The *ns* Manual
(formerly *ns* Notes and Documentation)[1]

The VINT Project

A Collaboration between researchers at
UC Berkeley, LBL, USC/ISI, and Xerox PARC.

Kevin Fall ⟨kfall@ee.lbl.gov⟩, Editor
Kannan Varadhan ⟨kannan@catarina.usc.edu⟩, Editor

December 17, 2014

ns © is LBNL's Network Simulator [**?**]. The simulator is written in C++; it uses OTcl as a command and configuration interface. *ns* v2 has three substantial changes from *ns* v1: (1) the more complex objects in *ns* v1 have been decomposed into simpler components for greater flexibility and composability; (2) the configuration interface is now OTcl, an object oriented version of Tcl; and (3) the interface code to the OTcl interpreter is separate from the main simulator.

Ns documentation is available in html, Postscript, and PDF formats. See `http://www.isi.edu/nsnam/ns/ns-documentation.html` for pointers to these.

[1]The VINT project is a joint effort by people from UC Berkeley, USC/ISI, LBL, and Xerox PARC. The project is supported by the Defense Advanced Research Projects Agency (DARPA) at LBL under DARPA grant DABT63-96-C-0105, at USC/ISI under DARPA grant ABT63-96-C-0054, at Xerox PARC under DARPA grant DABT63-96-C-0105. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the DARPA.

Contents

Chapter 1

Introduction

> Let's start at the very beginning,
> a very nice place to start,
> when you sing, you begin with A, B, C,
> when you simulate, you begin with the topology,[1]
> ...

This document (*ns Notes and Documentation*) provides reference documentation for ns. Although we begin with a simple simulation script, resources like Marc Greis's tutorial web pages (originally at his web site, now at `http://www.isi.edu/nsnam/ns/tutorial/`) or the slides from one of the ns tutorials are problably better places to begin for the ns novice.

We first begin by showing a simple simulation script. This script is also available in the sources in ~*ns*/tcl/ex/simple.tcl.

This script defines a simple topology of four nodes, and two agents, a UDP agent with a CBR traffic generator, and a TCP agent. The simulation runs for $3s$. The output is two trace files, `out.tr` and `out.nam`. When the simulation completes at the end of $3s$, it will attempt to run a nam visualisation of the simulation on your screen.

```
# The preamble
set ns [new Simulator]                          ;# initialise the simulation

# Predefine tracing
set f [open out.tr w]
$ns trace-all $f
set nf [open out.nam w]
$ns namtrace-all $nf
```

[1] with apologies to Rodgers and Hammerstein

```
# so, we lied.  now, we define the topology
#
#       n0
#          \
#      5Mb \
#      2ms  \
#            \
#             n2 --------- n3
#            /     1.5Mb
#      5Mb  /       10ms
#      2ms /
#         /
#       n1
#
set n0 [$ns node]
set n1 [$ns node]
set n2 [$ns node]
set n3 [$ns node]

$ns duplex-link $n0 $n2 5Mb 2ms DropTail
$ns duplex-link $n1 $n2 5Mb 2ms DropTail
$ns duplex-link $n2 $n3 1.5Mb 10ms DropTail

# Some agents.
set udp0 [new Agent/UDP]                                    ;# A UDP agent
$ns attach-agent $n0 $udp0                                  ;# on node $n0
set cbr0 [new Application/Traffic/CBR]                      ;# A CBR traffic generator agent
$cbr0 attach-agent $udp0                                    ;# attached to the UDP agent
$udp0 set class_ 0                                          ;# actually, the default, but...

set null0 [new Agent/Null]                                  ;# Its sink
$ns attach-agent $n3 $null0                                 ;# on node $n3

$ns connect $udp0 $null0
$ns at 1.0 "$cbr0 start"

puts [$cbr0 set packetSize_]
puts [$cbr0 set interval_]

# A FTP over TCP/Tahoe from $n1 to $n3, flowid 2
set tcp [new Agent/TCP]
$tcp set class_ 1
$ns attach-agent $n1 $tcp

set sink [new Agent/TCPSink]
$ns attach-agent $n3 $sink

set ftp [new Application/FTP]                               ;# TCP does not generate its own traffic
$ftp attach-agent $tcp
$ns at 1.2 "$ftp start"

$ns connect $tcp $sink
$ns at 1.35 "$ns detach-agent $n0 $tcp ; $ns detach-agent $n3 $sink"
```

```
# The simulation runs for 3s.
# The simulation comes to an end when the scheduler invokes the finish{} procedure below.
# This procedure closes all trace files, and invokes nam visualization on one of the trace files.

$ns at 3.0 "finish"
proc finish {} {
        global ns f nf
        $ns flush-trace
        close $f
        close $nf

        puts "running nam..."
        exec nam out.nam &
        exit 0
}

# Finally, start the simulation.
$ns run
```

Chapter 2

Undocumented Facilities

Ns is often growing to include new protocols. Unfortunately the documention doesn't grow quite as often. This section lists what remains to be documented, or what needs to be improved.

(The documentation is in the doc subdirectory of the ns source code if you want to add to it. :-)

Interface to the Interpreter
- nothing currently

Simulator Basics
- LANs need to be updated for new wired/wireless support (Yuri updated this?)
- wireless support needs to be added (done)
- should explicitly list queueing options in the queue mgt chapter?

Support
- should pick a single list mgt package and document it
- should document the trace-post-processing utilities in bin

Routing
- The usage and design of link state and MPLS routing modules are not documented at all. (Note: link state and MPLS appeared only in daily snapshots and releases after 09/14/2000.)
- need to document hierarchical routing/addressing (Padma has done)
- need a chapter on supported ad-hoc routing protocols

Queueing
- CBQ needs documentation (can maybe build off of `ftp://ftp.ee.lbl.gov/papers/cbqsims.ps.Z`?)

Transport
- need to document MFTP
- need to document RTP (session-rtp.cc, etc.)
- need to document multicast building blocks
- should repair and document snoop and tcp-int

Traffic and scenarios (new section)
- should add a description of how to drive the simulator from traces
- should add discussion of the scenario generator
- should add discussion of http traffic sources

Application
- is the non-Haobo http stuff documented? no.

Scale
- should add disucssion of mixed mode (pending)

Emulation
- nothing currently

Other
- should document admission control policies?
- should add a validation chapter and snarf up the contents of ns-tests.html
- should snarf up Marc Greis' tutorial rather than just referring to it?

Part I

Interface to the Interpreter

Chapter 3

OTcl Linkage

ns is an object oriented simulator, written in C++, with an OTcl interpreter as a frontend. The simulator supports a class hierarchy in C++ (also called the compiled hierarchy in this document), and a similar class hierarchy within the OTcl interpreter (also called the interpreted hierarchy in this document). The two hierarchies are closely related to each other; from the user's perspective, there is a one-to-one correspondence between a class in the interpreted hierarchy and one in the compiled hierarchy. The root of this hierarchy is the class TclObject. Users create new simulator objects through the interpreter; these objects are instantiated within the interpreter, and are closely mirrored by a corresponding object in the compiled hierarchy. The interpreted class hierarchy is automatically established through methods defined in the class TclClass. user instantiated objects are mirrored through methods defined in the class TclObject. There are other hierarchies in the C++ code and OTcl scripts; these other hierarchies are not mirrored in the manner of TclObject.

3.1 Concept Overview

Why two languages? ns uses two languages because simulator has two different kinds of things it needs to do. On one hand, detailed simulations of protocols requires a systems programming language which can efficiently manipulate bytes, packet headers, and implement algorithms that run over large data sets. For these tasks run-time speed is important and turn-around time (run simulation, find bug, fix bug, recompile, re-run) is less important.

On the other hand, a large part of network research involves slightly varying parameters or configurations, or quickly exploring a number of scenarios. In these cases, iteration time (change the model and re-run) is more important. Since configuration runs once (at the beginning of the simulation), run-time of this part of the task is less important.

ns meets both of these needs with two languages, C++ and OTcl. C++ is fast to run but slower to change, making it suitable for detailed protocol implementation. OTcl runs much slower but can be changed very quickly (and interactively), making it ideal for simulation configuration. *ns* (via `tclcl`) provides glue to make objects and variables appear on both langauges.

For more information about the idea of scripting languages and split-language programming, see Ousterhout's article in IEEE Computer [**?**]. For more information about split level programming for network simulation, see the ns paper [**?**].

Which language for what? Having two languages raises the question of which language should be used for what purpose.

Our basic advice is to use OTcl:

- for configuration, setup, and "one-time" stuff

- if you can do what you want by manipulating existing C++ objects

and use C++:

- if you are doing *anything* that requires processing each packet of a flow
- if you have to change the behavior of an existing C++ class in ways that weren't anticipated

For example, links are OTcl objects that assemble delay, queueing, and possibly loss modules. If your experiment can be done with those pieces, great. If instead you want do something fancier (a special queueing dicipline or model of loss), then you'll need a new C++ object.

There are certainly grey areas in this spectrum: most routing is done in OTcl (although the core Dijkstra algorithm is in C++). We've had HTTP simulations where each flow was started in OTcl and per-packet processing was all in C++. This approache worked OK until we had 100s of flows starting per second of simulated time. In general, if you're ever having to invoke Tcl many times per second, you problably should move that code to C++.

3.2 Code Overview

In this document, we use the term "interpreter" to be synonymous with the OTcl interpreter. The code to interface with the interpreter resides in a separate directory, `tclcl`. The rest of the simulator code resides in the directory, `ns-2`. We will use the notation ~*tclcl*/⟨file⟩ to refer to a particular ⟨file⟩ in the `Tcl` directory. Similarly, we will use the notation, ~*ns*/⟨file⟩ to refer to a particular ⟨file⟩ in the `ns-2` directory.

There are a number of classes defined in ~*tclcl*/. We only focus on the six that are used in *ns*: The Class Tcl (Section **??**) contains the methods that C++ code will use to access the interpreter. The class TclObject (Section **??**) is the base class for all simulator objects that are also mirrored in the compiled hierarchy. The class TclClass (Section **??**) defines the interpreted class hierarchy, and the methods to permit the user to instantiate TclObjects. The class TclCommand (Section **??**) is used to define simple global interpreter commands. The class EmbeddedTcl (Section **??**) contains the methods to load higher level builtin commands that make configuring simulations easier. Finally, the class InstVar (Section **??**) contains methods to access C++ member variables as OTcl instance variables.

The procedures and functions described in this chapter can be found in ~*tclcl*/Tcl.{cc, h}, ~*tclcl*/Tcl2.cc, ~*tclcl*/tcl-object.tcl, and, ~*tclcl*/tracedvar.{cc, h}. The file ~*tclcl*/tcl2c++.c is used in building *ns*, and is mentioned briefly in this chapter.

3.3 Class Tcl

The `class Tcl` encapsulates the actual instance of the OTcl interpreter, and provides the methods to access and communicate with that interpreter. The methods described in this section are relevant to the *ns* programmer who is writing C++ code. The class provides methods for the following operations:

- obtain a reference to the Tcl instance;
- invoke OTcl procedures through the interpreter;
- retrieve, or pass back results to the interpreter;
- report error situations and exit in an uniform manner; and

- store and lookup "TclObjects".
- acquire direct access to the interpreter.

We describe each of the methods in the following subsections.

3.3.1 Obtain a Reference to the class Tcl instance

A single instance of the class is declared in ~*tclcl*/Tcl.cc as a static member variable; the programmer must obtain a reference to this instance to access other methods described in this section. The statement required to access this instance is:

```
Tcl& tcl = Tcl::instance();
```

3.3.2 Invoking OTcl Procedures

There are four different methods to invoke an OTcl command through the instance, `tcl`. They differ essentially in their calling arguments. Each function passes a string to the interpreter, that then evaluates the string in a global context. These methods will return to the caller if the interpreter returns TCL_OK. On the other hand, if the interpreter returns TCL_ERROR, the methods will call `tkerror{}`. The user can overload this procedure to selectively disregard certain types of errors. Such intricacies of OTcl programming are outside the scope of this document. The next section (Section **??**) describes methods to access the result returned by the interpreter.

- `tcl.eval`(char* s) invokes `Tcl_GlobalEval`() to execute s through the interpreter.
- `tcl.evalc`(const char* s) preserves the argument string s. It copies the string s into its internal buffer; it then invokes the previous `eval`(char* s) on the internal buffer.
- `tcl.eval`() assumes that the command is already stored in the class' internal `bp_`; it directly invokes `tcl.eval`(char* bp_). A handle to the buffer itself is available through the method `tcl.buffer`(void).
- `tcl.evalf`(const char* s, ...) is a `Printf`(3) like equivalent. It uses `vsprintf`(3) internally to create the input string.

As an example, here are some of the ways of using the above methods:

```
Tcl& tcl = Tcl::instance();
char wrk[128];
strcpy(wrk, "Simulator set NumberInterfaces_ 1");
tcl.eval(wrk);

sprintf(tcl.buffer(), "Agent/SRM set requestFunction_ %s", "Fixed");
tcl.eval();

tcl.evalc("puts stdout hello world");
tcl.evalf("%s request %d %d", name_, sender, msgid);
```

3.3.3 Passing Results to/from the Interpreter

When the interpreter invokes a C++ method, it expects the result back in the private member variable, `tcl_->result`. Two methods are available to set this variable.

- `tcl.result`(const char* s)

  Pass the result string s back to the interpreter.

- `tcl.resultf`(const char* fmt, ...)

  `varargs`(3) variant of above to format the result using `vsprintf`(3), pass the result string back to the interpreter.

```
if (strcmp(argv[1], "now") == 0) {
        tcl.resultf("%.17g", clock());
        return TCL_OK;
}
tcl.result("Invalid operation specified");
return TCL_ERROR;
```

Likewise, when a C++ method invokes an OTcl command, the interpreter returns the result in `tcl_->result`.

- `tcl.result`(void) must be used to retrieve the result. Note that the result is a string, that must be converted into an internal format appropriate to the type of result.

```
tcl.evalc("Simulator set NumberInterfaces_");
char* ni = tcl.result();
if (atoi(ni) != 1)
        tcl.evalc("Simulator set NumberInterfaces_ 1");
```

3.3.4 Error Reporting and Exit

This method provides a uniform way to report errors in the compiled code.

- `tcl.error`(const char* s) performs the following functions: write s to stdout; write `tcl_->result` to stdout; exit with error code 1.

```
tcl.resultf("cmd = %s", cmd);
tcl.error("invalid command specified");
/*NOTREACHED*/
```

Note that there are minor differences between returning TCL_ERROR as we did in the previous subsection (Section **??**), and calling `Tcl::error`(). The former generates an exception within the interpreter; the user can trap the exception and possibly recover from the error. If the user has not specified any traps, the interpreter will print a stack trace and exit. However, if the code invokes `error`(), then the simulation user cannot trap the error; in addition, *ns* will not print any stack trace.

3.3.5 Hash Functions within the Interpreter

ns stores a reference to every TclObject in the compiled hierarchy in a hash table; this permits quick access to the objects. The hash table is internal to the interpreter. *ns* uses the name of the `TclObject` as the key to enter, lookup, or delete the TclObject in the hash table.

- `tcl.enter`(TclObject* o) will insert a pointer to the TclObject o into the hash table.
  It is used by `TclClass::create_shadow`() to insert an object into the table, when that object is created.
- `tcl.lookup`(char* s) will retrieve the TclObject with the name s.
  It is used by `TclObject::lookup`().
- `tcl.remove`(TclObject* o) will delete references to the TclObject o from the hash table.
  It is used by `TclClass::delete_shadow`() to remove an existing entry from the hash table, when that object is deleted.

These functions are used internally by the class TclObject and class TclClass.

3.3.6 Other Operations on the Interpreter

If the above methods are not sufficient, then we must acquire the handle to the interpreter, and write our own functions.

- `tcl.interp`(void) returns the handle to the interpreter that is stored within the class Tcl.

3.4 Class TclObject

`class TclObject` is the base class for most of the other classes in the interpreted and compiled hierarchies. Every object in the class TclObject is created by the user from within the interpreter. An equivalent shadow object is created in the compiled hierarchy. The two objects are closely associated with each other. The class TclClass, described in the next section, contains the mechanisms that perform this shadowing.

In the rest of this document, we often refer to an object as a TclObject[1]. By this, we refer to a particular object that is either in the class TclObject, or in a class that is derived from the class TclObject. If it is necessary, we will explicitly qualify whether that object is an object within the interpreter, or an object within the compiled code. In such cases, we will use the abbreviations "interpreted object", and "compiled object" to distinguish the two. and within the compiled code respectively.

Differences from *ns* v1 Unlike *ns* v1, the class TclObject subsumes the earlier functions of the NsObject class. It therefore stores the interface variable bindings (Section **??**) that tie OTcl instance variables in the interpreted object to corresponding C++ member variables in the compiled object. The binding is stronger than in *ns* v1 in that any changes to the OTcl variables are trapped, and the current C++ and OTcl values are made consistent after each access through the interpreter. The consistency is done through the class InstVar (Section **??**). Also unlike *ns* v1, objects in the class TclObject are no longer stored as a global link list. Instead, they are stored in a hash table in the class Tcl (Section **??**).

Example configuration of a TclObject The following example illustrates the configuration of an SRM agent (`class Agent/SRM/Adaptive`).

```
set srm [new Agent/SRM/Adaptive]
$srm set packetSize_ 1024
$srm traffic-source $s0
```

[1] In the latest release of *ns* and *ns/tclcl*, this object has been renamed to `SplitObjefct`, which more accurately reflects its nature of existence. However, for the moment, we will continue to use the term TclObject to refer to these objects and this class.

By convention in *ns*, the class Agent/SRM/Adaptive is a subclass of Agent/SRM, is a subclass of Agent, is a subclass of TclObject. The corresponding compiled class hierarchy is the ASRMAgent, derived from SRMAgent, derived from Agent, derived from TclObject respectively. The first line of the above example shows how a TclObject is created (or destroyed) (Section **??**); the next line configures a bound variable (Section **??**); and finally, the last line illustrates the interpreted object invoking a C++ method as if they were an instance procedure (Section **??**).

3.4.1 Creating and Destroying TclObjects

When the user creates a new TclObject, using the procedures `new{}` and `delete{}`; these procedures are defined in ~*tclcl*/tcl-object.tcl. They can be used to create and destroy objects in all classes, including TclObjects.[2]. In this section, we describe the internal actions executed when a TclObject is created.

Creating TclObjects By using `new{}`, the user creates an interpreted TclObject. the interpreter will execute the constructor for that object, `init{}`, passing it any arguments provided by the user. *ns* is responsible for automatically creating the compiled object. The shadow object gets created by the base class TclObject's constructor. Therefore, the constructor for the new TclObject must call the parent class constructor first. `new{}` returns a handle to the object, that can then be used for further operations upon that object.

The following example illustrates the Agent/SRM/Adaptive constructor:

```
Agent/SRM/Adaptive instproc init args {
        eval $self next $args
        $self array set closest_ "requestor 0 repairor 0"
        $self set eps_    [$class set eps_]
}
```

The following sequence of actions are performed by the interpreter as part of instantiating a new TclObject. For ease of exposition, we describe the steps that are executed to create an Agent/SRM/Adaptive object. The steps are:

1. Obtain an unique handle for the new object from the TclObject name space. The handle is returned to the user. Most handles in *ns* have the form `_o`⟨NNN⟩, where ⟨NNN⟩ is an integer. This handle is created by `getid{}`. It can be retrieved from C++ with the `name(){}` method.
2. Execute the constructor for the new object. Any user-specified arguments are passed as arguments to the constructor. This constructor must invoke the constructor associated with its parent class.

   In our example above, the Agent/SRM/Adaptive calls its parent class in the very first line.

   Note that each constructor, in turn invokes its parent class' constructor *ad nauseum*. The last constructor in *ns* is the TclObject constructor. This constructor is responsible for setting up the shadow object, and performing other initializations and bindings, as we explain below. *It is preferable to call the parent constructors first before performing the initializations required in this class.* This allows the shadow objects to be set up, and the variable bindings established.
3. The TclObject constructor invokes the instance procedure `create-shadow{}` for the class Agent/SRM/Adaptive.
4. When the shadow object is created, *ns* calls all of the constructors for the compiled object, each of which may establish variable bindings for objects in that class, and perform other necessary initializations. Hence our earlier injunction that it is preferable to invoke the parent constructors prior to performing the class initializations.
5. After the shadow object is successfully created, `create_shadow`(void)

[2] As an example, the classes Simulator, Node, Link, or rtObject, are classes that are *not* derived from the class TclObject. Objects in these classes are not, therefore, TclObjects. However, a Simulator, Node, Link, or route Object is also instantiated using the `new` procedure in *ns*.

(a) adds the new object to hash table of TclObjects described earlier (Section **??**).

(b) makes `cmd{}` an instance procedure of the newly created interpreted object. This instance procedure invokes the `command()` method of the compiled object. In a later subsection (Section **??**), we describe how the `command` method is defined, and invoked.

Note that all of the above shadowing mechanisms only work when the user creates a new TclObject through the interpreter. It will not work if the programmer creates a compiled TclObject unilaterally. Therefore, the programmer is enjoined not to use the C++ new method to create compiled objects directly.

Deletion of TclObjects The `delete` operation destroys the interpreted object, and the corresponding shadow object. For example, `use-scheduler{`⟨scheduler⟩`}` uses the `delete` procedure to remove the default list scheduler, and instantiate an alternate scheduler in its place.

```
Simulator instproc use-scheduler type {
        $self instvar scheduler_

        delete scheduler_                          ;# first delete the existing list scheduler
        set scheduler_ [new Scheduler/$type]
}
```

As with the constructor, the object destructor must call the destructor for the parent class explicitly as the very last statement of the destructor. The TclObject destructor will invoke the instance procedure `delete-shadow`, that in turn invokes the equivalent compiled method to destroy the shadow object. The interpreter itself will destroy the interpreted object.

3.4.2 Variable Bindings

In most cases, access to compiled member variables is restricted to compiled code, and access to interpreted member variables is likewise confined to access via interpreted code; however, it is possible to establish bi-directional bindings such that both the interpreted member variable and the compiled member variable access the same data, and changing the value of either variable changes the value of the corresponding paired variable to same value.

The binding is established by the compiled constructor when that object is instantiated; it is automatically accessible by the interpreted object as an instance variable. *ns* supports five different data types: reals, bandwidth valued variables, time valued variables, integers, and booleans. The syntax of how these values can be specified in OTcl is different for each variable type.

- Real and Integer valued variables are specified in the "normal" form. For example,

```
$object set realvar 1.2e3
$object set intvar  12
```

- Bandwidth is specified as a real value, optionally suffixed by a 'k' or 'K' to mean kilo-quantities, or 'm' or 'M' to mean mega-quantities. A final optional suffix of 'B' indicates that the quantity expressed is in Bytes per second. The default is bandwidth expressed in bits per second. For example, all of the following are equivalent:

```
$object set bwvar 1.5m
$object set bwvar 1.5mb
$object set bwvar 1500k
```

```
$object set bwvar 1500kb
$object set bwvar .1875MB
$object set bwvar 187.5kB
$object set bwvar 1.5e6
```

- Time is specified as a real value, optionally suffixed by a 'm' to express time in milli-seconds, 'n' to express time in nano-seconds, or 'p' to express time in pico-seconds. The default is time expressed in seconds. For example, all of the following are equivalent:

```
$object set timevar 1500m
$object set timevar 1.5
$object set timevar 1.5e9n
$object set timevar 1500e9p
```

  Note that we can also safely add a s to reflect the time unit of seconds. *ns* will ignore anything other than a valid real number specification, or a trailing 'm', 'n', or 'p'.

- Booleans can be expressed either as an integer, or as 'T' or 't' for true. Subsequent characters after the first letter are ignored. If the value is neither an integer, nor a true value, then it is assumed to be false. For example,

```
$object set boolvar t                                   ;# set to true
$object set boolvar true
$object set boolvar 1                                ;# or any non-zero value

$object set boolvar false                               ;# set to false
$object set boolvar junk
$object set boolvar 0
```

The following example shows the constructor for the ASRMAgent[3].

```
ASRMAgent::ASRMAgent() {
        bind("pdistance_", &pdistance_);                    /* real variable */
        bind("requestor_", &requestor_);                  /* integer variable */
        bind_time("lastSent_", &lastSessSent_);             /* time variable */
        bind_bw("ctrlLimit_", &ctrlBWLimit_);            /* bandwidth variable */
        bind_bool("running_", &running_);                  /* boolean variable */
}
```

Note that all of the functions above take two arguments, the name of an OTcl variable, and the address of the corresponding compiled member variable that is linked. While it is often the case that these bindings are established by the constructor of the object, it need not always be done in this manner. We will discuss such alternate methods when we describe the class InstVar (Section **??**) in detail later.

Each of the variables that is bound is automatically initialised with default values when the object is created. The default values are specified as interpreted class variables. This initialisation is done by the routing `init-instvar{}`, invoked by methods in the class Instvar, described later (Section **??**). `init-instvar{}` checks the class of the interpreted object, and all of the parent class of that object, to find the first class in which the variable is defined. It uses the value of the variable in that class to initialise the object. Most of the bind initialisation values are defined in ~*ns*/tcl/lib/ns-default.tcl.

For example, if the following class variables are defined for the ASRMAgent:

[3]Note that this constructor is embellished to illustrate the features of the variable binding mechanism.

```
Agent/SRM/Adaptive set pdistance_ 15.0
Agent/SRM set pdistance_ 10.0
Agent/SRM set lastSent_ 8.345m
Agent set ctrlLimit_    1.44M
Agent/SRM/Adaptive set running_ f
```

Therefore, every new Agent/SRM/Adaptive object will have `pdistance_` set to 15.0; `lastSent_` is set to 8.345m from the setting of the class variable of the parent class; `ctrlLimit_` is set to 1.44M using the class variable of the parent class twice removed; `running` is set to false; the instance variable `pdistance_` is not initialised, because no class variable exists in any of the class hierarchy of the interpreted object. In such instance, `init-instvar{}` will invoke `warn-instvar{}`, to print out a warning about such a variable. The user can selectively override this procedure in their simulation scripts, to elide this warning.

Note that the actual binding is done by instantiating objects in the class InstVar. Each object in the class InstVar binds one compiled member variable to one interpreted member variable. A TclObject stores a list of InstVar objects corresponding to each of its member variable that is bound in this fashion. The head of this list is stored in its member variable `instvar_` of the TclObject.

One last point to consider is that *ns* will guarantee that the actual values of the variable, both in the interpreted object and the compiled object, will be identical at all times. However, if there are methods and other variables of the compiled object that track the value of this variable, they must be explicitly invoked or changed whenever the value of this variable is changed. This usually requires additional primitives that the user should invoke. One way of providing such primitives in *ns* is through the `command`() method described in the next section.

3.4.3 Variable Tracing

In addition to variable bindings, TclObject also supports tracing of both C++ and Tcl instance variables. A traced variable can be created and configured either in C++ or Tcl. To establish variable tracing at the Tcl level, the variable must be visible in Tcl, which means that it must be a bounded C++/Tcl or a pure Tcl instance variable. In addition, the object that owns the traced variable is also required to establish tracing using the Tcl `trace` method of TclObject. The first argument to the `trace` method must be the name of the variable. The optional second argument specifies the trace object that is responsible for tracing that variable. If the trace object is not specified, the object that own the variable is responsible for tracing it.

For a TclObject to trace variables, it must extend the C++ `trace` method that is virtually defined in TclObject. The Trace class implements a simple `trace` method, thereby, it can act as a generic tracer for variables.

```
class Trace : public Connector {
        ...
        virtual void trace(TracedVar*);
};
```

Below is a simple example for setting up variable tracing in Tcl:

```
# $tcp tracing its own variable cwnd_
$tcp trace cwnd_

# the variable ssthresh_ of $tcp is traced by a generic $tracer
set tracer [new Trace/Var]
$tcp trace ssthresh_ $tracer
```

For a C++ variable to be traceable, it must belong to a class that derives from TracedVar. The virtual base class TracedVar keeps track of the variable's name, owner, and tracer. Classes that derives from TracedVar must implement the virtual method `value`, that takes a character buffer as an argument and writes the value of the variable into that buffer.

```
class TracedVar {
        ...
        virtual char* value(char* buf) = 0;
protected:
        TracedVar(const char* name);
        const char* name_;      // name of the variable
        TclObject* owner_;      // the object that owns this variable
        TclObject* tracer_;     // callback when the variable is changed
        ...
};
```

The TclCL library exports two classes of TracedVar: `TracedInt` and `TracedDouble`. These classes can be used in place of the basic type int and double respectively. Both TracedInt and TracedDouble overload all the operators that can change the value of the variable such as assignment, increment, and decrement. These overloaded operators use the `assign` method to assign the new value to the variable and call the tracer if the new value is different from the old one. TracedInt and TracedDouble also implement their `value` methods that output the value of the variable into string. The width and precision of the output can be pre-specified.

3.4.4 `command` Methods: Definition and Invocation

For every TclObject that is created, *ns* establishes the instance procedure, `cmd{}`, as a hook to executing methods through the compiled shadow object. The procedure `cmd{}` invokes the method `command`() of the shadow object automatically, passing the arguments to `cmd{}` as an argument vector to the `command`() method.

The user can invoke the `cmd{}` method in one of two ways: by explicitly invoking the procedure, specifying the desired operation as the first argument, or implicitly, as if there were an instance procedure of the same name as the desired operation. Most simulation scripts will use the latter form, hence, we will describe that mode of invocation first.

Consider the that the distance computation in SRM is done by the compiled object; however, it is often used by the interpreted object. It is usually invoked as:

```
$srmObject distance? ⟨agentAddress⟩
```

If there is no instance procedure called `distance?`, the interpreter will invoke the instance procedure `unknown{}`, defined in the base class TclObject. The unknown procedure then invokes

```
$srmObject cmd distance? ⟨agentAddress⟩
```

to execute the operation through the compiled object's `command`() procedure.

Ofcourse, the user could explicitly invoke the operation directly. One reason for this might be to overload the operation by using an instance procedure of the same name. For example,

```
Agent/SRM/Adaptive instproc distance? addr {
```

```
        $self instvar distanceCache_
        if ![info exists distanceCache_($addr)] {
                set distanceCache_($addr) [$self cmd distance? $addr]
        }
        set distanceCache_($addr)
}
```

We now illustrate how the `command()` method using `ASRMAgent::command()` as an example.

```
int ASRMAgent::command(int argc, const char*const*argv) {
        Tcl& tcl = Tcl::instance();
        if (argc == 3) {
                if (strcmp(argv[1], "distance?") == 0) {
                        int sender = atoi(argv[2]);
                        SRMinfo* sp = get_state(sender);
                        tcl.tesultf("%f", sp->distance_);
                        return TCL_OK;
                }
        }
        return (SRMAgent::command(argc, argv));
}
```

We can make the following observations from this piece of code:

- The function is called with two arguments:

  The first argument (`argc`) indicates the number of arguments specified in the command line to the interpreter.

  The command line arguments vector (`argv`) consists of

  — `argv[0]` contains the name of the method, "`cmd`".

  — `argv[1]` specifies the desired operation.

  — If the user specified any arguments, then they are placed in `argv[2...(argc - 1)]`.

  The arguments are passed as strings; they must be converted to the appropriate data type.

- If the operation is successfully matched, the match should return the result of the operation using methods described earlier (Section **??**).

- `command()` itself must return either `TCL_OK` or `TCL_ERROR` to indicate success or failure as its return code.

- If the operation is not matched in this method, it must invoke its parent's command method, and return the corresponding result.

  This permits the user to concieve of operations as having the same inheritance properties as instance procedures or compiled methods.

  In the event that this `command` method is defined for a class with multiple inheritance, the programmer has the liberty to choose one of two implementations:

  1) Either they can invoke one of the parent's `command` method, and return the result of that invocation, or

  2) They can each of the parent's `command` methods in some sequence, and return the result of the first invocation that is successful. If none of them are successful, then they should return an error.

In our document, we call operations executed through the `command()` *instproc-like*s. This reflects the usage of these operations as if they were OTcl instance procedures of an object, but can be very subtly different in their realisation and usage.

3.5 Class TclClass

This compiled class (`class TclClass`) is a pure virtual class. Classes derived from this base class provide two functions: construct the interpreted class hierarchy to mirror the compiled class hierarchy; and provide methods to instantiate new TclObjects. Each such derived class is associated with a particular compiled class in the compiled class hierarchy, and can instantiate new objects in the associated class.

As an example, consider a class such as the class `RenoTcpClass`. It is derived from class `TclClass`, and is associated with the class `RenoTcpAgent`. It will instantiate new objects in the class `RenoTcpAgent`. The compiled class hierarchy for `RenoTcpAgent` is that it derives from `TcpAgent`, that in turn derives from `Agent`, that in turn derives (roughly) from `TclObject`. `RenoTcpClass` is defined as

```
static class RenoTcpClass: public TclClass {
public:
        RenoTcpClass() : TclClass("Agent/TCP/Reno") {}
        TclObject* create(int argc, const char*const* argv) {
                return (new RenoTcpAgent());
        }
} class_reno;
```

We can make the following observations from this definition:

1. The class defines only the constructor, and one additional method, to `create` instances of the associated TclObject.
2. *ns* will execute the `RenoTcpClass` constructor for the static variable `class_reno`, when it is first started. This sets up the appropriate methods and the interpreted class hierarchy.
3. The constructor specifies the interpreted class explicitly as `Agent/TCP/Reno`. This also specifies the interpreted class hierarchy implicitly.

   Recall that the convention in *ns* is to use the character slash ('/') is a separator. For any given class `A/B/C/D`, the class `A/B/C/D` is a sub-class of `A/B/C`, that is itself a sub-class of `A/B`, that, in turn, is a sub-class of `A`. `A` itself is a sub-class of `TclObject`.

   In our case above, the TclClass constructor creates three classes, `Agent/TCP/Reno` sub-class of `Agent/TCP` sub-class of `Agent` sub-class of `TclObject`.
4. This class is associated with the class `RenoTcpAgent`; it creats new objects in this associated class.
5. The `RenoTcpClass::create` method returns TclObjects in the class `RenoTcpAgent`.
6. When the user specifies `new Agent/TCP/Reno`, the routine `RenoTcpClass::create` is invoked.
7. The arguments vector (`argv`) consists of

   — `argv[0]` contains the name of the object.

   — `argv[1...3]` contain `$self`, `$class`, and `$proc`.Since `create` is called through the instance procedure `create-shadow`, `argv[3]` contains `create-shadow`.

   — `argv[4]` contain any additional arguments (passed as a string) provided by the user.

The `class Trace` illustrates argument handling by TclClass methods.

```
class TraceClass : public TclClass {
public:
```

```
        TraceClass() : TclClass("Trace") {}
        TclObject* create(int args, const char*const* argv) {
                if (args >= 5)
                        return (new Trace(*argv[4]));
                else
                        return NULL;
        }
} trace_class;
```

A new Trace object is created as

```
new Trace "X"
```

Finally, the nitty-gritty details of how the interpreted class hierarchy is constructed:

1. The object constructor is executed when *ns* first starts.
2. This constructor calls the TclClass constructor with the name of the interpreted class as its argument.
3. The TclClass constructor stores the name of the class, and inserts this object into a linked list of the TclClass objects.
4. During initialization of the simulator, `Tcl_AppInit`(void) invokes `TclClass::bind`(void)
5. For each object in the list of TclClass objects, `bind`() invokes `register`{}, specifying the name of the interpreted class as its argument.
6. `register`{} establishes the class hierarchy, creating the classes that are required, and not yet created.
7. Finally, `bind`() defines instance procedures `create-shadow` and `delete-shadow` for this new class.

3.5.1 How to Bind Static C++ Class Member Variables

In Section **??**, we have seen how to expose member variables of a C++ object into OTcl space. This, however, does not apply to static member variables of a C++ class. Of course, one may create an OTcl variable for the static member variable of every C++ object; obviously this defeats the whole meaning of static members.

We cannot solve this binding problem using a similar solution as binding in TclObject, which is based on InstVar, because InstVars in TclCL require the presence of a TclObject. However, we can create a method of the corresponding TclClass and access static members of a C++ class through the methods of its corresponding TclClass. The procedure is as follows:

1. Create your own derived TclClass as described above;
2. Declare methods `bind`() and `method`() in your derived class;
3. Create your binding methods in the implementation of your `bind`() with `add_method("your_method")`, then implement the handler in `method`() in a similar way as you would do in `TclObject::command`(). Notice that the number of arguments passed to `TclClass::method`() are different from those passed to `TclObject::command`(). The former has two more arguments in the front.

As an example, we show a simplified version of `PacketHeaderClass` in ~*ns*/packet.cc. Suppose we have the following class `Packet` which has a static variable `hdrlen_` that we want to access from OTcl:

```
class Packet {
        ......
        static int hdrlen_;
};
```

Then we do the following to construct an accessor for this variable:

```
class PacketHeaderClass : public TclClass {
protected:
        PacketHeaderClass(const char* classname, int hdrsize);
        TclObject* create(int argc, const char*const* argv);
                                                /* These two implements OTcl class access methods */
        virtual void bind();
        virtual int method(int argc, const char*const* argv);
};

void PacketHeaderClass::bind()
{
                                                /* Call to base class bind() must precede add_method() */
        TclClass::bind();
        add_method("hdrlen");
}

int PacketHeaderClass::method(int ac, const char*const* av)
{
        Tcl& tcl = Tcl::instance();
                  /* Notice this argument translation; we can then handle them  as if in TclObject::command() */
        int argc = ac - 2;
        const char*const* argv = av + 2;
        if (argc == 2) {
                if (strcmp(argv[1], "hdrlen") == 0) {
                        tcl.resultf("%d", Packet::hdrlen_);
                        return (TCL_OK);
                }
        } else if (argc == 3) {
                if (strcmp(argv[1], "hdrlen") == 0) {
                        Packet::hdrlen_ = atoi(argv[2]);
                        return (TCL_OK);
                }
        }
        return TclClass::method(ac, av);
}
```

After this, we can then use the following OTcl command to access and change values of `Packet::hdrlen_`:

```
        PacketHeader hdrlen 120
        set i [PacketHeader hdrlen]
```

3.6 Class TclCommand

This class (`class TclCommand`) provides just the mechanism for *ns* to export simple commands to the interpreter, that can then be executed within a global context by the interpreter. There are two functions defined in ~*ns*/misc.cc: `ns-random` and `ns-version`. These two functions are initialized by the function `init_misc`(void), defined in ~*ns*/misc.cc; `init_misc` is invoked by `Tcl_AppInit`(void) during startup.

- `class VersionCommand` defines the command `ns-version`. It takes no argument, and returns the current *ns* version string.

```
% ns-version                                   ;# get the current version
2.0a12
```

- `class RandomCommand` defines the command `ns-random`. With no argument, `ns-random` returns an integer, uniformly distributed in the interval $[0, 2^{31} - 1]$.

  When specified an argument, it takes that argument as the seed. If this seed value is 0, the command uses a heuristic seed value; otherwise, it sets the seed for the random number generator to the specified value.

```
% ns-random                                    ;# return a random number
2078917053
% ns-random 0                                  ;#set the seed heuristically
858190129
% ns-random 23786                              ;#set seed to specified value
23786
```

Note that, it is generally not advisable to construct top-level commands that are available to the user. We now describe how to define a new command using the example `class say_hello`. The example defines the command `hi`, to print the string "hello world", followed by any command line arguments specified by the user. For example,

```
% hi this is ns [ns-version]
hello world, this is ns 2.0a12
```

1. The command must be defined within a class derived from the `class TclCommand`. The class definition is:

```
class say_hello : public TclCommand {
public:
        say_hello();
        int command(int argc, const char*const* argv);
};
```

2. The constructor for the class must invoke the TclCommand constructor with the command as argument; *i.e.*,

```
say_hello() : TclCommand("hi") {}
```

   The `TclCommand` constructor sets up "hi" as a global procedure that invokes `TclCommand::dispatch_cmd`().

3. The method `command`() must perform the desired action.

   The method is passed two arguments. The first argument, `argc`, contains the number of actual arguments passed by the user.

The actual arguments passed by the user are passed as an argument vector (`argv`) and contains the following:

— `argv[0]` contains the name of the command (`hi`).

— `argv[1...(argc - 1)]` contains additional arguments specified on the command line by the user.

`command`() is invoked by `dispatch_cmd`().

```
#include <streams.h>                          /* because we are using stream I/O */

int say_hello::command(int argc, const char*const* argv) {
        cout << "hello world:";
        for (int i = 1; i < argc; i++)
                cout << ' ' << argv[i];
        cout << '\ n';
        return TCL_OK;
}
```

4. Finally, we require an instance of this class. `TclCommand` instances are created in the routine `init_misc`(void).

```
new say_hello;
```

Note that there used to be more functions such as `ns-at` and `ns-now` that were accessible in this manner. Most of these functions have been subsumed into existing classes. In particular, `ns-at` and `ns-now` are accessible through the scheduler TclObject. These functions are defined in ~*ns*/tcl/lib/ns-lib.tcl.

```
% set ns [new Simulator]                    ;# get new instance of simulator
_o1
% $ns now                                   ;# query simulator for current time
0
% $ns at ...                                ;# specify at operations for simulator
...
```

3.7 Class EmbeddedTcl

ns permits the development of functionality in either compiled code, or through interpreter code, that is evaluated at initialization. For example, the scripts ~*tclcl*/tcl-object.tcl or the scripts in ~*ns*/tcl/lib. Such loading and evaluation of scripts is done through objects in the `class EmbeddedTcl`.

The easiest way to extend *ns* is to add OTcl code to either ~*tclcl*/tcl-object.tcl or through scripts in the ~*ns*/tcl/lib directory. Note that, in the latter case, *ns* sources ~*ns*/tcl/lib/ns-lib.tcl automatically, and hence the programmer must add a couple of lines to this file so that their script will also get automatically sourced by *ns* at startup. As an example, the file ~*ns*/tcl/mcast/srm.tcl defines some of the instance procedures to run SRM. In ~*ns*/tcl/lib/ns-lib.tcl, we have the lines:

```
source tcl/mcast/srm.tcl
```

to automatically get srm.tcl sourced by *ns* at startup.

Three points to note with EmbeddedTcl code are that firstly, if the code has an error that is caught during the eval, then *ns* will not run. Secondly, the user can explicitly override any of the code in the scripts. In particular, they can re-source the entire

script after making their own changes. Finally, after adding the scripts to ~*ns*/tcl/lib/ns-lib.tcl, and every time thereafter that they change their script, the user must recompile *ns* for their changes to take effect. Of course, in most cases[4], the user can source their script to override the embedded code.

The rest of this subsection illustrate how to integrate individual scripts directly into *ns*. The first step is convert the script into an EmbeddedTcl object. The lines below expand ns-lib.tcl and create the EmbeddedTcl object instance called `et_ns_lib`:

```
tclsh bin/tcl-expand.tcl tcl/lib/ns-lib.tcl | \
                        ../Tcl/tcl2c++ et_ns_lib > gen/ns_tcl.cc
```

The script, ~*ns*/bin/tcl-expand.tcl expands `ns-lib.tcl` by replacing all `source` lines with the corresponding source files. The program, ~*tclcl*/tcl2cc.c, converts the OTcl code into an equivalent EmbeddedTcl object, `et_ns_lib`.

During initialization, invoking the method `EmbeddedTcl::load` explicitly evaluates the array.

— ~*tclcl*/tcl-object.tcl is evaluated by the method `Tcl::init`(void); `Tcl_AppInit`() invokes `Tcl::Init`(). The exact command syntax for the load is:

```
et_tclobject.load();
```

— Similarly, ~*ns*/tcl/lib/ns-lib.tcl is evaluated directly by `Tcl_AppInit` in ~*ns*/ns_tclsh.cc.

```
et_ns_lib.load();
```

3.8 Class InstVar

This section describes the internals of the `class InstVar`. This class defines the methods and mechanisms to bind a C++ member variable in the compiled shadow object to a specified OTcl instance variable in the equivalent interpreted object. The binding is set up such that the value of the variable can be set or accessed either from within the interpreter, or from within the compiled code at all times.

There are five instance variable classes: `class InstVarReal`, `class InstVarTime`, `class InstVarBandwidth`, `class InstVarInt`, and `class InstVarBool`, corresponding to bindings for real, time, bandwidth, integer, and boolean valued variables respectively.

We now describe the mechanism by which instance variables are set up. We use the `class InstVarReal` to illustrate the concept. However, this mechanism is applicable to all five types of instance variables.

When setting up an interpreted variable to access a member variable, the member functions of the class InstVar assume that they are executing in the appropriate method execution context; therefore, they do not query the interpreter to determine the context in which this variable must exist.

In order to guarantee the correct method execution context, a variable must only be bound if its class is already established within the interpreter, and the interpreter is currently operating on an object in that class. Note that the former requires that when a method in a given class is going to make its variables accessible via the interpreter, there must be an associated

[4]The few places where this might not work are when certain variables might have to be defined or undefined, or otherwise the script contains code other than procedure and variable definitions and executes actions directly that might not be reversible.

class TclClass (Section **??**) defined that identifies the appropriate class hierarchy to the interpreter. The appropriate method execution context can therefore be created in one of two ways.

An implicit solution occurs whenever a new TclObject is created within the interpreter. This sets up the method execution context within the interpreter. When the compiled shadow object of the interpreted TclObject is created, the constructor for that compiled object can bind its member variables of that object to interpreted instance variables in the context of the newly created interpreted object.

An explicit solution is to define a `bind-variables` operation within a `command` function, that can then be invoked via the `cmd` method. The correct method execution context is established in order to execute the `cmd` method. Likewise, the compiled code is now operating on the appropriate shadow object, and can therefore safely bind the required member variables.

An instance variable is created by specifying the name of the interpreted variable, and the address of the member variable in the compiled object. The constructor for the base class InstVar creates an instance of the variable in the interpreter, and then sets up a trap routine to catch all accesses to the variable through the interpreter.

Whenever the variable is read through the interpreter, the trap routine is invoked just prior to the occurrence of the read. The routine invokes the appropriate `get` function that returns the current value of the variable. This value is then used to set the value of the interpreted variable that is then read by the interpreter.

Likewise, whenever the variable is set through the interpreter, the trap routine is invoked just after to the write is completed. The routine gets the current value set by the interpreter, and invokes the appropriate `set` function that sets the value of the compiled member to the current value set within the interpreter.

Part II

Simulator Basics

Chapter 4

The Class Simulator

The overall simulator is described by a Tcl `class Simulator`. It provides a set of interfaces for configuring a simulation and for choosing the type of event scheduler used to drive the simulation. A simulation script generally begins by creating an instance of this class and calling various methods to create nodes, topologies, and configure other aspects of the simulation. A subclass of Simulator called `OldSim` is used to support *ns* v1 backward compatibility.

The procedures and functions described in this chapter can be found in ~*ns*/tcl/lib/ns-lib.tcl, ~*ns*/scheduler.{cc,h}, and, ~*ns*/heap.h.

4.1 Simulator Initialization

When a new simulation object is created in tcl, the initialization procedure performs the following operations:

- initialize the packet format (calls `create_packetformat`)
- create a scheduler (defaults to a calendar scheduler)
- create a "null agent" (a discard sink used in various places)

The packet format initialization sets up field offsets within packets used by the entire simulation. It is described in more detail in the following chapter on packets (Chapter **??**). The scheduler runs the simulation in an event-driven manner and may be replaced by alternative schedulers which provide somewhat different semantics (see the following section for more detail). The null agent is created with the following call:

```
set nullAgent_ [new Agent/Null]
```

This agent is generally useful as a sink for dropped packets or as a destination for packets that are not counted or recorded.

4.2 Schedulers and Events

The simulator is an event-driven simulator. There are presently four schedulers available in the simulator, each of which is implemented using a different data structure: a simple linked-list, heap, calendar queue (default), and a special type called

"real-time". Each of these are described below. The scheduler runs by selecting the next earliest event, executing it to completion, and returning to execute the next event.Unit of time used by scheduler is seconds. Presently, the simulator is single-threaded, and only one event in execution at any given time. If more than one event are scheduled to execute at the same time, their execution is performed on the first scheduled – first dispatched manner. Simultaneous events are not re-ordered anymore by schedulers (as it was in earlier versions) and all schedulers should yeild the same order of dispatching given the same input.

No partial execution of events or pre-emption is supported.

An *event* generally comprises a "firing time" and a handler function. The actual definition of an event is found in ~*ns*/scheduler.h:

```
class Event {
public:
        Event* next_;                                   /* event list */
        Handler* handler_;                  /* handler to call when event ready */
        double time_;                           /* time at which event is ready */
        int uid_;                                       /* unique ID */
        Event() : time_(0), uid_(0) {}
};
/*
 *  The base class for all event handlers.  When an event's scheduled
 *  time arrives, it is passed to handle which must consume it.
 * i.e., if it needs to be freed it, it must be freed by the handler.
 */
class Handler {
 public:
        virtual void handle(Event* event);
};
```

Two types of objects are derived from the base `class Event`: packets and "at-events". Packets are described in detail in the next chapter (Chapter **??**). An at-event is a tcl procedure execution scheduled to occur at a particular time. This is frequently used in simulation scripts. A simple example of how it is used is as follows:

```
...
set ns_ [new Simulator]
$ns_ use-scheduler Heap
$ns_ at 300.5 "$self complete_sim"
...
```

This tcl code fragment first creates a simulation object, then changes the default scheduler implementation to be heap-based (see below), and finally schedules the function `$self complete_sim` to be executed at time 300.5 (seconds)(Note that this particular code fragment expects to be encapsulated in an object instance procedure, where the appropriate reference to `$self` is correctly defined.). At-events are implemented as events where the handler is effectively an execution of the tcl interpreter.

4.2.1 The List Scheduler

The list scheduler (`class Scheduler/List`) implements the scheduler using a simple linked-list structure. The list is kept in time-order (earliest to latest), so event insertion and deletion require scanning the list to find the appropriate entry. Choosing the next event for execution requires trimming the first entry off the head of the list. This implementation preserves event execution in a FIFO manner for simultaneous events.

4.2.2 the heap scheduler

The heap scheduler (`class Scheduler/Heap`) implements the scheduler using a heap structure. This structure is superior to the list structure for a large number of events, as insertion and deletion times are in $O(\log n)$ for n events. This implementation in *ns* v2 is borrowed from the MaRS-2.0 simulator [**?**]; it is believed that MaRS itself borrowed the code from NetSim [**?**], although this lineage has not been completely verified.

4.2.3 The Calendar Queue Scheduler

The calendar queue scheduler (`class Scheduler/Calendar`) uses a data structure analogous to a one-year desk calendar, in which events on the same month/day of multiple years can be recorded in one day. It is formally described in [**?**], and informally described in Jain (p. 410) [**?**]. The implementation of Calendar queues in *ns* v2 was contributed by David Wetherall (presently at MIT/LCS).

The calendar queue scheduler since *ns* v2.33 is improved by the following three algorithms:

- A heuristic improvement that changes the linear search direction in enqueue operations. The original implementation searches the events in a bucket in *chronological order* to find the in-order spot for the event that is being inserted. The new implementation searches the bucket in *reverse chronological order* because the event being inserted is usually later than most of the events that are already in the bucket.
- A new bucket width estimation that uses the average interval of *dequeued events* as the estimation of bucket width. It is stated in [**?**] that the optimal bucket width should be the *average inverval of all events in the future*. The original implementation uses the average interval of *future events currently in the most crowded bucket* as the estimation. This estimation is unstable because it is very likely that many future events will be inserted into the bucket after this estimation, significantly changing the averaged event interval in the bucket. The new implementation uses the observed event interval in the past, which will not change, to estimate the event interval in future.
- SNOOPy Calendar Queue: a Calendar queue variant that dynamically tunes the bucket width according to the cost trade-off between enqueue operation and dequeue operation. The SNOOPy queue improvement is described in [**?**]. In this implementation, there is one tcl parameter `adjust_new_width_interval_` specifying the interval with which the SNOOPy queue should re-calculate the bucket width. Setting this parameter to 0 turns off the SNOOPy queue algorithm and degrades the scheduler back to the original Calendar Queue. In general, normal simulation users are not expected to change this parameter.

The details of these improvements are described in [**?**].

The implementation of these three improvements was contributed by Xiaoliang (David) Wei at Caltech/NetLab.

4.2.4 The Real-Time Scheduler

The real-time scheduler (`class Scheduler/RealTime`) attempts to synchronize the execution of events with real-time. It is currently implemented as a subclass of the list scheduler. The real-time capability in ns is still under development, but is used to introduce an *ns* simulated network into a real-world topology to experiment with easily-configured network topologies, cross-traffic, etc. This only works for relatively slow network traffic data rates, as the simulator must be able to keep pace with the real-world packet arrival rate, and this synchronization is not presently enforced.

4.2.5 Precision of the scheduler clock used in ns

Precision of the scheduler clock can be defined as the smallest time-scale of the simulator that can be correctly represented. The clock variable for ns is represented by a double. As per the IEEE std for floating numbers, a double, consisting of 64 bits must allocate the following bits between its sign, exponent and mantissa fields.

```
sign     exponent      mantissa
 1 bit   11 bits        52 bits
```

Any floating number can be represented in the form ($X * 2^n$) where X is the mantissa and n is the exponent. Thus the precision of timeclock in ns can be defined as ($1/2^(52)$). As simulation runs for longer times the number of remaining bits to represent the time educes thus reducing the accuracy. Given 52 bits we can safely say time upto around ($2^(40)$) can be represented with considerable accuracy. Anything greater than that might not be very accurate as you have remaining 12 bits to represent the time change. However ($2^(40)$) is a very large number and we donot anticipate any problem regarding precision of time in ns.

4.3 Other Methods

The `Simulator` class provides a number of methods used to set up the simulation. They generally fall into three categories: methods to create and manage the topology (which in turn consists of managing the nodes (Chapter **??**) and managing the links (Chapter **??**)), methods to perform tracing (Chapter **??**), and helper functions to deal with the scheduler. The following is a list of the non-topology related simulator methods:

```
Simulator instproc now                                  ;# return scheduler's notion of current time
Simulator instproc at args                              ;# schedule execution of code at specified time
Simulator instproc cancel args                          ;# cancel event
Simulator instproc run args                             ;# start scheduler
Simulator instproc halt                                 ;# stop (pause) the scheduler
Simulator instproc flush-trace                          ;# flush all trace object write buffers
Simulator instproc create-trace  type files src dst     ;# create trace object
Simulator instproc create_packetformat                  ;# set up the simulator's packet format
```

4.4 Commands at a glance

Synopsis:
ns <otclfile> <arg> <arg>..

Description:
Basic command to run a simulation script in ns.
The simulator (ns) is invoked via the ns interpreter, an extension of the vanilla otclsh command shell. A simulation is defined by a OTcl script (file). Several examples of OTcl scripts can be found under *ns*/tcl/ex directory.

The following is a list of simulator commands commonly used in simulation scripts:

set ns_ [new Simulator]

This command creates an instance of the simulator object.

set now [$ns_ now]

The scheduler keeps track of time in a simulation. This returns scheduler's notion of current time.

$ns_ halt

This stops or pauses the scheduler.

$ns_ run

This starts the scheduler.

$ns_ at <time> <event>

This schedules an <event> (which is normally a piece of code) to be executed at the specified <time>.
e.g $ns_ at $opt(stop) "puts ŃS EXITING..¨ ; $ns_ halt"
or, $ns_ at 10.0 "$ftp start"

$ns_ cancel <event>

Cancels the event. In effect, event is removed from scheduler’s list of ready to run events.

$ns_ create-trace <type> <file> <src> <dst> <optional arg: op>

This creates a trace-object of type <type> between <src> and <dst> objects and attaches trace-object to <file> for writing trace-outputs. If op is defined as "nam", this creates nam tracefiles; otherwise if op is not defined, ns tracefiles are created on default.

$ns_ flush-trace

Flushes all trace object write buffers.

$ns_ gen-map

This dumps information like nodes, node components, links etc created for a given simulation. This may be broken for some scenarios (like wireless).

$ns_ at-now <args>

This is in effect like command "$ns_ at $now $args". Note that this function may not work because of tcl’s string number resolution.

These are additional simulator (internal) helper functions (normally used for developing/changing the ns core code) :

$ns_ use-scheduler <type>

Used to specify the type of scheduler to be used for simulation. The different types of scheduler available are List, Calendar, Heap and RealTime. Currently Calendar is used as default.

$ns_ after <delay> <event>

Scheduling an <event> to be executed after the lapse of time <delay>.

$ns_ clearMemTrace

Used for memory debugging purposes.

$ns_ is-started

This returns true if simulator has started to run and false if not.

$ns_ dumpq

Command for dumping events queued in scheduler while scheduler is halted.

$ns_ create_packetformat

This sets up simulator’s packet format.

Chapter 5

Nodes and Packet Forwarding

This chapter describes one aspect of creating a topology in *ns*, *i.e.*, creating the nodes. In the next chapter (Chapter **??**), we will describe second aspect of creating the topology, *i.e.*, connecting the nodes to form links.

Recall that each simulation requires a single instance of the `class Simulator` to control and operate that simulation. The class provides instance procedures to create and manage the topology, and internally stores references to each element of the topology. We begin by describing the procedures in the class Simulator (Section **??**). We then describe the instance procedures in the class Node (Section **??**) to access and operate on individual nodes. We conclude with detailed descriptions of the Classifier (Section **??**) from which the more complex node objects are formed.

The procedures and functions described in this chapter can be found in ~*ns*/tcl/lib/ns-lib.tcl, ~*ns*/tcl/lib/ns-node.tcl, ~*ns*/tcl/lib/ns-rtmodule.tcl, ~*ns*/rtmodule.{cc,h}, ~*ns*/classifier.{cc, h}, ~*ns*/classifier-addr.cc, ~*ns*/classifier-mcast.cc, ~*ns*/classifier-mpath.cc, and, ~*ns*/replicator.cc.

5.1 Node Basics

The basic primitive for creating a node is

```
set ns [new Simulator]
$ns node
```

The instance procedure `node` constructs a node out of more simple classifier objects (Section **??**). The Node itself is a standalone class in OTcl. However, most of the components of the node are themselves TclObjects. The typical structure of a (unicast) node is as shown in Figure **??**. This simple structure consists of two TclObjects: an address classifer (`classifer_`) and a port classifier (`dmux_`). The function of these classifiers is to distribute incoming packets to the correct agent or outgoing link.

All nodes contain at least the following components:

- an address or `id_`, monotonically increasing by 1 (from initial value 0) across the simulation namespace as nodes are created,
- a list of neighbors (`neighbor_`),

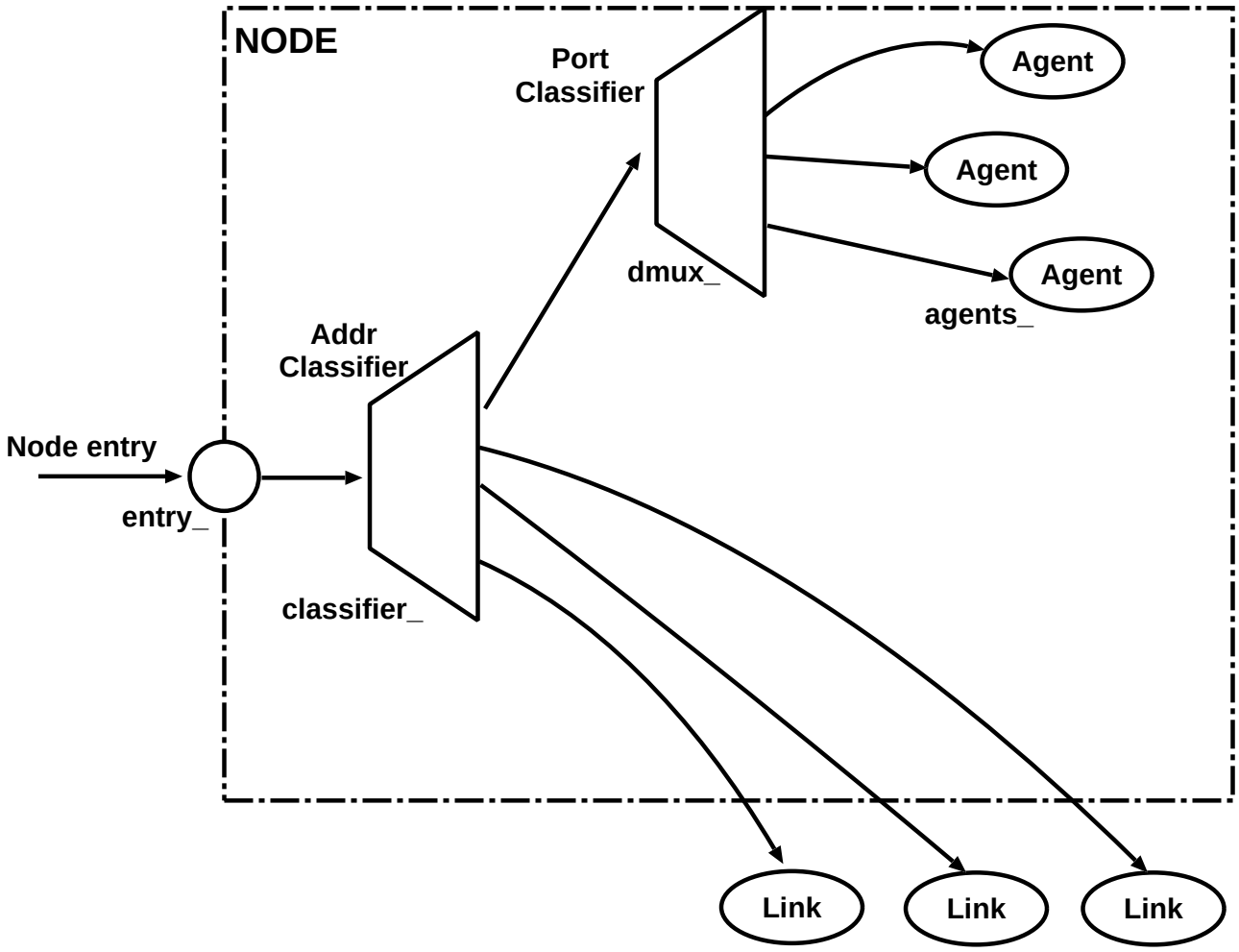

Figure 5.1: Structure of a Unicast Node. Notice that entry_ is simply a label variable instead of a real object, e.g., the classifier_.

- a list of agents (`agent_`),
- a node type identifier (`nodetype_`), and
- a routing module (described in Section **??** below)

By default, nodes in *ns* are constructed for unicast simulations. In order to enable multicast simulation, the simulation should be created with an option "-multicast on", e.g.:

```
set ns [new Simulator -multicast on]
```

The internal structure of a typical multicast node is shown in Figure **??**.

When a simulation uses multicast routing, the highest bit of the address indicates whether the particular address is a multicast address or an unicast address. If the bit is 0, the address represents a unicast address, else the address represents a multicast address.

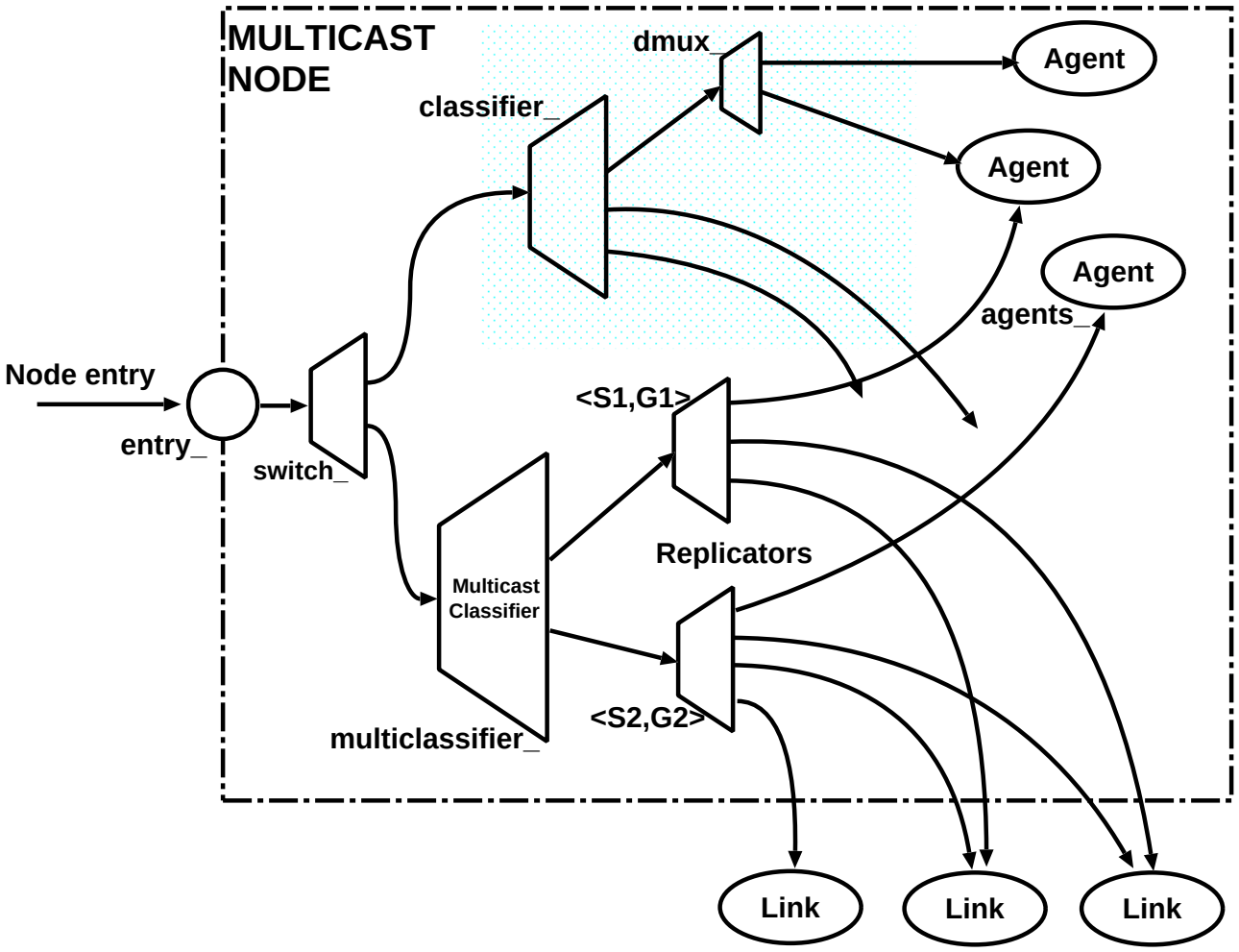

Figure 5.2: Internal Structure of a Multicast Node.

5.2 Node Methods: Configuring the Node

Procedures to configure an individual node can be classified into:

- Control functions
- Address and Port number management, unicast routing functions
- Agent management
- Adding neighbors

We describe each of the functions in the following paragraphs.

Control functions

1. `$node entry` returns the entry point for a node. This is the first element which will handle packets arriving at that node.

The Node instance variable, `entry_`, stores the reference this element. For unicast nodes, this is the address classifier that looks at the higher bits of the destination address. The instance variable, `classifier_` contains the reference to this classifier. However, for multicast nodes, the entry point is the `switch_` which looks at the first bit to decide whether it should forward the packet to the unicast classifier, or the multicast classifier as appropriate.

2. `$node reset` will reset all agents at the node.

Address and Port number management The procedure `$node id` returns the node number of the node. This number is automatically incremented and assigned to each node at creation by the class Simulator method, `$ns node`.The class Simulator also stores an instance variable array[1], `Node_`, indexed by the node id, and contains a reference to the node with that id.

The procedure `$node agent` ⟨`port`⟩ returns the handle of the agent at the specified port. If no agent at the specified port number is available, the procedure returns the null string.

The procedure `alloc-port` returns the next available port number. It uses an instance variable, `np_`, to track the next unallocated port number.

The procedures, `add-route` and `add-routes`, are used by unicast routing (Chapter **??**) to add routes to populate the `classifier_` The usage syntax is `$node add-route` ⟨`destination id`⟩ ⟨`TclObject`⟩. `TclObject` is the entry of `dmux_`, the port demultiplexer at the node, if the destination id is the same as this node's id, it is often the head of a link to send packets for that destination to, but could also be the the entry for other classifiers or types of classifiers.

`$node add-routes` ⟨`destination id`⟩ ⟨`TclObjects`⟩ is used to add multiple routes to the same destination that must be used simultaneously in round robin manner to spread the bandwidth used to reach that destination across all links equally. It is used only if the instance variable `multiPath_` is set to 1, and detailed dynamic routing strategies are in effect, and requires the use of a multiPath classifier. We describe the implementation of the multiPath classifier later in this chapter (Section **??**); however, we defer the discussion of multipath routing (Chapter **??**) to the chapter on unicast routing.

The dual of `add-routes{}` is `delete-routes{}`. It takes the id, a list of `TclObjects`, and a reference to the simulator's `nullagent`. It removes the TclObjects in the list from the installed routes in the multipath classifier. If the route entry in the classifier does not point to a multipath classifier, the routine simply clears the entry from `classifier_`, and installs the `nullagent` in its place.

Detailed dynamic routing also uses two additional methods: the instance procedure `init-routing{}` sets the instance variable `multiPath_` to be equal to the class variable of the same name. It also adds a reference to the route controller object at that node in the instance variable, `rtObject_`. The procedure `rtObject?{}` returns the handle for the route object at the node.

Finally, the procedure `intf-changed{}` is invoked by the network dynamics code if a link incident on the node changes state. Additional details on how this procedure is used are discussed later in the chapter on network dynamics (Chapter **??**).

Agent management Given an ⟨agent⟩, the procedure `attach{}` will add the agent to its list of `agents_`, assign a port number the agent and set its source address, set the target of the agent to be its (*i.e.*, the node's) `entry{}`, and add a pointer to the port demultiplexer at the node (`dmux_`) to the agent at the corresponding slot in the `dmux_` classifier.

Conversely, `detach{}`will remove the agent from `agents_`, and point the agent's target, and the entry in the node `dmux_` to `nullagent`.

[1] *i.e.*, an instance variable of a class that is also an array variable

Tracking Neighbors Each node keeps a list of its adjacent neighbors in its instance variable, `neighbor_`. The procedure `add-neighbor{}` adds a neighbor to the list. The procedure `neighbors{}` returns this list.

5.3 Node Configuration Interface

NOTE: This API, especially its internal implementation which is messy at this point, is still a moving target. It may undergo significant changes in the near future. However, we will do our best to maintain the same interface as described in this chapter. In addition, this API currently does not cover all existing nodes in the old format, namely, nodes built using inheritance, and parts of mobile IP. It is principally oriented towards wireless and satellite simulation. [Sep 15, 2000; updated June 2001].

`Simulator::node-config{}` accommodates flexible and modular construction of different node definitions within the same base Node class. For instance, to create a mobile node capable of wireless communication, one no longer needs a specialized node creation command, e.g., `dsdv-create-mobile-node{}`; instead, one changes default configuration parameters, such as

```
$ns node-config -adhocRouting dsdv
```

before actually creating the node with the command: `$ns node`. Together with routing modules, this allows one to combine "arbitrary" routing functionalities within a single node without resorting to multiple inheritance and other fancy object gimmicks. We will describe this in more detail in Section **??**. The functions and procedures relevant to the new node APIs may be found in ~*ns*/tcl/lib/ns-node.tcl.

The node configuration interface consists of two parts. The first part deals with node configuration, while the second part actually creates nodes of the specified type. We have already seen the latter in Section **??**, in this section we will describe the configuration part.

Node configuration essentially consists of defining the different node characteristics before creating them. They may consist of the type of addressing structure used in the simulation, defining the network components for mobilenodes, turning on or off the trace options at Agent/Router/MAC levels, selecting the type of adhoc routing protocol for wireless nodes or defining their energy model.

As an example, node-configuration for a wireless, mobile node that runs AODV as its adhoc routing protocol in a hierarchical topology would be as shown below. We decide to turn tracing on at the agent and router level only. Also we assume a topology has been instantiated with "set topo [new Topography]". The node-config command would look like the following:

```
$ns_ node-config -addressType hierarchical \
                 -adhocRouting AODV \
                 -llType LL \
                 -macType Mac/802_11 \
                 -ifqType Queue/DropTail/PriQueue \
                 -ifqLen 50 \
                 -antType Antenna/OmniAntenna \
                 -propType Propagation/TwoRayGround \
                 -phyType Phy/WirelessPhy \
                 -topologyInstance $topo \
                 -channel Channel/WirelessChannel \
                 -agentTrace ON \
                 -routerTrace ON \
                 -macTrace OFF \
                 -movementTrace OFF
```

The default values for all the above options are NULL except `-addressingType` whose default value is flat. The option `-reset` can be used to reset all node-config parameters to their default value.

Note that the config command can be broken down into separate lines like

```
$ns_ node-config -addressingType hier
$ns_ node-config -macTrace ON
```

The options that need to be changed may only be called. For example after configuring for AODV mobilenodes as shown above (and after creating AODV mobilenodes), we may configure for AODV base-station nodes in the following way:

```
$ns_ node-config -wiredRouting ON
```

While all other features for base-station nodes and mobilenodes are same, the base-station nodes are capable of wired routing, while mobilenodes are not. In this way we can change node-configuration only when it is required.

All node instances created after a given node-configuration command will have the same property unless a part or all of the node-config command is executed with different parameter values. And all parameter values remain unchanged unless they are expicitly changed. So after creation of the AODV base-station and mobilenodes, if we want to create simple nodes, we will use the following node-configuration command:

```
$ns_ node-config -reset
```

This will set all parameter values to their default setting which basically defines configuration of a simple node.

Currently, this type of node configuration is oriented towards wireless and satellite nodes. Table 5.1 lists the available options for these kinds of nodes. The example scripts ~*ns*/tcl/ex/simple-wireless.tcl and ~*ns*/tcl/ex/sat-mixed.tcl provide usage examples.

5.4 The Classifier

The function of a node when it receives a packet is to examine the packet's fields, usually its destination address, and on occasion, its source address. It should then map the values to an outgoing interface object that is the next downstream recipient of this packet.

In *ns*, this task is performed by a simple *classifier* object. Multiple classifier objects, each looking at a specific portion of the packet forward the packet through the node. A node in *ns* uses many different types of classifiers for different purposes. This section describes some of the more common, or simpler, classifier objects in *ns*.

We begin with a description of the base class in this section. The next subsections describe the address classifier (Section **??**), the multicast classifier (Section **??**), the multipath classifier (Section **??**), the hash classifier (Section **??**), and finally, the replicator (Section **??**).

A classifier provides a way to match a packet against some logical criteria and retrieve a reference to another simulation object based on the match results. Each classifier contains a table of simulation objects indexed by *slot number*. The job of a classifier is to determine the slot number associated with a received packet and forward that packet to the object referenced by that particular slot. The C++ `class Classifier` (defined in ~*ns*/classifier.h) provides a base class from which other classifiers are derived.

option	available values	default
general		
addressType	flat, hierarchical	flat
MPLS	ON, OFF	OFF
both satellite- and wireless-oriented		
wiredRouting	ON, OFF	OFF
llType	LL, LL/Sat	""
macType	Mac/802_11, Mac/Csma/Ca, Mac/Sat, Mac/Sat/UnslottedAloha, Mac/Tdma	""
ifqType	Queue/DropTail, Queue/DropTail/PriQueue	""
phyType	Phy/WirelessPhy, Phy/Sat	""
wireless-oriented		
adhocRouting	DIFFUSION/RATE, DIFFUSION/PROB, DSDV, DSR, FLOODING, OMNIMCAST, AODV, TORA, M-DART PUMA	""
propType	Propagation/TwoRayGround, Propagation/Shadowing	""
propInstance	Propagation/TwoRayGround, Propagation/Shadowing	""
antType	Antenna/OmniAntenna	""
channel	Channel/WirelessChannel, Channel/Sat	""
topoInstance	<topology file>	""
mobileIP	ON, OFF	OFF
energyModel	EnergyModel	""
initialEnergy	<value in Joules>	""
rxPower	<value in W>	""
txPower	<value in W>	""
idlePower	<value in W>	""
agentTrace	ON, OFF	OFF
routerTrace	ON, OFF	OFF
macTrace	ON, OFF	OFF
movementTrace	ON, OFF	OFF
errProc	UniformErrorProc	""
FECProc	?	?
toraDebug	ON, OFF	OFF
satellite-oriented		
satNodeType	polar, geo, terminal, geo-repeater	""
downlinkBW	<bandwidth value, e.g. "2Mb">	""

Table 5.1: Available options for node configuration (see tcl/lib/ns-lib.tcl).

```
class Classifier : public NsObject {
public:
        ~Classifier();
        void recv(Packet*, Handler* h = 0);
 protected:
        Classifier();
        void install(int slot, NsObject*);
        void clear(int slot);
        virtual int command(int argc, const char*const* argv);
        virtual int classify(Packet *const) = 0;
        void alloc(int);
        NsObject** slot_;                    /* table that maps slot number to a NsObject */
        int nslot_;
        int maxslot_;
```

```
};
```

The `classify`() method is pure virtual, indicating the class `Classifier` is to be used only as a base class. The `alloc`() method dynamically allocates enough space in the table to hold the specified number of slots. The `install`() and `clear`() methods add or remove objects from the table. The `recv`() method and the OTcl interface are implemented as follows in ~*ns*/classifier.cc:

```
/*
 * objects only ever see "packet" events, which come either
 * from an incoming link or a local agent (i.e., packet source).
 */
void Classifier::recv(Packet* p, Handler*)
{
        NsObject* node;
        int cl = classify(p);
        if (cl < 0 || cl >= nslot_ || (node = slot_[cl]) == 0) {
                Tcl::instance().evalf("%s no-slot %d", name(), cl);
                Packet::free(p);
                return;
        }
        node->recv(p);
}

int Classifier::command(int argc, const char*const* argv)
{
        Tcl& tcl = Tcl::instance();
        if (argc == 3) {
                /*
                 * $classifier clear $slot
                 */
                if (strcmp(argv[1], "clear") == 0) {
                        int slot = atoi(argv[2]);
                        clear(slot);
                        return (TCL_OK);
                }
                /*
                 * $classifier installNext $node
                 */
                if (strcmp(argv[1], "installNext") == 0) {
                        int slot = maxslot_ + 1;
                        NsObject* node = (NsObject*)TclObject::lookup(argv[2]);
                        install(slot, node);
                        tcl.resultf("%u", slot);
                        return TCL_OK;
                }
                if (strcmp(argv[1], "slot") == 0) {
                        int slot = atoi(argv[2]);
                        if ((slot >= 0) || (slot < nslot_)) {
                                tcl.resultf("%s", slot_[slot]->name());
                                return TCL_OK;
                        }
                        tcl.resultf("Classifier: no object at slot %d", slot);
                        return (TCL_ERROR);
```

```
                }
        } else if (argc == 4) {
                /*
                 * $classifier install $slot $node
                 */
                if (strcmp(argv[1], "install") == 0) {
                        int slot = atoi(argv[2]);
                        NsObject* node = (NsObject*)TclObject::lookup(argv[3]);
                        install(slot, node);
                        return (TCL_OK);
                }
        }
        return (NsObject::command(argc, argv));
}
```

When a classifier `recv`()'s a packet, it hands it to the `classify`() method. This is defined differently in each type of classifier derived from the base class. The usual format is for the `classify`() method to determine and return a slot index into the table of slots. If the index is valid, and points to a valid TclObject, the classifier will hand the packet to that object using that object's `recv`() method. If the index is not valid, the classifier will invoke the instance procedure `no-slot`{} to attempt to populate the table correctly. However, in the base class `Classifier::no-slot`{} prints and error message and terminates execution.

The `command`() method provides the following instproc-likes to the interpreter:

- `clear`{⟨slot⟩} clears the entry in a particular slot.
- `installNext`{⟨object⟩} installs the object in the next available slot, and returns the slot number.

  Note that this instproc-like is overloaded by an instance procedure of the same name that stores a reference to the object stored. This then helps quick query of the objects installed in the classifier from OTcl.
- `slot`{⟨index⟩} returns the object stored in the specified slot.
- `install`{⟨index⟩, ⟨object⟩} installs the specified ⟨object⟩ at the slot ⟨index⟩.

  Note that this instproc-like too is overloaded by an instance procedure of the same name that stores a reference to the object stored. This is also to quickly query of the objects installed in the classifier from OTcl.

5.4.1 Address Classifiers

An address classifier is used in supporting unicast packet forwarding. It applies a bitwise shift and mask operation to a packet's destination address to produce a slot number. The slot number is returned from the `classify`() method. The `class AddressClassifier` (defined in ~*ns*/classifier-addr.cc) ide defined as follows:

```
class AddressClassifier : public Classifier {
public:
        AddressClassifier() : mask_(~0), shift_(0) {
                bind("mask_", (int*)&mask_);
                bind("shift_", &shift_);
        }
protected:
        int classify(Packet *const p) {
                IPHeader *h = IPHeader::access(p->bits());
                return ((h->dst() >> shift_) & mask_);
```

```
        }
        nsaddr_t mask_;
        int shift_;
};
```

The class imposes no direct semantic meaning on a packet's destination address field. Rather, it returns some number of bits from the packet's `dst_` field as the slot number used in the `Classifier::recv`() method. The `mask_` and `shift_` values are set through OTcl.

5.4.2 Multicast Classifiers

The multicast classifier classifies packets according to both source and destination (group) addresses. It maintains a (chained hash) table mapping source/group pairs to slot numbers. When a packet arrives containing a source/group unknown to the classifier, it invokes an Otcl procedure `Node::new-group`{} to add an entry to its table. This OTcl procedure may use the method `set-hash` to add new (source, group, slot) 3-tuples to the classifier's table. The multicast classifier is defined in ~*ns*/classifier-mcast.cc as follows:

```
static class MCastClassifierClass : public TclClass {
public:
        MCastClassifierClass() : TclClass("Classifier/Multicast") {}
        TclObject* create(int argc, const char*const* argv) {
                return (new MCastClassifier());
        }
} class_mcast_classifier;

class MCastClassifier : public Classifier {
public:
        MCastClassifier();
        ~MCastClassifier();
protected:
        int command(int argc, const char*const* argv);
        int classify(Packet *const p);
        int findslot();
        void set_hash(nsaddr_t src, nsaddr_t dst, int slot);
        int hash(nsaddr_t src, nsaddr_t dst) const {
                u_int32_t s = src ^ dst;
                s ^= s >> 16;
                s ^= s >> 8;
                return (s & 0xff);
        }
        struct hashnode {
                int slot;
                nsaddr_t src;
                nsaddr_t dst;
                hashnode* next;
        };
        hashnode* ht_[256];
        const hashnode* lookup(nsaddr_t src, nsaddr_t dst) const;
};

int MCastClassifier::classify(Packet *const pkt)
```

```
        {
                IPHeader *h = IPHeader::access(pkt->bits());
                nsaddr_t src = h->src() >> 8; /*XXX*/
                nsaddr_t dst = h->dst();
                const hashnode* p = lookup(src, dst);
                if (p == 0) {
                        /*
                         * Didn't find an entry.
                         * Call tcl exactly once to install one.
                         * If tcl doesn't come through then fail.
                         */
                        Tcl::instance().evalf("%s new-group %u %u", name(), src, dst);
                        p = lookup(src, dst);
                        if (p == 0)
                                return (-1);
                }
                return (p->slot);
        }
```

The `class MCastClassifier`mplements a chained hash table and applies a hash function on both the packet source and destination addresses. The hash function returns the slot number to index the `slot_` table in the underlying object. A hash miss implies packet delivery to a previously-unknown group; OTcl is called to handle the situation. The OTcl code is expected to insert an appropriate entry into the hash table.

5.4.3 MultiPath Classifier

This object is devised to support equal cost multipath forwarding, where the node has multiple equal cost routes to the same destination, and would like to use all of them simultaneously. This object does not look at any field in the packet. With every succeeding packet, it simply returns the next filled slot in round robin fashion. The definitions for this classifier are in ~*ns*/classifier-mpath.cc, and are shown below:

```
class MultiPathForwarder : public Classifier {
public:
        MultiPathForwarder() : ns_(0), Classifier() {}
        virtual int classify(Packet* const) {
                int cl;
                int fail = ns_;
                do {
                        cl = ns_++;
                        ns_ %= (maxslot_ + 1);
                } while (slot_[cl] == 0 && ns_ != fail);
                return cl;
        }
private:
        int ns_;                                    /* next slot to be used.  Probably a misnomer? */
};
```

5.4.4 Hash Classifier

This object is used to classify a packet as a member of a particular *flow*. As their name indicates, hash classifiers use a hash table internally to assign packets to flows. These objects are used where flow-level information is required (e.g. in flow-specific queuing disciplines and statistics collection). Several "flow granularities" are available. In particular, packets may be assigned to flows based on flow ID, destination address, source/destination addresses, or the combination of source/destination addresses plus flow ID. The fields accessed by the hash classifier are limited to the `ip` header: `src(), dst(), flowid()` (see `ip.h`).

The hash classifier is created with an integer argument specifying the initial size of its hash table. The current hash table size may be subsequently altered with the `resize` method (see below). When created, the instance variables `shift_` and `mask_` are initialized with the simulator's current NodeShift and NodeMask values, respectively. These values are retrieved from the `AddrParams` object when the hash classifier is instantiated. The hash classifier will fail to operate properly if the `AddrParams` structure is not initialized. The following constructors are used for the various hash classifiers:

```
Classifier/Hash/SrcDest
Classifier/Hash/Dest
Classifier/Hash/Fid
Classifier/Hash/SrcDestFid
```

The hash classifier receives packets, classifies them according to their flow criteria, and retrieves the classifier *slot* indicating the next node that should receive the packet. In several circumstances with hash classifiers, most packets should be associated with a single slot, while only a few flows should be directed elsewhere. The hash classifier includes a `default_` instance variable indicating which slot is to be used for packets that do not match any of the per-flow criteria. The `default_` may be set optionally.

The methods for a hash classifier are as follows:

```
$hashcl set-hash buck src dst fid slot
$hashcl lookup buck src dst fid
$hashcl del-hash src dst fid
$hashcl resize nbuck
```

The `set-hash`() method inserts a new entry into the hash table within the hash classifier. The `buck` argument specifies the hash table bucket number to use for the insertion of this entry. When the bucket number is not known, `buck` may be specified as `auto`. The `src, dst` and `fid` arguments specify the IP source, destination, and flow IDs to be matched for flow classification. Fields not used by a particular classifier (e.g. specifying `src` for a flow-id classifier) is ignored. The `slot` argument indicates the index into the underlying slot table in the base `Classifier` object from which the hash classifier is derived. The `lookup` function returns the name of the object associated with the given `buck/src/dst/fid` tuple. The `buck` argument may be `auto`, as for `set-hash`. The `del-hash` function removes the specified entry from the hash table. Currently, this is done by simply marking the entry as inactive, so it is possible to populate the hash table with unused entries. The `resize` function resizes the hash table to include the number of buckets specified by the argument `nbuck`.

Provided no default is defined, a hash classifier will perform a call into OTcl when it receives a packet which matches no flow criteria. The call takes the following form:

```
$obj unknown-flow src dst flowid buck
```

Thus, when a packet matching no flow criteria is received, the method `unknown-flow` of the instantiated hash classifier object is invoked with the source, destination, and flow id fields from the packet. In addition, the `buck` field indicates the hash

bucket which should contain this flow if it were inserted using `set-hash`. This arrangement avoids another hash lookup when performing insertions into the classifier when the bucket is already known.

5.4.5 Replicator

The replicator is different from the other classifiers we have described earlier, in that it does not use the classify function. Rather, it simply uses the classifier as a table of n slots; it overloads the `recv`() method to produce n copies of a packet, that are delivered to all n objects referenced in the table.

To support multicast packet forwarding, a classifier receiving a multicast packet from source S destined for group G computes a hash function $h(S, G)$ giving a "slot number" in the classifier's object table. In multicast delivery, the packet must be copied once for each link leading to nodes subscribed to G minus one. Production of additional copies of the packet is performed by a `Replicator` class, defined in `replicator.cc`:

```
/*
 * A replicator is not really a packet classifier but
 * we simply find convenience in leveraging its slot table.
 * (this object used to implement fan-out on a multicast
 * router as well as broadcast LANs)
 */
class Replicator : public Classifier {
public:
        Replicator();
        void recv(Packet*, Handler* h = 0);
        virtual int classify(Packet* const) {};
protected:
        int ignore_;
};

void Replicator::recv(Packet* p, Handler*)
{
        IPHeader *iph = IPHeader::access(p->bits());
        if (maxslot_ < 0) {
                if (!ignore_)
                        Tcl::instance().evalf("%s drop %u %u", name(),
                                iph->src(), iph->dst());
                Packet::free(p);
                return;
        }
        for (int i = 0; i < maxslot_; ++i) {
                NsObject* o = slot_[i];
                if (o != 0)
                        o->recv(p->copy());
        }
        /* we know that maxslot is non-null */
        slot_[maxslot_]->recv(p);
}
```

As we can see from the code, this class does not really classify packets. Rather, it replicates a packet, one for each entry in its table, and delivers the copies to each of the nodes listed in the table. The last entry in the table gets the "original" packet. Since the `classify`() method is pure virtual in the base class, the replicator defines an empty `classify`() method.

5.5 Routing Module and Classifier Organization

As we have seen, a *ns* node is essentially a collection of classifiers. The simplest node (unicast) contains only one address classifier and one port classifier, as shown in Figure **??**. When one extends the functionality of the node, more classifiers are added into the base node, for instance, the multicast node shown in Figure **??**. As more function blocks is added, and each of these blocks requires its own classifier(s), it becomes important for the node to provide a *uniform* interface to organize these classifiers and to bridge these classifiers to the route computation blocks.

The classical method to handle this case is through class inheritance. For instance, if one wants a node that supports hierarchical routing, one simply derive a Node/Hier from the base node and override the classifier setup methods to insert hierarchical classifiers. This method works well when the new function blocks are independent and cannot be "arbitrarily" mixed. For instance, both hierarchical routing and ad hoc routing use their own set of classifiers. Inheritance would require that we have Node/Hier that supports the former, and Node/Mobile for the latter. This becomes slightly problematic when one wants an ad hoc routing node that supports hierarchical routing. In this simple case one may use multiple inheritance to solve the problem, but this quickly becomes infeasible as the number of such function blocks increases.

The only method to solve this problem is object composition. The base node needs to define a set of interfaces for classifier access and organization. These interfaces should

- allow individual routing modules that implement their own classifiers to insert their classifiers into the node;
- allow route computation blocks to populate routes to classifiers in all routing modules that need this information,
- provide a single point of management for existing routing modules.

In addition, we should also define a uniform interface for routing modules to connect to the node interfaces, so as to provide a systematic approach to extending node functionality. In this section we will describe the design of routing modules as well as that of the corresponding node interfaces.

5.5.1 Routing Module

In general, every routing implementation in *ns* consists of three function blocks:

- *Routing agent* exchanges routing packet with neighbors,
- *Route logic* uses the information gathered by routing agents (or the global topology database in the case of static routing) to perform the actual route computation,
- *Classifiers* sit inside a Node. They use the computed routing table to perform packet forwarding.

Notice that when implementing a new routing protocol, one does not necessarily implement all of these three blocks. For instance, when one implements a link state routing protocol, one simply implement a routing agent that exchanges information in the link state manner, and a route logic that does Dijkstra on the resulting topology database. It can then use the same classifiers as other unicast routing protocols.

When a new routing protocol implementation includes more than one function blocks, especially when it contains its own classifier, it is desirable to have another object, which we call a *routing module*, that manages all these function blocks and to interface with node to organize its classifiers. Figure **??** shows functional relation among these objects. Notice that routing modules may have direct relationship with route computation blocks, i.e., route logic and/or routing agents. However, route computation MAY not install their routes directly through a routing module, because there may exists other modules that

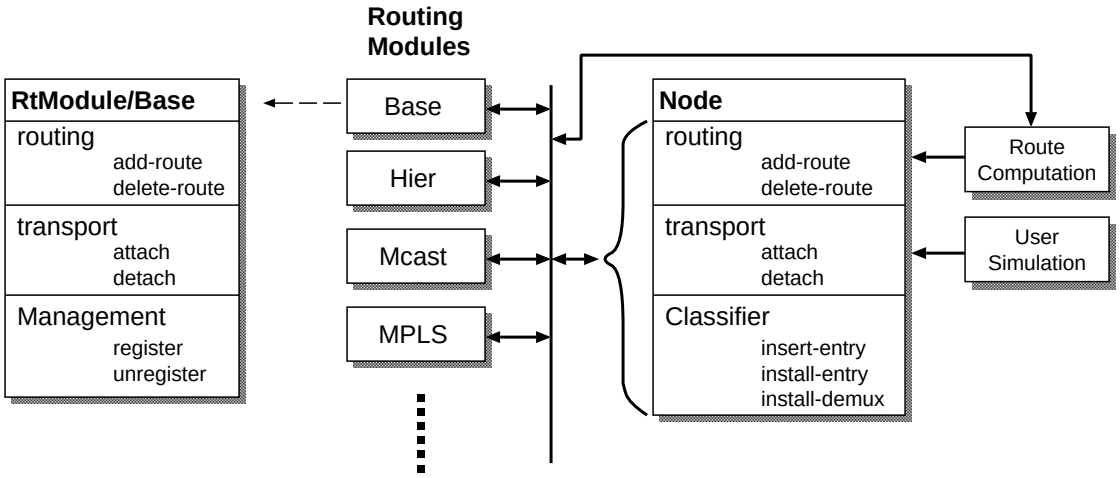

Figure 5.3: Interaction among node, routing module, and routing. The dashed line shows the details of one routing module.

are interested in learning about the new routes. This is not a requirement, however, because it is possible that some route computation is specific to one particular routing module, for instance, label installation in the MPLS module.

A routing module contains three major functionalities:

1. A routing module initializes its connection to a node through `register{}`, and tears the connection down via `unregister{}`. Usually, in `register{}` a routing module (1) tells the node whether it interests in knowing route updates and transport agent attachments, and (2) creates its classifiers and install them in the node (details described in the next subsection). In `unregister{}` a routing module does the exact opposite: it deletes its classifiers and removes its hooks on routing update in the node.
2. If a routing module is interested in knowing routing updates, the node will inform the module via `RtModule::add-route{dst, target}` and `RtModule::delete-route{dst, nullagent}`.
3. If a routing module is interested in learning about transport agent attachment and detachment in a node, the node will inform the module via `RtModule::attach{agent, port}` and `RtModule::detach{agent, nullagent}`.

There are two steps to write your own routing module:

1. You need to declare the C++ part of your routing module (see ~*ns*/rtmodule.{cc,h}). For many modules this only means to declare a virtual method `name()` which returns a string descriptor of the module. However, you are free to implement as much functionality as you like in C++; if necessary you may later move functionality from OTcl into C++ for better performance.
2. You need to look at the above interfaces implemented in the base routing module (see ~*ns*/tcl/lib/ns-rtmodule.tcl) and decide which one you'll inherit, which one you'll override, and put them in OTcl interfaces of your own module.

There are several derived routing module examples in ~*ns*/tcl/lib/ns-rtmodule.tcl, which may serve as templates for your modules.

Currently, there are six routing modules implemented in *ns*:

Module Name	Functionality
RtModule/Base	Interface to unicast routing protocols. Provide basic functionality to add/delete route and attach/detach agents.
RtModule/Mcast	Interface to multicast routing protocols. Its only purpose is establishes multicast classifiers. All other multicast functionalities are implemented as instprocs of Node. This should be converted in the future.
RtModule/Hier	Hierarchical routing. It's a wrapper for managing hierarchical classifiers and route installation. Can be combined with other routing protocols, e.g., ad hoc routing.
RtModule/Manual	Manual routing.
RtModule/VC	Uses virtual classifier instead of vanilla classifier.
RtModule/MPLS	Implements MPLS functionality. This is the only existing module that is completely self-contained and does not pollute the Node namespace.

Table 5.2: Available routing modules

5.5.2 Node Interface

To connect to the above interfaces of routing module, a node provides a similar set of interfaces:

- In order to know which module to register during creation, the Node class keeps a list of modules as a class variable. The default value of this list contains only the base routing module. The Node class provides the following two *procs* to manipulate this module list:

  `Node::enable-module`{name} If module `RtModule/[name]` exists, this proc puts [name] into the module list.

  `Node::disable-module`{name} If [name] is in the module list, remove it from the list.

  When a node is created, it goes through the module list of the Node class, creates all modules included in the list, and register these modules at the node.

  After a node is created, one may use the following instprocs to list modules registered at the node, or to get a handle of a module with a particular name:

  `Node::list-modules`{} Return a list of the handles (shadow objects) of all registered modules.

  `Node::get-module`{name} Return a handle of the registered module whose name matches the given one. Notice that any routing module can only have a single instance registered at any node.

- To allow routing modules register their interests of routing updates, a node object provide the following instprocs:

  `Node::route-notify`{module} Add `module` into route update notification list.

  `Node::unreg-route-notify`{module} Remove `module` from route update notification list.

  Similarly, the following instprocs provide hooks on the attachment of transport agents:

  `Node::port-notify`{module} Add `module` into agent attachment notification list.

  `Node::unreg-port-notify`{module} Remove `module` from agent attachment notification list.

  Notice that in all of these instprocs, parameter `module` should be a module handle instead of a module name.

- Node provides the following instprocs to manipulate its address and port classifiers:

  – `Node::insert-entry`{module, clsfr, hook} inserts classifier `clsfr` into the entry point of the node. It also associates the new classifier with `module` so that if this classifier is removed later, `module` will be unregistered. If `hook` is specified as a number, the existing classifier will be inserted into slot `hook` of the new classifier. In this way, one may establish a "chain" of classifiers; see Figure **??** for an example. **NOTE**: `clsfr` needs NOT

to be a classifier. In some cases one may want to put an agent, or any class derived from Connector, at the entry point of a node. In such cases, one simply supplies `target` to parameter `hook`.

– `Node::install-entry`{module, clsfr, hook} differs from `Node::insert-entry` in that it deletes the existing classifier at the node entry point, unregisters any associated routing module, and installs the new classifier at that point. If `hook` is given, and the old classifier is connected into a classifier chain, it will connect the chain into slot `hook` of the new classifier. As above, if `hook` equals to `target`, `clsfr` will be treated as an object derived from Connector instead of a classifier.

– `Node::install-demux`{demux, port} places the given classifier `demux` as the default demultiplexer. If `port` is given, it plugs the existing demultiplexer into slot `port` of the new one. Notice that in either case it does not delete the existing demultiplexer.

5.6 Commands at a glance

Following is a list of common node commands used in simulation scripts:

`$ns_ node [<hier_addr>]`
Command to create and return a node instance. If <hier_addr> is given, assign the node address to be <hier_addr>. Note that the latter MUST only be used when hierarchical addressing is enabled via either `set-address-format hierarchical`{} or `node-config -addressType hierarchical`{}.

`$ns_ node-config -<config-parameter> <optional-val>`
This command is used to configure nodes. The different config-parameters are addressingType, different type of the network stack components, whether tracing will be turned on or not, mobileIP flag is truned or not, energy model is being used or not etc. An option -reset maybe used to set the node configuration to its default state. The default setting of node-config, i.e if no values are specified, creates a simple node (base class Node) with flat addressing/routing. For the syntax details see Section **??**.

`$node id`
Returns the id number of the node.

`$node node-addr`
Returns the address of the node. In case of flat addressing, the node address is same as its node-id. In case of hierarchical addressing, the node address in the form of a string (viz. "1.4.3") is returned.

`$node reset`
Resets all agent attached to this node.

`$node agent <port_num>`
Returns the handle of the agent at the specified port. If no agent is found at the given port, a null string is returned.

`$node entry`
Returns the entry point for the node. This is first object that handles packet receiving at this node.

`$node attach <agent> <optional:port_num>`
Attaches the <agent> to this node. Incase no specific port number is passed, the node allocates a port number and binds the agent to this port. Thus once the agent is attached, it receives packets destined for this host (node) and port.

`$node detach <agent> <null_agent>`
This is the dual of "attach" described above. It detaches the agent from this node and installs a null-agent to the port this agent was attached. This is done to handle transit packets that may be destined to the detached agent. These on-the-fly packets are then sinked at the null-agent.

`$node neighbors`
This returns the list of neighbors for the node.

`$node add-neighbor <neighbor_node>`
This is a command to add `<neighbor_node>` to the list of neighbors maintained by the node.

Following is a list of internal node methods:

`$node add-route <destination_id> <target>`
This is used in unicast routing to populate the classifier. The target is a Tcl object, which may be the entry of `dmux_` (port demultiplexer in the node) incase the `<destination_id>` is same as this node-id. Otherwise it is usually the head of the link for that destination. It could also be the entry for other classifiers.

`$node alloc-port <null_agent>`
This returns the next available port number.

`$node incr-rtgtable-size`
The instance variable `rtsize_` is used to keep track of size of routing-table in each node. This command is used to increase the routing-table size every time an routing-entry is added to the classifiers.

There are other node commands that supports hierarchical routing, detailed dynamic routing, equal cost multipath routing, manual routing, and energy model for mobile nodes. These and other methods described earlier can be found in ~*ns*/tcl/lib/ns-node.tcl and ~*ns*/tcl/lib/ns-mobilenode.tcl.

Chapter 6

Links: Simple Links

This is the second aspect of defining the topology. In the previous chapter (Chapter **??**), we had described how to create the nodes in the topology in *ns*. We now describe how to create the links to connect the nodes and complete the topology. In this chapter, we restrict ourselves to describing the simple point to point links. *ns* supports a variety of other media, including an emulation of a multi-access LAN using a mesh of simple links, and other true simulation of wireless and broadcast media. They will be described in a separate chapter. The CBQlink is derived from simple links and is a considerably more complex form of link that is also not described in this chapter.

We begin by describing the commands to create a link in this section. As with the node being composed of classifiers, a simple link is built up from a sequence of connectors. We also briefly describe some of the connectors in a simple link. We then describe the instance procedures that operate on the various components of defined by some of these connectors (Section **??**). We conclude the chapter with a description the connector object (Section **??**), including brief descriptions of the common link connectors.

The `class Link` is a standalone class in OTcl, that provides a few simple primitives. The `class SimpleLink` provides the ability to connect two nodes with a point to point link. *ns* provides the instance procedure `simplex-link{}` to form a unidirectional link from one node to another. The link is in the class SimpleLink. The following describes the syntax of the simplex link:

```
set ns [new Simulator]
$ns simplex-link ⟨node0⟩ ⟨node1⟩ ⟨bandwidth⟩ ⟨delay⟩ ⟨queue_type⟩
```

The command creates a link from ⟨`node0`⟩ to ⟨`node1`⟩, with specified ⟨`bandwidth`⟩ and ⟨`delay`⟩ characteristics. The link uses a queue of type ⟨`queue_type`⟩. The procedure also adds a TTL checker to the link. Five instance variables define the link:

`head_`	Entry point to the link, it points to the first object in the link.
`queue_`	Reference to the main queue element of the link. Simple links usually have one queue per link. Other more complex types of links may have multiple queue elements in the link.
`link_`	A reference to the element that actually models the link, in terms of the delay and bandwidth characteristics of the link.
`ttl_`	Reference to the element that manipulates the ttl in every packet.
`drophead_`	Reference to an object that is the head of a queue of elements that process link drops.

In addition, if the simulator instance variable, `$traceAllFile_`, is defined, the procedure will add trace elements that

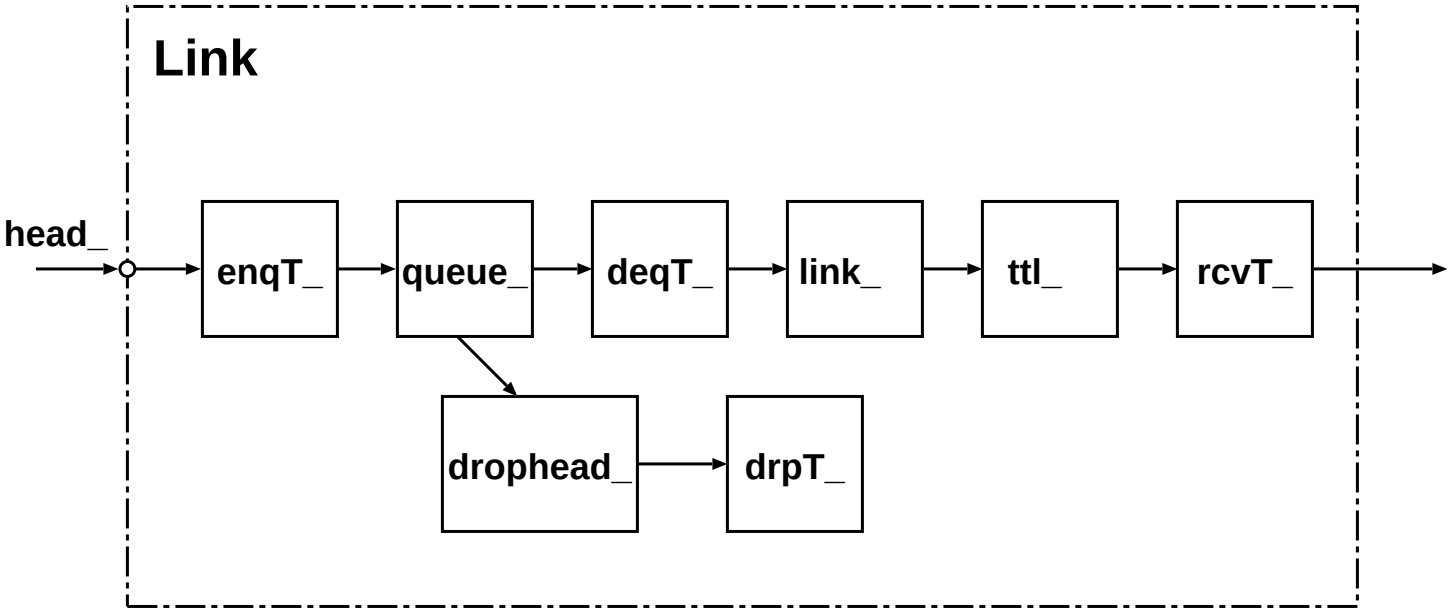

Figure 6.1: Composite Construction of a Unidirectional Link

track when a packet is enqueued and dequeued from `queue_`. Furthermore, tracing interposes a drop trace element after the `drophead_`. The following instance variables track the trace elements:

`enqT_`	Reference to the element that traces packets entering `queue_`.
`deqT_`	Reference to the element that traces packets leaving `queue_`.
`drpT_`	Reference to the element that traces packets dropped from `queue_`.
`rcvT_`	Reference to the element that traces packets received by the next node.

Note however, that if the user enable tracing multiple times on the link, these instance variables will only store a reference to the last elements inserted.

Other configuration mechanisms that add components to a simple link are network interfaces (used in multicast routing), link dynamics models, and tracing and monitors. We give a brief overview of the related objects at the end of this chapter (Section **??**), and discuss their functionality/implementation in other chapters.

The instance procedure `duplex-link{}` constructs a bi-directional link from two simplex links.

6.1 Instance Procedures for Links and SimpleLinks

Link procedures The `class Link` is implemented entirely in Otcl. The OTcl `SimpleLink` class uses the C++ `LinkDelay` class to simulate packet delivery delays. The instance procedures in the class Link are:

`head{}`	returns the handle for `head_`.
`queue{}`	returns the handle for `queue_`.
`link{}`	returns the handle for the delay element, `link_`.
`up{}`	set link status to "up" in the `dynamics_` element. Also, writes out a trace line to each file specified through the procedure `trace-dynamics{}`.
`down{}`	As with `up{}`, set link status to "down" in the `dynamics_` element. Also, writes out a trace line to each file specified through the procedure `trace-dynamics{}`.
`up?{}`	returns status of the link. Status is "up" or "down"; status is "up" if link dynamics is not enabled.
`all-connectors{}`	Apply specified operation to all connectors on the link.p An example of such usage is `link all-connectors reset`.
`cost{}`	set link cost to value specified.
`cost?{}`	returns the cost of the link. Default cost of link is 1, if no cost has been specified earlier.

SimpleLink Procedures The Otcl `class SimpleLink` implements a simple point-to-point link with an associated queue and delay[1]. It is derived from the base Otcl class Link as follows:

```
Class SimpleLink -superclass Link
SimpleLink instproc init { src dst bw delay q { lltype "DelayLink" } } {
        $self next $src $dst
        $self instvar link_ queue_ head_ toNode_ ttl_
        ...
        set queue_ $q
        set link_ [new Delay/Link]
        $link_ set bandwidth_ $bw
        $link_ set delay_ $delay

        $queue_ target $link_
        $link_ target [$toNode_ entry]

        ...
        # XXX
        # put the ttl checker after the delay
        # so we don't have to worry about accounting
        # for ttl-drops within the trace and/or monitor
        # fabric
        #
        set ttl_ [new TTLChecker]
        $ttl_ target [$link_ target]
        $link_ target $ttl_
}
```

Notice that when a `SimpleLink` object is created, new `Delay/Link` and `TTLChecker` objects are also created. Note also that, the `Queue` object must have already been created.

There are two additional methods implemented (in OTcl) as part of the `SimpleLink` class: `trace` and `init-monitor`. These functions are described in further detail in the section on tracing (Chapter **??**).

[1]The current version also includes an object to examine the network layer "ttl" field and discard packets if the field reaches zero.

6.2 Connectors

Connectors, unlink classifiers, only generate data for one recipient; either the packet is delivered to the `target_` neighbor, or it is sent to he `drop-target_`.

A connector will receive a packet, perform some function, and deliver the packet to its neighbor, or drop the packet. There are a number of different types of connectors in *ns.* Each connector performs a different function.

networkinterface	labels packets with incoming interface identifier—it is used by some multicast routing protocols. The class variable "Simulator NumberInterfaces_ 1" tells *ns* to add these interfaces, and then, it is added to either end of the simplex link. Multicast routing protocols are discussed in a separate chapter (Chapter **??**).
DynaLink	Object that gates traffic depending on whether the link is up or down. It expects to be at the head of the link, and is inserted on the link just prior to simulation start. It's `status_` variable control whether the link is up or down. The description of how the DynaLink object is used is in a separate chapter (Chapter **??**).
DelayLink	Object that models the link's delay and bandwidth characteristics. If the link is not dynamic, then this object simply schedules receive events for the downstream object for each packet it receives at the appropriate time for that packet. However, if the link is dynamic, then it queues the packets internally, and schedules one receives event for itself for the next packet that must be delivered. Thus, if the link goes down at some point, this object's `reset`() method is invoked, and the object will drop all packets in transit at the instant of link failure. We discuss the specifics of this class in another chapter (Chapter **??**).
Queues	model the output buffers attached to a link in a "real" router in a network. In *ns*, they are attached to, and are considered as part of the link. We discuss the details of queues and different types of queues in *ns*in another chapter (Chapter **??**).
TTLChecker	will decrement the ttl in each packet that it receives. If that ttl then has a positive value, the packet is forwarded to the next element on the link. In the simple links, TTLCheckers are automatically added, and are placed as the last element on the link, between the delay element, and the entry for the next node.

6.3 Object hierarchy

The base class used to represent links is called Link. Methods for this class are listed in the next section. Other link objects derived from the base class are given as follows:

- SimpleLink Object A SimpleLink object is used to represent a simple unidirectional link. There are no state variables or configuration parameters associated with this object. Methods for this class are: `$simplelink enable-mcast <src> <dst>`
  This turns on multicast for the link by creating an incoming network interface for the destination and adds an outgoing interface for the source.

  `$simplelink trace <ns> <file> <optional:op>`
  Build trace objects for this link and update object linkage. If op is specified as "nam" create nam trace files.

  `$simplelink nam-trace <ns> <file>`
  Sets up nam tracing in the link.

  `$simplelink trace-dynamics <ns> <file> <optional:op>`
  This sets up tracing specially for dynamic links. <op> allows setting up of nam tracing as well.

`$simplelink init-monitor <ns> <qtrace> <sampleInterval>`
Insert objects that allow us to monitor the queue size of this link. Return the name of the object that can be queried to determine the average queue size.

`$simplelink attach-monitors <insnoop> <outsnoop> <dropsnoop> <qmon>`
This is similar to init-monitor, but allows for specification of more of the items.

`$simplelink dynamic`
Sets up the dynamic flag for this link.

`$simplelink errormodule <args>`
Inserts an error module before the queue.

`$simpleilnk insert-linkloss <args>`
Inserts the error module after the queue.

//Other link objects derived from class SimpleLink are FQLink, CBQLink and IntServLink.

Configuration parameters for FQLink are:

queueManagement_ The type of queue management used in the link. Default value is DropTail.

No configuration parameters are specified for CBQLink and IntServLink objects.

- DelayLink Object The DelayLink Objects determine the amount of time required for a packet to traverse a link. This is defined to be size/bw + delay where size is the packet size, bw is the link bandwidth and delay is the link propagation delay. There are no methods or state variables associated with this object.

  Configuration Parameters are:

  bandwidth_ Link bandwidth in bits per second.

  delay_ Link propagation delay in seconds.

6.4 Commands at a glance

Following is a list of common link commands used in simulation scripts:

`$ns_ simplex-link <node1> <node2> <bw> <delay> <qtype> <args>`
This command creates an unidirectional link between node1 and node2 with specified bandwidth (BW) and delay characteristics. The link uses a queue type of <qtype> and depending on the queue type different arguments are passed through <args>.

`$ns_ duplex-link <node1> <node2> <bw> <delay> <qtype> <args>`
This creates a bi-directional link between node1 and node2. This procedure essentially creates a duplex-link from two simplex links, one from node1 to node2 and the other from node2 to node1. The syntax for duplex-link is same as that of simplex-link described above.

`$ns_ duplex-intserv-link <n1> <n2> <bw> <dly> <sched> <signal> <adc> <args>`
This creates a duplex-link between n1 and n2 with queue type of intserv, with specified BW and delay. This type of queue implements a scheduler with two level services priority. The type of intserv queue is given by <sched>, with admission control unit type of <adc> and signal module of type <signal>.

`$ns_ simplex-link-op <n1> <n2> <op> <args>`
This is used to set attributes for a simplex link. The attributes may be the orientation, color, label, or queue-position.

`$ns_ duplex-link-op <n1> <n2> <op> <args>`
This command is used to set link attributes (like orientation of the links, color, label, or queue-position) for duplex links.

`$ns_ link-lossmodel <lossobj> <from> <to>`
This function generates losses (using the loss model <lossobj> inserted in the link between <from> node and <to> node) in the link that can be visualized by nam.

`$ns_ lossmodel <lossobj> <from> <to>`
This is used to insert a loss module in regular links.

Following is a list of internal link-related procedures:

`$ns_ register-nam-linkconfig <link>`
This is an internal procedure used by `"$link orient"` to register/update the order in which links should be created in nam.

`$ns_ remove-nam-linkconfig <id1> <id2>`
This procedure is used to remove any duplicate links (duplicate links may be created by GT-ITM topology generator).

`$link head`
Returns the instance variable `head_` for the link. The `head_` is the entry pont to the link and it points to the first object in the link.

`$link add-to-head <connector>`
This allows the <connector> object to be now pointed by the `head_` element in the link, i.e, <connector> now becomes the first object in the link.

`$link link`
Returns the instance variable `link_`. The `link_` is the element in the link that actually models the link in terms of delay and bandwidth characteristics of the link.

`$link queue`
Returns the instance variable `queue_`. `queue_` is queue element in the link. There may be one or more queue elements in a particular link.

`$link cost <c>`
This sets a link cost of <c>.

`$link cost?`
Returns the cost value for the link. Default cost of link is set to 1.

`$link if-label?`
Returns the network interfaces associated with the link (for multicast routing).

`$link up`
This sets the link status to "up". This command is a part of network dynamics support in *ns*.

`$link down`
Similar to up, this command marks the link status as "down".

`$link up?`
Returns the link status. The status is always "up" as default, if link dynamics is not enabled.

`$link all-connectors op`
This command applies the specified operation <op> to all connectors in the link. Like, `$link all-connectors reset` or `$link all-connectors isDynamic`.

`$link install-error <errmodel>`
This installs an error module after the `link_` element.

In addition to the Link and link-related commands listed above, there are other procedures to support the specific requirements of different types of links derived from the base class "Link" like simple-link (SimpleLink), integrated service (IntServLink), class-based queue (CBQLink), fair queue (FQLink) and procedures to support multicast routing, sessionsim, nam etc. These and the above procedures may be found in *ns*/tcl/lib(ns-lib.tcl, ns-link.tcl, ns-intserv.tcl, ns-namsupp.tcl, ns-queue.tcl), *ns*/tcl/mcast/(McastMonitor.tcl, ns-mcast.tcl), *ns*/tcl/session/session.tcl.

Chapter 7

Queue Management and Packet Scheduling

Queues represent locations where packets may be held (or dropped). Packet scheduling refers to the decision process used to choose which packets should be serviced or dropped. Buffer management refers to any particular discipline used to regulate the occupancy of a particular queue. At present, support is included for drop-tail (FIFO) queueing, RED buffer management, CBQ (including a priority and round-robin scheduler), and variants of Fair Queueing including, Fair Queueing (FQ), Stochastic Fair Queueing (SFQ), and Deficit Round-Robin (DRR). In the common case where a *delay* element is downstream from a queue, the queue may be *blocked* until it is re-enabled by its downstream neighbor. This is the mechanism by which transmission delay is simulated. In addition, queues may be forcibly blocked or unblocked at arbitrary times by their neighbors (which is used to implement multi-queue aggregate queues with inter-queue flow control). Packet drops are implemented in such a way that queues contain a "drop destination"; that is, an object that receives all packets dropped by a queue. This can be useful to (for example) keep statistics on dropped packets.

7.1 The C++ Queue Class

The `Queue` class is derived from a `Connector` base class. It provides a base class used by particular types of (derived) queue classes, as well as a call-back function to implement blocking (see next section). The following definitions are provided in `queue.h`:

```
class Queue : public Connector {
 public:
        virtual void enque(Packet*) = 0;
        virtual Packet* deque() = 0;
        void recv(Packet*, Handler*);
        void resume();
        int blocked();
        void unblock();
        void block();
 protected:
        Queue();
        int command(int argc, const char*const* argv);
        int qlim_;                              /* maximum allowed pkts in queue */
        int blocked_;
        int unblock_on_resume_;                 /* unblock q on idle? */
        QueueHandler qh_;
```

```
};
```

The `enque` and `deque` functions are pure virtual, indicating the `Queue` class is to be used as a base class; particular queues are derived from `Queue` and implement these two functions as necessary. Particular queues do not, in general, override the `recv` function because it invokes the the particular `enque` and `deque`.

The `Queue` class does not contain much internal state. Often these are special monitoring objects (Chapter **??**). The `qlim_` member is constructed to dictate a bound on the maximum queue occupancy, but this is not enforced by the `Queue` class itself; it must be used by the particular queue subclasses if they need this value. The `blocked_` member is a boolean indicating whether the queue is able to send a packet immediately to its downstream neighbor. When a queue is blocked, it is able to enqueue packets but not send them.

7.1.1 Queue blocking

A queue may be either blocked or unblocked at any given time. Generally, a queue is blocked when a packet is in transit between it and its downstream neighbor (most of the time if the queue is occupied). A blocked queue will remain blocked as long as it downstream link is busy and the queue has at least one packet to send. A queue becomes unblocked only when its `resume` function is invoked (by means of a downstream neighbor scheduling it via a callback), usually when no packets are queued. The callback is implemented by using the following class and methods:

```
class QueueHandler : public Handler {
public:
        inline QueueHandler(Queue& q) : queue_(q) {}
        void handle(Event*); /* calls queue_.resume() */
 private:
        Queue& queue_;
};
void QueueHandler::handle(Event*)
{
        queue_.resume();
}

Queue::Queue() : drop_(0), blocked_(0), qh_(*this)
{
        Tcl& tcl = Tcl::instance();
        bind("limit_", &qlim_);
}
void Queue::recv(Packet* p, Handler*)
{
        enque(p);
        if (!blocked_) {
                /*
                 * We're not block.  Get a packet and send it on.
                 * We perform an extra check because the queue
                 * might drop the packet even if it was
                 * previously empty!  (e.g., RED can do this.)
                 */
                p = deque();
                if (p != 0) {
                        blocked_ = 1;
                        target_->recv(p, &qh_);
```

```
                }
        }
}
void Queue::resume()
{
        Packet* p = deque();
        if (p != 0)
                target_->recv(p, &qh_);
        else {
                if (unblock_on_resume_)
                        blocked_ = 0;
                else
                        blocked_ = 1;
        }
}
```

The handler management here is somewhat subtle. When a new `Queue` object is created, it includes a `QueueHandler` object (`qh_`) which is initialized to contain a reference to the new `Queue` object (`Queue& QueueHandler::queue_`). This is performed by the `Queue` constructor using the expression `qh_(*this)`. When a Queue receives a packet it calls the subclass (i.e. queueing discipline-specific) version of the `enque` function with the packet. If the queue is not blocked, it is allowed to send a packet and calls the specific `deque` function which determines which packet to send, blocks the queue (because a packet is now in transit), and sends the packet to the queue's downstream neighbor. Note that any future packets received from upstream neighbors will arrive to a blocked queue. When a downstream neighbor wishes to cause the queue to become unblocked it schedules the QueueHandler's `handle` function by passing `&qh_` to the simulator scheduler. The `handle` function invokes `resume`, which will send the next-scheduled packet downstream (and leave the queue blocked), or unblock the queue when no packet is ready to be sent. This process is made more clear by also referring to the `LinkDelay::recv`() method (Section **??**).

7.1.2 PacketQueue Class

The `Queue` class may implement buffer management and scheduling but do not implement the low-level operations on a particular queue. The `PacketQueue` class is used for this purpose, and is defined as follows (see `queue.h`):

```
class PacketQueue {
public:
        PacketQueue();
        int length(); /* queue length in packets */
        void enque(Packet* p);
        Packet* deque();
        Packet* lookup(int n);
        /* remove a specific packet, which must be in the queue */
        void remove(Packet*);
protected:
        Packet* head_;
        Packet** tail_;
        int len_;               // packet count
};
```

This class maintains a linked-list of packets, and is commonly used by particular scheduling and buffer management disciplines to hold an ordered set of packets. Particular scheduling or buffer management schemes may make use of several

PacketQueue objects. The PacketQueue class maintains current counts of the number of packets held in the queue which is returned by the length() method. The enque function places the specified packet at the end of the queue and updates the len_ member variable. The deque function returns the packet at the head of the queue and removes it from the queue (and updates the counters), or returns NULL if the queue is empty. The lookup function returns the nth packet from the head of the queue, or NULL otherwise. The remove function deletes the packet stored in the given address from the queue (and updates the counters). It causes an abnormal program termination if the packet does not exist.

7.2 Example: Drop Tail

The following example illustrates the implementation of the Queue/DropTail object, which implements FIFO scheduling and drop-on-overflow buffer management typical of most present-day Internet routers. The following definitions declare the class and its OTcl linkage:

```
/*
 *  A bounded, drop-tail queue
 */
class DropTail : public Queue {
 protected:
        void enque(Packet*);
        Packet* deque();
        PacketQueue q_;
};
```

The base class Queue, from which DropTail is derived, provides most of the needed functionality. The drop-tail queue maintains exactly one FIFO queue, implemented by including an object of the PacketQueue class. Drop-tail implements its own versions of enque and deque as follows:

```
/*
 *  drop-tail
 */
void DropTail::enque(Packet* p)
{
        q_.enque(p);
        if (q_.length() >= qlim_) {
                q_.remove(p);
                drop(p);
        }
}

Packet* DropTail::deque()
{
        return (q_.deque());
}
```

Here, the enque function first stores the packet in the internal packet queue (which has no size restrictions), and then checks the size of the packet queue versus qlim_. Drop-on-overflow is implemented by dropping the packet most recently added to the packet queue if the limit is reached or exceeded. *Note:* in the implementation of enque above, setting qlim_ to n actually means a queue size of n-1. Simple FIFO scheduling is implemented in the deque function by always returning the first packet in the packet queue.

7.3 Different types of Queue objects

A queue object is a general class of object capable of holding and possibly marking or discarding packets as they travel through the simulated topology. Configuration Parameters used for queue objects are:

limit_ The queue size in packets.

blocked_ Set to false by default, this is true if the queue is blocked (unable to send a packet to its downstream neighbor).

unblock_on_resume_ Set to true by default, indicates a queue should unblock itself at the time the last packet packet sent has been transmitted (but not necessarily received).

Other queue objects derived from the base class Queue are drop-tail, FQ, SFQ, DRR, RED, CBQ, CoDel, and SFQ-CoDel queue objects. Each are described as follows:

- Drop-tail objects: Drop-tail objects are a subclass of Queue objects that implement simple FIFO queue. There are no methods, configuration parameter, or state variables that are specific to drop-tail objects.
- FQ objects: FQ objects are a subclass of Queue objects that implement Fair queuing. There are no methods that are specific to FQ objects. Configuration Parameters are:

  secsPerByte_

  There are no state variables associated with this object.
- SFQ objects: SFQ objects are a subclass of Queue objects that implement Stochastic Fair queuing. There are no methods that are specific to SFQ objects. Configuration Parameters are:

  maxqueue_

  buckets_

  There are no state variables associated with this object.
- DRR objects: DRR objects are a subclass of Queue objects that implement deficit round robin scheduling. These objects implement deficit round robin scheduling amongst different flows (A particular flow is one which has packets with the same node and port id OR packets which have the same node id alone). Also unlike other multi-queue objects, this queue object implements a single shared buffer space for its different flows. Configuration Parameters are:

  buckets_ Indicates the total number of buckets to be used for hashing each of the flows.

  blimit_ Indicates the shared buffer size in bytes.

  quantum_ Indicates (in bytes) how much each flow can send during its turn.

  mask_ mask_, when set to 1, means that a particular flow consists of packets having the same node id (and possibly different port ids), otherwise a flow consists of packets having the same node and port ids.
- RED objects: RED objects are a subclass of Queue objects that implement random early-detection gateways. The object can be configured to either drop or “mark” packets. There are no methods that are specific to RED objects. Configuration Parameters are:

  bytes_ Set to "true" to enable “byte-mode” RED, where the size of arriving packets affect the likelihood of marking (dropping) packets.

  queue-in-bytes_ Set to "true" to measure the average queue size in bytes rather than packets. Enabling this option also causes thresh_ and maxthresh_ to be automatically scaled by mean_pktsize_ (see below).

  thresh_ The minimum threshold for the average queue size in packets.

maxthresh_ The maximum threshold for the average queue size in packets.

mean_pktsize_ A rough estimate of the average packet size in bytes. Used in updating the calculated average queue size after an idle period.

q_weight_ The queue weight, used in the exponential-weighted moving average for calculating the average queue size.

wait_ Set to true to maintain an interval between dropped packets.

linterm_ As the average queue size varies between "thresh_" and "maxthresh_", the packet dropping probability varies between 0 and "1/linterm".

setbit_ Set to "true" to mark packets by setting the congestion indication bit in packet headers rather than drop packets.

drop-tail_ Set to true to use drop-tail rather than randomdrop when the queue overflows or the average queue size exceeds "maxthresh_". For a further explanation of these variables, see [2].

None of the state variables of the RED implementation are accessible.

- CBQ objects: CBQ objects are a subclass of Queue objects that implement class-based queueing.

  `$cbq insert <class>`
  Insert traffic class class into the link-sharing structure associated with link object cbq.

  `$cbq bind <cbqclass> <id1> [$id2]`
  Cause packets containing flow id id1 (or those in the range id1 to id2 inclusive) to be associated with the traffic class cbqclass.

  `$cbq algorithm <alg>`
  Select the CBQ internal algorithm. <alg> may be set to one of: "ancestor-only", "top-level", or "formal".

- CBQ/WRR objects: CBQ/WRR objects are a subclass of CBQ objects that implement weighted round-robin scheduling among classes of the same priority level. In contrast, CBQ objects implement packet-by-packet round-robin scheduling among classes of the same priority level. Configuration Parameters are:

  maxpkt_ The maximum size of a packet in bytes. This is used only by CBQ/WRR objects in computing maximum bandwidth allocations for the weighted round-robin scheduler.

- CoDel objects: CoDel objects are a subclass of Queue objects that implement the Controlled Delay (CoDel) active queue manager. Configuration Parameters are:

  interval_ The CoDel measurement interval. This is typically set to a value that is on the order of the worst-case RTT of connections utilizing the queue.

  target_ The CoDel latency target. This is an upper bound on acceptable standing queue delay.

- SFQ-CoDel objects: SFQ-CoDel objects are a subclass of Queue objects that implement the Stochastic Flow Queuing - Controlled Delay queue manager. Configuration Parameters are:

  interval_ The CoDel measurement interval. This is typically set to a value that is on the order of the worst-case RTT of connections utilizing the queue.

  target_ The CoDel latency target. This is an upper bound on acceptable standing queue delay.

  maxbins_ The number of SFQ "bins" implemented by the SFQ-CoDel queue.

  quantum_ The deficit-round-robin quantum used for dequeuing packets from the SFQ structure.

CBQCLASS OBJECTS

CBQClass objects implement the traffic classes associated with CBQ objects.

`$cbqclass setparams <parent> <okborrow> <allot> <maxidle> <prio> <level>`
Sets several of the configuration parameters for the CBQ traffic class (see below).

`$cbqclass parent <cbqcl|none>`
specify the parent of this class in the link-sharing tree. The parent may be specified as “none” to indicate this class is a root.

`$cbqclass newallot <a>`
Change the link allocation of this class to the specified amount (in range 0.0 to 1.0). Note that only the specified class is affected.

`$cbqclass install-queue <q>`
Install a Queue object into the compound CBQ or CBQ/WRR link structure. When a CBQ object is initially created, it includes no internal queue (only a packet classifier and scheduler).

Configuration Parameters are:

okborrow_ is a boolean indicating the class is permitted to borrow bandwidth from its parent.

allot_ is the maximum fraction of link bandwidth allocated to the class expressed as a real number between 0.0 and 1.0.

maxidle_ is the maximum amount of time a class may be required to have its packets queued before they are permitted to be forwarded

priority_ is the class' priority level with respect to other classes. This value may range from 0 to 10, and more than one class may exist at the same priority. Priority 0 is the highest priority.

level_ is the level of this class in the link-sharing tree. Leaf nodes in the tree are considered to be at level 1; their parents are at level 2, etc.

extradelay_ increase the delay experienced by a delayed class by the specified time

QUEUE-MONITOR OBJECTS
QueueMonitor Objects are used to monitor a set of packet and byte arrival, departure and drop counters. It also includes support for aggregate statistics such as average queue size, etc.

`$queuemonitor`
reset all the cumulative counters described below (arrivals, departures, and drops) to zero. Also, reset the integrators and delay sampler, if defined.

`$queuemonitor set-delay-samples <delaySamp_>`
Set up the Samples object delaySamp_ to record statistics about queue delays. delaySamp_ is a handle to a Samples object i.e the Samples object should have already been created.

`$queuemonitor get-bytes-integrator`
Returns an Integrator object that can be used to find the integral of the queue size in bytes.

`$queuemonitor get-pkts-integrator`
Returns an Integrator object that can be used to find the integral of the queue size in packets.

`$queuemonitor get-delay-samples`
Returns a Samples object delaySamp_ to record statistics about queue delays.
There are no configuration parameters specific to this object.
State Variables are:

size_ Instantaneous queue size in bytes.

pkts_ Instantaneous queue size in packets.

parrivals_ Running total of packets that have arrived.

barrivals_ Running total of bytes contained in packets that have arrived.

pdepartures_ Running total of packets that have departed (not dropped).

bdepartures_ Running total of bytes contained in packets that have departed (not dropped).

pdrops_ Total number of packets dropped.

bdrops_ Total number of bytes dropped.

bytesInt_ Integrator object that computes the integral of the queue size in bytes. The sum_ variable of this object has the running sum (integral) of the queue size in bytes.

pktsInt_ Integrator object that computes the integral of the queue size in packets. The sum_ variable of this object has the running sum (integral) of the queue size in packets.

QUEUEMONITOR/ED OBJECTS

This derived object is capable of differentiating regular packet drops from early drops. Some queues distinguish regular drops (e.g. drops due to buffer exhaustion) from other drops (e.g. random drops in RED queues). Under some circumstances, it is useful to distinguish these two types of drops.
State Variables are:

epdrops_ The number of packets that have been dropped "early".

ebdrops_ The number of bytes comprising packets that have been dropped "early".

Note: because this class is a subclass of QueueMonitor, objects of this type also have fields such as pdrops_ and bdrops_. These fields describe the total number of dropped packets and bytes, including both early and non-early drops.

QUEUEMONITOR/ED/FLOWMON OBJECTS

These objects may be used in the place of a conventional QueueMonitor object when wishing to collect per-flow counts and statistics in addition to the aggregate counts and statistics provided by the basic QueueMonitor.

`$fmon classifier <cl>`

This inserts (read) the specified classifier into (from) the flow monitor object. This is used to map incoming packets to which flows they are associated with.

`$fmon dump`

Dump the current per-flow counters and statistics to the I/O channel specified in a previous attach operation.

`$fmon flows`

Return a character string containing the names of all flow objects known by this flow monitor. Each of these objects are of type QueueMonitor/ED/Flow.

`$fmon attach <chan>`

Attach a tcl I/O channel to the flow monitor. Flow statistics are written to the channel when the dump operation is executed.

Configuration Parameters are:

enable_in_ Set to true by default, indicates that per-flow arrival state should be kept by the flow monitor. If set to false, only the aggregate arrival information is kept.

enable_out_ Set to true by default, indicates that per-flow departure state should be kept by the flow monitor. If set to false, only the aggregate departure information is kept.

enable_drop_ Set to true by default, indicates that per-flow drop state should be kept by the flow monitor. If set to false, only the aggregate drop information is kept.

enable_edrop_ Set to true by default, indicates that per-flow early drop state should be kept by the flow monitor. If set to false, only the aggregate early drop information is kept.

QUEUEMONITOR/ED/FLOW OBJECTS
These objects contain per-flow counts and statistics managed by a QueueMonitor/ED/Flowmon object. They are generally created in an OTcl callback procedure when a flow monitor is given a packet it cannot map on to a known flow. Note that the flow monitor's classifier is responsible for mapping packets to flows in some arbitrary way. Thus, depending on the type of classifier used, not all of the state variables may be relevant (e.g. one may classify packets based only on flow id, in which case the source and destination addresses may not be significant). State Variables are:

src_ The source address of packets belonging to this flow.

dst_ The destination address of packets belonging to this flow.

flowid_ The flow id of packets belonging to this flow.

7.4 Commands at a glance

Following is a list of queue commands used in simulation scripts:

`$ns_ queue-limit <n1> <n2> <limit>`
This sets a limit on the maximum buffer size of the queue in the link between nodes <n1> and <n2>.

`$ns_ trace-queue <n1> <n2> <optional:file>`
This sets up trace objects to log events in the queue. If tracefile is not passed, it uses `traceAllFile_` to write the events.

`$ns_ namtrace-queue <n1> <n2> <optional:file>`
Similar to trace-queue above, this sets up nam-tracing in the queue.

`$ns_ monitor-queue <n1> <n2> <optional:qtrace> <optional:sampleinterval>`
This command inserts objects that allows us to monitor the queue size. This returns a handle to the object that may be queried to determine the average queue size. The default value for sampleinterval is 0.1.

7.5 Queue/JoBS

JoBS is developed and contributed by Nicolas Christin <nicolas@cs.virginia.edu>

This chapter describes the implementation of the Joint Buffer Management and Scheduling (JoBS) algorithm in *ns*. This chapter is in three parts. The first part summarizes the objectives of the JoBS algorithm. The second part explains how to configure a JoBS queue in *ns*. The third part focuses on the tracing mechanisms implemented for JoBS.

The procedures and functions described in this chapter can be found in *ns*/jobs.{cc, h}, *ns*/marker.{cc, h}, *ns*/demarker.{cc, h}. Example scripts can be found in *ns*/tcl/ex/jobs-{lossdel, cn2002}.tcl.

Additional information can be found at http://qosbox.cs.virginia.edu.

7.5.1 The JoBS algorithm

This section gives an overview of the objectives the JoBS algorithm aims at achieving, and of the mechanisms employed to reach these objectives. The original JoBS algorithm, as described in [**?**], was using the solution to a non-linear optimization problem. This *ns-2* implementation uses the feedback-control based heuristic described in [**?**].

Important Note: This *ns-2* implementation results from the merge between old code for *ns-2.1b5*, and code derived from the BSD kernel-level implementation of the JoBS algorithm. **It is still considered experimental.** Due to the absence of binding facilities for arrays between Tcl and C++ in *tclcl* at the moment, *the number of traffic classes is statically set to 4 and cannot be changed without modifying the C++ code.*

Objective

The objective of the JoBS algorithm is to provide absolute and relative (proportional) loss and delay differentiation independently at each node for *classes* of traffic. JoBS therefore provides service guarantees on a *per-hop* basis. The set of performance requirements are specified to the algorithm as a set of per-class Qualtiy of Service (QoS) constraints. As an example, for three classes, the QoS constraints could be of the form:

- Class-1 Delay $\approx 2 \cdot$ Class-2 Delay,
- Class-2 Loss Rate $\approx 10^{-1} \cdot$ Class-3 Loss Rate, or
- Class-3 Delay $\leq 5\ ms$.

Here, the first two constraints are relative constraints and the last one is an absolute constraint. The set of constraints can be any mix of relative and absolute constraints. More specifically, JoBS supports the five following types of constraints:

- **Relative delay constraints (RDC)** specify a proportional delay differentiation between classes. As an example, for two classes 1 and 2, the RDC enforces a relationship

$$\frac{\text{Delay of Class 2}}{\text{Delay of Class 1}} \approx \text{constant}\ .$$

- **Absolute delay constraints (ADC)**: An ADC on class i requires that the delays of class i satisfy a worst-case bound d_i.
- **Relative loss constraints (RLC)** specify a proportional loss differentiation between classes.
- **Absolute loss constraints (ALC)**: An ALC on class i requires that the loss rate of class i be bounded by an upper bound L_i.
- **Absolute rate constraints (ARC)**: An ARC on class i means that the throughput of class i is bounded by a lower bound μ_i.

JoBS does not rely on admission control or traffic policing, nor does it make any assumption on traffic arrivals. Therefore, a system of constraints may become infeasible, and some constraints may need to be relaxed. QoS constraints are prioritized in the following order.

$$\text{ALC} > \text{ADC, ARC} > \text{Relative Constraints}\ .$$

That is, if JoBS is unable to satisfy both absolute and relative constraints, it will give preference to the absolute constraints.

Mechanisms

JoBS performs scheduling and buffer management in a single pass. JoBS dynamically allocates service rates to classes in order to satisfy the delay constraints. The service rates needed for enforcing absolute delay constraints are allocated upon each packet arrival, while service rates derived from relative delay constraints are computed only every N packet arrivals. If no feasible service rate allocation exists[1], or if the packet buffer overflows, packets are dropped according to the loss constraints.

The service rates are translated into packet scheduling decisions by an algorithm resembling Deficit Round Robin. That is, the scheduler tries to achieve the desired service rates by keeping track of the difference between the actual transmission rate for each class and the desired service rate for each class. Scheduling in JoBS is work-conserving.

7.5.2 Configuration

Running a JoBS simulation requires to create and configure the JoBS "link(s)", to create and configure the Markers and Demarkers in charge of marking/demarking the traffic, to attach an application-level data source (traffic generator), and to start the traffic generator.

Initial Setup

```
set ns [new Simulator]                                  ;# preamble initialization

Queue/JoBS set drop_front_ false                        ;# use drop-tail
Queue/JoBS set trace_hop_ true                          ;# enable statistic traces
Queue/JoBS set adc_resolution_type_ 0                   ;# see ''commands at a glance''
Queue/JoBS set shared_buffer_ 1                         ;# all classes share a common buffer
Queue/JoBS set mean_pkt_size_ 4000                      ;# we expect to receive 500-Byte pkts
Queue/Demarker set demarker_arrvs1_ 0                   ;# reset arrivals everywhere
Queue/Demarker set demarker_arrvs2_ 0
Queue/Demarker set demarker_arrvs3_ 0
Queue/Demarker set demarker_arrvs4_ 0
Queue/Marker set marker_arrvs1_ 0
Queue/Marker set marker_arrvs2_ 0
Queue/Marker set marker_arrvs3_ 0
Queue/Marker set marker_arrvs4_ 0

set router(1) [$ns node]                                ;# set first router
set router(2) [$ns node]                                ;# set second router
set source [$ns node]                                   ;# set source
set sink [$ns node]                                     ;# set traffic sink
set bw 10000000                                         ;# 10 Mbps
set delay 0.001                                         ;# 1 ms
set buff 500                                            ;# 500 packets
```

Creating the JoBS links

```
$ns duplex-link $router(1) $router(2) $bw $delay JoBS   ;# Creates the JoBS link
```

[1] For instance, if the sum of the service rates needed is greater than the output link capacity.

```
$ns_ queue-limit $router(1) $router(2) $buff
set l [$ns_ get-link $router(1) $router(2)]
set q [$l queue]
$q init-rdcs -1 2 2 2     ;# Classes 2, 3 and 4 are bound by proportional delay differentiation with a factor of 2
$q init-rlcs -1 2 2 2     ;# Classes 2, 3 and 4 are bound by proportional loss differentiation with a factor of 2
$q init-alcs 0.01 -1 -1 -1                    ;# Class 1 is provided with a loss rate bound of 1%
$q init-adcs 0.005 -1 -1 -1                   ;# Class 1 is provided with a delay bound of 5 ms
$q init-arcs -1 -1 -1 500000          ;# Class 4 is provided with a minimumthroughput of 500 Kbps
$q link [$l link]                             ;# The link is attached to the queue (required)
$q trace-file jobstrace                ;# Trace per-hop, per-class metrics to the file jobstrace
$q sampling-period 1                          ;# Reevaluate rate allocation upon each arrival
$q id 1                                       ;# Assigns an ID of 1 to the JoBS queue
$q initialize                                 ;# Proceed with the initialization
```

Marking the traffic

Marking the traffic is handled by Marker objects. Markers are FIFO queues that set the class index of each packet. To ensure accuracy of the simulations, it is best to configure these queues to have a very large buffer, so that no packets are dropped in the Marker. Demarkers are used to gather end-to-end delay statistics.

```
$ns_ simplex-link $source $router(1) $bw $delay Marker          ;# set-up marker
$ns_ queue-limit $source $router(1) [expr $buff*10]      ;# Select huge buffers for markers
$ns_ queue-limit $router(1) $source [expr $buff*10]              ;# to avoid traffic drops
set q [$ns_ get-queue $source $router(1)]                                ;#in the marker
$q marker_type 2                                                  ;# Statistical marker
$q marker_frc 0.1 0.2 0.3 0.4          ;# 10% Class 1, 20% Class 2, 30% Class 3, 40% Class 4.
$ns_ simplex-link $router(2) $sink $bw $delay Demarker          ;# set-up demarker
$ns_ queue-limit $router(2) $sink [expr $buff*10]
$q trace-file e2e                                  ;# trace end-to-end delays to file e2e
```

The remaining steps (attaching agents and traffic generators or applications to the nodes) are explained in Chapters **??** and **??**, and are handled as usual. We refer to these chapters and the example scripts provided with your *ns* distribution.

7.5.3 Tracing

Tracing in JoBS is handled internally, by the scheduler. Each JoBS queue can generate a trace file containing the following information. Each line of the tracefile consists of 17 columns. The first column is the simulation time, columns 2 to 5 represent the loss rates over the current busy period for classes 1 to 4, columns 6 to 9 represent the delays for each class (average over a 0.5 seconds sliding window), columns 10 to 13 represent the average service rates allocated to each class over the last 0.5 seconds, and columns 14 to 17 represent the instantaneous queue length in packets. Additionally, Demarkers can be used to trace end-to-end delays.

7.5.4 Variables

This section summarizes the variables that are used by JoBS, Marker and Demarker objects.

JoBS objects

trace_hop_ Can be true or false. If set to true, per-hop, per-class metrics will be traced. (Trace files have then to be specified, using `<JoBS object> trace-file <filename>`.) Defaults to false.

drop_front_ Can be true or false. If set to true, traffic will be dropped from the front of the queue. Defaults to false (drop-tail).

adc_resolution_type_ Can be 0 or 1. If set to 0, traffic will be dropped from classes that have an ADC if the ADC cannot be met by adjusting the service rates. If set to 1, traffic will be dropped from all classes. A resolution mode set to 1 is therefore fairer, in the sense that the pain is shared by all classes, but can lead to more deadline violations. Defaults to 0.

shared_buffer_ Can be 0 or 1. If set to 0, all classes use a separate per-class buffer (which is required if only rate guarantees are to provided). All per-class buffers have the same size. If set to 1, all classes share the same buffer (which is required if loss differentiation is to be provided). Defaults to 1.

mean_pkt_size_ Used to set the expected mean packet size of packets arriving at a JoBS link. Setting this variable is required to ensure proper delay differentiation.

Marker objects

marker_arrvs1_ Number of Class-1 packets to have entered a Marker link.

marker_arrvs2_ Number of Class-2 packets to have entered a Marker link.

marker_arrvs3_ Number of Class-3 packets to have entered a Marker link.

marker_arrvs4_ Number of Class-4 packets to have entered a Marker link.

Demarker objects

demarker_arrvs1_ Number of Class-1 packets to have entered a Demarker link.

demarker_arrvs2_ Number of Class-2 packets to have entered a Demarker link.

demarker_arrvs3_ Number of Class-3 packets to have entered a Demarker link.

demarker_arrvs4_ Number of Class-4 packets to have entered a Demarker link.

7.5.5 Commands at a glance

The following is a list of commands used to configure the JoBS, Marker and Demarker objects.

JoBS objects

`set q [new Queue/JoBS]`
This creates an instance of the JoBS queue.

`$q init-rdcs <k1> <k2> <k3> <k4>`
This assigns the RDCs for the four JoBS classes. For instance, using a value of 4 for k2 means that Class-3 delays will be roughly equal to four times Class-2 delays. A value of -1 indicates that the class is not concerned by RDCs.

Important Note: Since RDCs bound two classes, one would expect only three parameters to be passed (k1, k2, and k3, since k4 theoretically binds Classes 4 and 5, and Class 5 does not exist). However, in this prototype implementation, it is imperative to specify a value different from 0 and -1 to k4 if Class 4 is to be concerned by RDCs.

Examples: `$q init-rdcs -1 2 1 -1` specifies that classes 2 and 3 are bound by a delay differentiation factor of 2, `$q init-rdcs 4 4 4 4` specifies that all classes are bound by a delay differentiation factor of 4 and is equivalent to `$q init-rdcs 4 4 4 1`, since the last coefficient is only used to specify that Class 4 is to be bound by proportional differentiation.

`$q init-rlcs <k'1> <k'2> <k'3> <k'4>`
This assigns the RLCs for the four JoBS classes. For instance, using a value of 3 for k1 means that Class-2 loss rates will be roughly equal to four times Class-2 loss rates. A value of -1 indicates that the class is not concerned by RLCs. As with RDCs, each RLC binds two classes, thus, one would expect only three parameters to be passed (k'1, k'2, and k'3, since k'4 theoretically bounds Classes 4 and 5, and Class 5 does not exist). As explained above, it is imperative to specify a value different from 0 and -1 to k'4 if Class 4 is to be concerned by RLCs.

`$q init-alcs <L1> <L2> <L3> <L4>`
This assigns the absolute loss guarantees (ALCs) to all four classes. L1 to L4 are given in fraction of 1. For instance, setting L1 to 0.05 means that Class-1 loss rate will be guarantees to be less than 5%. A value of -1 indicates that the corresponding class is not subject to an ALC.

`$q init-adcs <D1> <D2> <D3> <D4>`
This assigns the absolute loss guarantees (ADCs) to all four classes. D1 to D4 are given in milliseconds. A value of -1 indicates that the corresponding class is not subject to an ADC.

`$q trace-file <filename>`
This specifies the trace file for all per-hop metrics. JoBS uses an internal module to trace loss and delays, service rates, and per-class queue lengths in packets. If filename is set to **null**, no trace will be provided.

`$q link [<link-object> link]`
This command is required to bind a link to a JoBS queue. Note that JoBS needs to know the capacity of the link. Thus, this command **has to** be issued before the simulation is started.

`$q sampling-period <sampling-interval>`
This command specifies the sampling interval (in packets) at which the service rate adjustments for proportional differentiation will be performed. The default is a sampling interval of 1 packet, meaning that the rate allocation is reevaluated upon each packet arrival. Larger sampling intervals speed up the simulations, but typically result in poorer proportional differentiation.

`$q id <num_id>`
This command affects a numerical ID to the JoBS queue.

`$q initialize`
This command is required, and should be run after all configuration operations have been performed. This command will perform the final checks and configuration of the JoBS queue.

`$q copyright-info`
Displays authors and copyright information.

A simple example script (with nam output), fully annotated and commented can be found in *ns*/tcl/ex/jobs-lossdel.tcl. A more realistic example of a simulation with JoBS queues can be found in *ns*/tcl/ex/jobs-cn2002.tcl. This script is very similar to what was used in a simulation presented in [**?**]. Associated tracefiles and *gnuplot* scripts for visualization (in case you favor *gnuplot* over *xgraph* can be found in *ns*/tcl/ex/jobs-lossdel, and *ns*/tcl/ex/jobs-cn2002.

Marker objects

`$q marker_type <1|2>`
Selects the type of marker. 1 is DETERMINISTIC, 2 is STATISTICAL.

`$q marker_class <1|2|3|4>`
For a deterministic marker, selects which class packets should be marked with.

`$q marker_frc <f1> <f2> <f3> <f4>`
For a statistical marker, gives the fraction of packets that should be marked from each class. For instance, using 0.1 for f1 means that 10 percent of the traffic coming to the Marker link will be marked as Class 1.

Demarker objects

`$q trace-file <filename>`
This command specifies the trace file used for the demarker object. filename.1 will contain the end-to-end delays of each Class-1 packet to have reached the Demarker link, filename.2 will contain the end-to-end delays of each Class-2 packet to have reached the Demarker link, and so forth. (There will of course be 4 trace files, one for each class.)

Chapter 8

DOCSIS links

ns-2 contains models for sending Internet traffic over cable modems using the Data Over Cable Service Interface Specification (DOCSIS) specification: http://www.cablemodem.com. These models directly simulate DOCSIS 1.1 and DOCSIS 2.0 links and can be used to simulate DOCSIS 3.0 links and DOCSIS 3.1 SC-QAM links. Channel bonding for DOCSIS 3.x links is simulated by setting the link rate equal to the aggregate link rate for the bonding group.

DelayTb (`Link/DelayTb`) models a DOCSIS downstream link (from CMTS to the cable modem). More specifically, it models a single downstream service flow providing service to a single cable modem. It takes the following parameters:

`rate_`	"Maximum Sustained Traffic Rate": i.e. Token bucket rate (bits/s)
`bucket_`	"Maximum Traffic Burst": i.e. Token bucket maximum size (bytes)
`peakrate_`	"Peak Traffic Rate": i.e. Peak rate token generation rate (bits/s)
`peakbucket_`	Peak rate token bucket maximum rate (bytes): leave at 1522 to model DOCSIS

As per the DOCSIS 3.0/3.1 specifications, DelayTb uses two token buckets for rate shaping that will accumulate tokens according to their parameters. A departing packet gets the peak or normal transmission rate depending on the available tokens. To model DOCSIS 1.1/2.0, set `peakrate_` equal to the line rate.

DocsisLink (`Link/DocsisLink`) models a DOCSIS upstream link (from cable modem to the CMTS). More specifically, it models a single upstream service flow with best effort scheduling service. It takes the following parameters:

`mapint_`	The MAP interval (seconds); typically 2ms
`maxgrant_`	The maximum grant size (bytes) per MAP interval
`mgvar_`	The variability of maximum grant size (0..100: percentage)
`rate_`	Token generation rate (bits/s)
`bucket_`	Token bucket maximum size (bytes)
`peakrate_`	Peak rate token generation rate (bits/s)
`peakbucket_`	Peak rate token bucket maximum rate (bytes)

DOCSIS's upstream transmission is scheduled at a regular interval called "MAP interval". Before the beginning of each MAP interval, the cable modem receives a grant for how many bytes it can send. This byte count varies as a result of congestion from other users on the shared upstream link; `maxgrant_` and `mgvar_` are for emulating this congestion. The parameter `maxgrant_` is used to cap the average available capacity of the upstream link, and `mgvar_` provides a way to simulate the variability of congestion.

The remaining DocsisLink parameters implement the DOCSIS token bucket rate shaping, just like DelayTb.