2. The *ns–3* Network Simulator

George F. Riley (Georgia Tech)
Thomas R. Henderson (University of Washington, and Boeing Research & Technology)

2.1 Introduction

As networks of computing devices grow larger and more complex, the need for highly accurate and scalable network simulation technologies becomes critical. Despite the emergence of large-scale testbeds for network research, simulation still plays a vital role in terms of *scalability* (both in size and in experimental speed), *reproducibility*, *rapid prototyping*, and *education*. With simulation based studies, the approach can be studied in detail at varying scales, with varying data applications, varying field conditions, and will result in reproducible and analyzable results.

For many years, the venerable *ns–2* network simulation tool[81] was the de–facto standard for academic research into networking protocols and communications methods. Countless research papers were written reporting results obtained using *ns–2*, and hundreds of new models were written and contributed to the *ns–2* code base. Despite this popularity, and despite the large number of alternative network simulators documented later in this book and elsewhere, the authors and other researchers undertook a project in 2005 to design a new network simulator to replace *ns–2* for networking research. Why create a new tool? As the Introduction to this book states, this book is about *how to model* network stacks, and the decision to develop a new tool was motivated by a particular view of how to model networks in a manner that best suits network research, and by the authors' collective experiences in using and maintaining predecessor tools. As this tool was designed to replace *ns–2*, the name chosen for this tool was *ns–3* (http://www.nsnam.org).

One of the fundamental goals in the *ns–3* design was to improve the *realism* of the models; i.e., to make the models closer in implementation to the actual software implementations that they represent. Different simulation tools have taken different approaches to modeling, including the use of modeling-specific languages and code generation tools, and the use of component-based programming paradigms. While high-level modeling languages and simulation-specific programming paradigms have certain advantages, modeling actual implementations is not typically one of their strengths. In the authors' experience, the higher level of abstraction can cause simulation results to diverge too much from experimental results, and therefore an emphasis was placed on realism. For example, *ns–3* chose C++ as the

programming language in part because it better facilitated the inclusion of C-based implementation code. *ns–3* also is architected similar to Linux computers, with internal interfaces (network to device driver) and application interfaces (sockets) that map well to how computers are built today. As we describe later, *ns–3* also emphasizes emulation capabilities that allow *ns–3* to be used on testbeds and with real devices and applications, again with the goal of reducing the possible discontinuities when moving from simulation to experiment.

Another benefit of realism is *reuse*. *ns–3* is not purely a new simulator but a synthesis of several predecessor tools, including *ns–2* itself (random number generators, selected wireless and error models, routing protocols), the Georgia Tech Network Simulator (GTNetS)[393], and the YANS simulator[271]. The software that automates the construction of network routing tables for static topologies was ported from the quagga routing suite. *ns–3* also prioritizes the use of standard input and output file formats so that external tools (such as packet trace analyzers) can be used. Users are also able to link external libraries such as the GNU Scientific Library or IT++.

A third emphasis has been on *ease of debugging* and better alignment with current languages. Architecturally, this led the *ns–3* team away from *ns–2*'s mixture of object-oriented Tcl and C++, which was hard to debug and was unfamiliar (Tcl) to most students. Instead, the design chosen was to emphasize purely C++-based models for performance and ease of debugging, and to provide a Python-based scripting API that allows *ns–3* to be integrated with other Python-based environments or programming models. Users of *ns–3* are free to write their simulations as either C++ main() programs or Python programs. *ns–3*'s low-level API is oriented towards the power-user but more accessible "helper" APIs are overlaid on top of the low-level API.

Finally, *ns–3* is not a commercially-supported tool, and there are limited resources to perform long-term maintenance of an ever-growing codebase. Therefore, software *maintenance* was a key design issue. Two problems with *ns–2* led the *ns–3* team, after careful consideration, to abandon the goal of backward compatibility with or extension of *ns–2*. First, *ns–2* did not enforce a coding standard, and accepted models with inconsistent software testing and model verification, as well as a lack of overall system design considerations. This policy allowed the tool to grow considerably over time but ultimately led users to lose confidence in the results, made the software less flexible to reconfiguration, and created challenges and disincentives for maintainers to maintain software once the personnel maintaining the simulator changed. *ns–3* elected to prioritize the use of a single programming language while exporting bindings to Python and potentially other scripting languages in the future. A more rigorous coding standard, code review process, and test infrastructure has been put into place. It would have been possible to build *ns–3* with full backward compatibility at the Tcl scripting level, but the *ns–3* project does not have the resources to maintain such a

backward-compatibility layer. Therefore, the decision was made to create a new simulator by porting the pieces of *ns–2* that could be reused without compromising the long-term maintainability of *ns–3*.

With significant backing from the U.S. National Science Foundation, INRIA and the French government, the Georgia Institute of Technology, the University of Washington, and Google's Summer of Code program, *ns–3* has also been operated as a free, open source software project from the onset, and has accepted contributions from over forty contributors at the time of this writing. The remainder of this chapter will further describe a number of the design decisions that were incorporated into *ns–3*, and gives some simple examples of actual simulations created using *ns–3*.

2.2 Modeling the Network Elements in *ns–3*

As in virtually all network simulation tools, the *ns–3* simulator has models for all of the various network elements that comprise a computer network. In particular there are models for:

1. Network *nodes*, which represent both end–systems such as desktop computers and laptops, as well as network routers, hubs and switches.
2. Network *devices* which represent the physical device that connects a node to communications channel. This might be a simple Ethernet network interface card, or a more complex wireless IEEE 802.11 device.
3. Communications *channels* which represent the medium used to send the information between network devices. These might be fiber–optic point–to–point links, shared broadcast–based media such as Ethernet, or the wireless spectrum used for wireless communications.
4. Communications *protocols*, which model the implementation of protocol descriptions found in the various Internet *Request for Comments* documents, as well as newer experimental protocols not yet standardized. These protocol objects typically are organized into a *protocol stack* where each *layer* in the stack performs some specific and limited function on network packets, and then passes the packet to another layer for additional processing.
5. Protocol *headers* which are subsets of the data found in network packets, and have specific formats for each of the protocol objects they are associated with. For example, the IPv4 protocol described in RFC760 has a specified layout for the protocol header associated with IPv4. Most protocols have a well-defined format for storing the information related to that protocol in network packets.
6. Network *packets* are the fundamental unit of information exchange in computer networks. Nearly always a network packet contains one or more protocol headers describing the information needed by the protocol implementation at the endpoints and various hops along the way. Further,

the packets typically contain *payload* which represents the actual data (such as the web page being retrieved) being sent between end systems. It is not uncommon for packets to have no payload however, such as packets containing only header information about sequence numbers and window sizes for reliable transport protocols.

In addition to the models for the network elements mentioned above, *ns–3* has a number of helper objects that assist in the execution and analysis of the simulation, but are not directly modeled in the simulation. These are:

1. Random variables can be created and sampled to add the necessary randomness in the simulation. For example, the behavior of a web browser model is controlled by a number of random variables specifying distributions for think time, request object size, response object size, and objects per web page. Further, various well–known distributions are provided, including uniform, normal, exponential, Pareto, and Weibull.
2. Trace objects facilitate the logging of performance data during the execution of the simulation, that can be used for later performance analysis. Trace objects can be connect to nearly any of the other network element models, and can create the trace information in several different formats. A popular trace format in *ns–3* is the well known *packet capture* log, known as *pcap*. These *pcap* traces can then be visualized and analyzed using one of several analysis tools designed for analyzing actual network traces, such as *WireShark*. Alternately, a simple text–based format can be used that writes in human readable format the various information about the flow of packets in the simulated network.
3. Helper objects are designed to assist with and hide some of the details for various actions needed to create and execute an *ns–3* simulation. For example, the *Point to Point Helper* (described later in this chapter) provides an easy method to create a point–to–point network.
4. Attributes are used to configure most of the network element models with a reasonable set of default values (such as the initial time–to–live TTL value specified when a new IPv4 packet is created). These default values are easily changed either by specifying new values on the command line when running the *ns–3* simulation, or by calling specific API functions in the default value objects.

2.3 Simulating a Computer Network in *ns–3*

The *ns–3* simulator is developed and distributed completely in the *C++* programming language.[1] To construct a simulation using *ns–3*, the user writes a *C++* main program that constructs the various elements needed to describe

[1] The distribution does include some *Python bindings* for most of the publicly available API.

the communication network being simulated and the network activity desired for that network. The program is then compiled, and linked with the library of network models distributed with *ns–3*.

In writing the *C++* simulation program, there are four basic steps to perform:

1. Create the *network topology*. This consists of instantiating *C++* objects for the nodes, devices, channels, and network protocols that are being modeled in the simulation.
2. Create the *data demand* on the network. This consist of creating simulation models of various network applications that send and receive information from a network, and cause packets to be either created or accepted and processed.
3. Execute the simulation. Typically, this results in the simulator entering the *main event loop*, which reads and removes events in timestamp order from the sorted event data structure described earlier. This process repeats continually until either the event list becomes empty, or a predetermined *stop time* has been reached.
4. Analyze the results. This is typically done by post-analysis of the trace information produced by the *ns–3* program execution. The trace files will usually have enough information to compute average link utilization on the communication channels in the simulation, average queue sizes at the various queue, and drop rate in the queues, just to name a few. Using the optional *pcap* trace format, any of the various publicly available tool for analyzing *pcap* traces can be used.

To illustrate these steps in the context of *ns–3*, we next discuss an actual, albeit quite simple, *ns–3* simulation program in detail. The script in question is a simple two–node point–to–point network that sends one packet from node zero to node one. The *C++* program is shown in listing 2-1.

1. The use of a `NodeContainer` helper is shown in lines 5 – 6. The `Create` method for the `NodeContainer` object is used to construct exactly two network nodes. In this particular example, the two nodes both represent end systems, with one creating and transmitting a packet and the second receiving the packet.
2. The use of a `PointToPointHelper` is shown in lines 8 – 10. The `SetDeviceAttribute` method in the helper illustrates the use of the attribute system, and the ability to override the default values for most configuration items with the values desired for the particular simulation execution. Then the `Install` method is called at line 12, passing in the `NodeContainer` object, and returning a new `NetDeviceContainer` which contains the network devices that were created when installing the point–to–point network connecting the two nodes.
3. The use of the `InternetStackHelper` is shown in lines 14 – 15. The `Install` method in the `InternetStackHelper` is called at line 15, which

```
// Simple ns3 simulation with two node point to point network

int main (int argc, char** argv)
{
  NodeContainer nodes;
  nodes.Create (2);

  PointToPointHelper pointToPoint;
  pointToPoint.SetDeviceAttribute ("DataRate", StringValue ("5Mbps"));
  pointToPoint.SetChannelAttribute ("Delay", StringValue ("2ms"));

  NetDeviceContainer devices = pointToPoint.Install (nodes);

  InternetStackHelper stack;
  stack.Install (nodes);

  Ipv4AddressHelper address;
  address.SetBase ("10.1.1.0", "255.255.255.0");

  Ipv4InterfaceContainer interfaces = address.Assign (devices);

  UdpEchoServerHelper echoServer (9);

  ApplicationContainer serverApps = echoServer.Install (nodes.Get (1));
  serverApps.Start (Seconds (1.0));
  serverApps.Stop (Seconds (10.0));

  UdpEchoClientHelper echoClient (interfaces.GetAddress (1), 9);
  echoClient.SetAttribute ("MaxPackets", UintegerValue (1));
  echoClient.SetAttribute ("Interval", TimeValue (Seconds (1.)));
  echoClient.SetAttribute ("PacketSize", UintegerValue (1024));

  ApplicationContainer clientApps = echoClient.Install (nodes.Get (0));
  clientApps.Start (Seconds (2.0));
  clientApps.Stop (Seconds (10.0));

  Simulator::Run ();
  Simulator::Destroy ();
}
```

Program 2-1 first.cc

adds the protocol objects normally associated with a typical Internet protocol stack, including Address Resolution Protocol (ARP), Internet Protocol (IPv4), User Datagram Protocol (UDP), and Transmission Control Protocol (TCP).

4. Next, the use of the `Ipv4AddressHelper` is shown in lines 17 – 20. The `SetBase` call at line 18 specifies the network address and the network mask for the sequential *IPv4* addresses to be assigned. The `Assign` call at line 20 assigns sequential addresses to each of the network devices in the `devices` container, and returns a new `Ipv4InterfaceContainer` which holds all of the *IPv4* software interfaces created by the address assignment. Note that in *ns–3*, as in actual networks, Internet addresses are assigned to network layer interface instances, rather than to network nodes.
5. Lines 22 – 35 illustrate the use of several application helper objects. The `UdpEchoServerHelper` constructor at line 22 creates the helper and specifies the port number on which the *UDP Echo* application will listen for packets (9 in this case). Next, the `Install` method is called on the `UdpEchoServerHelper` object, which creates an instance of the *UDP Echo Server* on the specified node. Note the use of the call to the `Get` method on the `NodeContainer`. This returns a pointer to the node at index one in the node container object. The `Install` method returns a container of echo server applications. In this particular example, there is only one application in the container since we only passed a single node object pointer to the `Install` method. Had we passed a container object, the echo server application would have been installed on every node in the container, and the returned `ApplicationContainer` would contain a pointer to the application at each node.
6. The use of the `Start` and `Stop` calls in the `ApplicationContainer` object is illustrated in lines 25 and 26. The times specified represent the time the application is to begin processing (the *Start time*) and when it is to stop processing (the *Stop* time).
7. Lines 28 – 35 illustrate several more features of the application helpers, including the constructor at line 28 that initializes the destination address and port for the echo data, and the `SetAttribute` methods that again override default values as we saw above.
8. The echo client application is finally installed on a single node (node zero in the `NodeContainer object`), and start/stop times are specified in lines 33 – 35.
9. Lastly, the `Simulator` method `Run` is called at line 37, which causes the simulation be start executing simulated events. In this particular example, the echo client only sends one packet and receives one reply, after which there are no more pending events and the simulation terminates and the `Run` method returns to the caller.

10. The `Destroy` method is called explicitly to allow all objects in the *ns–3* environment to exit cleanly and return all allocated memory. This call is not strictly necessary in order to obtain correct simulation results, but does allow thorough memory leak checking to be done.

2.4 Smart Pointers in *ns–3*

```
1  // Code snippet from the ns-3 OnOff application
2  void OnOffApplication::SendPacket()
3  {
4    NS_LOG_FUNCTION_NOARGS ();
5    NS_LOG_LOGIC ("sending packet at " << Simulator::Now());
6    NS_ASSERT (m_sendEvent.IsExpired ());
7    Ptr<Packet> packet = Create<Packet> (m_pktSize);
8    m_txTrace (packet);
9    m_socket->Send (packet);
10   m_totBytes += m_pktSize;
11   m_lastStartTime = Simulator::Now();
12   m_residualBits = 0;
13   ScheduleNextTx();
14 }
```

Program 2-2 onoff-app.cc

All network simulation tools written in the *C++* programming languages make extensive use of *dynamic memory*, utilizing the built-in `new` and `delete` operators to allocate and free memory as needed by the application. However, when allocating memory in this way, a common occurrence is a *memory leak*. A leak results when some memory is allocated with the `new` operator, but due to a programming mistake the memory is never returned. Memory leaks are prevalent in *C++* programs, and sometimes result in long–running programs aborting due to apparent memory exhaustion. Additionally, memory mismanagement by referring to memory that has already been freed often results in erroneous behavior or fatal crashes in program execution.

The *ns–3* simulator makes extensive use of *smart pointers* to help alleviate these concerns. When using smart pointers, the simulation models do not call `new` and `delete` directly. Rather, they call a special templated `Create` method that both allocates the requested memory and increments a special *reference count* value associated with the allocated memory. Whenever a smart pointer is copied (for example when passing the pointer to another function by value), the associated reference counter is incremented, indicating there are additional pointer variables pointing to the underlying memory region. Whenever an instance of the smart pointer goes out of scope (resulting in a call to the destructor for the smart pointer object), the reference count is

decremented. If the value decrements to zero, then all outstanding references to the memory have been destroyed, and at that time the underlying memory will be returned with the `delete` operator. This approach greatly eases the programming of handling dynamic memory, at the expense of some minor overhead during object creation, copying, and deletion, and the avoidance of reference cycles among the pointers.

The code snippet in listing 2-2 illustrates the use of a smart pointer to manage an *ns–3* `Packet` object. At line 7 an object of type `Ptr<Packet>` is defined and created by a call to the global static function `Create`. The type `Ptr<Packet>` is an object with several member variables, but primarily consists of the pointer to the actual memory of the `Packet` object, and a pointer to a shared reference counter. The `Ptr` objects have all of the semantics of pointer variables, and for all intents and purposes can be considered to be pointers. Also in this example, note that the `Create` function has an argument `m_pktSize`. During the `Packet` object creation, the argument is passed to the constructor for the `Packet` object. This allows large packet payloads to be modeled without actually requiring that memory be allocated for the simulated payload. In *ns–3*, Packet objects may also carry real data buffers if the application requires them to do so.

Later, at line 9, the packet smart pointer object "`packet`" is passed as a parameter to the `Send` method for the associated socket. It is likely that the packet will eventually be forwarded to another *ns–3* `Node` object by scheduling a future event. However, since the smart pointer was passed by *value* to the `Send` method, the reference counter has been incremented to two, indicating that two different packet pointers to the same packet are in existence. When the `SendPacket` function exits at line 14, the reference counter is decremented to one. Since it is still non–zero, the underlying packet data is not freed. When the packet eventually reaches the final destination, the reference count will then become zero and the packet data will be freed and returned to the available pool.

Readers familiar with the sockets application programming interface (API) may be surprised to see packets being passed at the application/socket boundary. However, the *ns–3* Packet object at this layer of the simulation can be simply considered to be a fancy byte buffer. *ns–3* also supports a variant of the Send method that takes a traditional byte buffer as an argument.

2.5 Representing Packets in *ns–3*

A fundamental requirement for all network simulation tools is the ability to represent network packets, including both the packet payload and the set of protocol headers associated with the packet. Further, it is important to allow for actual payload data in the packets in the case where the payload is meaningful (such as routing table updates in routing protocols), or to simply

represent the existence of the payload but not actual contents (such as when measuring the behavior of a transport protocol that is data agnostic). Further, the design must support the presence of any number of protocol headers of any sizes. Finally, the design should support fragmentation and reassembly by network or link layer protocol models.

The design of *ns–3* allows for all of these capabilities. The size of the so–called *dummy payload* can be specified on the object constructor for the `Packet` object. If so, the payload size is simply stored as an integer, but no actual payload is represented.

Then, any object that is a subclass of the class `Header` can be added to the packet using the `AddHeader` method for packets, and any object that is a subclass of the class `Trailer` can be added at the end of the packet data using the `AddTrailer` method. Removing headers or trailers can easily be accomplished using the defined `RemoveHeader` and `RemoveTrailer` functions.

Another feature of *ns–3* packets is the inclusion of *copy on write* semantics for packet pointers. Consider the simple example discussed earlier in listing 2-2. At line 9 the newly created packet is passed as a parameter to the socket `Send` function. This will undoubtedly result in the addition of several protocol headers to the packet as it progresses down the protocol stack. However, semantically, passing an object by value, as is done here, should not result in any changes to the objects passed as arguments. In the *ns–3* design this is accomplished by the implementation of *copy on write*. In this design, the actual packet data and all protocol headers are stored in a separate helper object called a `Buffer`. Any time a `Buffer` associated with a packet is modified and holders of pointers to the packet need to access the different views of the buffer contents, the original buffer is replicated and the original packet buffer pointer points to the original buffer. The replicated packet pointer object gets pointed to the newly revised buffer, so that two packet pointers for the same logical packet in fact see two different representation of the packet data. This is implemented in an efficient manner that avoids actual data copying as much as possible.

2.6 Object Aggregation in *ns–3*

In a program design intended to be continually modified and enhanced by a number of users over a long period of time, it becomes important to allow flexibility in design, while at the same time having efficiency in memory usage and simplicity in object class implementation. For example, *ns–3* defines a `Node` object to that is a model of a network end system or router. However, not all `Node` objects have the same requirements. For example, some may want an implementation of IP version 6 (IPv6) and others may not. Some may need an indication of physical location while others may not. Some may

need instances of specialized routing protocols, but others may not. Clearly, the notion of a *one size fits all* design for `Node` objects is not appropriate.

The *ns–3* design solves this problem by using a methodology loosely modeled after the Microsoft *Component Object Model* (*COM*) design approach[73]. In this approach, objects deriving from a special base class can be aggregated (associated with) other such objects. After the objects have been aggregated, later queries to the objects can determine if an object of a specified type has been previously aggregated, and if so, a pointer to the associated object is returned. In the example cited above, the model developer might create an object representing a two–dimensional location value that should be associated with a given `Node` object. To achieve this, the `Location` object is created, given a value, and then aggregated to the specific `Node` object. Later, the model can query each `Node` and retrieve a pointer to the `Location` object associated with each node.

```
 1  // Revised Code snippet from the Dumbbell topology object
 2  // Add a node location object to the left side router
 3
 4  // Get a pointer to the left side router node
 5  Ptr<Node> lr = GetLeft();
 6  // See if a node location object is already present on this node
 7  Ptr<NodeLocation> loc = lr->GetObject<NodeLocation>();
 8  if (loc == 0)
 9    { // If not, create one and aggregate it to the node.
10      loc = CreateObject<NodeLocation>();
11      lr->AggregateObject(loc);
12    }
13  // Set the associated position for the left side router
14  Vector lrl(leftX, leftY, leftZ);
15  loc->SetLocation(lrl);
```

Program 2-3 aggregation.cc

This approach is illustrated in the code snippet shown in listing 2-3. The code snippet is a slightly simplified excerpt from the dumbbell topology helper object. This illustrates the creation of an object of class `NodeLocation` and subsequent aggregation of that object to a specified `Node` object.

First, line 5 simply obtains a smart pointer to a particular node, the left side dumbbell router in this case. Line 7 starts by querying if an object of type `NodeLocation` is already aggregated to the node by calling the `GetObject` method. The returned value is a smart pointer to a `NodeLocation` object if one is found, or a null pointer if not. Line 10 creates the new `NodeLocation` object (if one was not found), and then line 11 aggregates the location object with the node by calling the `AggregateObject` method. Finally, the desired location information is specified for the `NodeLocation` object starting at line 14.

2.7 Events in *ns–3*

In discrete event simulations the engine maintains a sorted list of future events (sorted in ascending order of event timestamp), and then simply removes the earliest event, advances simulation time to the time for that event, and then calls the appropriate event handler for the event. Earlier network simulators written in *C++*, notably *ns–2* and *GTNetS*, accomplish this design by defining a base class called `Handler` that include a pure virtual function called `Handle`. The argument to this `Handle` function is simply a pointer to any object that subclasses from a base class `Event`. Then, each object in the simulation that is designed to handle events (such as the network device to handle packet reception events) simply subclasses from the base class `Handler` and implements the `Handle` function.

This approach is simple, easy to understand, and easy to implement. However, it is cumbersome when a given model must process several different types of events. For example, a network device is likely required to process packet reception events, transmission complete events, and link up or down events, just to name a few. This requires somewhat tedious type casting and definition of a number of different event objects to represent the necessary data for each different event.

The *ns–3* simulator takes a different approach that results in considerably more flexibility for the model developer, at the expense of substantial complexity in the design and implementation of the event scheduler and simulator main loop. In *ns–3*, any static function, or any public member function for any object can be an event handler. Rather than defining new event subclasses for each event type, the *ns–3* approach simply specifies the required information as arguments to the function that creates and schedules new events. This is implemented by a complex set of templated functions in the simulator object.

A simple example illustrating this approach to event scheduling and event handling is shown in listing 2-4. This code snippet is excerpted from the `mac-low.cc` implementation of the IEEE 802.11 protocol in *ns–3*.

Line 4 demonstrates creating and scheduling a new event. The first three arguments are common to all member function event scheduling calls, and specify the amount of time in the future the event occurs, the address of the member function to call, and the object pointer for the event handler object. In this example, the future time is `GetSifs()`, the member function is `MacLow::SendCtsAfterRts`, and the object pointer is `this`. The next four parameters are those specifically required by the event handler function. In this case those are a 48–bit address of type `Mac48Address`, a duration of type `Time`, an enumeration value of type `WifiMode`, and a signal to noise ratio of type `double`.

Line 13 shows the specified handler function `SendCtsAfterRts`. Note that the parameter list expect four arguments, and types of those arguments match

```
1  // Code snippet from mac-low.cc, ilustrating event scheduling
2
3  // Excerpt from function MacLow::ReceiveOk
4  m_sendCtsEvent = Simulator::Schedule (
5                                  GetSifs (),
6                                  MacLow::SendCtsAfterRts, this,
7                                  hdr.GetAddr2 (),
8                                  hdr.GetDuration (),
9                                  txMode,
10                                 rxSnr);
11
12 // Excerpt from function MacLow::SendCtsAfterRts
13 void MacLow::SendCtsAfterRts (
14                                Mac48Address source,
15                                Time duration,
16                                WifiMode rtsTxMode,
17                                double rtsSnr)
18 {
19   NS_LOG_FUNCTION (this << source << duration
20                         << rtsTxMode << rtsSnr);
21   // Remainder of code removed for brevity.
```

Program 2-4 schedule.cc

those specified earlier during the event scheduling. Should any of the argument types not match, a compile–time error occurs.

The end result of the call to the `Schedule` function is that the specified member function will be called at the appropriate time in the future (`GetSifs()` seconds in this case), and the parameters specified on the `Schedule` call will be passed to the event handler. It is easy to see that such an approach avoids the need to introduce an intermediate, generic event handler object that later dispatches events to specific model functions; instead, the functions themselves can be the event handlers.

2.8 Compiling and Running the Simulation

As discussed above, *ns–3* programs are typically *C++* programs that link against a library providing the *ns–3* core and simulation models. The project uses the Waf build system to configure and manage the build of the simulator and its documentation. Waf is a Python-based framework supporting configuration, build, installation, packaging, and testing. Once a simulation program is built, the final executable will be placed in a `build/` directory, where it can be run from a shell like any other program. Waf also provides a custom shell, which features integration of dynamic library path discovery and support for debugging tools and memory checkers, that can be used to run programs such as typing `./waf -run my-program` for a program `my-program.cc`.

Because the *ns–3* API is also exported as Python bindings, users can also write Python programs instead of *C++* programs, such as in listing 2-5 that corresponds to listing 2-1 above.

```
1  import ns3
2  def main(argv):
3      nodes = ns3.NodeContainer()
4      nodes.Create(2)
5
6      pointToPoint = ns3.PointToPointHelper()
7      ...
```

Program 2-5 first.py

2.9 Animating the Simulation

The *ns–3* tool has the ability to create a trace file specifically designed to facilitate the animation of the flow of packets through the simulation, allowing visual confirmation that the packets are indeed flowing through the simulated network as desired. The addition of the animation trace file output is quite simple, and is illustrated in listing 2-6. This particular example is a snippet from the `test-dumbbell.cc` example program. Presently, the animation interface supports only the *point–to–point* network devices and channels, with support of other device types planned.

To facilitate the animation, the only requirement is to specify a *location* for each node object in the simulation, and to create and configure an object of class `AnimationInterface`. In the example, line 11 simply creates a dumbbell topology with the specified number of leaf nodes on the left and right side, and the specified helper objects to connect the nodes together. The majority of the animation work is done by the call to `BoundingBox` at line 23. This specifies the upper left X and Y coordinates and the lower right X and Y coordinates that will contain the nodes in the dumbbell. They are dimensionless units, and the nodes in the dumbbell are positioned in this box in such a way to result in a symmetric and visually pleasing animation.

The `AnimationInterface` object is created at line 26. Then a file name specified on the command line argument is assigned as the name of the output trace file. Once all nodes are created and given node locations, the `StartAnimation` function is called, which results in the complete list of nodes, locations, and connectivity being written to the specified trace file. During the simulation execution initiated by the `Run` call at line 38, all packet transmission events are written to the trace file, along with sufficient information to later animate the path of that packet during the animation. Finally, the

`StopAnimation` function at line 40 causes the output file to be closed and all remaining trace data to be flushed to the trace file.

A sample animation visualization for the `test-dumbbell` program is shown in Figure 2.1.

2.10 Scalability with Distributed Simulation

In order to achieve scalability to a very large number of simulated network elements, the *ns–3* simulation tools supports *distributed simulation*. Rather than running a single simulation instance that must handle all events and all network object models, the distributed simulation approach allows the execution of the simulation on multiple, independent computing platforms. By doing this, the overall scalability of the simulated networks can increase considerably.

The approach to supporting distributed simulation in *ns–3* is derived from prior work designing distributed simulation for the Georgia Tech Network Simulator (*GTNetS*). Consider the simple topology shown in Figure 2.2. To execute an *ns–3* simulation using distributed simulation, one approach would be to use four separate simulation processes, each maintaining its own timestamp–ordered event list, and each modeling a subset of the overall topology. For example, simulator 0 would model all of the network elements in subnet 0, simulator 1 would model subnet 1, and so on. However, if this approach is used, then the complete global topology is not known by any one simulator instance. Lacking the global topology picture, the simulators cannot easily make routing decisions globally, and must resort to models for routing protocols and the corresponding overhead for maintaining routing tables.

An alternative approach is shown in Figure 2.3. Here, each simulator instantiates *ns–3* objects for *every* network element in the complete topology. However, each instance only maintains the state (and processes events) for the topology subset assigned to it, as described above. For the remaining network elements (those assigned to other simulator instances) the only state created is the existence of the node, device, and link objects. This is called a *Ghost Node* in simulation parlance. In the figure below, simulator 0 is responsible for all elements in subnet 0 (as described above), and additionally creates ghost nodes for the remaining three subnets. However, no events are scheduled or processed for ghost nodes. Rather, the responsible simulator for those nodes handle those events. In this example, simulator 1 would create complete model elements for the subnet1 objects, and create ghost objects for subnets 0, 2, and 3.

The *ns–3* simulator provides an easy way to create the ghost elements, by assigning a global *simulator id* to the simulator instance, and individual *node id* values to each network element. If the node id does not match the

```
1  // Excerpt from the test-dumbbell.cc illustrating the animation interface
2
3  // Create the point-to-point link helpers
4  PointToPointHelper pointToPointRouter;
5  pointToPointRouter.SetDeviceAttribute  ("DataRate", StringValue ("10Mbps"));
6  pointToPointRouter.SetChannelAttribute ("Delay", StringValue ("1ms"));
7  PointToPointHelper pointToPointLeaf;
8  pointToPointLeaf.SetDeviceAttribute    ("DataRate", StringValue ("10Mbps"));
9  pointToPointLeaf.SetChannelAttribute   ("Delay", StringValue ("1ms"));
10
11 Dumbbell d(nLeftLeaf, pointToPointLeaf,
12            nRightLeaf, pointToPointLeaf,
13            pointToPointRouter);
14
15 // Assign IP Addresses
16 d.AssignAddresses(Ipv4AddressHelper("10.1.1.0", "255.255.255.0"),
17                   Ipv4AddressHelper("10.2.1.0", "255.255.255.0"),
18                   Ipv4AddressHelper("10.3.1.0", "255.255.255.0"));
19 // Install on/off app on all right side nodes
20 // Omitted for brevity
21
22 // Set the bounding box for animation
23 d.BoundingBox(1, 1, 10, 10);
24
25 // Create the animation object and configure for specified output
26 AnimationInterface anim;
27 // Check if a file name specified on command line, and set it if so
28 (!animFile.empty())
29    {
30      anim.SetOutputFile(animFile);
31    }
32 anim.StartAnimation();
33
34 // Set up the acutal simulation
35 Ipv4GlobalRoutingHelper::PopulateRoutingTables();
36
37 // Run the simulation
38 Simulator::Run();
39 Simulator::Destroy();
40 anim.StopAnimation();
41 return 0;
```

Program 2-6 anim.cc

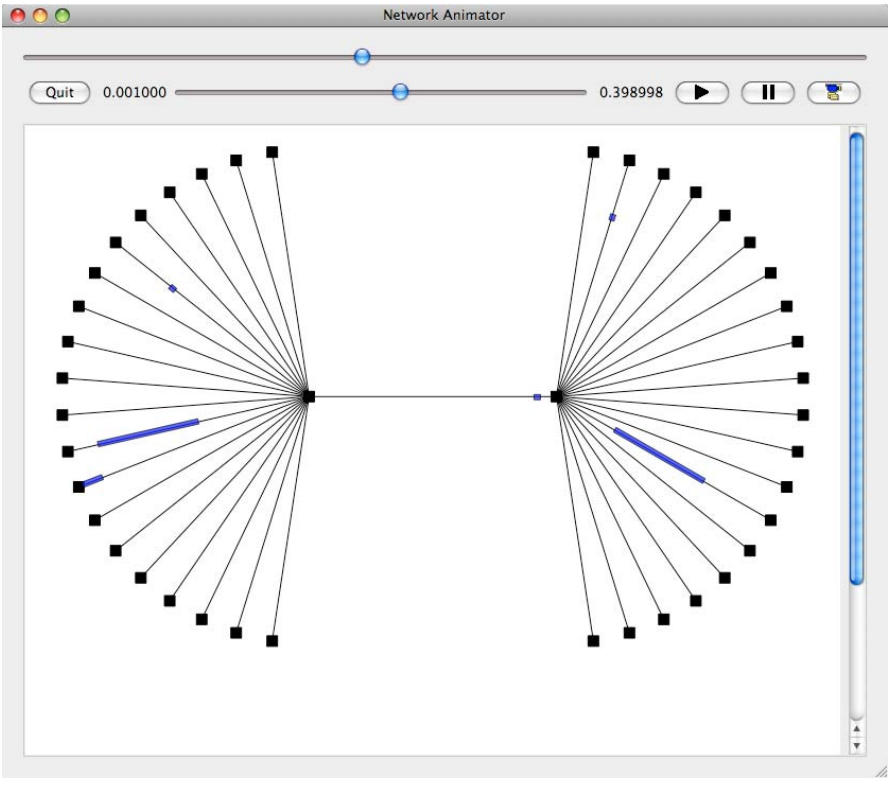

Fig. 2.1: Sample *ns–3* Animation

simulator id, then a ghost node is created. No applications or protocol stacks should be created for the ghost nodes.

2.11 Emulation Capabilities

In the past decade, experimental research in networking has migrated towards testbeds and virtualization environments, in large part because of the realism that such environments provide compared to simulation abstractions, and also because real implementation code can be reused. A design goal in *ns–3* has been to offer several options for the support for emulation, virtualization, and the running of real implementation code, to minimize the experimental discontinuities when moving between simulation, emulation, and live experiments, and to enable experiments that may want to combine the techniques.

The first emulation capability that was integrated to *ns–3* was the Network Simulation Cradle (NSC)[240]. NSC is a framework that largely auto-

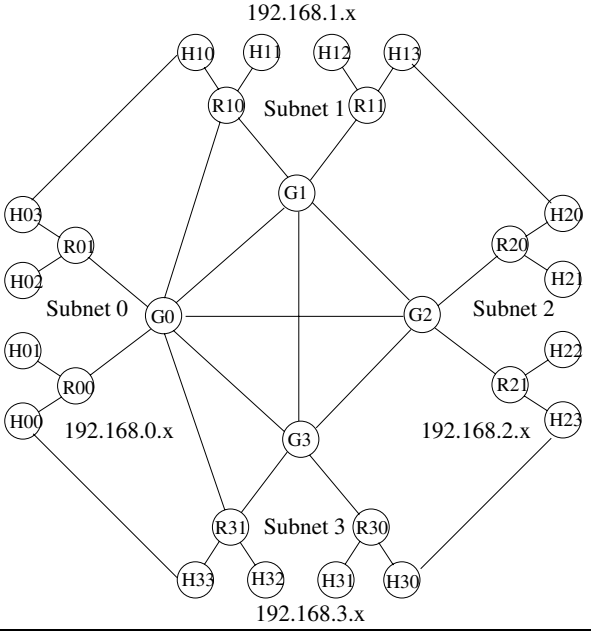

Fig. 2.2: Simple Topology

mates the porting of kernel code from several networking stacks to a simulation environment. The NSC-enabled stack in *ns–3* allows researchers to use the actual TCP implementation from recent Linux kernels.

Another emulation capability that has been used to integrate *ns–3* with experimental wireless testbeds is the emulation NetDevice, which allows an *ns–3* process on a physical computer to bind a simulation-based network interface to a physical interface on the host machine. This capability also requires the use of a real-time scheduler in *ns–3* that aligns the simulation clock with the host machine clock. One testbed in which this has been used is the ORBIT testbed at Rutgers University[385]. ORBIT consists of a deployment of a two dimensional grid of four hundred computers in a large arena, on which various radios (802.11, Bluetooth, and others) are deployed. *ns–3* has been integrated with ORBIT by using their imaging process to load and run ns-3 simulations on the ORBIT array. The technique uses an emulation NetDevice to drive the hardware in the testbed, and results are gathered either using the ns-3 tracing and logging functions or the native ORBIT data management framework.

The inverse of the above capability is also an important use case. Rather than run *ns–3* protocol stacks over real network interfaces, one can run real systems over an *ns–3*-provided emulation of a (typically wireless) network. For example, lightweight virtual machines have been developed that provide varying degrees of system isolation between instances. Specially tailored ma-

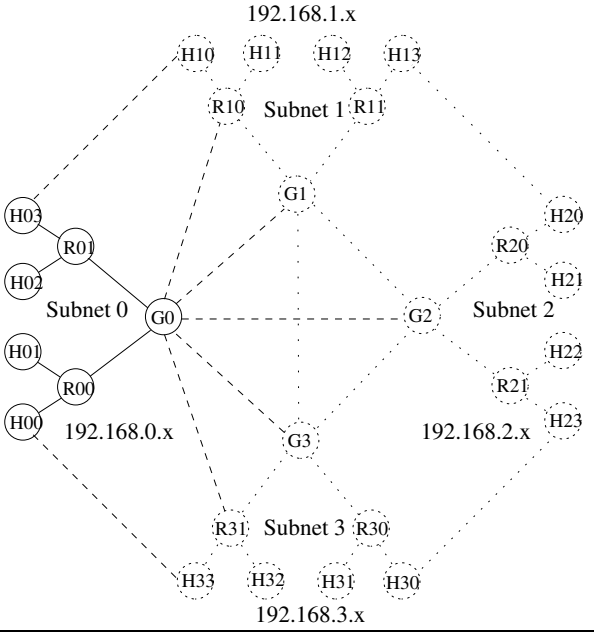

Fig. 2.3: Distributed Topology

chines can be used to virtualize instances of the network stack and provide partly shared and partly private file systems and access to system resources. The term "slicing" has often been used to describe this type of virtualization, such as in PlanetLab[366]. While slices on an experimental testbed run over real (distributed) network segments, by using *ns–3* as an underlay, they can also be run over simulated networks. This hybrid of emulation and (real-time) simulation is particularly useful for wireless networking, which is not always provided or is provided only in a limited fashion on virtualized testbeds.

As the maturity and cost-effectiveness of virtualization software continues to increase, the authors foresee that blending the use of *ns–3* with lightweight virtual machines will become a very useful research capability. The project is also exploring new techniques to run multiple instances of unmodified application processes in simulations on a single host.

2.12 Analyzing the Results

The design goal for *ns–3* has been to equip the core of the simulator with tools that allow for highly customizable extraction of event logs and output statistics, to provide a framework for managing large numbers of simulation runs and output data, and to allow third-party analysis tools to be used where possible.

ns–3 includes a *tracing* subsystem that allows for the export of simulation data from *trace sources* to *trace sinks*. The key ideas are that, by decoupling the data generation (trace source) from the consumption of the data (trace sink), users can customize and write their own trace sinks to generate whatever output they desire, without having to edit the core of the simulator. Model authors declare various *trace sources* in their models, such as the arrival of a packet or the change in value of a variable such as a congestion window. Users who want to trace selected behavior from a model will simply attach their own trace sinks (implemented as C++ functions) to the trace sources of interest. *ns–3* also provides a set of stock trace sinks for common trace events such as the generation of typical packet traces.

Users often do not run a single instance of a simulation; they run multiple independent replications with different random variables, or they run a set of simulations with each set of runs changing slightly the configuration. Frameworks are needed to manage the simulation runs and to manage and organize the large amounts of output and configuration data that is generated. *ns–3* is presently evaluating a custom statistics framework that provides support for each of these functions. The framework is organized around the following principles:

- Define a *trial* as one instance of a simulation program;
- Provide a control script to execute instances of the simulation, varying parameters as necessary;
- Collect data and marshal into persistent storage for plotting and analysis using external scripts and tools;
- Provide a basic statistical framework for core statistics as well as to perform simulation run-length control based on observed data; and
- Use the *ns–3* tracing framework to instrument custom code.

This framework defines metadata to collect run information, and provides support to dump simulation configuration and output data to a relational database or to other existing output formats.

ns–3 also supports standardized output formats for trace data, such as the `pcap` format used by network packet analysis tools such as `tcpdump`, and supports standardized input formats such as importing mobility trace files from *ns-2*. By aligning to existing data standards, *ns–3* allows users to reuse a wide range of existing analysis tools. Likewise, the project makes use of existing support libraries such as the GNU Scientific Library (GSL) to avoid reinventing statistical and analysis tools.