. This pre–pended address is used by `ssfgwd` to forward the packet to the correct *reader* thread. Finally, we modified the server to route packets using a pre–pended IP address rather than the destination address found in the IP header. The last two modifications are necessary to allow emulation of routers, switches, and the like. See 8.1.7 for more details.

8.1.6 OpenVPN Client

The OpenVPN client source code is unmodified so any standard OpenVPN client can be used to connect to our modified OpenVPN Sever. When a OpenVPN client connects to a OpenVPN server, a virtual (IP) interface is added to the system. This interface is, as far as applications are cornered, a real interface. When an application targets an IP address in simulation, it is routed out this virtual interface allowing the application to interact with the simulation the same was as it interacts with the real world.

8.1.7 Emulation of Routers, Switches, etc

Recall that an OpenVPN client creates a virtual interface which has the IP address of the virtual host it is emulating. More than one OpenVPN client can be used on the same machine. Every OpenVPN client on a machine will create a virtual interface. To emulated a host with more than IP address (i.e. more than one interface) an OpenVPN client for each IP address is needed.

If the host was simply an end host, the pre–pended addresses (and all the supporting modifications) used to route packets to and from the correct VPN client/virtual host pair would not be necessary. Instead, `ssfgwd` could run independently from the VPN server and they could communicate using a standard IP tunnel devices. For example, immediately upon start-up, `ssfgwd` could open a tunnel device, activate and configure the device by assigning a special IP address to it, and then modify the kernel routing table to connect the tunnel device with the one created by the VPN server. The kernel routing table could be set up so that packets arriving from the VPN server's tunnel device be forwarded by the kernel to `ssfgwd`'s tunnel device and similarly packets written by `ssfgwd` to its tunnel device be forwarded to the VPN server, which are then delivered to the corresponding clients. Packets arrived at the simulation gateway from a client machine could be sent to the simulation process and the corresponding virtual host according their source IP addresses. Similarly, packets arrived at the simulation gateway from the simulator could be distinguished by their destination IP addresses and sent to the corresponding client machines via the corresponding VPN connections.

This scheme, however, does not work if one wants to emulate a router with multiple network interfaces (e.g., to test a real routing protocol implementation). Consider an example shown in Fig. 3, where we emulate the router B in a virtual network of three nodes. Since router B has two network

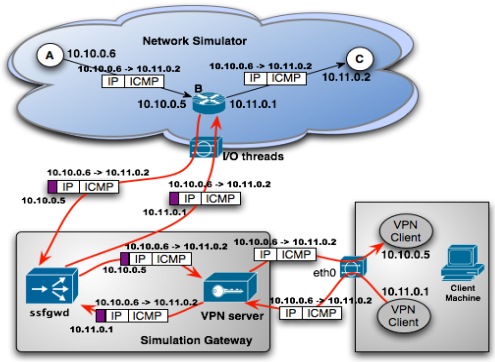

Figure 3: Emulation of a Router with Multiple Network Interfaces

interfaces with IP addresses 10.10.0.5 and 10.11.0.1, respectively, we run two separate VPN clients on the client machine. Suppose that node A pings node C. An IP packet with a source address of 10.10.0.6, a destination address of 10.11.0.2, and a payload of an ICMP `ECHO-REQUEST` message reaches the network interface 10.10.0.5 and is then exported to the simulation gateway as expected. Since the packet carries a source address of 10.0.0.2, which is not emulated, the `ssfgwd` process will have trouble forwarding the packet to the client machine.

To overcome this problem, each packet sent between the network simulator and the simulation gateway is pre-pended by the IP address from which the intercepted or is to be injected.

8.2 Data Conversion

8.2.1 Virtual to Real Packet Translation

When a virtual packet (`prime:ssfnet:ProtocolMessage`) enters the emulation session and it is determined that the packet should be exported, the emulation session will cast the `prime:ssfnet:Prot` as an `prime:ssfnet::IPMessage` instance and call `createEmulationPacket()` on it. `createEmulationPacket()` eventually calls `toRealBytes()` on the IPMessage.

8.2.2 Real to Virtual Packet Translation

When a real packet (is converted to a `prime::ssfnet::EmuMessage` the`prime::ssfnet::EmuMessag` simply takes ownership of the raw byte buffer. The parsing of the raw buffer is postponed until `detach_payload()` is called by the emulation when the packet is inserted into a virtual stack. `detach_payload()` will create an appropriate ProtocolMessge: `prime:ssfnet::ICMPMessage` for icmp, `prime:ssfnet::UDPMessage` for udp, `prime:ssfnet::TCPMessage` for tcp (that is all that is currently supported). The ProtocolMessage converts the raw bytes into the appropriate data structure by called `fromRealBytes()` on itself and passing in the buffer.

8.3 Set Up a Network Emulation

To setup a network emulation PRIME must be compiled with emulation support turned on and OpenVPN support turned on. Emulation support is turned on during build configuration with the option `--enable-ssfnet-emulation=yes`

When setting up a network model (DML), if emulation is going to be used, there must be a `Emulation` block. With the emulation block you specify one or more simulation gateways. A gateway simulation consists of a name, an ip address or name, and a port. The following DML snippet is an emulation block with one simulation gateway:

```
 Emulation [
           gatewayOne [
                  ip myGatewayName.mydomain
                  port 1000
           ]
]
```

To specify more than one simulation simply add another simulation gateway block as seen in the following snippet:

```
 Emulation [
           gatewayOne [
                  ip myGatewayName.aDomain
                  port 1000
           ]
           gatewayTwo [
                  ip anotherGatewayName.aDomain
                  port 1100
           ]
]
```

In the previous snippet the port changed between the two simuation gateways. This is not necessary. However, it is neccesary for the address–port pair to be unique for each gateway.

In addition to the `Emulation` block, a `EmuSession` protocol session must be inserted into the protocol graph specification of every virtual host that is to be emulated. The following snippet shows the specification of a virtual host with one interface, supports IP and ICMP, and in addition is also emulated:

```
host [ id 1
  graph [
    ProtocolSession [ name emu use "SSF.OS.Emulation.EmuSession" ]
    ProtocolSession [ name icmp use "SSF.OS.ICMP.ICMPSession" ]
    ProtocolSession [ name ip  use "SSF.OS.IP" ]
  ]
  interface [ id 0 _extends .dict.iface ]
]
```

To bring it together, lets look at a simple DML example that builds on the previous snippets. The following DML specifies a network with two hosts. One host will function as a simple HTTP server and the other host will be emulated. Other than the addition of the `Emulation` block and `EmuSession` in the protocol graph, the DML file does not need to be changed. See section 2 for details on the remainder of the DML.

```
Emulation [
 gateway [ip roar.mines.edu port 1000]
]
Net [
 host [ id 1
  graph [
    ProtocolSession [ name emu use "SSF.OS.Emulation.EmuSession" ]
    ProtocolSession [ name icmp use "SSF.OS.ICMP.ICMPSession" ]
    ProtocolSession [ name ip  use "SSF.OS.IP" ]
  ]
  interface [ id 0 _extends .dict.iface ]
 ]
 host [ id 2
  graph [
    ProtocolSession [ name app use "SSF.OS.APP.SimpleHTTPServer" ]
    ProtocolSession [ name socket use "SSF.OS.Socket.blockingSocketMaster" ]
    ProtocolSession [ name tcp use "SSF.OS.TCP.tcpSessionMaster"
                      _find .dict.tcpinit ]
    ProtocolSession [ name icmp use "SSF.OS.ICMP.ICMPSession" ]
    ProtocolSession [ name ip  use "SSF.OS.IP" ]
  ]
  interface [ id 0 _extends .dict.iface ]
 ]
 link [ attach 1(0) attach 2(0) delay 0.2 ]
]
dict [
  iface [ bitrate 1e8 latency 0 ]
  tcpinit[
    ISS 10000               # initial sequence number
    MSS 960                 # maximum segment size
    RcvWndSize  10000       # receive buffer size
    SendWndSize 10000       # maximum send window size
    SendBufferSize 10000    # send buffer size
    MaxRexmitTimes 12       # maximum retransmission times before drop
    TCP_SLOW_INTERVAL 0.5   # granularity of TCP slow timer
    TCP_FAST_INTERVAL 0.2   # granularity of TCP fast(delay-ack) timer
    MSL 60.0                # maximum segment lifetime
    MaxIdleTime 600.0       # maximum idle time for drop a connection
    delayed_ack false       # delayed ack option
```

```
    version reno            # tcp version
  ]
]
```

Just like a DML without emulation `dmlenv` must be run on the DML file. Assume the above DML file is saved as `simple.dml`. The following series of commands will produce all the necessary files needed to configure and run `ssfgwd`, OpenVPN server, and a OpenVPN client.[12]

```
$SSFNET_DIR/dmlenv -b 10.10.0.0 simple.dml > simple-env.dml
$SSFNET_DIR/dmlenv -r all simple.dml > simple-rt.dml
$SSFNET_DIR/dmlenv -e -b 10.10.0.0 simple.dml > simple.vpnconf
perl $SSFNET_DIR/src/emuproxy/vpnsetup/vpnscript.pl -d
   $SSFNET_DIR/src/emuproxy/openvpn -s 10.8.0.0 simple.vpnconf
perl $SSFNET_DIR/src/emuproxy/vpnsetup/vpnscript.pl --clean
```

The first and second commands generate configuration information needed by prime (see section 2). The additional command line parameter, `-e`, tells `dmlenv` to generate the vpn configuration information which we stored in `simple.vpnconf`. The third command generates a configuration file that is used by `vpnscript` to create configuration files for the OpenVPN server and clients. The fourth command calls the vpn setup script which creates all the necessary configuration files.

Going back to the third command, Notice `10.8.0.0` is different from `10.10.0.0`, the address used in previous commands. `-s 10.8.0.0` configures the VPN server to allocate vpn resources on the `10.8.0.0` subnet. The OpenVPN server will have an IP address in this address space. Any vpn client that connects to the server will have access to both the `10.8.0.0` and `10.10.0.0` address spaces. The last command simply cleans up temporary files generated by the previous command. [13]

The previous four commands generates `simple-rt.dml`, `simple-env.dml`, `simple.vpnconf` `10.10.0.1.tgz`, and `roar.mines.edu.tgz`, in addition to `simple.dml`. See section 2 for discussion of files ending with '.dml'. `simple.vpnconf` contains the following text:

```
server roar.mines.edu 1000 36685
client 10.10.0.1
subnet 10.10.0.0 255.255.255.248
```

The first line specifies information about the simulation gateway. `roar.mines.edu` is the address of the machine hosting `ssfgwd` and the associated OpenVPN server. `1000` is the port on which `ssfgwd` will listen for connections from PRIME, and `36685` is the port on which OpenVPN will listen for connections from OpenVPN clients. This line would exist for every simulation gateway in the `Emulation` block specified in the original dml. The second line states that there is one client whose address should be `10.0.0.1` – there would be an entry like this for every

[12] It is assumed that the local environment path includes the `SSFNET_DIR` variable set correctly.

[13] Run `vpnscript` with `--help` as the command line parameter to find details on the supported command line options of `vpnscript`.

virtual host that is specified. The third line specifies what address range should be routed into the simulation from the vpn clients.

The remaining two files, `roar.mines.edu.tgz` and `10.10.0.1.tgz`, are compressed archives which contain the necessary files to start a OpenVPN server and client respectively.

Move `roar.mines.edu.tgz` and a copy of the OpenVPN server (found where PRIME install the ssfnet binary) to the machine that will host the simulation gateway. In this case, move the file to `roar.mines.edu` and unpack the archive. The archive will extract its contents into `./openvpn_server_roar.mines.edu`. In that directory is a shell script (`runserver.sh`) that will start `ssfgwd` and the OpenVPN server. Run the script as root and the simulation gateway is ready to accept connections from PRIME and the vpn client(s)![14]

Move `10.10.0.1.tgz` to any machine with a valid OpenVPN Client and unpack it. `10.10.0.1.tgz` will extract its contents into `./openvpn_client_10.10.0.1`. In that directory is a shell script (`runclient.sh`) that will start the client. If a virtual host has multiple interfaces to be emulated, one archive is still made for the client and `runclient.sh` starts all the necessary vpn clients. Run the script as root and the client host is ready able to send and exchange traffic with a PRIME simulation!

9 Advanced Topics

9.1 The Fluid Model

Not yet finished.

10 Caveats

10.1 Using C++ Standard Template Library (STL)

Not yet finished.

[14] It is necessary to start all simulation gateway(s) *before* any vpn client is started and *before* PRIME is started. This is necessary because the I/O threads and vpn clients are contacting the gateway and if it is not up, the will fail to start correctly!.