A User Guide to Pydtn and Simlpy

Michael Marsh
University of Maryland

Copyright 2008 Michael Marsh, University of Maryland
Some rights reserved.

This work is licensed under the Creative Commons Attribution-Share Alike 3.0 Unported License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/3.0/; or, (b) send a letter to

Creative Commons
171 2nd Street, Suite 300
San Francisco, California, 94105, USA.

Contents

I Simulator Basics 1

1 Introduction 3
- 1.1 Notation 5

2 Basic Simlpy Usage 7

3 Pydtn Usage 11
- 3.1 Static Networks 11
  - 3.1.1 Configuring the Network 11
  - 3.1.2 Sending Data 13
  - 3.1.3 Controlling Nodes using Policies 15
  - 3.1.4 More Complicated Actions 16
- 3.2 Mobile Networks 17
- 3.3 Data Collection 23
  - 3.3.1 Global Statistics 23
  - 3.3.2 Tracers 24
  - 3.3.3 `sim.collect()` 24
  - 3.3.4 `Entity.get()` 25
  - 3.3.5 Scheduled Python Function Calls 26
  - 3.3.6 Writing Output to Files 26

II Extending the Simulator 29

4 Writing Modules 31
- 4.1 Python Modules 31
  - 4.1.1 Preliminaries 31
  - 4.1.2 Configuration 32
  - 4.1.3 Collection 34
  - 4.1.4 Triggering 35
  - 4.1.5 Using the Module 36
- 4.2 C++ Modules 37
  - 4.2.1 The Source Code 37

4.2.2 Compiling . . . 45
4.2.3 Advanced Options . . . 47

5 A Brief Introduction to Data Structures 53
5.1 The dtn Library . . . 53
5.1.1 Bundle . . . 53
5.1.2 Node . . . 55
5.1.3 BundleStore . . . 55
5.1.4 NodePolicy . . . 56
5.1.5 Consumer . . . 58
5.2 The pydtn Module . . . 59
5.2.1 WrapNode . . . 60
5.2.2 WrapLink . . . 61
5.2.3 SimLink . . . 61
5.3 The mobility Module . . . 62
5.3.1 MobileNode . . . 62
5.3.2 WirelessLink . . . 63
5.3.3 MobileApp . . . 63
5.3.4 MockLink . . . 63

6 Tracers 65
6.1 Structure of a Tracer . . . 65
6.2 flowtrace . . . 66
6.3 Modifying a Tracer . . . 67
6.3.1 Setup . . . 67
6.3.2 The Header Files . . . 68
6.3.3 The Class Definition File . . . 71
6.3.4 Bootstrapping . . . 72
6.3.5 Makefile . . . 72

7 Drop Cause Classes 75

8 Applications 79
8.1 Ping . . . 80
8.1.1 Class Declaration . . . 80
8.1.2 Sending a Ping . . . 82
8.1.3 Responding to a Ping . . . 82
8.1.4 Finishing the Round-Trip . . . 83
8.1.5 Bootstrapping . . . 83
8.2 Mobile Forwarding Protocol . . . 85
8.2.1 Structure of the Library . . . 85
8.2.2 A Do-Nothing Protocol . . . 86
8.2.3 Basic Epidemic Forwarding . . . 88

8.2.4 Modifying the Epidemic Protocol 89

9 Policies **93**
9.1 Forwarding 93
9.1.1 `forward()` 93
9.1.2 `forwardOn()` 95
9.1.3 `cache()` 96
9.2 VolatileStore 97
9.3 Custody 98
9.3.1 `takeCustody()` 98
9.3.2 `policyRemove()` 99
9.3.3 `retry()` 99

10 Wireless Links **101**
10.1 Simple Disk Model 102

11 Mobile Nodes **105**
11.1 Random Waypoint Geometries 105
11.1.1 Existing Geometries 105
11.1.2 Creating New Geometries 108
11.2 Other Mobility Patterns 108
11.2.1 Parametrized Movement 108
11.2.2 Hybrid Patterns 109

Part I

Simulator Basics

Chapter 1

Introduction

The purpose of this guide is to help you get started with and make the most of the *pydtn* delay-tolerant network simulator. The earlier chapters cover the basics of running the simulator and creating simulation scenarios of increasing complexity using the built-in features of pydtn. Later chapters delve into extending the basic simulator with your own additions.

pydtn uses the *simlpy* (pronounced **sim**·'l·pē) generic simulator, which was written specifically for pydtn. Both are written in a combination of C++ (for the guts of the simulator) and python (for the interface). Functionality is added by loading modules into the simulator. The interface between python and C++ can be a bit cumbersome, but since simlpy includes the complete python interpreter, many modules provide python functions or objects that make the calls into the C++ guts with a friendlier python syntax.

We will delve into this in depth later, but it is perhaps worthwhile to get a taste of the general simulation structure and what you will see in essentially fully specified configurations. In an effort to make the simulator more easily extensible, there is a great deal of complexity in the basic design. In particular, the basic "DTN simulator" consists of three separate packages, and that does not include any node mobility. One of these packages is the simlpy framework, which provides the event-passing and synchronization mechanisms. A second package is the actual DTN implementation, which provides the common mechanisms for any DTN node, regardless of specific configuration (such as the link protocol or mobility) and agnostic as to environment. That is, it is not tied to simlpy. The third basic package is pydtn, which connects the DTN implementation with simlpy, and is essentially (in the language of DTN) a convergence layer.

The functionality provided by these three packages is minimal. You can configure a fixed-topology simulation using table-based routing, but that's about all. Additional functionality comes from loading additional packages, called modules, into the simulator. These range from the basic mobility module to bundle tracing to forwarding algorithms. Most serious users of pydtn will likely want to write their own modules for functionality not provided by any of the core modules, and this will be covered later in some detail.

Here is an example of a simple scenario employing mobile nodes using epidemic forwarding of bundles. Don't worry about whether the script makes sense right now. The scenario defined is ten mobile nodes, starting on a line and moving randomly, each producing a single bundle for a destination at the far end of the mobility range. Nodes exchange messages as they meet, with delivery occuring whenever a mobile node sees the data sink and has as-yet undelivered bundles.

```
# This is an example of configuring a mobile network using random waypoints.
import pydtn
from pydtn.mobility import *
import pydtn.mobility.rf as rf # disk model antennae
import pydtn.mobility.epidemic # epidemic routing
import pydtn.flowtrace
import pydtn.sampleapp

# Set some overrideable defaults:
seed=None
beacon_interval=2
wireless_range=1.35
min_speed=2
max_speed=5
sim_time=720

# Process command-line arguments, which should be passed as:
#  -a variable=value
clargs = sim.args()
for a in clargs: exec(a)

# Set the default lifetime for a bundle to 1 day.
pydtn.config("bundle_lifetime",86400)
pydtn.config("resend_period",0)

# Trace traffic by flows, and write the data to the file specified.
pydtn.flowtrace.setup("epidemic_example.flow")

# Set up the mobility model.
if seed is None:
    s = sim.seed()
else:
    s = sim.seed(seed)
set_beacon_default(beacon_interval)
random_waypoint()

# Set the defaults for link characteristics.
rf.set_range(wireless_range)  # in meters
rf.set_latency(0)             # in seconds
rf.set_bandwidth(10)          # in Mbps

# Create the nodes.
mobility_range = SimpleRectangleRange( lower_corner=(0,0), upper_corner=(40,40) )
speeds = SpeedRange(min_speed,max_speed)
source = []
for i in range(10):
    source.append( node( 'source%d' % (i), position=[0,2*i],
                         waypoint_range=mobility_range, speed_range=speeds ) )
sink = node( 'sink', position=[40,40] )

# Attach our simple example application to all nodes.
applyToNodes(pydtn.sampleapp.attach)

# Send data from a to b.  The objects we have include both the node
# and the link, so we have to specify which is sending data.
send_time = 0.000001
for i in range(10):
    pydtn.sampleapp.send(source[i].node, sink.node, 'hello world', send_time )

sim.stopAt(sim_time)
sim.run()

sys.exit()
```

1.1 Notation

Text that you would type at a command prompt or in a script is presented in `teletype font`. Scripts will generally be enclosed in boxes to visually set them apart, as above. Commands typed at the shell prompt will be preceded by `$`, which is a fairly common shell prompt. Commands typed at the python prompt will be preceded by the standard python prompt `>>>`, with continuation lines (for code blocks) preceded by the prompt `...`.

Chapter 2

Basic Simlpy Usage

The simlpy simulator is designed to be easy to use. It only takes a few command-line arguments:

`-L <dir>` Add the directory `<dir>` to python's module search path. In addition to the standard python path, simlpy also includes its configured library directory in the search path. Use this for finding modules installed elsewhere.

`-i <module>` Import `<module>` when starting the simlator. By default, only the **`sys`** module and the internal **`sim`** module are loaded. You can also import modules with the standard python `import` command, so this should rarely be needed.

`-p <path>` The path to the python libraries with which the simulator was linked. This should almost never be needed.

`-v` Increase the verbosity level of output. This may be called multiple times for greater verbosity.

`-a <arg>` Adds `<arg>` to the list of parameters made available to scripts. This may be specified multiple times to pass multiple arguments to scripts. Arguments will be stored in a list ordered as they are on the command line.

`<script file>` Other options are interpreted as python scripts, and are run in the order provided.

When called without any script files, or when the scripts finish without calling `sys.exit()`, simlpy presents the user with the python prompt. Anything that can be done from the python prompt or a script can be done in simlpy interactively.

simlpy works by scheduling and processing *events*. An event holds as data the time for which it is scheduled, and the object that acts as its handler, called an *entity*. Some types of events will contain additional data, as we will see later. The simulator maintains two lists of events: a list of events that *must* be processed and a list of events that *might* be processed. We call these, respectively, *scheduled* and *tentative* events. As long as there are scheduled events, the simulation will continue. Tentative events are generally used for periodic actions that must be performed as long as the simulation is running, but that should not themselves cause the simulation to continue.

For example, mobile nodes will send out periodic beacons to inform potential neighbors of their presence, but these beacons are only meaningful as long as the simulation has something else to do, such as moving data.

As mentioned above, simlpy has its own module within python, called **sim**. The command

```
>>> help(sim)
```

will provide you with fairly detailed documentation. Briefly, the commands provided are:

sim.args() Returns the list of arguments passed to simlpy on the command line using the -a flag. Interpreting these arguments, including their order, is specific to a particular script. A useful scripting idiom is to specify assignment statements as arguments, which can then be executed by the script to override default values (see below for an example).

sim.collect() For any entities created by the sim module, this will call their collect() methods. What this does depends on the entity.

sim.random() Returns the next number from the simulator's pseudo-random number generator (RANLUX), as a real-valued number in the range $[0, 1)$.

sim.run() Starts the simulation, which runs until there are no more events to process, a stopping event is processed, or sys.exit() is called.

sim.schedule(<t>,<f>) Runs the python function <f> at time <t>.

sim.seed([<n>]) Seeds the random number generator with the integer <n>, if provided, or a seed constructed from the current (real) time, if not. The random number seed is returned.

sim.stopAt(<t>) Schedule a stopping event for the simulation time <t>, either as a (floating-point or integer) number of seconds or in the format described below.

sim.time() Returns the current time, as a 2-element list in struct timeval format. That is, the first element is the number of seconds from the start of simlation, and the second element is the number of microseconds after the first element.

sim.Entity(<identifier>) Creates a new entity of the type registered for <identifier>. The registry system will be discussed in Chapter 4.

We can see some of this functionality in action, even without other modules loaded in:

```
$ simlpy -a v=2
>>> sim.time()
[0L, 0L]
>>> sim.stopAt([2,3000])
>>> sim.time()
[0L, 0L]
>>> sim.run()
```

```
>>> sim.time()
[2L, 3000L]
>>> v = 1
>>> print v
1
>>> for arg in sim.args():
...     exec(arg)
...
>>> print v
2
>>> sys.exit()
$
```

Chapter 3

Pydtn Usage

The DTN simulation framework **`pydtn`** runs as a dynamically loaded module within simlpy. Modules are loaded in simlpy just as they are in python, with the `import` command:

```
import pydtn
```

As with the base simulator, there is in-python help provided for **`pydtn`**, as well as all of its subpackages:

```
>>> help(pydtn)
```

The interactive help is likely to be more complete than this user guide, as it documents all available functions, classes, and arguments.

We will first see how to configure a static wired network in **`pydtn`**, which uses many of the common building blocks that also pertain to wireless networks. We will then move to wireless networks, which are likely to be of more interest for DTN simulations. Finally, we will examine different ways of collecting data from a simulation.

3.1 Static Networks

3.1.1 Configuring the Network

There are two types of objects on which we'll focus here: *nodes* and *links*. Nodes are, as you might expect, representations of the physical hosts that comprise the network. Similarly, links are the communications channels between these nodes. All links are unidirectional. If node A and node B can communicate symmetrically, then there must be separate links both from A to B and from B to A.

Both nodes and links are simlpy *entities*, and can be created by calls to `sim.Entity()`:

```
import pydtn

n = sim.Entity('node')
l = sim.Entity('link')
```

In practice, this is an inconvenient way to create these entities, since they are created completely unconfigured. For example, to set up a simple node named "A" with a volatile storage capacity (essentially queue size) of 4MB and a stable storage capacity of 10MB, you would have to do the following:

```
import pydtn

a = sim.Entity('node')
a.config('addr','A')
a.config('capacity',4096)
a.config('stable',10240)
```

Even with this configuration, the node will not have any way of forwarding bundles. To simplify things, there is a sub-module within **pydtn** called **network**, which reduces the above to:

```
import pydtn
import pydtn.network

a = pydtn.network.addNode('A',capacity=4096,stable=10240)
```

This helper function also attaches a default forwarding algorithm to the node, and puts the node in a list for later routing computations.

Links may be created in a similar way using the **network** module:

```
import pydtn
import pydtn.network

a = pydtn.network.addNode('A',capacity=4096,stable=10240)
b = pydtn.network.addNode('B',capacity=4096,stable=10240)
l_ab = pydtn.network.addLink( a, b, latency=0.003, bandwidth=100 )
l_ba = pydtn.network.addLink( b, a, latency=0.003, bandwidth=100 )
```

Here we have two nodes connected with a bidirectional link with a latency of 3ms and bandwidth of 100Mbps in each direction.

At this point, however, the network still has no routing. All of the basic structures are in place, but the routes have to be computed. Because the **network** module has kept track of all the nodes and links created through it, it can provide the route-computation functionality:

```
import pydtn
import pydtn.network

a = pydtn.network.addNode('A',capacity=4096,stable=10240)
b = pydtn.network.addNode('B',capacity=4096,stable=10240)
l_ab = pydtn.network.addLink( a, b, latency=0.003, bandwidth=100 )
l_ba = pydtn.network.addLink( b, a, latency=0.003, bandwidth=100 )

pydtn.network.route()
```

Arbitrarily complex networks can be created in this fashion, though the routing computation, which performs Dijkstra's shortest path algorithm pairwise for all nodes, will take increasing lengths of time. A call to `pydtn.network.construct()` before calling `route()` will cause an output file to be generated with the complete node, link, and routing commands specified. This can then be imported in subsequent scripts to avoid the route computation step.

3.1.2 Sending Data

Once a network has been created, you can send bundles from one node to another using a node entity's `emit()` method, which creates an event to be processed at a specified time:

```
import pydtn
import pydtn.network

a = pydtn.network.addNode('A',capacity=4096,stable=10240)
b = pydtn.network.addNode('B',capacity=4096,stable=10240)
l_ab = pydtn.network.addLink( a, b, latency=0.003, bandwidth=100 )
l_ba = pydtn.network.addLink( b, a, latency=0.003, bandwidth=100 )

pydtn.network.route()

a.emit(0.1, 'data', b, 'hello world')

sim.run()
sys.exit()
```

Here we've told node `A` to send a data event to `B` at time 0.1s. The payload of the bundle is the string `hello world`. The final two lines tell the simulation to begin, and once the events have all been processed, to terminate.

Running this script without options produces no output, but if we pass simlpy the `-v` option, we get the following:

```
(t={0,100000}) node 41 stored a bundle
(t={0,100000}) sending 41 bytes from 41 to 42
(t={0,103004}) received 41 bytes at 42 from 41
(t={0,103004}) node 42 consuming data bundle 0 from 41
   11 bytes
   68 65 6C 6C 6F 20 77 6F 72 6C 64
(t={0,103004}) sending 30 bytes from 42 to 41
(t={0,106007}) received 30 bytes at 41 from 42
(t={0,106007}) node 41 received an ACK for 0
(t={1,150000}) node 41 deleted a bundle
```

Let's take a close look at this output; understanding the verbose bundle information is useful for diagnosing many problems. First, the time is expressed in seconds and microseconds, rather than as a single decimal value. Next, we see that the nodes are referred to as "41" and "42." These are the (hex) ASCII values for the characters "A" and "B," respectively. The verbose output takes the conservative approach of printing everything in hexadecimal, since it is considerably simpler than trying to determine whether a byte string is printable ASCII or not.

We record several types of events. The first event we see is the originator of the bundle, node *A*, storing the bundle in its persistent store. This happens at the time the "emit" event is processed. *A* then sends the bundle to *B* (at the same time), as recorded on the following line. After that, about 3ms later, *B* receives the bundle from *A*. Both the send and receive lines record the total size of the bundle, as well. Having reached its destination, the next line reports that *B* is consuming the bundle. Here we see both the source of the bundle (*A*) and the bundle sequence number from the source, in this case 0, as it's the first bundle created by *A*. The consumer also prints out the bundle's payload, again in hexadecimal. The eleven bytes of data are the ASCII string `hello world`.

After consuming the bundle, *B* generates an acknowledgment, the 30-byte bundle sent on the next line. *A* receives this after another 3ms. The bundle is finally deleted from *A*'s persistent store a little more than a second after it was created, because one second is the default lifetime of a bundle.

A Higher-Level Interface

The call to `emit()` is, like the calls to `configure()`, somewhat low-level and inconvenient. This becomes even more the case when we consider that most of the traffic we'll want to send is going to be specific to some algorithm or simulated service. Messages with semantic meaning to them are generally exchanged by *applications*. A simple example of this is the **`sampleapp`** application:

```
import pydtn
import pydtn.network
import pydtn.sampleapp

a = pydtn.network.addNode('A',capacity=4096,stable=10240)
b = pydtn.network.addNode('B',capacity=4096,stable=10240)
l_ab = pydtn.network.addLink( a, b, latency=0.003, bandwidth=100 )
l_ba = pydtn.network.addLink( b, a, latency=0.003, bandwidth=100 )

pydtn.network.route()
pydtn.network.applyToNodes(pydtn.sampleapp.attach)

pydtn.sampleapp.send(a,b,'hello world',0.1)

sim.run()
sys.exit()
```

The above script uses **`sampleapp`** to generate traffic. The application is attached to all nodes in the network by calling **`network`**'s special `applyToNodes()` method. We could, instead, have called

```
pydtn.sampleapp.attach(a)
pydtn.sampleapp.attach(b)
```

explicitly. This is useful when only some of the nodes will have a particular application attached to them. If you run the above script, you will see that **`sampleapp`** prints out a message when it receives a bundle, while emitting a bundle event directly from the node does not.

The sample application generates the following message when a bundle is delivered:

```
sample app: got a bundle of size 50
```

Note that the bundle is nine bytes larger than before. This is because the bundle now carries the identifier `sampleapp` to identify its application. Interspersed with the verbose output as above, we have:

```
(t={0,100000}) node 41 stored a bundle
(t={0,100000}) sending 50 bytes from 41 to 42
(t={0,103004}) received 50 bytes at 42 from 41
(t={0,103004}) node 42 consuming data bundle 0 from 41
   11 bytes
```

```
   68 65 6C 6C 6F 20 77 6F 72 6C 64
sample app: got a bundle of size 50
(t={0,103004}) sending 30 bytes from 42 to 41
(t={0,106007}) received 30 bytes at 41 from 42
(t={0,106007}) node 41 received an ACK for 0
(t={1,150000}) node 41 deleted a bundle
```

Aside from the larger bundle size and the application-specific output, this is identical to the previous version.

3.1.3 Controlling Nodes using Policies

The behavior of a node in pydtn is determined by the *policies* registered with that node. There are three types of policies, which control different aspects of the node's behavior.

Forwarding Policies

A simple example of a forwarding policy, and the particular policy used by default in static pydtn networks, is a routing table lookup. That is, for a bundle's destination, the policy looks in its routing table to find the appropriate next hop, and retrieves the link to that node. If the link is available, the bundle is forwarded.

Routing tables become less useful as connectivity decreases, so once we begin to discuss networks of mobile nodes, we will look at other types of forwarding policies.

Volatile Storage Policies

If a bundle could not be forwarded, then we have to determine whether to store it temporarily for later delivery. This is controlled by the node's volatile storage policy. This is analogous to queueing, though pydtn's nodes employ a single shared volatile data store for all pending bundles. By default, a node is configured with a simple drop-tail policy: a bundle is stored if there's still enough room for it. This is based on bundle sizes, not a bundle-granularity count.

Custody Policies

The final policy type determines whether a node should take custody of a bundle. That is, whether the bundle should be added to the node's stable storage. This is done before invoking the forwarding policy. If there is a bundle in stable storage, the node will periodically attempt to send it until it receives an acknowledgement from either the destination or another node accepting custody. The resend period can be configured for a node with:

```
a = pydtn.network.addNode(...)
a.config('resend',5)
```

which will try to send stored bundles every five seconds.

By default, nodes do not have a custody policy configured. A simple space-available policy (similar to the default drop-tail volatile storage policy) is defined, and can be added to a node with:

```
a = pydtn.network.addNode(...)
a.config('custody','spaceavail')
```

Using a little python, and the `applyToNodes` function we saw before, we can configure all nodes with a custody policy and resend period:

```
def custodial_node( n ):
    n.config('custody','spaceavail')
    n.config('resend',5)

pydtn.network.applyToNodes(custodial_node)
```

3.1.4 More Complicated Actions

It's probably a safe bet that at some point you will want to run a simulation that sends more than one bundle. In fact, you will probably not want the hassle of scheduling all of the bundles individually.

If you want to send bundles regularly, with some fixed period, or in fact perform any action regularly, you can use the `periodically` function. Here's how it works:

```
import pydtn

def speaking_clock():
    print 'At the tone, the time will be ', sim.time(), '... BEEP!'

pydtn.periodically(start_time=0,interval=1,function=speaking_clock)

sim.stopAt(20)
sim.run()
sys.exit()
```

The parameter `start_time` is the first instant at which `function` should be run, and it is run subsequently every `interval` seconds. The output of this script will be:

```
At the tone, the time will be  [0L, 0L] ... BEEP!
At the tone, the time will be  [1L, 0L] ... BEEP!
At the tone, the time will be  [2L, 0L] ... BEEP!
At the tone, the time will be  [3L, 0L] ... BEEP!
At the tone, the time will be  [4L, 0L] ... BEEP!
At the tone, the time will be  [5L, 0L] ... BEEP!
At the tone, the time will be  [6L, 0L] ... BEEP!
At the tone, the time will be  [7L, 0L] ... BEEP!
At the tone, the time will be  [8L, 0L] ... BEEP!
At the tone, the time will be  [9L, 0L] ... BEEP!
At the tone, the time will be  [10L, 0L] ... BEEP!
At the tone, the time will be  [11L, 0L] ... BEEP!
At the tone, the time will be  [12L, 0L] ... BEEP!
At the tone, the time will be  [13L, 0L] ... BEEP!
At the tone, the time will be  [14L, 0L] ... BEEP!
At the tone, the time will be  [15L, 0L] ... BEEP!
At the tone, the time will be  [16L, 0L] ... BEEP!
At the tone, the time will be  [17L, 0L] ... BEEP!
At the tone, the time will be  [18L, 0L] ... BEEP!
At the tone, the time will be  [19L, 0L] ... BEEP!
```

For sending data, you could use:

```
def send_data():
    pydtn.sampleapp.send(a,b,'hello world',sim.time())

pydtn.periodically(start_time=0,interval=1,function=send_data)
```

This will tell **sampleapp** to send a data bundle from `A` to `B` every second, scheduled for the time at which the function `send_data` is called.

The `periodically` function is actually a simplified version of a traffic generation utility. To use this, you would have something like the following in your script:

```
def td(now):
    return [now[0]+1, now[1]]

def send_data():
    pydtn.sampleapp.send(a,b,'hello world',sim.time())

tg = sim.Entity('trafficgen')
tg.config('time_dist',td)
tg.config('generator',send_data)
tg.emit(0)
```

The effect of this is identical to the (much shorter) snippet above. This form can be handy if you have non-uniform times, though. For example, if you want to simulate a Poisson arrival process, you would define `td` appropriately.

Note also that we can simplify things slightly with python's lambda-forms:

```
tg = sim.Entity('trafficgen')
tg.config('time_dist', lambda t: [t[0]+1,t[1]] )
tg.config('generator',
          lambda: pydtn.sampleapp.send(a,b,'hello world',sim.time()) )
tg.emit(0)
```

Whether this is simpler to read depends on how comfortable you are with the functional programming style.

3.2 Mobile Networks

For mobile nodes, or rather for nodes with spatial positions and wireless antennae, configuring the network is a bit different. The basic structure of nodes, links, applications, and scheduling events is the same, however.

The first obvious change is that we need to include a separate **mobility** package:

```
import pydtn
import pydtn.mobility
```

The **mobility** package has its own version of the functions in the **network** package, so we won't bother loading the latter from this point on.

The base **mobility** package has mobile nodes and a simple mobility model, but it doesn't define a wireless link, nor does it provide a meaningful wireless forwarding policy. There is,

however, a sub-package of **mobility** that provides a simple disk model, where nodes that are within a configured radius can see one another and communicate without loss, while nodes outside of this range cannot communicate. We can load this as

```
import pydtn
import pydtn.mobility
import pydtn.mobility.rf
```

By importing the **rf** package, we also set it as the default link model for mobile nodes. There are some package-wide configurations that we can set for the links:

```
import pydtn
import pydtn.mobility
import pydtn.mobility.rf

pydtn.mobility.rf.set_range(5)
pydtn.mobility.rf.set_latency(0)
pydtn.mobility.rf.set_bandwidth(11)
```

This sets a range of 5m with a latency of 0s and a bandwidth of 11Mbps.

The nested python namespaces are starting to get unwieldy, so we'll modify this to use some special features of the `import` command:

```
import pydtn
import pydtn.mobility as mbl
import pydtn.mobility.rf as rf

rf.set_range(5)
rf.set_latency(0)
rf.set_bandwidth(11)
```

We still don't have a forwarding policy, though. Another package provided by pydtn is the DTN epidemic forwarding algorithm, which we can add to our script:

```
import pydtn
import pydtn.mobility as mbl
import pydtn.mobility.rf as rf
import pydtn.mobility.epidemic as ep

rf.set_range(5)
rf.set_latency(0)
rf.set_bandwidth(11)
```

We're now in a position to create nodes. Because each node has only one link (its wireless antenna), **mobility** provides a simple interface:

```
import pydtn
import pydtn.mobility as mbl
import pydtn.mobility.rf as rf
import pydtn.mobility.epidemic as ep

rf.set_range(5)
rf.set_latency(0)
rf.set_bandwidth(11)

a = mbl.node('A', position=[0,0])
b = mbl.node('B', position=[5,5])
```

We now have two mobile nodes, `A` and `B`. One sits at the origin, and the other at $x = y = 5$m. Positions are three-dimensional, but if left unspecified z is assumed to be 0. Note that the objects returned by `mbl.node()` are *not* entities, they are encapsulations of the node and link. We can reference these entities as:

```
node = a.node
link = a.link
```

These nodes can be configured with waypoints as follows:

```
a.node.emit(1,'waypoint',3,0,[1,0])
```

This tells node `A` that at $t = 1$s, it should begin moving towards the point $(1, 0, 0)$. It's scheduled to arrive there at $t = 3$s, and should not wait before starting towards its next waypoint.

You can use this interface to schedule all of your nodes' waypoints, but it's likely that you will want to use something a bit more automated, such as a random-waypoint mobility model. This is provided for you, as well, with fairly minimal changes to your script:

```
import pydtn
import pydtn.mobility as mbl
import pydtn.mobility.rf as rf
import pydtn.mobility.epidemic as ep

mbl.random_waypoint()
sim.seed()
rf.set_range(5)
rf.set_latency(0)
rf.set_bandwidth(11)

mobility_range = mbl.SimpleRectangleRange( lower_corner=(0,0), upper_corner=(40,40) )
speeds = mbl.SpeedRange( 2, 5 )
a = mbl.node('A', position=[0,0], waypoint_range=mobility_range, speed_range=speeds)
b = mbl.node('B', position=[5,5], waypoint_range=mobility_range, speed_range=speeds)
```

The changes we've made are to tell the **`mobility`** package that we're using random waypoint mobility, defining a rectangular range of motion for the nodes, and a range of speeds with which the nodes should move. The call to `sim.seed()` seeds the random number generator.

The bounds of the mobility range are the points $(0, 0)$, $(0, 40)$, $(40, 40)$, and $(40, 0)$, where all values are in meters. The node speeds are distributed uniformly between 2m/s and 5m/s. A different waypoint or speed range could be provided for each node, if desired. There is also a more complex waypoint range that defines a parallelogram. See

```
>>> help(pydtn.mobility.ParallelogramRange)
```

for details.

Just as in the **`network`** package, **`mobility`** provides a hook to call a function for all nodes. This allows us to attach an application in essentially the same way as before:

```
import pydtn
import pydtn.mobility as mbl
import pydtn.mobility.rf as rf
import pydtn.mobility.epidemic as ep
import pydtn.sampleapp

mbl.random_waypoint()
sim.seed()
rf.set_range(5)
rf.set_latency(0)
rf.set_bandwidth(11)

mobility_range = mbl.SimpleRectangleRange( lower_corner=(0,0), upper_corner=(40,40) )
speeds = mbl.SpeedRange( 2, 5 )
a = mbl.node('A', position=[0,0], waypoint_range=mobility_range, speed_range=speeds)
b = mbl.node('B', position=[5,5], waypoint_range=mobility_range, speed_range=speeds)

mbl.applyToNodes(pydtn.sampleapp.attach)
```

Note that **`epidemic`** configures all nodes with its own custody policy. That means we're ready to send data through the simulation:

```
import pydtn
import pydtn.mobility as mbl
import pydtn.mobility.rf as rf
import pydtn.mobility.epidemic as ep
import pydtn.sampleapp

pydtn.config('bundle_lifetime',1000)
mbl.random_waypoint()
sim.seed()
rf.set_range(5)
rf.set_latency(0)
rf.set_bandwidth(11)

mobility_range = mbl.SimpleRectangleRange( lower_corner=(0,0), upper_corner=(40,40) )
speeds = mbl.SpeedRange( 2, 5 )
a = mbl.node('A', position=[0,0], waypoint_range=mobility_range, speed_range=speeds)
b = mbl.node('B', position=[5,5], waypoint_range=mobility_range, speed_range=speeds)

mbl.applyToNodes(pydtn.sampleapp.attach)

pydtn.sampleapp.send(a.node,b.node,'hello world',0)

sim.stopAt(100)
sim.run()
sys.exit()
```

We've set the bundle lifetime to 1000s, since 1s is far too short for successful delivery in this mobile scenario. Typical verbose output might look like:

```
Random number seed = 3617528207 (d79f198f)
(t={0,0}) node 41 stored a bundle, persistent store now 50 bytes
(t={0,0}) sending 30 bytes from 41 to 41
(t={0,0}) sending 30 bytes from 42 to 42
(t={0,22}) dropping bundle of size 50 at node 41 (no route found)
(t={5,703867}) sending 30 bytes from 41 to 41
(t={6,503732}) sending 30 bytes from 42 to 42
(t={15,244755}) sending 30 bytes from 41 to 41
(t={17,351542}) sending 30 bytes from 42 to 42
```

```
(t={24,132747}) sending 30 bytes from 41 to 41
(t={27,101525}) sending 30 bytes from 42 to 42
(t={27,101547}) received 30 bytes at 41 from 42  (broadcast)
sending summary vector from A to B [0x050000000041]
(t={27,101547}) sending 45 bytes from 41 to 42
(t={27,101547}) node 41 UNKNOWN BUNDLE TYPE
(t={27,101580}) received 45 bytes at 42 from 41
(t={27,101580}) node 42 consuming data bundle 5 from 41
   7 bytes
   00 05 00 00 00 00 41
processing a summary vector at B from A
sending data request from B to A [0x01050000000041]
(t={27,101580}) sending 45 bytes from 42 to 41
(t={27,101613}) sending 30 bytes from 42 to 41
(t={27,101613}) received 45 bytes at 41 from 42
(t={27,101613}) node 41 consuming data bundle 4 from 42
   7 bytes
   01 05 00 00 00 00 41
(t={27,101613}) sending 50 bytes from 41 to 42
(t={27,101635}) received 30 bytes at 41 from 42
(t={27,101635}) node 41 received an ACK for 5
(t={27,101650}) sending 30 bytes from 41 to 42
(t={27,101650}) received 50 bytes at 42 from 41
(t={27,101650}) node 42 consuming data bundle 0 from 41
   11 bytes
   68 65 6C 6C 6F 20 77 6F 72 6C 64
sample app: got a bundle of size 50
(t={27,101650}) sending 30 bytes from 42 to 41
(t={27,101672}) received 30 bytes at 42 from 41
(t={27,101672}) node 42 received an ACK for 4
(t={27,101672}) received 30 bytes at 41 from 42
(t={27,101672}) node 41 received an ACK for 0
(t={36,531371}) sending 30 bytes from 41 to 41
(t={36,926469}) sending 30 bytes from 42 to 42
(t={46,924481}) sending 30 bytes from 41 to 41
(t={46,924503}) received 30 bytes at 42 from 41  (broadcast)
(t={46,924503}) node 42 UNKNOWN BUNDLE TYPE
(t={51,343437}) sending 30 bytes from 42 to 42
(t={57,945781}) sending 30 bytes from 41 to 41
(t={60,467480}) sending 30 bytes from 42 to 42
(t={64,736603}) sending 30 bytes from 41 to 41
(t={70,675584}) sending 30 bytes from 42 to 42
(t={75,513601}) sending 30 bytes from 41 to 41
(t={76,505457}) sending 30 bytes from 42 to 42
(t={87,157533}) sending 30 bytes from 41 to 41
(t={89,753775}) sending 30 bytes from 42 to 42
(t={98,151097}) sending 30 bytes from 41 to 41
(t={98,620267}) sending 30 bytes from 42 to 42
```

There is a lot more going on here than in the static case. To start with, the data bundle is created and stored at the beginning of the simulation:

```
(t={0,0}) node 41 stored a bundle, persistent store now 50 bytes
```

and is almost immediately dropped:

```
(t={0,22}) dropping bundle of size 50 at node 41 (no route found)
```

This is an artifact of the epidemic forwarding algorithm, which does not provide routes specifically for individual bundles. When the bundle can be forwarded, it will be pulled from the persistent store and enqueued.

In addition, there are many lines of the form

```
... sending 30 bytes from 41 to 41
```

These are broadcast messages that serve as heartbeats. The purpose of these is to announce a node's existence to any other nodes in range, so that neighbor lists might be constructed. We see receipt of one of these in

```
(t={27,101547}) received 30 bytes at 41 from 42  (broadcast)
```

Recall that '41' is ASCII for 'A'. This then triggers:

```
sending summary vector from A to B [0x050000000041]
(t={27,101547}) sending 45 bytes from 41 to 42
(t={27,101547}) node 41 UNKNOWN BUNDLE TYPE
```

The epidemic algorithm at *A* sends a summary vector, which is shown as a hexadecimal string, to *B*. After the epidemic algorithm has handled the broadcast bundle, it's passed to the normal verbose output mechanism, which simply reports an unrecognized format (the broadcast is not flagged as a data bundle).

The next step of the exchange is

```
(t={27,101580}) received 45 bytes at 42 from 41
(t={27,101580}) node 42 consuming data bundle 5 from 41
   7 bytes
   00 05 00 00 00 00 41
processing a summary vector at B from A
sending data request from B to A [0x01050000000041]
(t={27,101580}) sending 45 bytes from 42 to 41
(t={27,101613}) sending 30 bytes from 42 to 41
```

Here we see *B* receive the data from *A*, compose its response, and send that response back to *A*. The last line is an acknowledgment of the summary vector data bundle, which is enqueued after the response. Obviously, this ordering of messages is an artifact of a zero-processing-time assumption.

Note that after the exchange of data, *A* and *B* are out of range. Later, they are once again in range. However, there is no summary vector exchange. While this might seem odd at first glance, the neighborhood information is never perfect, and so nodes will not notice one another's departure until they are out of range for a sufficient length of time. *A* and *B* come back into range before this timeout has occurred.

3.3 Data Collection

Pydtn provides a number of ways to collect data during your simulation. Each has its uses, though there is some duplication of functionality.

3.3.1 Global Statistics

Nodes have a built-in bundle data gatherer that tracks some basic statistical data. It is invoked by having the following in your simulation script:

```
import pydtn

pydtn.stats('file.stats', 1000)
```

The first argument is the file into which the statistical data should be written, and the second argument is the time in microseconds between outputs.

An example of the output from this is:

```
{0,541000} originated unique 0 0 nan±nan 1016 159267 156.759±1058.57
{0,541000} originated duplicate 0 0 nan±nan 1006 46294 46.0179±0.327163
{0,541000} delivered unique 0 0 nan±nan 1012 118167 116.766±847.782
{0,541000} delivered duplicate 0 0 nan±nan 274 12610 46.0219±0.361811
{0,541000} delivered hops 1286 2572 2±0 1286 2572 2±0
{0,541000} sent 0 0 nan±nan 5606 416768 74.3432±578.969
{0,541000} received 0 0 nan±nan 5606 416768 74.3432±578.969
{0,541000} dropped bytes 0 0 nan±nan 1936 11770580 6079.85±nan
{0,541000} dropped hops 1936 775 0.40031±0.489961 1936 775 0.40031±0.489961
{0,541000} fdelivered 0.996063±0.00196463
{0,541000} fdropped 1.50078±nan
{0,541000} storage 133575 20484096 0.0130444
{0,541000} exhausted 0 0
```

All lines begin with the time at which the current time slice ended. The next thing on the line is the category of data being reported, for example bundles delivered that duplicate already-delivered bundles. For most categories, what follows is two sets of triples, one for the current time slice, and the other for the total run of the simulator to the current time. The triples are, in order, the number of bundles, the total size of these bundles (in bytes), and the average bundle size with its standard deviation. `fdelivered` is the fraction of bundles created that have been delivered, while `fdropped` is the fraction of bundles created that have been dropped *at some point*. It is not uncommon for `fdropped` to be greater than 1, if a bundle has to be sent by either its originator or an intermediate custodian multiple times before successful delivery. `storage` gives the current usage of stable storage across the entire network at the recorded time, which `exhausted` records whether any node has exhausted its stable storage at any point during the current time slice or since the beginning of the simulation.

Generally, the statistics reported are useful only in debugging an algorithm or scenario. Generally, you will want to employ a more tailored approach to data aggregation, though the existing code might be a useful place to start.

3.3.2 Tracers

The statistics-gathering mechanism is incapable of distinguishing between operations at different nodes. Often, this will be critical information for your experiments. Tracers can fill this need.

Tracers are hooked into the various bundle-handling operations at a node. When a particular operation is invoked, the tracer's hook is called with the bundle and node as arguments. As a result, the tracer can output bundle-specific and node-specific information.

There are a two tracers that might be of interest:

`flowtrace` This traces when bundles are created, delivered, or dropped. Enough information is given to uniquely identify the bundle, and generally if part of a data flow this too is deducible.

`storetrace` This traces when a node's stable storage changes. That is, when a bundle is added to or removed from stable storage.

Both of these are configured in the same way:

```
import pydtn.flowtrace

pydtn.flowtrace.setup("example.flow")
```

The argument to `setup()` is the file into which the trace is written.

As with the statistics collection, these tracers are unlikely to provide the functionality you need beyond early-stage testing. However, in many cases you should be able to adapt one or the other of these to generate more relevant data.

What distinguished tracers from the collection methods to follow is that tracing is triggered by operations on bundles when they occur, while we will now move on to specifically scheduled data collection methods.

3.3.3 `sim.collect()`

Any entity can define a default data-collection behavior. Nodes, for example, have a list of bundle-data collector objects that have been registered, each of which is called as part of data collection.

When `sim.collect()` is called, every entity created by the simulator has its data collection method invoked. To see this in operation, add

```
import pydtn.storeprofile

pydtn.storeprofile.setup("example.storprof")
```

to your script before any nodes have been created, and add

```
sim.collect()
```

after `sim.run()` has returned. For each node, you will get the total number of bytes of data and acknowledgement bundles in its volatile storage (`qd` and `qa`) and stable storage (`sd` and `sa`). In essence, `sim.collect()` gives you a fairly simple way of recording data available from C++.

3.3.4 Entity.get()

Some entities, such as nodes and links, provide access to a great deal of information directly from python. This is done through the `get()` method. Rather than calling `sim.collect()` to gather data from every entity, you might want to access data from only the nodes using something like the following:

```
def gather(n):
     volatile_used = n.get('volatile_used')
     stable_used = n.get('persistent_used')
     print "%s %d %d" % (n.get('addr'), volatile_used[0], stable_used[0])

mbl.applyToNodes(gather)
```

For nodes defined as in the script at the end of Section 3.2, the following data is available through `get()`:

addr The node's address.

bundle_lifetime The default time-to-live for newly created bundles, in seconds.

resend_period The time in seconds between attempts to re-send bundles held in stable storage.

volatile_capacity The capacity in bytes of the node's volatile storage.

persistent_capacity The capacity in bytes of node's stable storage.

volatile_used The number of bytes of volatile storage currently in use.

persistent_used The number of bytes of stable storage currently in use.

position A three-vector with the node's current coordinates

beacon_interval The time in seconds between neighbor-discovery beacons.

last_beacon The time in timeval format of the last neighbor-discovery beacon sent.

speed_range A two-element list `[min_speed,max_speed]`, both in meters per second.

Links provide somewhat less information:

bandwidth The link's bandwidth, in Mbps.

latency The link's latency, in seconds.

state A string, either `'up'` or `'down'`.

range The range of the antenna, in meters.

visible A list of addresses for the nodes currently in range.

3.3.5 Scheduled Python Function Calls

There is limited utility in collecting data only at points when the simulation is not actively running. Consequently, it is useful to schedule data collection as a simulation event. There are two easy ways to do this, using a mechanism similar to that of the previous section.

If you want to run 10 seconds of simulation to damp out transients, take one measurement, and then finish the simulation before taking another measurement, you might do the following:

```
sim.stopAt(10)
sim.run()
mbl.applyToNodes(gather)
sim.stopAt(sim_duration)
sim.run()
mbl.applyToNodes(gather)
```

A simpler way to accomplish this is with the `sim.schedule()` method:

```
def gather_all():
   mbl.applyToNodes(gather)
sim.stopAt(sim_duration)
sim.schedule(10,gather_all)
sim.run()
gather_all()
```

or

```
sim.stopAt(sim_duration)
sim.schedule(10, lambda: mbl.applyToNodes(gather))
sim.run()
mbl.applyToNodes(gather)
```

More likely, you will want to perform data collection periodically throughout the simulation. We have already seen this in Section 3.1.4 with the `pydtn.periodically()` method:

```
sim.stopAt(sim_duration)
pydtn.periodically(start_time=10, interval=1, function=lambda: mbl.applyToNodes(gather))
sim.run()
mbl.applyToNodes(gather)
```

This begins calling `gather()` on all nodes at 10 seconds, repeating the call every second thereafter. The final call after `sim.run()` returns ensures that the final data collection is performed.

3.3.6 Writing Output to Files

The pre-defined data collectors with which we began our discussion wrote to files, while the collectors we have more recently discussed wrote to the terminal. We can, of course, write to files from python as well, which is very useful when different types of output are being generated.

In python, we open a file with a call like:

```
f = open( fname, 'w' )
```

where `fname` is a string variable containing the filename (we could also subtitute a string literal). Writing to the file can be done with:

```
f.write('Hello world\n')
print >>f, 'Hello world'
sys.stdout = f
print 'Hello world'
```

Our output file would then contain:

```
Hello world
Hello world
Hello world
```

Resetting `sys.stdout` is probably not what you want to do, unless you always want to capture the full output of the simulator (except for errors). The second form is probably the most useful.

We will now put everything together into one complete script that collects data and writes it into a file:

```
import pydtn
import pydtn.mobility as mbl
import pydtn.mobility.rf as rf
import pydtn.mobility.epidemic as ep
import pydtn.sampleapp

fname = 'outfile'
seed = None
for clArg in sim.args():
     exec(clArg)
ofile = open(fname,'w')

def gather(n):
     volatile_used = n.get('volatile_used')
     stable_used = n.get('persistent_used')
     tl = sim.time()
     t = tl[0] + 0.000001*tl[1]
     print >>ofile,"%f %s %d %d" % (t, n.get('addr'), volatile_used[0], stable_used[0])

mbl.random_waypoint()
if seed is None:
     sim.seed()
else:
     sim.seed(seed)
rf.set_range(5)
rf.set_latency(0)
rf.set_bandwidth(11)

mobility_range = mbl.SimpleRectangleRange( lower_corner=(0,0), upper_corner=(40,40) )
speeds = mbl.SpeedRange( 2, 5 )
a = mbl.node('A', position=[0,0], waypoint_range=mobility_range, speed_range=speeds)
b = mbl.node('B', position=[5,5], waypoint_range=mobility_range, speed_range=speeds)

mbl.applyToNodes(pydtn.sampleapp.attach)

pydtn.sampleapp.send(a.node,b.node,'hello world',0)

sim.stopAt(30)
pydtn.periodically(start_time=10, interval=1, function=lambda: mbl.applyToNodes(gather))
sim.run()
mbl.applyToNodes(gather)
sys.exit()
```

If we save this script as `test.py` and run it with (noting the escaped quotes):

```
$ simlpy test.py -a seed=123 -a fname=\'output.file\'
```

the contents of `output.file` are:

```
10.000000 A 0 50
10.000000 B 0 0
11.000000 A 0 50
11.000000 B 0 0
...
30.000000 A 0 50
30.000000 B 0 0
```

Part II

Extending the Simulator

Chapter 4

Writing Modules

There are two ways of getting the simulator to do what you want: *scripts* and *modules*. The former are useful for setting up specific scenarios and individual tests. The latter allow you to extend the functionality of the simulator.

A module for simlpy is essentially just a module for python, though it has access to simlpy's functionality as well. The entire DTN simulation framework is, in fact, a set of simlpy modules. Modules can be written in two ways: in python or in C++. Actually, C++ modules will typically have a small amount of python as well, but we'll discuss that later.

4.1 Python Modules

Writing a module in python can be as simple as writing a single python file, or as complex as creating a directory hierarchy of interconnected python files. The **`pydtn`** module contains two python submodules: **`Dijkstra`** and **`network`**. We've already seen the **`network`** module; the **`Dijkstra`** module is called by **`network`** to perform shortest-path route computation.

In this section we'll see how to construct a simple data collection module for a mobile network, based on what we saw in Section 3.3. We'll call our module **`gather`**, and will include it in a script using

```
import gather
```

When the python interpreter reads this import statement, it will look for `gather.pyc`, `gather.py`, or `gather.so`. A file ending in `.pyc` is a compiled python file—this will be generated automatically if there is a newer file with the same base ending in `.py`. A `.so` file is a compiled (C or C++) module. In our case, we will be working with the file `gather.py`.

4.1.1 Preliminaries

This module is specific to **`pydtn`** and relies on a mobile network, so we will need to import both of these modules:

```
gather.py

import pydtn
import pydtn.mobility
```

We have the dependencies specified, which is an important start, but how do we get meaningful documentation into the module? That is, if we type

```
>>> help(gather)
```

how do we construct the text to display? This is done using python's triple-quoting syntax:

```
gather.py

"""Data gathering module.

This module will periodically collect information about a mobile
network and write it to a file."""

import pydtn
import pydtn.mobility
```

The quoted text (which must be the first thing in the file) will be displayed verbatim in simlpy as the module's help text.

4.1.2 Configuration

We will want to let users control certain aspects of the module. In particular, output will go to a file, and the data collection will be done periodically. First, we'll set up the variables, along with any default values.

```
gather.py

"""Data gathering module.

This module will periodically collect information about a mobile
network and write it to a file."""

import pydtn
import pydtn.mobility

_outfile = None  # no default output option
_start_time = 0  # start at t=0 by default
_period = None   # no default period
```

Now we want to provide methods to set these parameters:

gather.py

```
"""Data gathering module.

This module will periodically collect information about a mobile
network and write it to a file."""

import pydtn
import pydtn.mobility

_outfile = None  # no default output option
_start_time = 0  # start at t=0 by default
_period = None   # no default period

def file( f ):
   """Set the output file.

   If the argument passed is a string, open a file with that
   name for writing.  Otherwise, the argument should be a
   writable file object."""
   import types
   global _outfile
   if type(f) in types.StringTypes:
      _outfile = open(f,'w')
   else:
      _outfile = f

def start(t):
   """Set the time at which data collection should begin.

   This is when the first data point will be taken.  If not
   specified, collection will begin at t=0."""
   global _start_time
   _start_time = t

def period(t):
   """Set the periodicity of data collection.

   Data will be collected every t seconds."""
   global _period
   _period = t
```

There are a few things to note here. First is that the module variables have to be declared `global` in the functions, or else only local versions are modified. Second is that we use the **`types`** module in `file()` to check the input parameter type, but we only need to import it locally in the function. Third is that we provide function-specific documentation using the same triple-quoted format that we used for the module as a whole, again at the very beginning of what we're documenting.

4.1.3 Collection

To keep things simple, we'll use the same data collection method we defined earlier:

gather.py

```
"""Data gathering module.

This module will periodically collect information about a mobile
network and write it to a file."""

import pydtn
import pydtn.mobility

_outfile = None  # no default output option
_start_time = 0  # start at t=0 by default
_period = None   # no default period

def file( f ):
   """Set the output file.

   If the argument passed is a string, open a file with that
   name for writing.  Otherwise, the argument should be a
   writable file object."""
   import types
   global _outfile
   if type(f) in types.StringTypes:
      _outfile = open(f,'w')
   else:
      _outfile = f

def start(t):
   """Set the time at which data collection should begin.

   This is when the first data point will be taken.  If not
   specified, collection will begin at t=0."""
   global _start_time
   _start_time = t

def period(t):
   """Set the periodicity of data collection.

   Data will be collected every t seconds."""
   global _period
   _period = t

def collect(n):
   volatile_used = n.get('volatile_used')
   stable_used = n.get('persistent_used')
   tl = sim.time()
   t = tl[0] + 0.000001*tl[1]
   print >>_outfile,"%f %s %d %d" % (t, n.get('addr'), volatile_used[0], stable_used[0])
```

4.1.4 Triggering

At this point, all that we're missing is a way to set this going. We'll create a method `gather.enable()` to do this:

gather.py

```
"""Data gathering module.

This module will periodically collect information about a mobile
network and write it to a file."""

import pydtn
import pydtn.mobility

_outfile = None  # no default output option
_start_time = 0  # start at t=0 by default
_period = None   # no default period

def file( f ):
   """Set the output file.

   If the argument passed is a string, open a file with that
   name for writing.  Otherwise, the argument should be a
   writable file object."""
   import types
   global _outfile
   if type(f) in types.StringTypes:
      _outfile = open(f,'w')
   else:
      _outfile = f

def start(t):
   """Set the time at which data collection should begin.

   This is when the first data point will be taken.  If not
   specified, collection will begin at t=0."""
   global _start_time
   _start_time = t

def period(t):
   """Set the periodicity of data collection.

   Data will be collected every t seconds."""
   global _period
   _period = t

def collect(n):
   volatile_used = n.get('volatile_used')
   stable_used = n.get('persistent_used')
   tl = sim.time()
   t = tl[0] + 0.000001*tl[1]
   print >>_outfile,"%f %s %d %d" % (t, n.get('addr'), volatile_used[0], stable_used[0])

def enable():
   if _outfile is None:
      raise ValueError('No output file specified.')
   if _period is None:
      raise ValueError('No period specified.')
   pydtn.periodically( start_time=_start_time, interval=_period,
                       function=lambda: pydtn.mobility.applyToNodes(collect) )
```

4.1.5 Using the Module

We now have a complete module. To use it, we could modify our earlier script as follows:

```
test.py

import pydtn
import pydtn.mobility as mbl
import pydtn.mobility.rf as rf
import pydtn.mobility.epidemic as ep
import pydtn.sampleapp
import gather

fname = 'outfile'
seed = None
for clArg in sim.args():
     exec(clArg)
gather.start_time(10)
gather.period(1)
gather.file(fname)

mbl.random_waypoint()
if seed is None:
     sim.seed()
else:
     sim.seed(seed)
rf.set_range(5)
rf.set_latency(0)
rf.set_bandwidth(11)

mobility_range = mbl.SimpleRectangleRange( lower_corner=(0,0), upper_corner=(40,40) )
speeds = mbl.SpeedRange( 2, 5 )
a = mbl.node('A', position=[0,0], waypoint_range=mobility_range, speed_range=speeds)
b = mbl.node('B', position=[5,5], waypoint_range=mobility_range, speed_range=speeds)

mbl.applyToNodes(pydtn.sampleapp.attach)

pydtn.sampleapp.send(a.node,b.node,'hello world',0)

gather.enable()
sim.stopAt(30)
sim.run()
mbl.applyToNodes( gather.collect ) # get the end-of-simulation data
sys.exit()
```

While the final script is not much shorter than what we had before, it doesn't change if we make **`gather`** more complex. In addition, we can reuse the same data collection module in other scripts without having to copy and paste code, and then worrying about maintaining consistency.

Other modules that you might want to write in python include things like topology configuration or traffic generation. Anything you plan to use essentially unmodified (or with minimal configuration) in multiple scripts is a reasonable candidate to turn into a python module.

4.2 C++ Modules

Generally, modules written in C++ will be very different in nature than those written in python. However, to illustrate the process of creating a C++ module, we will recreate our python module from the previous section in C++.

Because we have to compile code, we add a good bit of complexity by moving from python to C++. While the number of files will grow, and we will have to start dealing with language-to-language interfaces, the result in this case should still be reasonably easy to follow. We will, at a minimum, need two files in our module: a source code file and a `Makefile`. The latter turns our source code into a loadable module for simlpy.

4.2.1 The Source Code

For this module, there will be only one source code file. In keeping with the structure of other pydtn modules, we will call this file `python_init.cc`. The bare minimum that we need to make this a valid module loadable in python is:

`python_init.cc`

```
#include "simlpy/interpreter_defs.h"

#ifndef PyMODINIT_FUNC
#define PyMODINIT_FUNC extern "C" void
#endif

static PyMethodDef ExampleMethods[] =
{
   {0,0,0,0} // sentinel
};

PyMODINIT_FUNC
initgather(void)
{
   (void) Py_InitModule("gather",ExampleMethods);
}
```

Strictly speaking, this is a little more than we need. We've included a few things that aren't yet being used, such as simlpy's python definitions, as well as an empty list of python-callable methods. We're about to use them, so there is little sense in defining the truly minimal file and immediately changing it. One important thing to note is that there is a very tight binding between the name of the initialization function, the first argument that it passes to `Py_InitModule`, and the name of the shared object that will be generated. Namely, if in python you call

```
>>> import gather
```

and the module is loaded from `gather.so`, the interpreter will look for a function called `initgather`. Any variation of this will **not work**. Similarly, since the module is imported under the name **`gather`**, that must be passed as the first argument to `Py_InitModule`.

Configuration

Following the development pattern of the previous section, we will first define the configuration variables and methods to set them. We start by adding a few more header files.

```
python_init.cc

#include "simlpy/interpreter_defs.h"
#include "simlpy/interpreter_hooks.h"
#include <iostream>
#include <fstream>
```

The file `interpreter_hooks.h` contains helper functions for parsing the arguments that python passes to us. The other two files are standard C++ I/O headers.

Next we define the module's variables.

```
python_init.cc

#ifndef PyMODINIT_FUNC
#define PyMODINIT_FUNC extern "C" void
#endif

static std::ostream*  gather_outfile = 0;
static Time           gather_start_time(0,0);
static Time           gather_period;
static bool           gather_period_set = false;
```

We keep track of whether the output file has been configured by using a pointer. For the period, we have to keep track of whether it has been configured with an additional boolean. The `Time` structure (which is included through `interpreter_hooks.h`) is a wrapper around the standard C `timeval`.

Now we have to define the functions. This should be done somewhere after the variables are declared and before the definition of `ExampleMethods`. We'll start with the easiest one, which sets the start time to something other than the default.

python_init.cc

```
static PyObject*
gather_set_start_time( PyObject* self, PyObject* args )
{
   Entity::ArgList argList;
   build_arglist( argList, args );
   if ( argList.size() < 1 )
   {
      std::cerr << "start_time() takes a single argument" << std::endl;
      return 0;
   }
   try
   {
     gather_start_time = parse_time( argList[0] );
   }
   catch ( ... )
   {
      clean_arglist( argList );
      return 0;
   }
   clean_arglist( argList );
   Py_INCREF(Py_None);
   return Py_None;
}
```

The ArgList class is provided by simlpy. parse_time() converts any valid python time expression (as defined by simlpy) into a Time structure. Functions called by python must return a value. We have nothing meaningful to return, so we return None, which in C++ is referred to as Py_None, after incrementing its reference count. On error, we return a null pointer, which causes python to raise an exception.

Slightly more complicated is setting the period. The only substantial difference is that we have to also set the boolean configuration-tracking variable to true.

python_init.cc

```
static PyObject*
gather_set_period( PyObject* self, PyObject* args )
{
   Entity::ArgList argList;
   build_arglist( argList, args );
   if ( argList.size() < 1 )
   {
      std::cerr << "period() takes a single argument" << std::endl;
      return 0;
   }
   try
   {
     gather_period = parse_time( argList[0] );
     gather_period_set = true;
   }
   catch ( ... )
   {
      clean_arglist( argList );
      return 0;
   }
   clean_arglist( argList );
   Py_INCREF(Py_None);
   return Py_None;
}
```

The last function we need to define is the one that sets the output file. Unlike the python version, we will only accept a file name as a valid argument.

```
python_init.cc

static PyObject*
gather_set_outfile( PyObject* self, PyObject* args )
{
   Entity::ArgList argList;
   build_arglist( argList, args );
   if ( argList.size() < 1 )
   {
      std::cerr << "file() takes a single argument" << std::endl;
      return 0;
   }
   try
   {
     std::string fname = parse_string(argList[0]);
     if ( 0 != gather_outfile ) delete gather_outfile;
     gather_outfile = new std::ofstream(fname.c_str());
   }
   catch ( ... )
   {
      clean_arglist( argList );
      return 0;
   }
   clean_arglist( argList );
   Py_INCREF(Py_None);
   return Py_None;
}
```

Note the use of `parse_string()`, which is another convenience function provided by simlpy.

The final change we need to make is to export the functions that we've defined to python. This is done via the `ExampleMethods` array:

```
python_init.cc

static PyMethodDef ExampleMethods[] =
{
   {"start_time",gather_set_start_time,METH_VARARGS,
     "Set the time at which data collection should begin."},
   {"period",gather_set_period,METH_VARARGS,"Set the periodicity of data collection."},
   {"file",gather_set_outfile,METH_VARARGS,"Set the output file."},
   {0,0,0,0} // sentinel
};
```

The structure of the elements should be fairly obvious. We begin with the string that will become the function name in python. This is followed by the function pointer for the function to call. `METH_VARARGS` tells python that the arguments should be converted into a tuple when passed to the function. The final field is the documentation string, as if it were specified in python with the triple-quote syntax.

Collection

Now we need to implement the data collection in C++. We'll be dealing with mobile nodes, so we need to include the appropriate header. We will also need to be able to access the simulator's

clock.

```
python_init.cc

#include "simlpy/interpreter_defs.h"
#include "simlpy/interpreter_hooks.h"
#include "simlpy/Clock.h"
#include "mobility/parsers.h"
#include "mobility/MobileNode.h"
#include <iostream>
#include <fstream>
```

The functions that we'll call from the node are, in fact, defined by the base class, but by extracting them as mobile nodes gives us a few more options.

Here is our data collection function:

```
python_init.cc

static PyObject*
gather_collect( PyObject* self, PyObject* args )
{
   Entity::ArgList argList;
   build_arglist( argList, args );
   if ( argList.size() < 1 )
   {
      PyErr_SetString(PyExc_TypeError,"collect() takes a single argument");
      return 0;
   }
   if ( 0 == gather_outfile )
   {
      std::cerr << "No output file configured for gather" << std::endl;
      return 0;
   }
   try
   {
      const Time& t = Clock::time();
      double dt = t.tv.tv_sec + 1.0*t.tv.tv_usec/Time::MILLION;
      Mobility::MobileNode* mn = Mobility::parse_mn( argList[0] );
      (*gather_outfile) << dt << " " << mn->addrString() << " "
                        << mn->node().usedVolatileCap() << " "
                        << mn->node().usedPersistentCap() << std::endl;
   }
   catch ( ... )
   {
      clean_arglist( argList );
      return 0;
   }
   clean_arglist( argList );
   Py_INCREF(Py_None);
   return Py_None;
}
```

The output from this should be identical to that of the python function we defined previously.

The function must be added to the list of exported methods, of course:

python_init.cc

```
static PyMethodDef ExampleMethods[] =
{
   {"start_time",gather_set_start_time,METH_VARARGS,
     "Set the time at which data collection should begin."},
   {"period",gather_set_period,METH_VARARGS,"Set the periodicity of data collection."},
   {"file",gather_set_outfile,METH_VARARGS,"Set the output file."},
   {"collect",gather_collect,METH_VARARGS,"Collect data from a mobile node."},
   {0,0,0,0} // sentinel
};
```

Triggering

Triggering the data collection is a bit more complex. This is because the list of mobile nodes is maintained by the **mobility** module in python, not C++. Consequently, we don't have a good way to access it. We could invoke the python interpreter from within our code, using **mobility**'s functionality directly. Instead, we'll use another facility that is provided by simlpy.

The simulator provides a list of all entities that it has created through the python interface. We can use the rather inefficient error-handling mechanism in `gather_collect()` to gather data from only the mobile nodes.

python_init.cc

```
static PyObject*
gather_collect_all( PyObject* self, PyObject* args )
{
   if ( 0 == gather_outfile )
   {
      std::cerr << "No output file configured for gather" << std::endl;
      return 0;
   }

   PyObject* t = PyTuple_New(1);
   unsigned int nument = num_entities();
   for ( unsigned int i = 0 ; i < nument ; ++i )
   {
      InterpreterItem ent = get_entity(i);
      Py_INCREF(ent);
      if ( 0 == PyTuple_SetItem(t,0,ent) )
      {
         PyObject* r = gather_collect(0,t);
         if ( 0 != r )
         {
           Py_DECREF(r);
         }
      }
   }
   Py_DECREF(t);

   Py_INCREF(Py_None);
   return Py_None;
}
```

There's a lot in this small block of code, so let's take a look in some detail. The two hooks `num_entities()` and `get_entity()` are provided by `interpreter_hooks.h`. They

give us a way to iterate over the simulator's entities, as shown in the `for` loop. Because the function `gather_collect()` expects a tuple as its second argument, we create one called `t` using `PyTuple_New()`. The argument to this function is the initial size of the tuple. Doing this outside of the loop reduces the amount of memory allocation and deallocation that we'll need. Now we have to add the entity `ent` that we obtained from `get_entity()` to `t`. The tuple will *steal* the reference to `ent`, so we first increment the reference count for `ent`. Calling `PyTuple_SetItem()` adds `ent` to `t`, and returns 0 if there were no errors. The previous item at that position will be replaced. At this point, we're ready to call `gather_collect()`. Because a successful call will return a reference-incremented `Py_None`, we must decrement the reference count of the return value (unless it was a NULL pointer). Once the loop is done, we have to clean up the tuple `t` by decrementing its reference count.

Now all we need to do is define the `enable` function and hook these last two functions into python. This involves adding a couple more header files:

`python_init.cc`

```
#include "simlpy/interpreter_defs.h"
#include "simlpy/interpreter_hooks.h"
#include "simlpy/Clock.h"
#include "mobility/parsers.h"
#include "mobility/MobileNode.h"
#include "pydtn/TrafficGenerator.h"
#include "pydtn/UniformTimeDist.h"
#include <iostream>
#include <fstream>
```

Defining a new "bundle maker" class:

`python_init.cc`

```
class GatherCollector : public BundleMaker
{
   public:
      GatherCollector() {}
      virtual ~GatherCollector() {}
      void generate() {
         PyObject* o = gather_collect_all(0,0);
         if ( 0 != o ) Py_DECREF(o);
      }
};
```

Writing the function to set things in motion:

python_init.cc

```
static PyObject*
gather_enable( PyObject* self, PyObject* args )
{
   if ( ! gather_period_set )
   {
      std::cerr << "No period configured for gather" << std::endl;
      return 0;
   }
   if ( 0 == gather_outfile )
   {
      std::cerr << "No output file configured for gather" << std::endl;
      return 0;
   }
   TrafficGenerator* tg = new TrafficGenerator();
   tg->setTimeDistribution( new UniformTimeDist(gather_period) );
   tg->setBundleMaker( new GatherCollector );
   Clock::schedule( new TrafficTrigger(tg,tg,gather_start_time) );

   Py_INCREF(Py_None);
   return Py_None;
}
```

And modifying the list of exported methods:

python_init.cc

```
static PyMethodDef ExampleMethods[] =
{
   {"start_time",gather_set_start_time,METH_VARARGS,
     "Set the time at which data collection should begin."},
   {"period",gather_set_period,METH_VARARGS,"Set the periodicity of data collection."},
   {"file",gather_set_outfile,METH_VARARGS,"Set the output file."},
   {"collect",gather_collect,METH_VARARGS,"Collect data from a mobile node."},
   {"collect_all",gather_collect_all,METH_VARARGS,"Collect data from all mobile nodes."},
   {"enable",gather_enable,METH_VARARGS,"Start the data collection."},
   {0,0,0,0} // sentinel
};
```

A few things here are new, so let's look at them in a bit more detail. The first is our use of `TrafficGenerator` and the `BundleMaker` base class. Obviously, we're not generating traffic, but that was the original intent of the classes. Since there's nothing specifically tied to bundles in the definition of either `TrafficGenerator` or `BundleMaker`, we can in fact define the `generate()` virtual method of a `BundleMaker` subclass to do anything we like. In this case, we're triggering our collect-from-all-nodes function. The arguments to this function aren't used, so we can pass null pointers. We have no interest in the returned object, so we can just decrement the reference count and forget about it.

If we now look at the enable function, after verifying that the module was sufficiently configured, we create a new traffic generator object. We're being somewhat lazy here by defining the variable locally through dynamic allocation, and letting ourselves lose the reference. While this will technically result in a memory leak, the object pointed to by `tg` will live the entire duration of the simulation. We don't have a good clean-up mechanism in any event, so this isn't a big deal. The traffic generator is configured with a simple time distribution object that adds a fixed period to the time at which it's called to determine the time of the next trigger. It is also configured with our

“bundle maker.”

The final difference worth noting is that in the enable function we see our first call to the `Clock::schedule()` method. This is the direct call in C++ corresponding to the `emit()` calls we made in python. The scheduling function takes a pointer to an event object, which contains the entity sending the event, the entity that should handle the event, and the scheduled time for the event. In our case, the sender and handler of the event are both `tg`, and the scheduled time is the configured start time.

Note that we’ve written this module to be completely compatible with the python version. That means the script `test.py` at the end of Section 4.1 can be used unmodified. We’ve defined an extra method, though, so the line

```
mbl.applyToNodes( gather.collect ) # get the end-of-simulation data
```

could be replaced with

```
gather.collect_all() # get the end-of-simulation data
```

4.2.2 Compiling

Before we can use the module, we have to compile it. If you have a single machine (or a set of nearly identical machines) on which you want to use your module, the easiest thing to do is probably to crib a `Makefile` from elsewhere. In this example, we’ll use **`sampleapp`**’s `Makefile`. Most of the changes will be near the beginning of the file, in the variable definitions:

```
Makefile

CXX := g++
SOSUFF := so
PYTHON := python
DOXYGEN := doxygen
INSTALL := /usr/bin/install -c
prefix := /home/user/code/dtn-lts/src
exec_prefix := ${prefix}
libdir := ${exec_prefix}/lib
bindir := ${exec_prefix}/bin
SIM_LIBS := /home/user/code/dtn-lts/src/lib
MODNAME := sampleapp
MODPATH := pydtn
moddir := $(libdir)/$(MODPATH)/$(MODNAME)
LIBRARY := lib$(MODNAME).$(SOSUFF)
SO := _$(MODNAME).$(SOSUFF)
SOSOURCE := python_init.cc
SOURCES := $(filter-out example% $(SOSOURCE),$(wildcard *.cc))
HEADERS := $(wildcard *.h)
OBJECTS := $(patsubst %.cc,%.o,$(SOURCES))
XTRALIBDIRS :=  . $(libdir) $(SIM_LIBS)
XTRAINCDIRS :=  /home/user/code/dtn-lts/src
XTRALIBS :=
LIBRARIES := pydtn dtn
XTRAINCDIRS += $(shell $(PYTHON) -c \
  "import distutils.sysconfig;print distutils.sysconfig.get_config_var('INCLUDEPY')")
INCLUDES := $(patsubst %,-I%,$(XTRAINCDIRS))
LIBDIRS := $(patsubst %,-L%,$(XTRALIBDIRS))
CXXFLAGS := -g -fPIC -Wall $(INCLUDES)
LIBS := $(patsubst %,-l%,$(LIBRARIES)) $(XTRALIBS)
MAKEDEPEND := makedepend
LIBDEPS := $(SIM_LIBS)/libpydtn.$(SOSUFF)
...
install : $(libdir)/$(LIBRARY) $(moddir)/$(SO) $(moddir)/__init__.py
```

Here's our modified version, which changed lines marked in bold:

```
Makefile

CXX := g++
SOSUFF := so
PYTHON := python
DOXYGEN := doxygen
INSTALL := /usr/bin/install -c
prefix := /home/user/code/dtn-lts/src
exec_prefix := ${prefix}
libdir := ${exec_prefix}/lib
bindir := ${exec_prefix}/bin
SIM_LIBS := /home/user/code/dtn-lts/src/lib
MODNAME := gather
MODPATH :=
moddir := $(libdir)/$(MODPATH)
LIBRARY := lib$(MODNAME).$(SOSUFF)
SO := $(MODNAME).$(SOSUFF)
SOSOURCE := python_init.cc
SOURCES := $(filter-out example% $(SOSOURCE),$(wildcard *.cc))
HEADERS := $(wildcard *.h)
OBJECTS := $(patsubst %.cc,%.o,$(SOURCES))
XTRALIBDIRS :=  . $(libdir) $(SIM_LIBS)
XTRAINCDIRS :=  /home/user/code/dtn-lts/src
XTRALIBS :=
LIBRARIES := mobility pydtn dtn
XTRAINCDIRS += $(shell $(PYTHON) -c \
  "import distutils.sysconfig;print distutils.sysconfig.get_config_var('INCLUDEPY')")
INCLUDES := $(patsubst %,-I%,$(XTRAINCDIRS))
LIBDIRS := $(patsubst %,-L%,$(XTRALIBDIRS))
CXXFLAGS := -g -fPIC -Wall $(INCLUDES)
LIBS := $(patsubst %,-l%,$(LIBRARIES)) $(XTRALIBS)
MAKEDEPEND := makedepend
LIBDEPS := $(SIM_LIBS)/libmobility.$(SOSUFF) $(SIM_LIBS)/libpydtn.$(SOSUFF)
...
install : $(libdir)/$(LIBRARY) $(moddir)/$(SO)
```

Of course, some of these variables will be different depending on your installation. The reason that `$SIM_LIBS` and `$libdir` differ, as well as the reason for including the source code directory explicitly in the search path, is that the core simulation packages might not be where you want (or can) build your own extensions. For example, you might have a system-wide installation of the simulator, in `/usr/local`. In general, you would not be able to put your own modules in `/usr/local/lib`, nor put the source code in `/usr/local/src`.

In many cases, you will only run `make`, not `make install`, so the final shared object `gather.so` would not be moved into `$libdir`.

4.2.3 Advanced Options

Adding Python to your Module

Some things are more easily done directly in python. You might also want to have extra features that you want to provide, for instance the sort of simplified node construction and bookkeeping used by the **`mobility`** module. Fortunately, incorporating python into your C++ module is easy.

When we first presented modules, we listed the things for which python looks to resolve an `import` statement. We left one thing out, which is important now. When presented with

```
import gather
```

python will *also* look for a *directory* named “gather” with a file `gather/__init__.py`.

How do we take advantage of this? First, let’s change a few lines in our `Makefile` back to their original values:

```
Makefile

moddir := $(libdir)/$(MODPATH)/$(MODNAME)
LIBRARY := lib$(MODNAME).$(SOSUFF)
SO := _$(MODNAME).$(SOSUFF)
...
install : $(libdir)/$(LIBRARY) $(moddir)/$(SO) $(moddir)/__init__.py
```

We now create a new file in the same directory as our source code:

```
__init__.py

from _gather import *
```

What we’ve done here is to rename our C++ module from “gather” to “˙gather”. When we import the **`gather`** module now, we’ll be calling the python script, which then imports everything from **`_gather`** into its own (**`gather`**’s) namespace.

This python file is just like the `gather.py` file we saw previously. In particular, we can add in module documentation:

```
__init__.py

"""Data gathering module.

This module will periodically collect information about a mobile
network and write it to a file."""
from _gather import *
```

We can also remove the functions `gather_collect_all()` and `gather_enable()`, as well as the class `GatherCollector`, and replace them with pure python:

```
__init__.py

"""Data gathering module.

This module will periodically collect information about a mobile
network and write it to a file."""
from _gather import *
import pydtn
import pydtn.mobility

def collect_all():
   """Collect data from all mobile nodes."""
   pydtn.mobility.applyToNodes(collect)

def enable():
   """Start the data collection."""
   pydtn.periodically( start_time=start_time(), interval=period(), function=collect_all )
```

Again, our script `test.py` doesn't need to change at all.

In addition, we need to modify our C++ source code:

python_init.cc

```
static PyObject*
gather_set_start_time( PyObject* self, PyObject* args )
{
   Entity::ArgList argList;
   build_arglist( argList, args );
   if ( argList.size() < 1 )
   {
      double dt = gather_start_time.tv.tv_sec + 1.0*gather_start_time.tv.tv_usec/Time::MILLION;
      return construct_double(dt);
   }
   try
   {
     gather_start_time = parse_time( argList[0] );
   }
   catch ( ... )
   {
      clean_arglist( argList );
      return 0;
   }
   clean_arglist( argList );
   double dt = gather_start_time.tv.tv_sec + 1.0*gather_start_time.tv.tv_usec/Time::MILLION;
   return construct_double(dt);
}
```

python_init.cc

```
static PyObject*
gather_set_period( PyObject* self, PyObject* args )
{
   Entity::ArgList argList;
   build_arglist( argList, args );
   if ( argList.size() < 1 )
   {
      if ( ! gather_period_set )
      {
         std::cerr << "No period configured for gather" << std::endl;
         return 0;
      }
      double dt = gather_period.tv.tv_sec + 1.0*gather_period.tv.tv_usec/Time::MILLION;
      return construct_double(dt);
   }
   try
   {
     gather_period = parse_time( argList[0] );
     gather_period_set = true;
   }
   catch ( ... )
   {
      clean_arglist( argList );
      return 0;
   }
   clean_arglist( argList );
   double dt = gather_period.tv.tv_sec + 1.0*gather_period.tv.tv_usec/Time::MILLION;
   return construct_double(dt);
}
```

What we've done is to change the behavior of `start_time()` and `period()` so that they return the current values, instead of `None`. Calling them without arguments is no longer an error, it's the way to get the values without specifying new ones. We've used the function `construct_double()` from `simlpy/interpreter_hooks.h`, which turns a C or C++ `double` into a python `Float`.

Autoconf

If your module needs to build on multiple machines, you might want to use *autoconf*. We will not discuss autoconf in any detail here, as it's a rather complicated program. Instead, this section will present a quick recipe for using autoconf starting with existing files.

The purpose of autoconf is to generate files with machine-specific configurations. Typically, your autoconf-enabled module will have two generated files: `Makefile` and `config.h`. You must provide autoconf with skeleton files for this, called `Makefile.in` and `config.h.in`. Most modules have essentially an empty file for the latter. For `Makefile.in`, copy it from another module as you did `Makefile` above, and make the corresponding changes to the skeleton, rather than to `Makefile`.

Generating the machine-specific files uses the `configure` script. This, in turn, is built by autoconf from `configure.ac`. Copy `configure.ac` from another module, such as **basicepidemic**. The only change you should need to make to this particular file is on the very first line. Change the first argument of `AC_INIT` to a short description of your module. Make sure the text is between square braces, or you're likely to encounter an error.

Once you have your modified `configure.ac`, you run the following command:

```
$ autoconf
```

This will generate `configure`, which you can then run as:

```
$ ./configure
```

At this point you can run `make` as usual. You only need to run autoconf to generate `configure` once — the resulting script can be run on any Unix-like system. It is common for packages that use autoconf to include both `configure.ac` (or, for older versions, `configure.in`) and `configure`.

Installation

The modules that come with pydtn are all automatically installed into a single library directory. simlpy is compiled and linked with this directory specified, so any needed shared objects are found by the loader as needed.

For shared objects not in this directory (or one of the standard system directories), the loader has to be told where to look. simlpy will also look in the current directory, but this is very limiting in where you can put things. In addition, the loader reads an environment variable `LD_LIBRARY_PATH` for a (colon-separated) list of directories that should be searched for shared

objects. You can also specify the "-L" flag on the command line to add a directory to the search path. You might need to do both, depending on how things have been installed.

If you have more than a couple of modules that you've written yourself, this quickly becomes cumbersome. To make things simpler, you can install your modules into a common directory. If you've used autoconf, then you can specify the installation directory when you run `configure`. The following are equivalent for this purpose:

```
$ ./configure --prefix=/installation/dir

$ ./configure --exec_prefix=/installation/dir

$ ./configure --libdir=/installation/dir/lib
```

Note that if you specify `libdir`, you can install the modules into any directory, while if you specify one of the prefixes, then `/lib` will be appended to the directory given. If you are not using autoconf, you will have to modify the `libdir` variable directly in `Makefile`.

To place modules in the installation directory, after you compile a module you should then run

```
$ make install
```

This will handle all of the necessary copying for you.

If you are able to, you might want to install your modules in the same installation directory that pydtn uses. This obviates the need to specify "-L" or `LD_LIBRARY_PATH`. If you cannot do this, or you want to keep your modules separate, then you might want to define an alias, or even a special script, to launch simlpy with the correct environment and options.

Chapter 5

A Brief Introduction to Data Structures

Before we proceed further, let's take a look at some of the data structures that you'll encounter through the next several chapters. We've seen some of this in the example code presented, but without any explanation of structure or meaning.

This chapter is organized by the layering of three packages containing the bulk of the interesting classes. The first package that we'll look at has the basic DTN implementation. The second package ties the DTN package into the simulator. The third package adds mobility.

5.1 The `dtn` Library

As mentioned in Chapter 1, we have a DTN implementation that is essentially platform-independent. This implementation is the **`dtn`** library. All of the basic DTN functionality is present in this library, some in largely skeletal form. All of the library's classes and methods reside in the `DTN` namespace.

A listing of all the header files in **`dtn`** reveals a large number of classes. We will begin our discussion with two of these: `Bundle` and `Node`. As you might expect, these will prove to be the two most significant classes in the library. Everything else in the library will tie in to one or (more commonly) both of them.

5.1.1 `Bundle`

A `Bundle` encapsulates the data flowing through the network. It also contains some minimal functionality of its own, relating to the current state of the `Bundle`. Much of the bundle's data is stored in `ByteString`s. These behave essentially like standard C++ strings, but with unsigned character data. A `ByteString` is used for any data that does not have a fixed size, including:

- the source address of the bundle
- the destination address of the bundle
- the address of the most recent custodian

- the last-hop sender's address
- the last-hop receiver's address
- the bundle payload
- the application identifier

Some of these may be empty. We use `ByteString` for addresses because the DTN addresses are more free-form than IP addresses, more akin to FQDNs. A sequence number and a hop count are stored as unsigned 32-bit integers. The time at which the bundle was created and the time at which it expires are stored as `timeval` structures. The final piece of data stored by `Bundle` is a `Bundle::BundleType`, which is typedef'ed to an unsigned character. This contains the bundle's control flags, OR'ed together, which can be any of:

`kData` this bundle contains data

`kACK` this bundle is an acknowledgement

`kNoACK` this bundle should **not** be acknowledged

`kCustodial` request custody transfer for the bundle

`kBcast` this is a broadcast bundle

Most of `Bundle`'s methods are fairly obvious matches to the data listed above. The methods that perform more interesting actions are:

`clone()` returns a newly allocated copy of the existing bundle [virtual]

`expired()` tests the bundle's expiration time against the current time; checked when forwarding [virtual]

`size()` the number of bytes used by this bundle, including all headers [virtual]

`incrHop()` increments the bundle's hop count; called when received

The first three methods are declared virtual because, in the simulator, it's occasionally useful to have different types of bundles with slightly different behaviors. For instance, there's a `MockBundle` type in the **`pydtn`** module that has no real payload, just an additional field specifying the *size* of the payload. This requires reimplementing both `clone()` and `size()`. The `expired()` method can be reimplemented to, for example, change the expiration semantics from time-based to hop-count-based.

5.1.2 Node

We operate on Bundle objects with Nodes. When constructing a Bundle, we typically need to invoke the following two methods at the creating Node:

nextSeq() get the next bundle sequence number for the node, incrementing the internal counter

addr() get the node's address

At the highest level, the path a Bundle b takes through a Node is:

1. recv(b) — The Bundle is passed to the Node and its hop count is incremented. If this is the destination, or the Bundle::kBcast flag is set, then consume(b) is called. Otherwise,

2. forward(b) — The Node examines b for further propagation. It hands the Bundle to its CustodyPolicy to see if b should be placed in stable storage. Regardless, it then passes b to its ForwardingPolicy. If this policy says that b could *not* be sent, then the VolatileStorePolicy determines if and how to cache b for later sending. Otherwise,

3. send(b,l) — The ForwardingPolicy calls the Node's sending method with the Link that the policy determines is appropriate for this next hop. At this point, the Node updates b's sender and receiver, passing the modified Bundle to the Link l.

There are also broadcast() and acknowledge() methods, but the rest of Node is configuration and bookkeeping. We will deal with these methods as they become necessary.

The Link class is pure virtual. A convergence layer needs to define this, because there is no meaningful default behavior for the dtn library to implement. We will see a concrete example in Section 5.2 when we look at SimLink.

5.1.3 BundleStore

A Node has two types of storage: volatile and persistent. Volatile storage holds Bundles that have been matched to Links for forwarding as soon as possible (essentially a shared queue). Stable storage holds Bundles for which the Node is the custodian, and these messages should persist even if the physical node is shut off and then turned back on.

Both types of storage employ the abstract BundleStore class, and this is what the Node stores. When configured, however, one of two subclasses is required (depending on the store being set): VolatileBundleStore or PersistentBundleStore. Both of these abstract subclasses perform cleanup of expired Bundles, with minor differences. The **dtn** library does not contain any concrete subclasses of VolatileBundleStore or PersistentBundleStore, because there is no way to generalize these usefully to arbitrary platforms. Instead, we leave it to the convergence layer to define appropriate BundleStore types.

The interface that BundleStore provides is fairly simple. The methods addBundle(b) and deleteBundle(b) do the obvious. Both take pointers, with the former taking ownership of the memory in the process (usually a Bundle produced by the Bundle::clone() method).

There are two version of the `getPointer()` method, both of which return a `BundlePointer` object. This is a special bookkeeping class specifically for `BundleStores`, and acts as an iterator. We do not go into the details of this class here. One version of `getPointer()` takes an existing `BundlePointer`, while the other takes a sender address and sequence number. In either case, if a suitable `Bundle` cannot be found, a null `BundlePointer` is returned. If a null `BundlePointer` is passed to the first version, the "head" of the `BundleStore` is returned.

In addition to asking for a `BundlePointer`, you can ask the `BundleStore` whether a cached `BundlePointer` is still valid. That is, does the referenced `Bundle` still exist in the store? This is done with the `validatePointer()` method.

The other methods that might be of interest are `bytesUsed()` and `bundles()`. The former reports on the actual amount of data stored, and the latter reports the number of `Bundles` in the store.

5.1.4 NodePolicy

`NodePolicy` is another abstract base class. There are three abstract subclasses that are used by `Node`: `ForwardingPolicy`, `CustodyPolicy`, and `VolatileStorePolicy`. Each controls some aspect of a node's behavior, and together they define the bulk of DTN's properties.

ForwardingPolicy

As we saw in Section 3.1.3, a `ForwardingPolicy` tells `Node` how to send a `Bundle`. The base class looks like:

```
class ForwardingPolicy : public NodePolicy
{
   public :
      ForwardingPolicy( Node* owner ) : NodePolicy( owner ) {}
      virtual ~ForwardingPolicy() {}

      virtual void cache( BundlePointer& p )
      virtual bool forward( Bundle* b ) = 0;
      virtual Bundle* forwardOn( const Link& l ) = 0;
};
```

We begin with the `forward()` method. Given a `Bundle`, the policy looks at its internal data structures for a `Link` capable of sending `b`. If the `Bundle` can be sent immediately, it should be, and `true` returned. Otherwise, this method should return `false`. `forward()` takes ownership of `b` if it will return `true`, otherwise it does not. Note that often ownership will be transferred immediately to the `Link`.

If `forward()` returns `false`, the `Node` will attempt to cache the `Bundle` in its volatile store. If it is able to, then `Node` calls `cache()` with the `BundlePointer` returned by the `BundleStore`. This allows the `ForwardingPolicy` to maintain its own data structure matching `Bundles` with `Links`. The policy must call `BundleStore::validatePointer()` before using a cached `BundlePointer`, however.

When a previously busy `Link` (or one that had been in the "down" state) becomes available, the `Node` calls the `forwardOn()` method, which includes the now-available `Link`. The

ForwardingPolicy must then look through the cached BundlePointers (either its own or the volatile store's) for a Bundle that can be sent along this Link. If a suitable Bundle is found, then it is returned. Otherwise, a null pointer should be returned. Ownership of the memory for the Bundle is being handed to the caller, so the BundleStore should be told to release the Bundle with the BundleStore::deleteBundle() method.

CustodyPolicy

The CustodyPolicy controls what goes into or gets removed from stable storage. The structure of the base class is:

```
class CustodyPolicy : public NodePolicy
{
   public :
      enum CustodyReturn
      {
         kCustodyTaken,
         kNoCustody,
         kDropBundle
      };

      CustodyPolicy( Node* owner ) : NodePolicy( owner ) {}
      virtual ~CustodyPolicy() {}

      virtual CustodyReturn takeCustody( const Bundle& b ) = 0;
      virtual bool policyRemove( const Bundle& b ) { return true; }
      virtual void retry() = 0;
};
```

The most complex method is takeCustody(), which we will discuss last. The policyRemove() method controls the stable storage behavior when a Bundle is acknowledged, either by the destination or the next custodian. In most cases, we want the Bundle to be removed. Some policies might want to keep Bundles in stable storage indefinitely, though. The retry() method is called occasionally by the Node, and should attempt to forward the Bundles in stable storage. Not every configuration will use this.

The bulk of the work is done by the takeCustody() method. One thing to note is that this method takes a constant reference to the Bundle, not a pointer. This is because we are not handing control of the Bundle to the policy. If it wants to store the Bundle, it must first make a copy with the Bundle::clone() method.

The next thing to note is that the return type is an enumeration value. This gives the CustodyPolicy an added expressiveness when compared with the ForwardingPolicy. These return values are:

kCustodyTaken This is equivalent to true from ForwardingPolicy::forward(); the Node takes custody of the Bundle, and a copy has been placed in the stable store.

kNoCustody This is equivalent to false from ForwardingPolicy::forward(); the Node does not take custody of the Bundle, so nothing was changed in the stable store.

kDropBundle We did not take custody of the Bundle. Furthermore, the Node should immediately drop the Bundle for policy-violation reasons.

VolatileStorePolicy

Just as CustodyPolicy controls the stable store, VolatileStorePolicy controls the volatile store. It is called by Node when ForwardingPolicy::forward() returns false. The class structure is:

```
class VolatileStorePolicy : public NodePolicy
{
   public :
      VolatileStorePolicy( Node* owner ) : NodePolicy( owner ) {}
      virtual ~VolatileStorePolicy() {}

      virtual BundlePointer store( Bundle* b ) = 0;
};
```

The only method in this policy is store(). If the Bundle is placed in the volatile store, ownership is transferred to the store and the volatile store's BundlePointer is returned. If the Bundle is not placed in the volatile store, a null BundlePointer should be returned, which will prompt the Node to drop the Bundle.

It is possible that, in storing the Bundle, the VolatileStorePolicy might specify that other Bundles in the volatile store must be dropped. This is done by calling the Node::drop() method, with an appropriate reason for the drop. We'll look at the DropCause classes in Chapter 6.

5.1.5 Consumer

A Consumer is an object that handles Bundles at their destinations. The base class also has hooks for a number of other Bundle operations:

```
class Consumer
{
   public :
      Consumer( Node* owner ) : m_owner(owner) {}
      virtual ~Consumer() {}

      virtual void setOwner( Node* owner ) {m_owner = owner;}

      virtual void operator()( const Bundle& b ) = 0;

      virtual void drop( const Bundle& b,
                         const DropCause& c=DropCause::inst ) {}
      virtual void custody( const Bundle& b ) {}
      virtual void persistentStore( const Bundle& b ) {}
      virtual void persistentRemove( const Bundle& b ) {}
      virtual void send( const Bundle& b ) {}
      virtual void recv( const Bundle& b ) {}
      virtual void exhausted( const Bundle& b ) {}

   protected :
      Node*   m_owner;
};
```

These hooks are called by Node, generally after the decision has been made to perform an action, but before the action has been carried out.

Because the intent is that a `Consumer` is an at-destination `Bundle` handler, the functional operator is declared as pure virtual. The other methods have default (no-op) actions. For those not familiar with the syntax of overloading `operator()`, the declaration means that a `Consumer` named `c` (of a concrete subtype) can be treated as if it were a function with prototype

```
void c( const Bundle& b );
```

Note that the `persistentStore()` hook will likely be called if `custody()` is called, but the converse is not necessarily true. For example, a `Node` that is initially sending a `Bundle` might place it in the stable store, but there is no custody decision to make. The `exhausted()` hook is called when the stable store is filled, or when it is insufficient to hold an incoming `Bundle`. A `CustodyPolicy` should generally trigger this hook by calling `Node::signalExhausted()`. `drop()` differs from the other hooks in that it takes an additional optional argument specifying the reason that the `Bundle` is being dropped. As mentioned previously, we will see how to use `DropCause` in the next chapter.

There's a special subclass of `Consumer` called `Application`. It *only* provides the at-destination processing of a `Bundle`, by re-declaring all of the hooks as `private`, and providing a `protected` method called `process()`. An `Application` also has a unique identifier string, accessed as `appID()`, which distinguishes one subclass of `Application` from another. This is used by `operator()()`, which only calls `process()` on `Bundles` with a matching application identifier. That is, the application identifier string acts like a *port number* in a standard IP network.

A `Node` only knows directly about one `Consumer`, but clearly we will often want several. For instance, one `Node` may have several application-layer protocols. To handle this, a convergence layer will typically configure a `Node` with a special concrete subtype of `Consumer` called `ConsumerChain`. In addition to *being* a `Consumer`, it also *contains a list* of `Consumers`. When a `Bundle` is passed to any of the chain's hooks, it in turn calls the hooks of all the other `Consumers` in its internal list. This gives us the ability to register an arbitrary number of `Consumers` (including `Applications`) with a `Node`, simply by calling the chain's `addConsumer()` method.

5.2 The `pydtn` Module

The basic interface between the **`dtn`** library and the simulator is the **`pydtn`** module. There are two major classes in this package: `WrapNode` and `WrapLink`. Another important class is `SimLink`, which implements `DTN::Link`. The **`pydtn`** module places all of its classes in the global namespace.

Both `WrapNode` and `WrapLink` inherit the `BundleHandler` class, which in turn inherits `Entity`. What distinguishes a `BundleHandler` is that it has a virtual method

```
bool BundleHandler::handler( BundleEvent& )
```

A `BundleEvent`, in turn, inherits `Event`, and contains a `DTN::Bundle` as data and a particu-

lar BundleHandler as its destination. There are also a few extra members of the BundleEvent class, which we will address as they arise. The result of this class structure is that a DTN::Bundle can be placed in a BundleEvent, which can be handled by a WrapNode or WrapLink.

5.2.1 **WrapNode**

The module's bootstrapping code registers WrapNode by calling

```
add_entity( new WrapNode );
```

This associates a dummy instance with the identifier "node", so that

```
sim.Entity('node')
```

generates a new WrapNode by cloning the instance in the registry.

As a BundleHandler, the WrapNode class implements a handler for BundleEvent objects. This method is invoked both for DTN::Bundles originating at this node and for those transitting the node. Because of this, the handler must determine why it was given the particular BundleEvent. A locally generated event with the local-storage flag set (determined by the event's storeLocally() method) is passed to DTN::Node::addPersistent(). If the bundle was newly generated, then DTN::Node::forward() or DTN::Node::broadcast() is called directly, the latter being similar in function to forward(). Otherwise, if this node was the intended receiver of the last hop (that is, its address matches the receiver field of the DTN::Bundle or the bundle is a broadcast), the bundle is passed to DTN::Node::recv(). If none of the above conditions is met, the DTN::Bundle is silently dropped. Finally, if the node holds any DTN::Bundles in persistent storage, and is not yet configured to resend stored DTN::Bundles, the WrapNode schedules a ResendEvent for itself.

This ResendEvent requires a separate handler

```
bool WrapNode::handler( ResendEvent& )
```

This simply calls DTN::Node::retryStored() and then checks to see whether it needs to schedule another ResendEvent. As above, this depends on whether there are still bundles in the persistent store. Both here and in the handler for BundleEvent, if the WrapNode is configured with a resend period of 0, no ResendEvent will be scheduled (this might change after an event is already scheduled).

In its constructor, WrapNode configures default values for the underlying DTN::Node as well as its own parameters, such as the lifetime of a DTN::Bundle and the time between resends from peristent storage. Many of these defaults are controlled by a global configuration class, which is accessed in python as pydtn.config(). A few things are hard-coded. The WrapNode always configures the DTN::Node with a DTN::ConsumerChain. To this is attached a DumpConsumer, which provides verbose output for DTN::Bundle operations (see the description of the -v flag in Chapter 2). If statistics collection is configured, a DTN::Consumer for this is added. Similarly, if any tracers (Chapter 6) have been established, a TraceConsumer

is added to the chain. The volatile store is configured as a `FlatStore`, with `DTN::DropTail` as the controlling policy. `PersistentStore` is used for stable storage. We do not describe these classes here.

The remaining interesting methods of `WrapNode` are the python-callable hooks. These have been discussed elsewhere. For the `collect()` method, the `WrapNode` maintains a list of `NodeCollector` functional objects, over which it loops. The `finalize()` method simply calls `collect()`.

5.2.2 WrapLink

`WrapLink` is a simpler class than `WrapNode`. It is bootstrapped into the simulator's registry (with the identifier "link") by calling

```
add_entity( new WrapLink );
```

Internally, the `WrapLink` holds pointers to two `WrapNodes`, the source and destination. We can make this distinction because all links are unidirectional. In addition, the `WrapLink` holds a `SimLink`, which implements the actual link functionality.

The `BundleEvent` handler simply schedules a new `BundleEvent` for the remote end of the link, unless the link is currently down. If this is the case, the bundle is dropped.

`WrapLink` handles two new event types: `LinkAvailEvent` and `LinkUpDownEvent`. The handler for `LinkAvailEvent` sets the link's busy state to `false` and calls the link source's `DTN::Node::forwardOn()`. The handler for `LinkUpDownEvent` sets the link state to either "up" or "down", depending on the event data.

5.2.3 SimLink

This class implements the actual link functionality. Because the simulator does not rely on physical devices, `SimLink` encapsulates these as:

latency the time it takes a single bit to traverse the link

bandwidth the number of bits per second that the link can transmit

busy whether an outgoing bundle is still placing bits on the line

busyUntil the time at which another bundle can begin placing bits on the line

earliestDelivery the earliest time at which a bundle can be delivered

up whether the link is currently operational

upSince the latest time at which the link entered the "up" state

The link is modelled as a FIFO queue of bits. The bandwidth governs when new bits may be enqueued, while the latency is added to this to determine the delivery time. The `earliestDelivery` time ensures that, even if the link characteristics change, the enqueued bundles will be delivered in FIFO order.

The method most commonly accessed is `send()`. This uses the above parameters to determine a delivery time, and schedules a `BundleEvent` at the `WrapLink` that owns the `SimLink`. It also recomputes `busyUntil` and schedules a `LinkAvailEvent` for this time, also setting the `busy` state to `true`.

If a `LinkUpDownEvent` causes the `WrapLink` to change the `SimLink`'s state from "down" to "up", the `SimLink` also calls `DTN::Node::forwardOn()` at the sending node.

5.3 The `mobility` Module

This module extends **`pydtn`** with mobile and wireless functionality. We will look at four classes in this section: `MobileNode`, `WirelessLink`, `MobileApp`, and `MockLink`. These classes are defined in the `Mobility` namespace.

5.3.1 `MobileNode`

This class inherits `WrapNode`. It provides mobility through a series of *waypoints*. Each waypoint comprises a location (in three dimensions), a deadline for reaching that location, and a pause time specifying the minimum delay between reaching the waypoint and departing for the next one. No specific mobility pattern is included, though a subclass `RandomWaypointNode` implements random waypoint mobility over an arbitrary `WaypointRange`.

The identifier for a `MobileNode` is "node:mobile". We use a (colon-separated) naming hierarchy so that parsers for the superclass can always attempt to match the longest-relevant prefix. That is, when attempting to parse a python object as a `WrapNode`, the identifier must start with the substring "node", but when attempting to parse the object as a `MobileNode`, the identifier must start with "node:mobile". `RandomWaypointNode`, for example, has an identifier "node:mobile:random_waypoint".

To maintain connectivity, the `MobileNode` periodically broadcasts *beacons*. As an optimization, any broadcast message is considered a beacon. The `MobileNode` schedules beacons with a `HelloTrigger` event. If the configured beacon period is τ, the next beacon is scheduled for a time selected uniformly at random in $[\tau/2, 3\tau/2)$ from the time of the last beacon (or broadcast).

Reacting to beacons is handled by a pair of classes. First a `BeaconConsumer` records the times of outgoing broadcasts (using the `send()` hook) and checks incoming messages (using the `recv()` hook) for previously unknown neighbors. If a new neighbor is discovered, the consumer calls `MobileNode::notifyApps()` with the neighbor's address. `MobileNode` maintains a list of `MobileApps` (Section 5.3.3) that respond to these new neighbor notifications with appropriate protocol messages. These `DTN::Application` subtypes are registered as normal `DTN::Consumers` as well as being stored in a `MobileNode`-specific data structure.

5.3.2 `WirelessLink`

The `WirelessLink` class inherits `WrapLink`. As with `MobileNode`, we use a hierarchical identifier name, in this case "link:wireless". Because a wireless link has only a fixed source, with an arbitrary number (possibly zero) of destinations, `WirelessLink` does some tweaking of the base-class functionality.

The principal change to `WrapLink`'s basic behavior is that the "connect" operation in the `WirelessLink::configure()` method takes only one node. This must be a `MobileNode`, and is recorded as both the source and destination of the `WrapLink`. In addition, this node is stored explicitly as a `MobileNode`, allowing `WirelessLink` to access the subtype's methods.

`WirelessLink` is, itself, an abstract class. It provides no handler for `BundleEvents`, nor a `create()` method: both are defined pure virtual. Additional packages provide the actual link functionality. For example, the **`pydtn.mobility.rf`** subpackage provides the `RF::RFLink` subclass ("link:wireless:rf"), which implements a simple interference-free disk model.

5.3.3 `MobileApp`

As mentioned previously, a `MobileApp` is a `DTN::Application` that handles new-neighbor protocol messages. It has a (pure virtual) `newNeighbor()` method. This class is abstract, with no meaningful functionality of its own. It's typically used in concert with a forwarding algorithm. See Section 8.2 for an example.

5.3.4 `MockLink`

`MockLink` is a very simple class. It contains a remote address and a pointer to a `DTN::Link`. Its sole purpose is to allow packages based on **`mobility`** to provide bookkeeping links for a `MobileNode`'s active neighbors. The `send()` method is overridden to call the aliased link's method. It can be regarded as an ephemeral static-network link.

Chapter 6

Tracers

As was mentioned in Section 3.3.2, tracers are useful for combining information about bundle operations with information about the nodes at which those operations are performed. In this chapter, we'll look at tracers in greater detail. We'll start with the general structure of a tracer, then move on to a thorough examination of an existing tracer, and finally see how to modify this to capture additional information.

6.1 Structure of a Tracer

The **pydtn** module defines a class `Tracer` from which all tracers are derived. The structure of this class is:

```
class Tracer
{
   public :
      Tracer() {}
      virtual ~Tracer() {}

      virtual void node( const WrapNode& wn ) = 0;
      virtual void link( const WrapLink& wl ) = 0;

      virtual void enqueue( const DTN::Bundle& b, const DTN::Node* n ) = 0;
      virtual void dequeue( const DTN::Bundle& b, const DTN::Node* n ) = 0;
      virtual void send( const DTN::Bundle& b, const DTN::Node* n ) = 0;
      virtual void receive( const DTN::Bundle& b, const DTN::Node* n ) = 0;
      virtual void drop( const DTN::Bundle& b,
                         const DTN::DropCause& c,
                         const DTN::Node* n ) = 0;
};
```

This is the definition from the file `pydtn/Tracer.h`, with all comments removed. The first thing to note is that most of the methods are pure virtual, and the base class holds no data itself. All of these methods must be defined by a derived class before it can be instantiated, though many of them could be empty.

We've divided the methods into three groups. The first group comprises the constructor and virtual destructor. These have to have concrete definitions, though they do nothing.

The second group of methods pertain to the creation of network elements. In many cases, we will not be interested in tracing this information. There is a module **namtrace** that generates tracefiles in the nam format for static networks, though, that uses these two methods to generate the input commands needed by nam.

The third group contains the tracing methods that are most likely to be useful. All of these methods take a constant reference to the bundle on which we're operating, and a constant pointer to the node at which the operation occurs. The drop() method takes an additional argument specifying the reason why the bundle was dropped (see Chapter 7 for details).

A tracer works as a component in a special DTN::Consumer, called TraceConsumer, and which is defined in the **pydtn** module. The GlobalTracer class provides a static instance, created by calling GlobalTracer::enable(). A WrapNode's constructor can test for the existence of this instance, and if it's present the node creates a new TraceConsumer with a pointer to the instance. The TraceConsumer calls the GlobalTracer's hooks from the normal DTN::Consumer hooks. The GlobalTracer, in turn, contains a list of Tracers, similar to the structure of DTN::ConsumerChain.

Adding a Tracer is then simpler than adding a DTN::Consumer to each WrapNode. Any module that initializes the GlobalTracer ensures that all subsequently created WrapNodes have a TraceConsumer, so the only thing that needs to be done is to add the Tracer to the chain by calling GlobalTracer::addTracer(). The calls to enable() and addTracer() are typically done during module initialization or setup.

6.2 flowtrace

This module provides a simple example of DTN::Bundle tracing. It traces three types of bundle events: the initial transmission of a bundle from its source, the receipt of a bundle at its destination, and a bundle drop. Note that a single bundle might be dropped multiple times, and that dropping a bundle does not preclude its later delivery.

The send() hook is instructive for this tracer in general:

```
void
FlowTracer::send( const DTN::Bundle& b, const DTN::Node* n )
{
   if ( 0 == n ) return;
   if ( n->addr() != b.source() ) return;
   traceBundle(b,"inject",*n);
}
```

Here we ensure that the pointer to the node isn't null, and then check whether this is the source. If so, we call the object's workhorse routine:

```
void
FlowTracer::traceBundle( const DTN::Bundle& b,
                         const std::string& func,
                         const DTN::Node& n,
                         const std::string& xtra )
{
   if ( b.type() & DTN::Bundle::kACK )
   {
      return;
   }
   Time t = Clock::time();
   m_stream << t.tv.tv_sec << "," << t.tv.tv_usec << " "
            << stringify(b.source()) << " "
            << stringify(b.destination()) << " "
            << stringify(b.app()) << " "
            << ntohl(b.seqNum()) << " "
            << func << " "
            << b.size() << " "
            << b.hopCount() << " "
            << stringify(n.addr()) << " "
            << xtra
            << std::endl;
}
```

The `stringify()` method takes a `DTN::ByteString` and converts it to printable ASCII. The `xtra` argument has a default empty string value, and is only used by `drop()` to report the reason that the bundle was dropped.

In the bootstrapping code, **`flowtrace`** defines the python-callable method `flowtrace_setup()` (called as `pydtn.flowtrace.setup()`). This takes a filename, and initializes the `FlowTracer` with an output stream that writes to that file (`m_stream` in the above code sample). It also enables the `GlobalTracer`. If the module is loaded, but `setup()` is not called, then the tracer will not be created, and no flow information will be generated.

6.3 Modifying a Tracer

As we've seen, `FlowTracer` is a pretty simple class. In fact, all of the tracers defined in the core pydtn distribution are fairly simple. Adding more functionality to `FlowTracer` is also simple. This makes it nearly ideal to describe the process of copying an existing module.

6.3.1 Setup

We'll assume that you want to put the new module in the current directory, and that the pydtn source code is in `/opt/pydtn/src`. Setting up the new module, which we'll call **`newflowtrace`**, begins with:

```
$ cp -r /opt/pydtn/src/flowtrace newflowtrace
```

We'll further assume that you aren't worried about cross-platform portability, and that you aren't interested in building source code documentation with doxygen. This means you can remove the following files in the **`newflowtrace`** directory:

- Doxyfile
- Doxyfile.in
- Makefile.in
- config.h.in
- configure
- configure.ac

The following files are generated during compilation and linking, and can also be removed:

- FlowTracer.o
- _flowtrace.so
- config.log
- config.status
- libflowtrace.so

You should now see the following files:

- FlowTracer.cc
- FlowTracer.h
- Makefile
- __init__.py
- config.h
- install-sh
- python_init.cc

The last step that we'll do is to rename a couple of these files:

```
$ mv FlowTracer.cc NewFlowTracer.cc
$ mv FlowTracer.h NewFlowTracer.h
```

6.3.2 The Header Files

We'll begin with `config.h`, because it is simpler. The file protects itself against multiple inclusion with a set of preprocessor statements. The file is short enough that we include it here verbatim:

```
config.h

/* config.h.  Generated from config.h.in by configure.  */
#ifndef __FLOW_TRACE_CONFIG_H__
#define __FLOW_TRACE_CONFIG_H__

// Configuration options from autoconf

#endif /* __FLOW_TRACE_CONFIG_H__ */
```

If we include files from both **flowtrace** and **newflowtrace**, we will have a problem with only the first version of `config.h` being included. Of course, since the files will both be empty, this isn't really a problem. It is also unlikely that we would include both. Still, to be safe we modify the header file as follows:

```
config.h

/* config.h.  Generated from config.h.in by configure.  */
#ifndef __NEW_FLOW_TRACE_CONFIG_H__
#define __NEW_FLOW_TRACE_CONFIG_H__

// Configuration options from autoconf

#endif /* __NEW_FLOW_TRACE_CONFIG_H__ */
```

That's it for `config.h`, which means it's time to look at `NewFlowTracer.h`. Stripping out all of the comments, we begin with:

NewFlowTracer.h

```
#ifndef __FLOW_TRACE_FLOW_TRACER_H__
#define __FLOW_TRACE_FLOW_TRACER_H__

#include "config.h"

#include "pydtn/Tracer.h"
#include "dtn/ByteString.h"
#include <ostream>

class FlowTracer : public Tracer
{
   public :
      FlowTracer( std::ostream& s );
      virtual ~FlowTracer();

      void node( const WrapNode& wn );
      void link( const WrapLink& wl );

      void enqueue( const DTN::Bundle& b, const DTN::Node* n );
      void dequeue( const DTN::Bundle& b, const DTN::Node* n );
      void send( const DTN::Bundle& b, const DTN::Node* n );
      void receive( const DTN::Bundle& b, const DTN::Node* n );
      void drop( const DTN::Bundle& b,
                 const DTN::DropCause& c,
                 const DTN::Node* n );

   protected :
      void traceBundle( const DTN::Bundle& b, const std::string& func,
                        const DTN::Node& n, const std::string& xtra="" );
      std::string stringify( const DTN::ByteString& b ) const;

   private :
      std::ostream&  m_stream;
};

#endif
```

The changes to this file will be minimal:

NewFlowTracer.h

```
#ifndef __NEW_FLOW_TRACE_NEW_FLOW_TRACER_H__
#define __NEW_FLOW_TRACE_NEW_FLOW_TRACER_H__

#include "config.h"

#include "pydtn/Tracer.h"
#include "dtn/ByteString.h"
#include <ostream>

class NewFlowTracer : public Tracer
{
   public :
      NewFlowTracer( std::ostream& s );
      virtual ~NewFlowTracer();

      void node( const WrapNode& wn );
      void link( const WrapLink& wl );
      ...
```

Note that all we've done is to rename the class and fix the first two preprocessor statements (just as we did in `config.h`).

6.3.3 The Class Definition File

Most of our modifications to `NewFlowTracer.cc` will be renaming, similar to what we did with the header file. Rather than showing all of the substitutions explicitly, we'll just observe that every occurrence of `FlowTracer` should be replaced with `NewFlowTracer`.

There isn't much point in copying a module if we don't modify its behavior. We're going to add two things to **`newflowtrace`**. The first is that we want to record any time a bundle is sent by a node. We do this by changing

NewFlowTracer.cc

```
void
NewFlowTracer::send( const DTN::Bundle& b, const DTN::Node* n )
{
   if ( 0 == n ) return;
   if ( n->addr() != b.source() ) return;
   traceBundle(b,"inject",*n);
}
```

to

NewFlowTracer.cc

```
void
NewFlowTracer::send( const DTN::Bundle& b, const DTN::Node* n )
{
   if ( 0 == n ) return;
   if ( n->addr() != b.source() )
      traceBundle(b,"forward",*n);
   else
      traceBundle(b,"inject",*n);
}
```

The next change that we'll make is to add a time-to-live field to the output in `traceBundle()`. First we compute the TTL for the bundle:

```
   Time t = Clock::time();
   Time ttl = b.expiry();
   ttl -= t;
```

Now we add this to the stream inserter statement:

```
   m_stream << t.tv.tv_sec << "," << t.tv.tv_usec << " "
            << stringify(b.source()) << " "
            << stringify(b.destination()) << " "
            << stringify(b.app()) << " "
            << ntohl(b.seqNum()) << " "
            << func << " "
            << b.size() << " "
            << b.hopCount() << " "
            << ttl.tv.tv_sec << "," << ttl.tv.tv_usec << " "
            << stringify(n.addr()) << " "
            << xtra
            << std::endl;
```

The time-to-live will appear between the hop count for the bundle and the address at which the bundle is currently being processed.

6.3.4 Bootstrapping

Now we need to modify `python_init.cc`. Again, this will be an exercise in renaming. The important things to change are `FlowTracer` to `NewFlowTracer` (don't forget the `#include` statement) and changing

python_init.cc

```
PyMODINIT_FUNC
init_flowtrace(void)
{
   (void) Py_InitModule("_flowtrace",ExampleMethods);
}
```

to

python_init.cc

```
PyMODINIT_FUNC
init_newflowtrace(void)
{
   (void) Py_InitModule("_newflowtrace",ExampleMethods);
}
```

Everything else in this file is either documentation or declared as "static" to the file, so there won't be any problems with naming collisions. It doesn't hurt to name things consistently, though.

The last bit of bootstrapping we need to address is the file `__init__.py`. This needs to be changed from

__init__.py

```
from _flowtrace import *
```

to

__init__.py

```
from _newflowtrace import *
```

6.3.5 Makefile

The last thing to change is the `Makefile`. There won't be much to change here. At a minimum, you will need to change

Makefile

```
MODNAME := flowtrace
MODPATH := pydtn
moddir := $(libdir)/$(MODPATH)/$(MODNAME)
```

to

```
Makefile

MODNAME := newflowtrace
MODPATH :=
moddir := $(libdir)/$(MODPATH)/$(MODNAME)
```

If you plan to run

```
$ make install
```

after you compile and link your module (which isn't a bad idea), you might need to modify the `prefix` variable from, say,

```
Makefile

prefix := /opt/pydtn/src
```

to a directory in which you can write. Installing all of your modules (including pure-python modules) into a common directory makes it much simpler to specify the appropriate search path when you invoke simlpy (see Chapter 2).

Chapter 7

Drop Cause Classes

When a `DTN::Bundle` is dropped by a `DTN::Node`, a reason for the drop is included (though there's a default value). The mechanism for specifying this reason, `DTN::DropCause`, is somewhat complex, so it's worth discussing at some length.

To begin, let's consider what we'd like, functionality-wise, from this mechanism. Clearly, we want some way to distinguish a drop due to, say, a bundle lifetime being exceeded from a drop due to a link failure. The simplest way to specify this would be to enumerate the various reasons why a bundle might be dropped, and pass this simple enumerator value. This, however, is extremely limited. If a later module introduces drops for a new reason, there's no way to change the enumeration without an invasive modification of a low-level file in the **`dtn`** library. This would necessitate a recompilation of almost the entire suite of packages.

Instead of an enumeration, we might assign constants to the various reasons for a drop. This is similar to placing the reasons in an enumeration, but new constants could be created arbitrarily by other modules. A `DTN::Consumer` that doesn't recognize the value passed to it would simply report the reason as "unknown." This, too, has a problem. While it doesn't require extensive recompilation, there's a risk of collision. Without some way to enforce the assignment of reasons to constants, sequentially numbered reasons are likely to conflict in third-party modules. We could recommend the use of random values from a large domain, but this is impossible to enforce.

A third option would be to pass strings for the reasons. This is certainly flexible. A particular unambiguous identifying string for a drop cause is not likely to satisfy every need, however. Even if we consider the primary use of the drop cause to be printing a readable message, one `DTN::Consumer` might want to indicate bundle expiry with "bundle expired", while another might prefer the shorter (and whitespace-free) "expry". In this case, the simple printable string would still require string comparison operations in many instances.

Another wrinkle is that we would like to be able to create a hierarchy of reasons. Perhaps we want a specialization of link failure for an antenna that has caught fire. We cannot do this with the constants-based drop causes. For strings, we could employ a method similar to the hierarchical entity identifiers: "link_failed:antenna_on_fire". Again, we're left with the issue of string comparisons.

Since the simulator is written in an object-oriented language with inheritance, a class hierarchy is an appealing option. We can create derived types from various drop causes, with-

out the base classes having to know or care. Further, if we pass pointers or references, then a `DTN::Consumer` that recognizes a base class but not the derived class can still handle the drop cause appropriately.

We run into a basic problem here, however. A language such as python has object reflection: you can examine the object's meta-data, such as its type, and the type itself can describe its base type. Given a drop cause instance, we could walk up its inheritence graph through the meta-data until we find a type we recognize. C++ does not have reflection, however.

To overcome this problem, we created the `DTN::DropCause` class, which adds its own very basic type reflection through static instances and pointer abuse. The base class declaration is:

```
class DropCause
{
   public:
      static DropCause inst;
      virtual const DropCause* n() const { return m_next; }
      virtual bool isType( const DropCause& c );
      virtual ~DropCause() {}

   protected:
      DropCause( DropCause* p=0 ) : m_next(p) {}
      DropCause* m_next;
};
```

Note that we have a static instance, and that the constructor (which includes the default) is protected, so we cannot create additional instances. We have a pointer to another instance and a method to retrieve it. For this base class, this pointer is null.

The `isType()` method is defined as follows:

```
bool
DropCause::isType( const DropCause& c )
{
   if ( &c == this ) return true;
   if ( 0 == c.n() ) return false;
   return isType(*(c.n()));
}
```

To understand this, it's important to know how it's used. When given an as-yet-unidentified `DropCause` named `c`, you pass it to the `isType()` method of one of the known static instances. The value returned is whether `c` is of the candidate instance's type *or a type derived from it*. Now, let's go through this method. First we compare the address of `c` with the address of this instance. If these match, they're the same type, and we return `true`. If not, we move to the *proximate* base class of `c`, as reflected by its `m_next` field, accessed as `c.n()`. If this pointer is null, there is no base class, and the match fails. Otherwise, we call `isType()` recursively with `c`'s base.

We'll look at one of the derived types, to see how these classes plug together. Since we've previously discussed bundle expiration as a reason for a drop, let's take a look at:

```
class ExpiryDrop : public DropCause
{
   public :
      static ExpiryDrop inst;
      virtual ~ExpiryDrop() {}

   protected :
      ExpiryDrop( DropCause* p=&(DropCause::inst) ) : DropCause(p) {}
};
```

The constructor for `ExpiryDrop::inst` passes the address of the base `DropCause::inst` as the "next" pointer.

Let's now take a look at the call:

```
DropCause::inst.isType( ExpiryDrop::inst )
```

The initial test compares the addresses of `ExpiryDrop::inst` and `DropCause::inst`. Obviously, these don't match, so we next look at `ExpiryDrop::inst.n()`. This isn't null, so we pass it to `DropCause::inst.isType()`. The "next" pointer of `ExpiryDrop::inst` is the address of `DropCause::inst`, however, so on the first recursion we find a match, and the method returns that `ExpiryDrop::inst` is in fact a type of `DropCause`. The recursion can proceed an arbitrary number of inheritence steps, but is guaranteed to stop once `DropCause::inst` is reached. That is, unless you incorrectly define a subclass with a loop.

Testing against the base class is not very interesting, though. We'd like to be able to distinguish betweeen different drop causes. Let's then take a look at another subclass:

```
class LinkFailureDrop : public DropCause
{
   public :
      static LinkFailureDrop inst;
      virtual ~LinkFailureDrop() {}

   protected :
      LinkFailureDrop( DropCause* p=&(DropCause::inst) ) : DropCause(p) {}
};
```

Again, we pass the address of `DropCause::inst` as the "next" pointer. Now, however, we can test the following:

```
LinkFailureDrop::inst.isType( ExpiryDrop::inst )
```

We first compare the addresses of `ExpiryDrop::inst` and `LinkFailureDrop::inst`. These won't match, so we'll then recursively call `LinkFailureDrop::inst.isType()` with the "next" pointer of `ExpiryDrop::inst`. This is the address of `DropCause::inst`, though, which again does not match. Now, when we look at the "next" pointer, we get a null, so the method returns `false`. Sure enough, an `ExpiryDrop` is not a `LinkFailureDrop`.

To see how we might use these in practice, we'll take a look at `DumpConsumer`, which provides verbose output for **`pydtn`**. This class has a static method `dropCauseString()`, which takes a constant reference to a `DropCause` and returns a printable string. The method is defined as:

```
std::string
DumpConsumer::dropCauseString( const DTN::DropCause& c )
{
   std::string retval = "unknown";
   if ( DTN::BitErrorDrop::inst.isType(c) )
   {
      retval = "bit error";
   }
   else if ( DTN::CustodyPolicyDrop::inst.isType(c) )
   {
      retval = "custody policy violation";
   }
   else if ( DTN::ExpiryDrop::inst.isType(c) )
   {
      retval = "bundle expired";
   }
   else if ( DTN::FullQueueDrop::inst.isType(c) )
   {
      retval = "queue is full";
   }
   else if ( DTN::LinkFailureDrop::inst.isType(c) )
   {
      retval = "link failed";
   }
   else if ( DTN::NoRouteDrop::inst.isType(c) )
   {
      retval = "no route found";
   }
   return retval;
}
```

The cascade of `if-else-ifs` is not much of a syntactic improvement over string comparisons, but each comparison in this cascade is a small number of pointer comparisons, regardless of the possible prefix substring matches.

What if we wanted to add a new drop cause for an antenna that has caught fire? This is a subclass of link failures, so we would define it as:

```
class AntennaOnFireDrop : public LinkFailureDrop
{
   public :
      static AntennaOnFireDrop inst;
      virtual ~AntennaOnFireDrop() {}

   protected :
      AntennaOnFireDrop( DropCause* p=&(LinkFailureDrop::inst) ) : DropCause(p) {}
};
```

This will match itself, of course, but for `DumpConsumer`, which only knows the causes listed above, not `AntennaOnFireDrop`, it will also match `LinkFailureDrop` and `DropCause`.

Chapter 8

Applications

Applications are the way to implement message protocols in the simulator. These could be low-level protocols, normally thought of at the network layer, or high-level protocols like FTP. The `DTN::Application` class is the base for all pydtn applications, and is defined as:

```
class Application : public Consumer
{
   public :
      Application() : Consumer(0) {}
      virtual ~Application() {}

   protected :
      virtual const ByteString& appID() const = 0;
      virtual void process( const Bundle& b ) = 0;

   private :
      void operator()( const Bundle& b );
      void drop( const Bundle& b,
                 const DropCause& c=DropCause::inst ) {}
      void custody( const Bundle& b ) {}
      void persistentStore( const Bundle& b ) {}
      void persistentRemove( const Bundle& b ) {}
      void send( const Bundle& b ) {}
      void recv( const Bundle& b ) {}
      void exhausted( const Bundle& b ) {}
};
```

Note that all of the `DTN::Consumer` hooks are declared private, and all but the functional operator are declared empty. Derived classes cannot access nor modify any of these definitions. Rather, the protected virtual methods `appID()` and `process()` provide the protocol-specific functionality.

A subclass of `DTN::Application` must define a unique string that is shared by all instances of that class. This string is returned by `appID()`. The `operator()()` method uses this string to identify `DTN::Bundles` for the protocol, since the same string is stored in the `DTN::Bundle`.

The `process()` method receives any `DTN::Bundle` passed to the functional operator that matches the application's identifier. This is how an application gets its messages. Messages can be created by any method or object that knows the appropriate identifier and message format. This might be the `process()` method itself (if responding to a message), another method defined in the subclass (the **`mobility`** module's `Mobility::MobileApp` has a method `newNeighbor()`

for this purpose), or a completely separate mechanism. In most cases, it makes sense to have an application class generate all of its own messages.

The remainder of this chapter will look at specific protocols. Familiarity with previously introduced concepts, especially Chapter 4, is assumed.

8.1 Ping

Because of its simplicity, a version of the *ping* protocol is a useful first example. Our implementation will provide a hook for scripts to generate a single ping bundle with a payload consisting of a one-byte type identifier (`kPing`). The destination will respond with a bundle containing a response identifier (`kResp`) and the timestamp from the original bundle. The original sender, on receipt of the response, will print out a message of the form:

```
<t> <src> -> <dest> : <rtt> s
```

8.1.1 Class Declaration

We will call our application class `PingApp`. Here is the header file (without comments):

```
PingApp.h

#ifndef __PING_APP_H__
#define __PING_APP_H__

#include "config.h"

#include "dtn/Application.h"
#include "simlpy/Time.h"

class PingApp : public DTN::Application
{
   public :
      static DTN::ByteString      kIdentifier;
      static const unsigned char  kPing;
      static const unsigned char  kResp;
      union SerialInt { uint32_t i; unsigned char c[4]; };

      PingApp( const Time& lifetime );
      virtual ~PingApp();

      void send( const DTN::ByteString& dest );

   protected :
      const DTN::ByteString& appID() const { return kIdentifier; }
      void process( const DTN::Bundle& b );

   private :
      void processPing( const DTN::Bundle& b );
      void processResp( const DTN::Bundle& b );

      Time  m_lifetime;
};

#endif
```

We're using static constants for both the application identifier (`kIdentifier`) and the one-byte message codes (`kPing` and `kResp`). The `SerialInt` union will make it easy to put the elements of a `timeval` struct into and read them out of a `DTN::Bundle` payload.

The application is configured with the appropriate lifetime to set for a bundle. This will be used when composing both a ping message and the response. Note that we've split the processing into two separate methods, `processPing()` and `processResp()`. This allows us to keep the `process()` method simple.

The initial structure of the definition file is:

PingApp.cc

```
#include "PingApp.h"
#include "dtn/Node.h"
#include "simlpy/Clock.h"
#include "pydtn/parsers.h"
#include <iostream>
#include <netinet/in.h>

DTN::ByteString PingApp::kIdentifier((unsigned char*)"ping");
const unsigned char PingApp::kPing = 0;
const unsigned char PingApp::kResp = 1;

PingApp::PingApp( const Time& lifetime ) : DTN::Application(), m_lifetime( lifetime )
{
}

PingApp::~PingApp()
{
}

void
PingApp::send( const DTN::ByteString& dest )
{
}

void
PingApp::process( const DTN::Bundle& b )
{
   if ( 0 == m_owner ) return;
   if ( b.payload().length() < 1 ) return;
   switch ( b.payload()[0] )
   {
      case kPing : processPing(b); break;
      case kResp : processResp(b); break;
      default: ;
   }
}

void
PingApp::processPing( const DTN::Bundle& b )
{
}

void
PingApp::processResp( const DTN::Bundle& b )
{
}
```

As mentioned before, the `process()` method is very simple. It ensures that the message is

minimally well-formed, and then dispatches it according to the message type.

8.1.2 Sending a Ping

The `send()` method creates the initial bundle and sends it. This method should be called at the time you wish to actually send the bundle; it does not schedule a bundle for later sending. The actual bundle payload is a single byte, with the `PingApp::kPing` message type.

```
PingApp.cc

void
PingApp::send( const DTN::ByteString& dest )
{
   if ( 0 == m_owner ) return;
   DTN::ByteString data;
   data.push_back( kPing );
   Time t = Clock::time();
   DTN::Bundle* b = new DTN::Bundle( m_owner->nextSeq(),
                                     m_owner->addr(),
                                     dest,
                                     data,
                                     t.tv,
                                     (t + m_lifetime).tv,
                                     DTN::Bundle::kData | DTN::Bundle::kCustodial,
                                     kIdentifier );
   m_owner->forward( b );
}
```

8.1.3 Responding to a Ping

Because the `DTN::Bundle` has `PingApp::kIdentifier` as its application identifier, at the destination it will be passed to `PingApp::process()`, which will then pass the bundle to `PingApp::processPing()`. This is where we generate the response:

```
PingApp.cc

void
PingApp::processPing( const DTN::Bundle& b )
{
   DTN::ByteString data;
   data.push_back( kResp );
   SerialInt ts, tus;
   ts.i = ::htonl(b.created().tv_sec);
   tus.i = ::htonl(b.created().tv_usec);
   data.append( ts.c, 4 );
   data.append( tus.c, 4 );
   Time t = Clock::time();
   DTN::Bundle* pB = new DTN::Bundle( m_owner->nextSeq(),
                                      m_owner->addr(),
                                      b.source(),
                                      data,
                                      t.tv,
                                      (t + m_lifetime).tv,
                                      DTN::Bundle::kData | DTN::Bundle::kCustodial,
                                      kIdentifier );
   m_owner->forward( pB );
}
```

The response is more complicated than the initial ping. We have to add the creation time of the incoming bundle to the payload of the response. We can't count on the convergence layer serializing the integers properly, so we have to convert them to network byte-order ourselves.

8.1.4 Finishing the Round-Trip

The last piece of the `PingApp` class that we need to write is the processing action for the response. Just as we had to convert the initial ping creation time from host byte-order to network byte-order, we have to convert the values back to host byte-order here.

PingApp.cc

```
void
PingApp::processResp( const DTN::Bundle& b )
{
   if ( b.payload().length() < 9 ) return;
   SerialInt ts, tus;
   for ( int i = 0 ; i < 4 ; ++i )
   {
      ts.c[i] = b.payload()[i+1];
      tus.c[i] = b.payload()[i+5];
   }
   ts.i = ::ntohl(ts.i);
   tus.i = ::ntohl(tus.i);
   Time now = Clock::time();
   double rtt = (now.tv.tv_sec - ts.i) + 1.0*(now.tv.tv_usec - tus.i)/Time::MILLION;
   std::cout << now.tv.tv_sec + 1.0*now.tv.tv_usec/Time::MILLION << " "
             << stringify( m_owner->addr() ) << " -> "
             << stringify( b.source() ) << " : "
             << rtt << " s" << std::endl;
}
```

8.1.5 Bootstrapping

In `python_init.cc`, we'll need the following to begin with (in addition to the standard skeleton header, structures, and methods):

python_init.cc

```
#include "PingApp.h"
#include "dtn/ByteString.h"
#include "pydtn/parsers.h"
#include <map>
typedef std::map< DTN::ByteString , PingApp* >  AppMap;
static AppMap  apps;
```

There are two python-callable methods that we'll need. The first is the routine to call from `applyToNodes`:

```
python_init.cc

static PyObject*
ping_attach( PyObject* self, PyObject* args )
{
   if ( 0 == PyTuple_Size( args ) )
   {
      PyErr_SetString(PyExc_TypeError, "attach expects an argument");
      return 0;
   }

   WrapNode* wn = parse_node( PyTuple_GetItem( args, 0 ) );
   if ( 0 == wn ) return 0;

   PingApp* app = new PingApp( wn->bundleLifetime() );
   wn->addApplication( app );
   apps[wn->addr()] = app;
   Py_INCREF(Py_None);
   return Py_None;
}
```

This is copied from **sampleapp**, with minor modifications (and simplifications) where needed. The `apps` map allows us to specify a node address and retrieve its attached `PingApp`.

The second method we need to define calls `PingApp::send()`. We'll want to call this method in python as

```
ping.send( srcnode, destnode )
```

We'll crib this method from **sampleapp** as well:

```
python_init.cc

static PyObject*
ping_send( PyObject* self, PyObject* args )
{
   if ( 2 > PyTuple_Size( args ) )
   {
      PyErr_SetString(PyExc_TypeError, "send expects two arguments");
      return 0;
   }

   ItemWrapper item;

   WrapNode* from = parse_node( PyTuple_GetItem( args, 0 ) );
   WrapNode* to = parse_node( PyTuple_GetItem( args, 1 ) );

   AppMap::iterator itr = apps.find( from->addr() );
   if ( apps.end() == itr )
   {
      PyErr_SetString(PyExc_ValueError,
                      "ping is not attached to this node");
      return 0;
   }
   PingApp* app = itr->second;
   if ( 0 != app )
   {
      app->send(to->addr());
   }

   Py_INCREF(Py_None);
   return Py_None;
}
```

8.2 Mobile Forwarding Protocol

The ping protocol in the previous section is a high-level protocol that depends only on itself and the facilities provided for it by the node. Another type of protocol that we can implement using `DTN::Application` is a low-level forwarding protocol. The **`mobileforwarding`** package provides a skeleton for writing these protocols, though it is by no means the only way of doing so.

8.2.1 Structure of the Library

The **`mobileforwarding`** package defines two classes, which reside in the `MobileForwarding` namespace. These classes are `Protocol` and `Forwarding`, and are tightly bound to one another. We will begin with the `Protocol` abstract class.

`Protocol`

The `Protocol` class inherits `Mobility::MobileApp`, a subclass of `DTN::Application` capable of receiving new-neighbor notifications. Most of the class functionality is abstract, and so will need to be defined by a concrete subclass. There are, however, a few things implemented directly in `Protocol`.

The abstract class holds a small amount of state common to (nearly) all forwarding protocols. It holds pointers to the `Mobility::MobileNode` to which it's attached and the `DTN::Link` corresponding to the antenna. It also holds the lifetime of the protocol's control message bundles. These data elements are declared "private," and so are accessible to subclasses only through protected accessors.

When the virtual `setOwner()` method is called, `Protocol` will call the normal base-class `Consumer::setOwner()` method, and then register an instance of `Forwarding` as the forwarding policy for the node. `Forwarding` is configured with the `DTN::Node`, the link, and a pointer to the `Protocol`.

`Protocol` provides a `visible()` method that takes a node address and forwards it to `Mobility::MobileNode::visible()`. This uses the **`mobility`** module's built-in tracking of what other mobile nodes are believed to be within range of the antenna. A `tweak()` method allows the protocol to modify bundles passing through; the default method definition does nothing, but may be overridden by subclasses.

Several methods are declared (or remain) abstract, and must be defined by a subclass. The standard `DTN::Application::appID()` and `DTN::Application::process()` methods are undefined, as is `Mobility::MobileApp::newNeighbor()`. Recall that `process()` and `newNeighbor()` are the triggering and message-handling methods for a mobile application. To these we add the `Protocol::getLink()` method, which takes an address and returns a pointer to a `DTN::Link`. This link will generally be of the type `Mobility::MockLink`, which aliases another link using a specific node's address as the remote endpoint.

Forwarding

Forwarding is a concrete subclass of DTN::ForwardingPolicy. It implements the forward() and forwardOn() methods. Mostly, this class acts as an interface between the DTN::Node and the Protocol.

Every DTN::Bundle that passes through a DTN::Node is passed to the forward() method, so this is where we call tweak(). Many protocols will not reimplement this method, but as we will see in Section 8.2.3, it can be useful. Next we test if the link is available. If so, and the bundle is a broadcast, we send it. If the link is available, but the bundle is not a broadcast, we ask the Protocol if the marked receiver is visible. If it is, we get the outgoing link and send the bundle. Note that a newly arrived bundle will never have a visible receiver, and so will not be forwarded immediately.

The forwardOn() method returns the first DTN::Bundle found in the volatile store that has a visible receiver. Bundles with no-longer-visible receivers are removed from the volatile store, since they cannot be routed.

You might have noticed that this forwarding policy seems incomplete. As presented, it appears that no bundles will ever be forwarded. This is where the interactions with Protocol come into play. We will see how this works shortly. First, we will look at a simple concrete subclass of Protocol that does not forward bundles.

8.2.2 A Do-Nothing Protocol

In this section, we define the following class:

```
class DoNothingProtocol : public MobileForwarding::Protocol
{
   public :
      static DTN::ByteString  kIdentifier;
      DoNothingProtocol( Mobility::MobileNode* n,
                         DTN::Link* l,
                         const struct timeval& t );
      virtual ~DoNothingProtocol();

      void newNeighbor( const DTN::ByteString& addr );
      DTN::Link* getLink( const DTN::ByteString& addr );

   protected :
      const DTN::ByteString& appID() const;
      void process( const DTN::Bundle& b );
};
```

As the name implies, this protocol will not actually do anything, but will serve as an example of how, at a minimum, to define a Protocol.

We provide the application identifier with the following code:

```
DTN::ByteString DoNothingProtocol::kIdentifier((unsigned char*)"donothing");

const DTN::ByteString&
DoNothingProtocol::appID() const
{
   return kIdentifier;
}
```

A more meaningful subclass would look virtually identical.

The constructor and destructor are similarly simple:

```
DoNothingProtocol::DoNothingProtocol( Mobility::MobileNode* n,
                                      DTN::Link* l,
                                      const struct timeval& t ) :
   MobileForwarding::Protocol(n,l,t) {}

DoNothingProtocol::~DoNothingProtocol() {}
```

This leaves the control methods. Two of these are easy, as they do not return anything:

```
void
DoNothingProtocol::newNeighbor( const DTN::ByteString& addr ) {}

void
DoNothingProtocol::process( const DTN::Bundle& b ) {}
```

The final method we need to provide, `getLink()`, returns a pointer to a `DTN::Link`, and is called by `Forwarding::forward()` and `Forwarding::forwardOn()`. In both cases, a null pointer indicates an un-forwardable bundle:

```
DTN::Link*
DoNothingProtocol::getLink( const DTN::ByteString& addr ) { return 0; }
```

The following bootstrapping code will allow us to compile and “use” this protocol:

```
static PyObject*
donothing_attach( PyObject* self, PyObject* args )
{
   if ( PyTuple_Size( args ) < 2 )
   {
      PyErr_SetString(PyExc_TypeError, "attach expects two arguments");
      return 0;
   }

   Mobility::MobileNode* mn = Mobility::parse_mn( PyTuple_GetItem(args,0) );
   if ( 0 == mn ) return 0;

   Mobility::WirelessLink* wl = Mobility::parse_wl( PyTuple_GetItem(args,1) );
   if ( 0 == wl ) return 0;

   DoNothingProtocol* prot = new DoNothingProtocol( mn, &(wl->link()), mn->bundleLifetime().tv );
   mn->addMobileApp(prot);

   Py_INCREF(Py_None);
   return Py_None;
}
```

Remember to include `mobileforwarding` in the `Makefile`’s `LIBRARIES` variable. The resulting module will load, and can be attached to nodes. Every bundle will be dropped, however. You will also want the following python code:

`__init__.py`

```
from _donothing import *
import pydtn.mobility
pydtn.mobility._default_forwarding = attach
```

8.2.3 Basic Epidemic Forwarding

The pydtn distribution comes with two epidemic forwarding packages. The **epidemic** package is a relatively full-featured implementation of epidemic forwarding, while **basicepidemic** is a much simpler protocol, and is implemented using the **mobileforwarding** library.

Here is the declaration of the epidemic protocol class from **basicepidemic**:

EpidemicApp.h

```
#include "mobileforwarding/Protocol.h"
#include "mobility/MockLink.h"
#include "dtn/Bundle.h"
#include "dtn/BundlePointer.h"
#include <list>

namespace BasicEpidemic
{
   class EpidemicApp : public MobileForwarding::Protocol
   {
      public :
         typedef std::list< DTN::ByteString >  DigestList;
         static DTN::ByteString  kIdentifier;

         static unsigned char kSummaryVector;
         static unsigned char kDataRequest;

         typedef std::map< DTN::ByteString , Mobility::MockLink* > LinkMap;

         EpidemicApp( Mobility::MobileNode* n, DTN::Link* l, const struct timeval& t );
         virtual ~EpidemicApp();

         void newNeighbor( const DTN::ByteString& addr );
         DTN::Link* getLink( const DTN::ByteString& addr );
         void tweak( DTN::Bundle* b );

      protected :
         const DTN::ByteString& appID() const;
         void process( const DTN::Bundle& b );

      private :
         static DTN::ByteString digest( const DTN::BundlePointer& p );
         static DigestList* parseSummaryVector( const DTN::ByteString& sv );

         DTN::BundlePointer storedBundle( const DTN::ByteString& d );

         LinkMap  m_linkMap;
   };
}
```

The first thing you might notice is that it's considerably longer than the declaration in the previous section for `DoNothingProtocol`. We've added a couple of `typedef`s, and two message types, `kSummaryVector` and `kDataRequest`. We're going to reimplement `tweak()`, and we've added private methods `digest()` and `parseSummaryVector()` (both static), and `storedBundle()`. Finally, we have a map of addresses to `Mobility::MockLinks`.

The identifier "basicepidemic" is set in the same manner as "donothing" in the previous section, and the constructor merely forwards its arguments to that of `MobileForwarding::Protocol`. The destructor cleans up the `LinkMap`. The remainder of the methods implement the interesting

parts of the protocol.

We begin the detailed discussion with the `tweak()` method. We use this in **basicepidemic** to suppress bundle acknowledgments by adding the `DTN::Bundle::kNoACK` flag to every bundle handled. These ACKs are an unnecessary use of bandwidth, given how epidemic forwarding operates, and can cause bookkeeping issues for the persistent store.

Epidemic forwarding uses message digests to identify bundles that have or have not been received by a node. We provide a static `digest()` method to compute the digest for a bundle (from a `DTN::BundlePointer`. These digests are concatenated to form a summary vector, and the static `parseSummaryVector()` method splits a summary vector into a list of individual message digests. The `storedBundle()` method takes a digest and checks the persistent store at the node for the corresponding bundle. This is possible because the digest is easily converted back into the address of the source and the bundle sequence number from that source.

In the `newNeighbor()` method, the protocol establishes a new `Mobility::MockLink` for the new neighbor, placing it in the link map, where it can be retrieved later by `getLink()`. If the node has any bundles in its persistent store, a summary vector is then composed from the digests and sent to the new neighbor. Calling the `DTN::Node::forward()` method will in turn call the `MobileForwarding::Forwarding::forward()` method, which will successfully retrieve a link from `EpidemicApp::getLink()`, as the link was just created and inserted into the map.

The `process()` method, when receiving a summary vector, first parses the summary vector into individual digests. It loops over these digests, looking up each one using `storedBundle()`. If this call fails, the digest is added to a data request message. If any digests were not found locally, the data request message is sent to the node that sent the summary vector. Receipt of the summary vector message triggers `newNeighbor()`, if appropriate, so the data request will match a link when passed to `MobileForwarding::Forwarding::forward()`.

When `process()` receives a data request, it again parses the list of bundles, which has the same format as the summary vector. For each digest in the request, the protocol finds the associated bundle. The bundle's receiver is set to be the request sender, for which we hold a `MockLink` in the map, and the bundle is passed to `DTN::Node::forward()`.

The bootstrapping code, in addition to creating the protocol and attaching it to the node, ensures that a `DTN::CustodyPolicy` has been set. If the node does not currently have a custody policy, then the simple `DTN::SpaceAvailCustody` policy is set. This is needed to handle the storage of bundles in the persistent store.

8.2.4 Modifying the Epidemic Protocol

All files in **mobileforwarding** and **basicepidemic** are extensively commented, so you are encouraged to look through the source code for additional insight into how it's structured. Copying **basicepidemic** is much the same as copying any other module, and its comparatively simple structure should make it a straight-forward base from which to build.

For instance, say we want to modify the simple epidemic forwarding implementation to prioritize last-hop delivery. That is, when we encounter a node, any bundles for which that node is the destination are sent immediately, rather than added to the protocol control message.

The only change we need to make, protocol-wise, is in `newNeighbor()`. We start with the version from **`basicepidemic`** (slightly simplified):

EpidemicApp.cc

```
void
EpidemicApp::newNeighbor( const DTN::ByteString& addr )
{
   if ( 0 == m_owner ) return;

   if ( m_linkMap.end() == m_linkMap.find(addr) )
      m_linkMap[addr] = new Mobility::MockLink(link(),addr);

   DTN::ByteString data;
   data.push_back( kSummaryVector );
   DTN::BundlePointer itr = m_owner->cachedPersistent();
   for ( ; ! itr.isNull() ; itr = itr.next() )
   {
      data.append( digest(itr) );
   }
   if ( 1 == data.length() ) return;

   struct timeval now = DTN::dtn_time();
   struct timeval exp = DTN::dtn_time();
   exp.tv_sec += lifetime().tv_sec;
   exp.tv_usec += lifetime().tv_usec;
   while ( exp.tv_usec > 1000000 ) { exp.tv_usec -= 1000000; exp.tv_sec += 1; }

   DTN::Bundle* b = new DTN::Bundle( m_owner->nextSeq(), m_owner->addr(), addr,
                                     data, now, exp, DTN::Bundle::kData, appID() );
   b->recv() = addr; // We have to set the receiver now.

   // Send the message via the node.
   m_owner->forward(b);
}
```

The part that we are going to change is the following loop:

```
   DTN::BundlePointer itr = m_owner->cachedPersistent();
   for ( ; ! itr.isNull() ; itr = itr.next() )
   {
      data.append( digest(itr) );
   }
```

We need to check each bundle's destination, to see if it matches the new neighbor:

```
   DTN::BundlePointer itr = m_owner->cachedPersistent();
   for ( ; ! itr.isNull() ; itr = itr.next() )
   {
      DTN::Bundle* pB = itr.repr()->bundle();
      if ( pB->destination() == addr )
      {
      }
      else
      {
         data.append( digest(itr) );
      }
   }
```

At this point, only bundles with destinations different from the newly seen node are added to the summary vector. Those that match have to be sent immediately:

```
DTN::BundlePointer itr = m_owner->cachedPersistent();
for ( ; ! itr.isNull() ; itr = itr.next() )
{
   DTN::Bundle* pB = itr.repr()->bundle();
   if ( pB->destination() == addr )
   {
      DTN::Bundle* copyB = pB->clone();
      copyB->recv() = addr;
      m_owner->forward(copyB);
   }
   else
   {
      data.append( digest(itr) );
   }
}
```

Note that we clone the `DTN::Bundle` held by the persistent store, since we're transferring ownership of the object's memory when we call `DTN::Node::forward()`.

This was a very simple modification. Other modifications might require that additional data structures be added to `EpidemicApp`. For example, if destinations send innoculating acknowledgments, then nodes would have to clean up their persistent stores, while the protocol would need to maintain an additional list of bundles that have been seen and (possibly) removed. Additional protocol messages might also be added, for more nuanced propagation decisions.

Chapter 9

Policies

Where applications affect node behavior through the exchange of protocol messages, policies affect node behavior through internal state. In many cases, a particular behavior will depend both on a policy and a protocol. For instance, the **`mobileforwarding`** package described in Section 8.2 contains both an abstract `Application` and a simple `ForwardingPolicy`. The three types of policies—forwarding, volatile storage, and custody—were discussed briefly in Section 5.1.4. Here we present more detail and some concrete examples.

The text of the examples will generally be reformatted and simplified slightly. The program logic, however, is identical to the actual source code, and direct comparison should be straightforward.

9.1 Forwarding

A `DTN::ForwardingPolicy` determines whether and how bundles are forwarded towards their destinations. For a static network, a simple example is a routing table that associates a next-hop node with every address or block of addresses. Mobile protocols might also employ routing tables (whether through base stations or ad-hoc), though in a DTN effective forwarding is less likely to rely on routing paths.

9.1.1 `forward()`

The prototype for this method is

```
bool DTN::ForwardingPolicy::forward( DTN::Bundle* b )
```

The `forward()` method makes an immediate decision about forwarding a bundle, sending it if appropriate. Generally, this involves determining first *how* a bundle should be forwarded and then if it *can* be forwarded. A couple of examples should illustrate this.

First, consider the implementation `DTN::RoutedForwarding::forward()`, which employs routing tables:

```
dtn/RoutedForwarding.cc

bool RoutedForwarding::forward( Bundle* b )
{
   if ( 0 == m_owner ) return false;
   try
   {
      Link& l = target( *b );
      if ( l.available() ) { m_owner->send( b, l ); return true; }
   } catch ( ... ) { return false; }
   return false;
}
```

The `target()` method (specific to `RoutedForwarding`) matches a bundle's destination with an appropriate outgoing link. How it does this is unimportant for our discussion. If that link is available, the bundle is sent and the method returns `true`. Otherwise, or if an error occurs, it returns `false`.

Route-based forwarding is relatively straightforward, so let's now look at a more complex example. The following is `MobileForwarding::Forwarding::forward()`:

```
mobileforwarding/Forwarding.cc

bool Forwarding::forward( DTN::Bundle* b )
{
   if ( 0 == b ) return false;
   if ( 0 == m_owner ) return false;
   if ( 0 == m_link ) return false;
   if ( 0 == m_prot ) return false;
   m_prot->tweak(b);
   if ( m_link->available() )
   {
      try
      {
         if ( DTN::Bundle::kBcast & b->type() ) { m_owner->send( b , *m_link ); return true; }
         if ( m_prot->visible( b->recv() ) )
         {
            DTN::Link* l = m_prot->getLink( b->recv() );
            if ( 0 != l ) { m_owner->send( b , *l ); return true; }
         }
      } catch ( ... ) {}
   }
   return false;
}
```

In many respects, this is not much different than the previous example. There is only one link, so we can test its availability first. We still do what amounts to a table lookup, but this time we're looking for a link *alias* based on the already-determined *next hop*, which must still appear to be in our transmission range. Since we already know that the link is available, we need only test that a valid alias was returned. The bundle is sent using the alias, rather than the actual link, in order to ensure that the bundle's fields are marked correctly. Special handling of broadcasts preempts the search for a specific neighbor.

Note that for `MobileForwarding::Forwarding` the matching of bundles to their next-hop recipients is done somewhere other than the `forward()` method. As we saw in Section 8.2, this is typically handled by a subclass of `MobileForwarding::Protocol` when exchanging

protocol messages.

There are a few important things to keep in mind when implementing a `forward()` method. First is that it's a request for immediate service. Second is that it should not, in general, modify state. Caching information about a bundle will come later, *if* the bundle is placed in volatile storage. Third, ownership of the memory is transferred *if and only if* the bundle can be forwarded. The return value must correspond to whether the bundle has changed hands.

9.1.2 `forwardOn()`

The prototype for this method is

```
DTN::Bundle* DTN::ForwardingPolicy::forwardOn( const DTN::Link& l )
```

The `forwardOn()` method searches the volatile store for a bundle that can be sent along the specified link when that link becomes available, either due to a state change or completed enqueuing of a previous bundle. If a suitable bundle is found, it must be removed from the volatile store without being deallocated. `forwardOn()` is generally more complex than `forward()`. Again, we illustrate with a couple of examples.

The `RoutedForwarding` class maintains per-link queues of stored bundles, which it uses to schedule the next transmission:

dtn/RoutedForwarding.cc

```
Bundle* RoutedForwarding::forwardOn( const Link& l )
{
   if ( 0 == m_owner ) throw Exception( __FILE__ , __LINE__ );
   BundleCache::iterator itr = m_cache.find( &l );
   if ( m_cache.end() == itr ) return 0;
   while ( 0 != itr->second.size() )
   {
      BundlePointer p = itr->second.front();
      itr->second.pop_front();
      if ( m_owner->validate(p) )
      {
         Bundle* b = p.repr()->bundle();
         m_owner->remove(b,false);
         return b;
      }
   }
   return 0;
}
```

The policy's cache is a map with a `DTN::Link*` as the key and a list of `DTN::BundlePointers` as the value. If the queue for a link is not empty, then the first valid bundle in the queue is returned, after unregistering it from the volatile store. Cached `BundlePointers` that refer to already-removed bundles are discarded. If there's no matching bundle, a null pointer is returned.

The `forwardOn()` method for `MobileForwarding::Forwarding` is, again, more complicated than that of the routing-table-based policy:

```
mobileforwarding/Forwarding.cc

DTN::Bundle* Forwarding::forwardOn( const DTN::Link& l )
{
   if ( 0 == m_owner ) return 0;
   DTN::Bundle* retval = 0;
   DTN::BundlePointer p = m_owner->cachedVolatile();
   DTN::BundlePointer prev;
   while ( ! p.isNull() )
   {
      DTN::Bundle* b = p.repr()->bundle();
      if ( ( !(DTN::Bundle::kBcast & b->type()) ) &&
           ( ( ! m_prot->visible( b->recv() ) ) || ( 0 == m_prot->getLink( b->recv() ) ) ) )
      {
         m_owner->remove(b,true,DTN::NoRouteDrop::inst);
         p = prev;
      }
      else if ( 0 == retval )
      {
         retval = b;
      }
      prev = p;
      if ( p.isNull() )
         p = m_owner->cachedVolatile();
      else
         p = p.next();
   }
   if ( 0 != retval ) m_owner->remove(retval,false);
   return retval;
}
```

This method cleans up the cache more aggressively. It looks at every bundle in the volatile store (it holds no cache of its own), and if the recipient is no longer visible (excluding broadcast bundles) the bundle is dropped. This cleanup comprises the bulk of the method. Since there is only one link, there's no need to match bundles to links. The first deliverable bundle we see is marked to be sent.

9.1.3 `cache()`

The prototype for this method is

```
void DTN::ForwardingPolicy::cache( DTN::BundlePointer& p )
```

The `cache()` method allows a `DTN::ForwardingPolicy` to keep track of the bundles in the volatile store. When a `DTN::Node` adds a bundle to the volatile store, it hands the returned `DTN::BundlePointer` to the policy's `cache()` method.

In the case of `DTN::RoutedForwarding`, this works as follows:

```
dtn/RoutedForwarding.cc

void RoutedForwarding::cache( BundlePointer& p )
{
   try
   {
      const Link& l = target( *(p.repr()->bundle()) );
      m_cache[ &l ].push_back( p ); // FIFO
   } catch ( ... ) {}
}
```

We use the same `target()` method to find the appropriate link for the bundle. The array-subscript access to the cache map creates an entry for the link, if necessary. We ignore any errors that occur.

For `MobileForwarding::Forwarding`, we have no internal cache. Consequently, we use the default implementation `DTN::ForwardingPolicy::cache()`, which does nothing.

9.2 VolatileStore

A `DTN::VolatileStorePolicy` determines whether a bundle that could not be forwarded should be placed in the volatile store. It also determines when a bundle should be removed from the store (aside from it being successfully forwarded). Compared to a `ForwardingPolicy`, `VolatileStorePolicy` is fairly simple.

There is, in fact, only one abstract method that a `VolatileStorePolicy` must implement:

```
DTN::BundlePointer DTN::VolatileStorePolicy::store( DTN::Bundle* b )
```

Given a pointer to a `DTN::Bundle`, the policy determines whether the bundle can be stored, and returns an appropriate `DTN::BundlePointer`. If the bundle could not be stored, a null pointer is returned and the bundle is dropped by the node; otherwise, the returned object contains a reference to the bundle in the node's `DTN::VolatileBundleStore`.

The **`dtn`** package provides a simple policy, called `DTN::DropTail`. As the name implies, it behaves much like a drop-tail queue. There are slight differences, however. The most notable is that it treats the volatile store as a shared queue among all outgoing links. Another notable difference is that the policy is size-based, not count-based. That is, a standard drop-tail queue typically allows for a certain maximum number of packets to be enqueued. The `DropTail` policy, however, allows a bundle to be stored as long as there is enough space for it in the bundle store. That means that, without any bundles being forwarded, one bundle might be rejected by the policy while the next bundle is accepted, since the first would overflow the store but the second would not.

Here is the definition of the `store()` method:

```
dtn/DropTail.cc

BundlePointer DropTail::store( Bundle* b )
{
   if ( ( 0 == b ) || ( 0 == m_owner ) || ( 0 == m_store ) ) return BundlePointer();
   unsigned int size = b->size();
   unsigned int avail = m_owner->volatileCap() - m_owner->usedVolatileCap();
   if ( size > avail ) return BundlePointer();
   return m_store->addBundle(b);
}
```

More complex policies might attempt to remove old or low-priority bundles in favor of an incoming bundle. This can address issues such as head-of-line blocking or quality of service.

9.3 Custody

A `DTN::CustodyPolicy` is similar to a `VolatileStorePolicy`, with a little added functionality. Like the latter, it controls whether bundles are placed in a bundle store, in this case the persistent store. The custody policy also provides a hook for re-sending stored bundles, which for the volatile store is handled by `DTN::ForwardingPolicy::forwardOn()`.

9.3.1 `takeCustody()`

The prototype for this method is

```
DTN::CustodyPolicy::CustodyReturn DTN::CustodyPolicy::takeCustody( const DTN::Bundle& b )
```

This differs from `DTN::VolatileStorePolicy::store()` in two major ways:

1. This method takes a constant reference to the `DTN::Bundle`, rather than a pointer to it.
2. The return value is a status code, not a reference into the store.

The first difference is because the persistent store must *copy* the bundle; the memory is still owned by the node at this point. The second difference is because the persistent store is designed to be longer-term, and the node is unlikely to need access to this particular stored bundle in the near term. We take advantage of this by using a more expressive return value than a simple boolean.

The return value can be `DTN::CustodyPolicy::kCustodyTaken`, which means that a copy of the bundle is now in the persistent store, or `DTN::CustodyPolicy::kNoCustody`, which means the bundle was not placed in the store, or `DTN::CustodyPolicy::kDropBundle`, which means that not only was the bundle not stored, but the original bundle should be dropped by the node. This last value might be used in cases where the node's semantics dictate that a non-broadcast data bundle *must* be stored custodially if it's going to be forwarded.

As an example, the **`dtn`** package includes `DTN::SpaceAvailCustody`, which is similar to the volatile store policy `DTN::DropTail`. Here is the `takeCustody()` method:

`dtn/SpaceAvailCustody.cc`

```
CustodyPolicy::CustodyReturn SpaceAvailCustody::takeCustody( const Bundle& b )
{
   if ( 0 == m_owner ) return kDropBundle;
   if ( 0 == m_store ) return kNoCustody;
   Bundle::BundleType t = b.type();
   if ( ( Bundle::kBcast & t ) || ( Bundle::kACK & t ) || !( Bundle::kCustodial & t ) )
      return kNoCustody;
   if ( m_owner->addr() == b.custodian() ) return kNoCustody;
   unsigned int size = b.size();
   unsigned int avail = m_owner->persistentCap() - m_owner->usedPersistentCap();
   if ( size > avail ) { m_owner->signalExhausted(b); return kNoCustody; }
   Bundle* pB = b.clone();
   pB->custodian() = m_owner->addr();
   BundlePointer ptr = m_store->addBundle( pB );
   if ( ptr.isNull() ) return kNoCustody;
   return kCustodyTaken;
}
```

Note that we return `kDropBundle` if the policy is mis-configured. We return `kNoCustody` if the store has insufficient capacity for the bundle, if the bundle is of a type that should not be stored custodially (broadcasts, acknowledgments, and bundles explicitly marked as non-custodial), if this node is *already* the custodian, or on error. `kCustodyTaken` is returned if the bundle was successfully copied and placed in the store.

One effect of this implementation is that if a bundle was stored custodially and then removed, on subsequent examination of the same bundle it will still record the current node as the custodian, and it will not be stored. An alternative (but less efficient) implementation would search the persistent store explicitly for the bundle.

9.3.2 `policyRemove()`

The prototype for this method is

```
bool DTN::CustodyPolicy::policyRemove( const DTN::Bundle& b )
```

It is called when a node receives an acknowledgment for a bundle, and determines whether the node should direct the persistent store to remove the bundle. The default implementation in `DTN::CustodyPolicy` always returns true, so that handing off a bundle to another custodian or the destination causes the bundle to be removed from the persistent store. The reimplementation in `Epidemic::EpidemicStore` always returns false.

9.3.3 `retry()`

The prototype for this method is

```
void DTN::CustodyPolicy::retry()
```

This is called by `DTN::Node::retryStored()`, which is in turn periodically called by `WrapNode`. The purpose of this method is for the custody policy to provide reliable delivery of stored bundles. Most implementations will likely look similar to that of `DTN::SpaceAvailCustody`:

`dtn/SpaceAvailCustody.cc`

```
void SpaceAvailCustody::retry()
{
   if ( ( 0 == m_owner ) || ( 0 == m_store ) ) return;
   BundlePointer itr = m_store->getPointer( BundlePointer() );
   for ( ; ! itr.isNull() ; itr = itr.next() )
   {
      m_owner->forward( itr.repr()->bundle()->clone() );
   }
}
```

Note that we clone the stored bundle, since the node will have to control the memory for the `DTN::Bundle` passed to `forward()`.

Chapter 10

Wireless Links

Mobile nodes employ links with only one fixed endpoint. We have seen this is Section 3.2 with the **pydtn.mobility.rf** module, which implements a simple disk model for antenna range and capacity. We'll discuss this module in a bit, after an overview of the abstract base on which it is built.

The **pydtn.mobility** module includes a class WirelessLink, which provides the abstraction for antennae and similar links. The structure of the class is:

```
class WirelessLink : public WrapLink
{
   public :
      WirelessLink();
      virtual ~WirelessLink();

      Entity* create() const = 0;
      void configure( const ArgList& args );
      bool handler( BundleEvent& event ) = 0;
      std::string identifier() const { return "link:wireless"; }
      MobileNode* node() { return m_node; }
      const MobileNode* node() const { return m_node; }

   protected :
      MobileNode*  m_node;
};
```

Note that this inherits WrapLink, not DTN::Link.

The configure() method redefines "connect" so that only one node is specified, and it must inherit MobileNode. This node is set as the source *and destination* of the underlying link, as well as the internally-held MobileNode. The reason we have to set the destination is that a non-null pointer is required by WrapLink. The result of this is that bundles sent through this link will appear to have the sender as the next-hop receiver. Most mobile forwarding protocols will also use the MockLink class (5.3.4), though this is unimportant for understanding the behavior of a WirelessLink.

We do not instantiate WirelessLink directly, so its create() method is pure virtual. Similarly, the handler() is left unspecified, since its behavior will depend upon the specific antenna model used.

10.1 Simple Disk Model

Currently, only one antenna model is implemented, in the module **`pydtn.mobility.rf`**. This is an extremely simple disk model, in which signal strength and quality is uniform within a configured radius from the node, and drops to nothing outside of this range. Moreover, this is a transmission range—any node within this range can receive messages, even if its own antenna has a smaller reach. There is perfect duplexing, and no interference.

The **`rf`** module defines one class, `RFLink`:

```
class RFLink : public Mobility::WirelessLink
{
   public :
      typedef std::list< Mobility::MobileNode* > NodeList;

      RFLink();
      virtual ~RFLink();

      Entity* create() const;
      void configure( const ArgList& args );
      InterpreterItem get( const ArgList& args );

      bool handler( BundleEvent& event );

      std::string identifier() const { return Mobility::WirelessLink::identifier() + ":rf"; }

      double range() const { return m_range; }

   private :
      static NodeList  listeners;
      double  m_range;
};
```

To start with the obvious, the disk model link adds a transmission range (in meters). The `get()` method is overridden to provide this configured value using the "range" directive.

`RFLink` also contains a list of mobile nodes, shared by all instances. When a link is connected to a node, that node is added to this list:

```
RFLink.cc

void
RFLink::configure( const ArgList& args )
{
   unsigned int nargs = args.size();
   // ...
   std::string command = parse_string(args[0]);
   if ( "connect" == command )
   {
      WirelessLink::configure(args);
      if ( 0 != m_node ) { listeners.push_back( m_node ); }
      return;
   }
   //...
}
```

Reachable nodes are determined on demand, both by the `handler()` and the "visible" directive to `get()`. The latter returns a list of the relevant nodes' addresses. If the link is "up,"

`handler()` loops over *all* nodes in its list of listeners, and send the bundle to any of them within range. For an iterator `itr` into the list of listeners, a relative position is calculated:

RFLink.cc

```
Mobility::Position  rcvrPos = (*itr)->position() - pos;
```

We then do a quick comparison, based on a cube centered on the sender:

RFLink.cc

```
if ( fabs( rcvrPos.x() ) > m_range ) continue;
if ( fabs( rcvrPos.y() ) > m_range ) continue;
if ( fabs( rcvrPos.z() ) > m_range ) continue;
```

This allows us to reject far-away nodes quickly. Only for nodes within this cube do we do the slower spherical comparison:

RFLink.cc

```
double norm = rcvrPos.x() * rcvrPos.x() + rcvrPos.y() * rcvrPos.y() + rcvrPos.z() * rcvrPos.z();
if ( norm > rnorm ) continue;
```

For any node passing all of these tests, the bundle is copied, encapsulated in a new `BundleEvent` with that node as the handler, and scheduled.

In the module's python code, we handle default link parameters:

__init__.py

```
_defaults = {}
_defaults['range'] = None
_defaults['latency'] = None
_defaults['bandwidth'] = None

def set_range( r ):
    """Configure the default transmission range, in meters."""
    global _defaults
    _defaults['range'] = r

def set_latency( l ):
    """Configure the default link latency, in seconds."""
    global _defaults
    _defaults['latency'] = l

def set_bandwidth( bw ):
    """Configure the default link bandwidth, in Mbps."""
    global _defaults
    _defaults['bandwidth'] = bw
```

We have seen these previously in Section 3.2. These default values are used in a link-creation function:

```
__init__.py

def _rf_link():
    global _defaults
    if _defaults['range'] is None:
        raise UnboundLocalError('No transmission range was specified.')
    if _defaults['latency'] is None:
        raise UnboundLocalError('No latency was specified.')
    if _defaults['bandwidth'] is None:
        raise UnboundLocalError('No bandwidth was specified.')

    l = sim.Entity('link:wireless:rf')
    for k in _defaults.keys():
        l.config(k,_defaults[k])
    return l
```

This, in turn, is automatically hooked into **`pydtn.mobility`**'s node-creation machinery:

```
__init__.py

pydtn.mobility._default_link = _rf_link
```

If all antenna models are designed similarly, then simply importing the appropriate module sets up the default wireless link, with the last one loaded taking precedence.

Chapter 11

Mobile Nodes

For the most part, the MobileNode class in **pydtn.mobility** should suit your needs. It supports waypoint-based mobility, including random-waypoint (through the RandomWaypointNode subclass). Regular MobileNodes can be configured with set mobility patterns for repeatable paths as other aspects of the simulation are varied. RandomWaypointNodes, in contrast, can be configured with a WaypointRange object that selects a new waypoint when needed. Other mobility patterns are possible, however.

Since configuring fixed waypoints is straightforward, we begin with a discussion of the existing WaypointRanges and defining new subclasses. We follow this with a discussion of other mobility models.

11.1 Random Waypoint Geometries

The base class WaypointRange provides a generic interface:

```
class WaypointRange
{
   public :
      WaypointRange() {}
      virtual ~WaypointRange() {}

      virtual Position next(const Position& p) = 0;
};
```

The relevant method of this class is next(), which takes the current waypoint and selects the next one. In RandomWaypointNode, the speed with which the node moves is chosen uniformly at random between a minimum and maximum value (in meters per second), and the schedule for the waypoint is computed from the distance to travel and the speed.

11.1.1 Existing Geometries

Simple Rectangles

The simplest geometry defined in **pydtn.mobility** is RectangularAlignedWaypointRange:

```
class RectangularAlignedWaypointRange : public WaypointRange
{
   public :
      RectangularAlignedWaypointRange( const Position& ll, const Position& ur );
      virtual ~RectangularAlignedWaypointRange();

      Position next( const Position& p );

   private :
      Position  m_lowerLeft;
      Position  m_extent;
};
```

The contructor is passed a lower-left corner and an upper-right corner. From these the vector is computed from the lower-left to the upper-right.

```
RectangularAlignedWaypointRange.cc

RectangularAlignedWaypointRange::RectangularAlignedWaypointRange( const Position& ll,
                                                                  const Position& ur ) :
   m_lowerLeft( ll ), m_extent( ur - ll ) {}
```

The cartesian components of this vector are used to generate random points, uniformly distributed, within the rectangular region defined.

```
RectangularAlignedWaypointRange.cc

Position
RectangularAlignedWaypointRange::next( const Position& p )
{
   double rx = prng();
   double ry = prng();
   double rz = prng();
   return ( m_lowerLeft + Position( rx*m_extent.x() , ry*m_extent.y() , rz*m_extent.z() ) );
}
```

The `prng()` function calls the RANLUX pseudorandom number generator, and is the call behind the `sim.random()` python method.

Note that the name of the class comes from the fact that the region defined has edges parallel to the cartesian axes. This is a very restrictive geometry, but is likely to be useful in many situations. Note also that the positions are three-dimensional, as are all mobile node positions, so technically the region defined is a rectangular solid.

Arbitrary Parallelepipeds

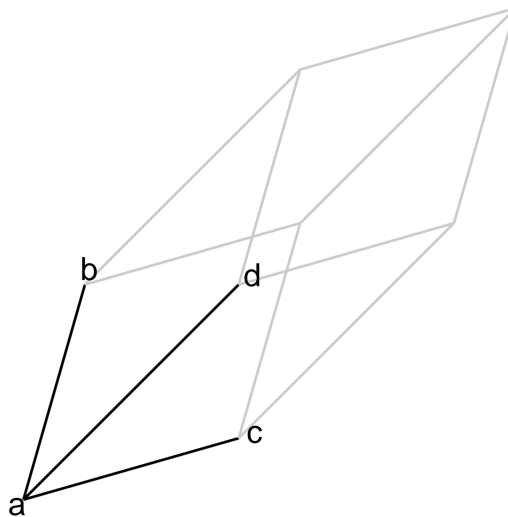
A slightly more complex geometry is supported by the class `ParallelogramWaypointRange`. Again, all positions are in three dimensions, making the range a parallelepiped, rather than a parallelogram.

```
class ParallelogramWaypointRange : public WaypointRange
{
   public :
      ParallelogramWaypointRange( const Position& a, const Position& b,
                                  const Position& c, const Position& d );
      virtual ~ParallelogramWaypointRange();

      Position next( const Position& p );

   private :
      Position  m_vertex;
      Position  m_side1;
      Position  m_side2;
      Position  m_side3;
};
```

The relation between these points and the region defined is:



ParallelogramWaypointRange.cc

```
ParallelogramWaypointRange::ParallelogramWaypointRange( const Position& a, const Position& b,
                                                        const Position& c, const Position& d ) :
   m_vertex(a), m_side1(b-a), m_side2(c-a), m_side3(d-a) {}
```

Selecting a point in this region is similar to the previous example, except that we are scaling the vectors defining the sides of the region, rather than the components of a single vector. That is:

ParallelogramWaypointRange.cc

```
Position
ParallelogramWaypointRange::next( const Position& p )
{
   double r1 = prng();
   double r2 = prng();
   double r3 = prng();
   return ( m_vertex + r1*m_side1 + r2*m_side2 + r3*m_side3 );
}
```

11.1.2 Creating New Geometries

In general, the most difficult part of defining a new subclass of WaypointRange is configuring the RandomWaypointNode to use it. The configure() method employs *duck typing* to interpret its arguments, and hence can only recognize geometries that are hard-coded in. RandomWaypointNode also provides a setRange() method that takes a WaypointRange. This gives you a mechanism by which to configure a node with your custom range.

For example, let's say you have a WaypointRange called Shell, which defines mobility at a fixed radius from the origin. Your module would then have a python-callable function that looks something like (omitting some details and error-checking):

```
static PyObject*
shell_attach( PyOBject* self, PyObject* args )
{
   Entity::ArgList argList;
   build_arglist( argList, args );
   ItemWrapper item = resolve_symbol( argList[0] );
   if ( 0 == item.type.find( Mobility::RandomWaypointNode::kIdentifier ) )
   {
      Mobility::RandomWaypointNode* n = (Mobility::RandomWaypointNode*)(item.item);
      double r = parse_double( argList[1] );
      n->setRange( new Shell(r) );
   }
   clean_arglist( argList );
   Py_INCREF(Py_None);
   return Py_None;
}
```

In your python scripts, you would use this similarly to the attach() methods we've seen for Applications.

11.2 Other Mobility Patterns

The MobileNode class is based on waypoints to define mobility, with nodes moving in straight-line paths at constant velocity between successive waypoints. Of course, this is an extremely limited mobility pattern, and will not suit all applications.

The way to change the basic behavior is, perhaps unsurprisingly, to define a new subclass of MobileNode. This is what RandomWaypointNode does in order to have automatically updating mobility. As we'll see, derived classes need not even concern themselves with waypoints.

11.2.1 Parametrized Movement

An obvious new mobility pattern is to specify the node's position as some sort of parametrized function. The node will still have waypoints, all MobileNodes do, but they need not have any real impact on the node's position.

To implement parametrized movement, your subclass would override MobileNode's position(t) method. That is, the version of position() that takes a Time argument. The version with a

void argument list simply calls the former with the current time, and so need not be redefined. Aside from a mechanism for setting the parameters, that's it.

As a trivial example, consider `LinearMotionNode`, which simply moves in a straight line at a constant speed. Here's what its `position()` method might look like:

```
Mobility::Position
LinearMotionNode::position( const Time& t )
{
   Time dt = t - m_startTime;
   double ddt = dt.tv.tv_sec + 0.000001*dt.tv.tv_usec;
   return (m_startPos + ddt*m_velocity);
}
```

11.2.2 Hybrid Patterns

Perhaps you want a basically parametrized mobility pattern, but you want some control via waypoints. For example, you might use a waypoint to set the initial position of a node, or you might want to use waypoints to schedule movement, but follow other than constant-velocity paths. For these, you would define a class with a *hybrid pattern*.

There is essentially no difference between defining a hybrid pattern and defining a corresponding parametrized pattern. All that substantially changes is that the waypoints are included in the position computation.

A simple example of this is given in the module **`pydtn.mobility.accelnode`**, which defines `AcceleratedMobileNode`. This class behaves identically to `Mobility::MobileNode`, except that the interpolation between waypoints is not at constant velocity. Rather, the node undergoes positive acceleration for half of its movement and negative acceleration for the other half. This involves copying the code for `Mobility::MobileNode::position(t)`, and changing the end of the method from

```
mobility/MobileNode.cc

   Presence current = before->towards( *after, t );
   return current.position();
```

to

```
accelnode/AcceleratedMobileNode.cc

   Time t0 = before->time() + before->duration();
   Time totalT = after->time() - t0;
   Time currT = t - t0;

   double r = ( 1.0*currT.tv.tv_sec*Time::MILLION + currT.tv.tv_usec ) /
      ( 1.0*totalT.tv.tv_sec*Time::MILLION + totalT.tv.tv_usec );
   double a = ( r < 0.5 ) ? (2*r*r) : (-1 + 4*r - 2*r*r);
   return before->position() * ( 1.0 - a ) + after->position() * a;
```

The python bootstrapping assigns its own function to `pydtn.mobility._default_node`. A similar copying could be done for `Mobility::RandomWaypointNode`, which is unable to take advantage of this particular class.