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

November 20, 1998

ns © is LBNL's Network Simulator [11]. 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.

[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]
> ...

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
```

Part I

Interface to the Interpreter

Chapter 2

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.

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, `Tcl`. The rest of the simulator code resides in the directory, `ns-2`. We will use the notation ~*Tcl*/⟨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 ~*Tcl*/. We only focus on the six that are used in *ns*: The Class Tcl (Section 2.1) contains the methods that C++ code will use to access the interpreter. The class TclObject (Section 2.2) is the base class for all simulator objects that are also mirrored in the compiled hierarchy. The class TclClass (Section 2.3) defines the interpreted class hierarchy, and the methods to permit the user to instantiate TclObjects. The class TclCommand (Section 2.4) is used to define simple global interpreter commands. The class EmbeddedTcl (Section 2.5) contains the methods to load higher level builtin commands that make configuring simulations easier. Finally, the class InstVar (Section 2.6) contains methods to access C++ member variables as OTcl instance variables.

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

2.1 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.

2.1.1 Obtain a Reference to the class Tcl instance

A single instance of the class is declared in ~*Tcl*/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();
```

2.1.2 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 2.1.3) 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);
```

2.1.3 Passing Results to/fro 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");
```

2.1.4 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 2.1.3), 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.

2.1.5 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.

2.1.6 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.

2.2 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 2.2.2) 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 2.6). 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 2.1.5).

[1] In the latest release of *ns* and *ns/Tcl*, 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.

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
```

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 2.2.1); the next line configures a bound variable (Section 2.2.2); and finally, the last line illustrates the interpreted object invoking a C++ method as if they were an instance procedure (Section 2.2.4).

2.2.1 Creating and Destroying TclObjects

When the user creates a new TclObject, using the procedures `new{}` and `delete{}`; these procedures are defined in ~*Tcl*/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{}`.
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.

[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*.

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)
   (a) adds the new object to hash table of TclObjects described earlier (Section 2.1.5).
   (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 2.2.4), 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.

2.2.2 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 2.6) in detail later.

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

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 2.6). `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:

```
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.

2.2.3 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.

2.2.4 `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 2.1.3).
- 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.

2.3 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[2...4]` contain `$self`, `$class`, and `$proc`. Since `create` is called through the instance procedure `create-shadow`, `argv[4]` contains `create-shadow`.

— `argv[5...]` contain any additional arguments 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.

2.4 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
...
```

2.5 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 ~*Tcl*/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 ~*Tcl*/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, ~*Tcl*/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.

[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.

— *~Tcl*/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();
```

2.6 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 class TclClass (Section 2.3) 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 3

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.

3.1 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 simple linked-list 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 10). 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.

3.2 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 (default), heap, calendar queue, 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. 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, they are first ordered (in some scheduler-dependent way) and executed serially. 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) = 0;
};
```

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 10.2.1). 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)[1]. At-events are implemented as events where the handler is effectively an execution of the tcl interpreter.

3.2.1 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.

[1]Note that this particular code fragment expects to be encapsulated in an object instance procedure, where the appropriate reference to `$self` is correctly defined.

3.2.2 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 [1]; it is believed that MaRS itself borrowed the code from NetSim [7], although this lineage has not been completely verified.

3.2.3 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 [4], and informally described in Jain (p. 410) [8]. The implementation of Calendar queues in *ns* v2 was contributed by David Wetherall (presently at MIT/LCS).

3.2.4 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.

3.3 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 4) and managing the links (Chapter 5)), methods to perform tracing (Chapter 14), 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
```

Chapter 4

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 5), 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 4.1). We then describe the instance procedures in the class Node (Section 4.2) to access and operate on individual nodes. We conclude with detailed descriptions of the Classifier (Section 4.3) 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-nodes.tcl, ~*ns*/classifier.{cc, h}, ~*ns*/classifier-addr.cc, ~*ns*/classifier-mcast.cc, ~*ns*/classifier-mpath.cc, and, ~*ns*/replicator.cc.

4.1 Simulator Methods: Creating the Topology

The basic primitive for acquiring a node is

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

The instance procedure `node` constructs a node out of more simple classifier objects (Section 4.3). 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 node is as shown in Figure 4.1.

The address of an agent in a node is 16 bits wide: the higher 8 bits define the node `id_`, the lower 8 bits identify the individual agent at the node. This limits the number of nodes in a simulation to 256 nodes. If the user needs to create a topology larger than 256 nodes, then they should first expand the address space before creating any nodes, as

```
Node expandaddr
```

This expands the address space to 30 bits, the higher 22 of which are used to assign node numbers.

By default, nodes in *ns* are constructed for unicast simulations. In order to create nodes for multicast simulation, the class variable, `EnableMcast_`, should be set to 1, as:

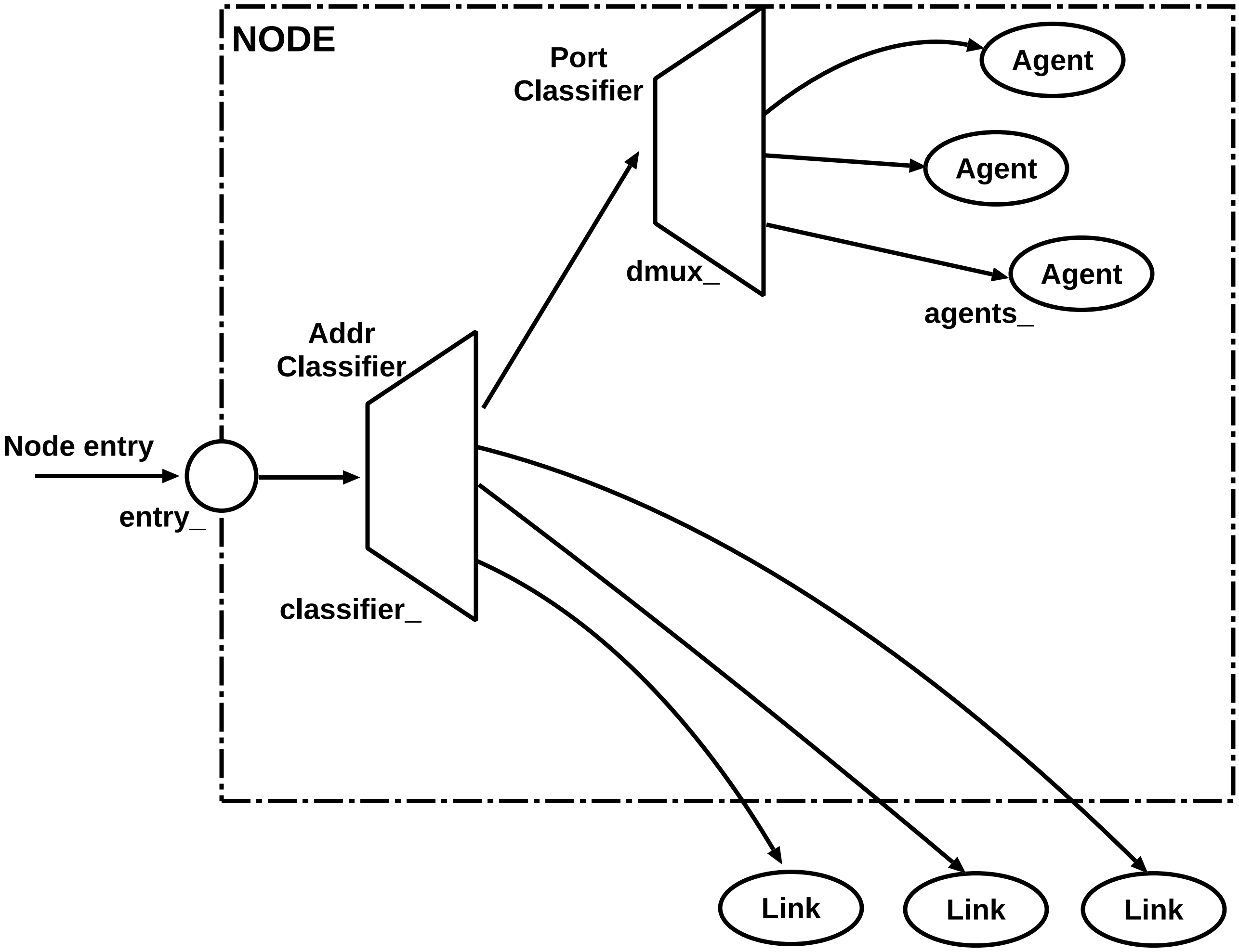


Figure 4.1: Structure of an Unicast Node

```
Simulator set EnableMcast_ 1
```

before any nodes are created. The internal structure of a typical multicast node is shown in Figure 4.2.

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. This implies that, by default, a multicast simulation is restricted to 128 nodes.

4.2 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

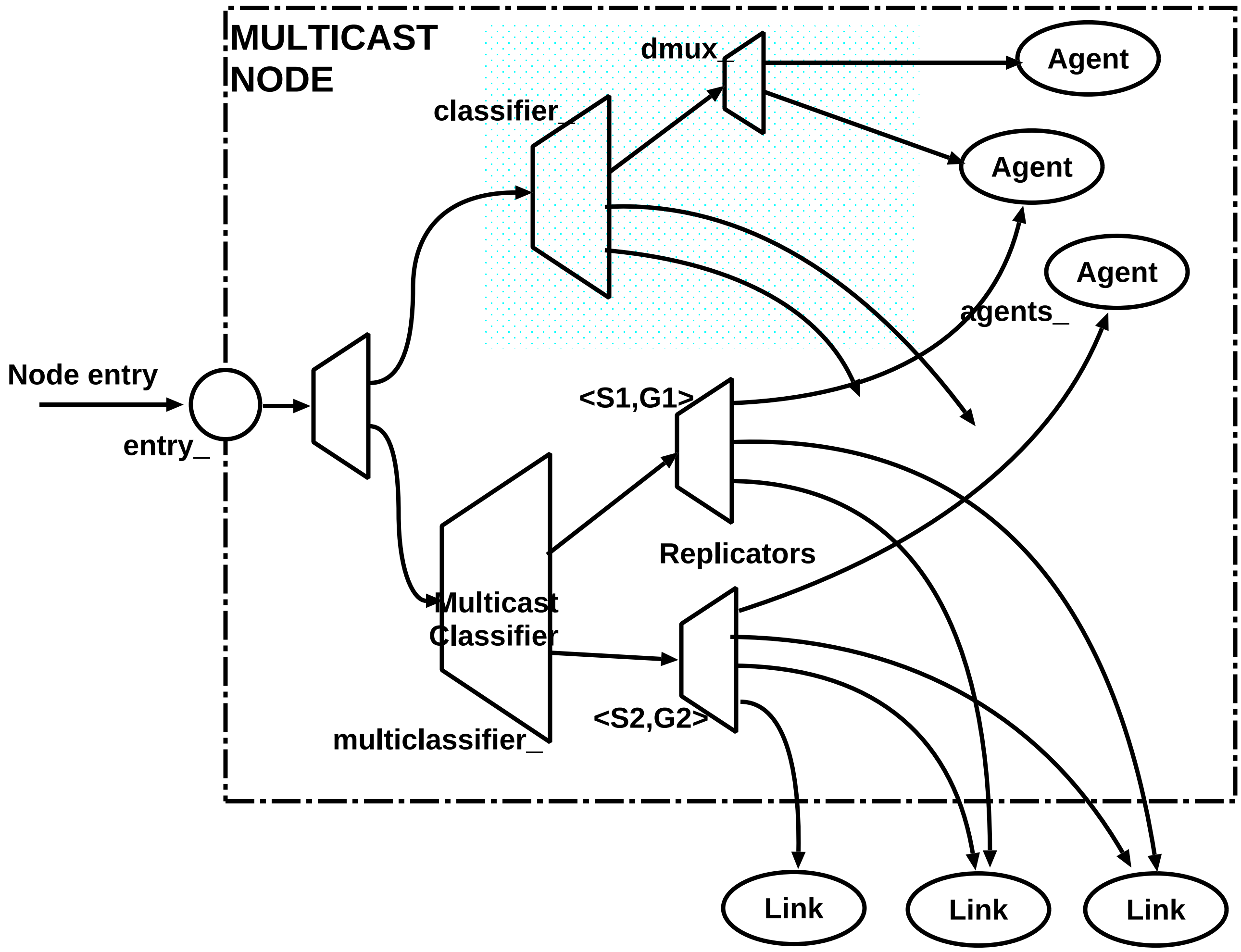


Figure 4.2: Internal Structure of a Multicast Node

— 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.

3. `$node enable-mcast` is an internal procedure used by the class simulator when creating a node. This procedure installs the additional classifiers needed to convert a unicast node to a multicast 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 15) 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 4.3); however, we defer the discussion of multipath routing (Chapter 15) 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 17).

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`.

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.

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

4.3 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 4.3.1), the multicast classifier (Section 4.3.2), the multipath classifier (Section 4.3.3), and finally, the replicator (Section 4.3.5).

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.

```
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.

4.3.1 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.

4.3.2 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.

4.3.3 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? */
};
```

4.3.4 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` flor 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.

4.3.5 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.

Chapter 5

Links: Simple Links

This is the second aspect of defining the topology. In the previous chapter (Chapter 4), 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 5.1). We conclude the chapter with a description the connector object (Section 5.2), 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

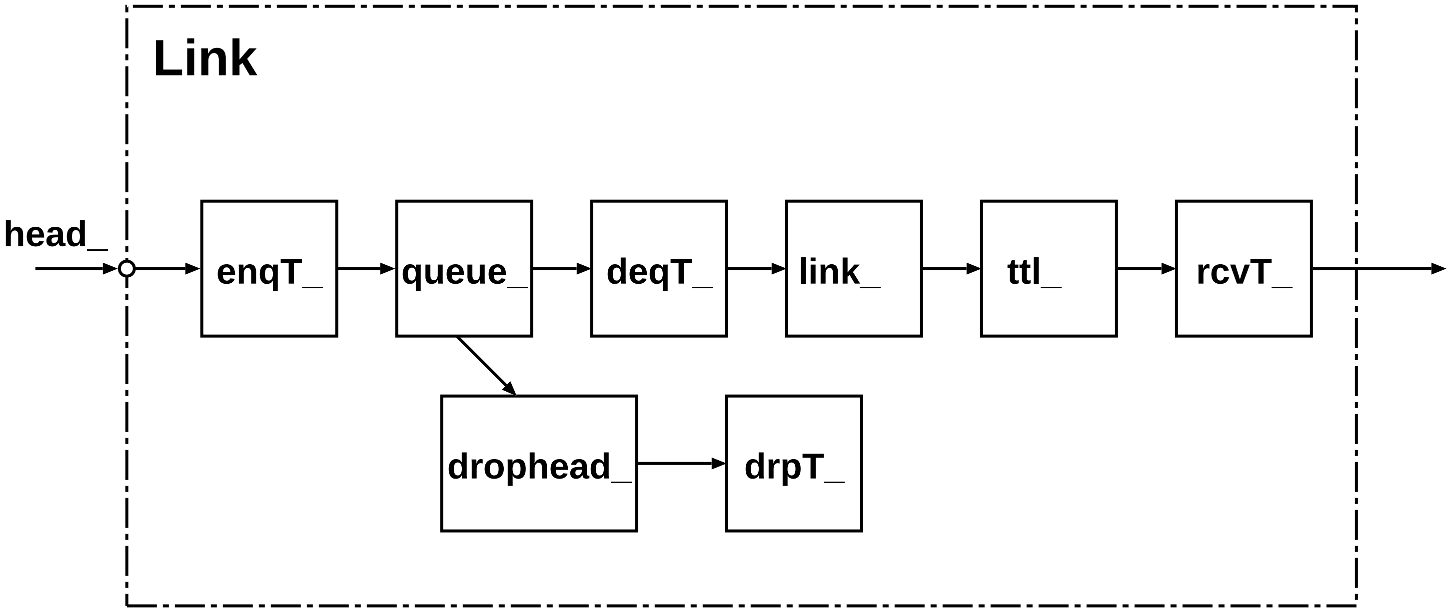


Figure 5.1: Composite Construction of a Unidirectional Link

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 5.2), and discuss their functionality/implementation in other chapters.

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

5.1 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 14).

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

5.2 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 16).

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 17).

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 7).

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 6).

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.

Chapter 6

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 Weighted Fair Queueing (WFQ), 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.

6.1 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 14). 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.

6.1.1 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).

6.1.2 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.

6.2 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. Simple FIFO scheduling is implemented in the deque function by always returning the first packet in the packet queue.

Chapter 7

Delays and Links

Delays represent the time required for a packet to traverse a link. A special form of this object ("dynamic link") also captures the possibility of a link failure. The amount of time required for a packet to traverse a link is defined to be $s/b + d$ where s is the packet size (as recorded in its IP header), b is the speed of the link in bits/sec, and d is the link delay in seconds. The implementation of link delays is closely associated with the blocking procedures described for Queues in Section6.1.1.

7.1 The LinkDelay Class

The `class LinkDelay` is derived from the base `class Connector`. Its definition is in ~*ns*/delay.cc, and is briefly excerpted below:

```
class LinkDelay : public Connector {
 public:
        LinkDelay();
        void recv(Packet* p, Handler*);
        void send(Packet* p, Handler*);
        void handle(Event* e);
        double delay();                              /* line latency on this link */
        double bandwidth();                          /* bandwidth on this link */
        inline double txtime(Packet* p) {            /* time to send pkt p on this link */
                hdr_cmn* hdr = (hdr_cmn*) p->access(off_cmn_);
                return (hdr->size() * 8. / bandwidth_);
        }

 protected:
        double bandwidth_;                           /* bandwidth of underlying link (bits/sec) */
        double delay_;                               /* line latency */
        int dynamic_;                                /* indicates whether or not link is ~ */
        Event inTransit_;
        PacketQueue* itq_;                           /* internal packet queue for dynamic links */
        Packet* nextPacket_;                         /* to be delivered for a dynamic link. */
        Event intr_;
};
```

The `recv()` method overrides the base class Connector version. It is defined as follows:

```
void LinkDelay::recv(Packet* p, Handler* h)
{
        double txt = txtime(p);
        Scheduler& s = Scheduler::instance();
        if (dynamic_) {
                Event* e = (Event*)p;
                e->time_ = s.clock() + txt + delay_;
                itq_->enque(p);
                schedule_next();
        } else {
                s.schedule(target_, p, txt + delay_);
        }
        /* XXX only need one intr_ since upstream object should
         * block until it's handler is called
         *
         * This only holds if the link is not dynamic. If it is, then
         * the link itself will hold the packet, and call the upstream
         * object at the appropriate time. This second interrupt is
         * called inTransit_, and is invoked through schedule_next()
         */
        s.schedule(h, &intr_, txt);
}
```

This object supports one instproc-like, `$object dynamic`, to set its variable, `dynamic_`. This variable determines whether the link is dynamic or not (*i.e.*, prone to fail/recover at appropriate times). The internal behavior of the link in each case is different.

For "non-dynamic" links, this method operates by receiving a packet, p, and scheduling two events. Assume these two events are called E_1 and E_2, and that event E_1 is scheduled to occur before E_2. E_1 is scheduled to occur when the upstream node attached to this delay element has completed sending the current packet (which takes time equal to the packet size divided by the link bandwidth). E_1 is usually associated with a `Queue` object, and will cause it to (possibly) become unblocked (see section 6.1.1). E_2 represents the packet arrival event at the downstream neighbor of the delay element. Event E_2 occurs a number of seconds later than E_1 equal to the link delay.

Alternately, when the link is dynamic, and receives p, then it will schedule E_1 to possibly unblock the queue at the appropriate time. However, E_2 is scheduled only if p is the only packet currently in transit. Otherwise, there is at least one packet in transit on the link that must be delivered before p at E_2. Therefore, packet p is held in the object's inTransit queue, `itq_`. When the packet just before p in transit on the link is delivered at the neighbor node, the DelayLink object will schedule an event for itself to fire at E_2. At that appropriate time then, it's `handle()` method will directly send p to its target. The object's internal `schedule_next()` method will schedule these events for packet sin transit at the appropriate time.

Chapter 8

Agents

Agents represent endpoints where network-layer packets are constructed or consumed, and are used in the implementation of protocols at various layers. The `class Agent` has an implementation partly in OTcl and partly in C++. The C++ implementation is contained in ~*ns*/agent.cc and ~*ns*/agent.h, and the OTcl support is in ~*ns*/tcl/lib/ns-agent.tcl.

8.1 Agent state

The C++ `class Agent` includes enough internal state to assign various fields to a simulated packet before it is sent. This state includes the following:

`addr_`	node address of myself (source address in packets)
`dst_`	where I am sending packets to
`size_`	packet size in bytes (placed into the common packet header)
`type_`	type of packet (in the common header, see packet.h)
`fid_`	the IP flow identifier (formerly *class* in ns-1)
`prio_`	the IP priority field
`flags_`	packet flags (similar to ns-1)
`defttl_`	default IP ttl value

These variables may be modified by any class derived from `Agent`, although not all of them may be needed by any particular agent.

8.2 Agent methods

The `class Agent` supports packet generation and reception. The following member functions are implemented by the C++ Agent class, and are generally *not* over-ridden by derived classes:

`Packet* allocpkt()`	allocate new packet and assign its fields
`Packet* allocpkt(int)`	allocate new packet with a data payload of n bytes and assign its fields

The following member functions are also defined by the class Agent, but *are* intended to be over-ridden by classes deriving from Agent:

`void timeout`(timeout number)	subclass-specific time out method
`void recv`(Packet*, Handler*)	receiving agent main receive path

The `allocpkt`() method is used by derived classes to create packets to send. The function fills in the following fields in the common packet header (Section 10): `uid, ptype, size`, and the following fields in the IP header: `src, dst, flowid, prio, ttl`. It also zero-fills in the following fields of the Flags header: `ecn, pri, usr1, usr2`. Any packet header information not included in these lists must be must be handled in the classes derived from `Agent`.

The `recv`() method is the main entry point for an Agent which receives packets, and is invoked by upstream nodes when sending a packet. In most cases, Agents make no use of the second argument (the handler defined by upstream nodes).

8.3 Protocol Agents

There are several agents supported in the simulator. These are their names in OTcl:

TCP	a "Tahoe" TCP sender (cwnd = 1 on any loss)
TCP/Reno	a "Reno" TCP sender (with fast recovery)
TCP/NewReno	a modified Reno TCP sender (changes fast recovery)
TCP/Sack1	a SACK TCP sender
TCP/Fack	a "forward" SACK sender TCP
TCP/FullTcp	a more full-functioned TCP with 2-way traffic
TCP/Vegas	a "Vegas" TCP sender
TCP/Vegas/RBP	a Vegas TCP with "rate based pacing"
TCP/Vegas/RBP	a Reno TCP with "rate based pacing"
TCP/Asym	an experimental Tahoe TCP for asymmetric links
TCP/Reno/Asym	an experimental Reno TCP for asymmetric links
TCP/Newreno/Asym	an experimental NewReno TCP for asymmetric links
TCPSink	a Reno or Tahoe TCP receiver (not used for FullTcp)
TCPSink/DelAck	a TCP delayed-ACK receiver
TCPSink/Asym	an experimental TCP sink for asymmetric links
TCPSink/Sack1	a SACK TCP receiver
TCPSink/Sack1/DelAck	a delayed-ACK SACK TCP receiver
UDP	a basic UDP agent
RTP	an RTP sender and receiver
RTCP	an RTCP sender and receiver
LossMonitor	a packet sink which checks for losses
IVS/Source	an IVS source
IVS/Receiver	an IVS receiver

CtrMcast/Encap	a “centralised multicast” encapsulator
CtrMcast/Decap	a “centralised multicast” de-encapsulator
Message	a protocol to carry textual messages
Message/Prune	processes multicast routing prune messages
SRM	an SRM agent with non-adaptive timers
SRM/Adaptive	an SRM agent with adaptive timers
Tap	interfaces the simulator to a live network
Null	a degenerate agent which discards packets
rtProto/DV	distance-vector routing protocol agent

Agents are used in the implementation of protocols at various layers. Thus, for some transport protocols (e.g. UDP) the distribution of packet sizes and/or inter-departure times may be dictated by some separate object representing the demands of an application. To this end, agents expose an application programming interface (API) to the application. For agents used in the implementation of lower-layer protocols (e.g. routing agents), size and departure timing is generally dictated by the agent's own processing of protocol messages.

8.4 OTcl Linkage

Agents may be created within OTcl and an agent's internal state can be modified by use of Tcl's `set` function and any Tcl functions an Agent (or its base classes) implements. Note that some of an Agent's internal state may exist only within OTcl, and is thus is not directly accessible from C++.

8.4.1 Creating and Manipulating Agents

The following example illustrates the creation and modification of an Agent in OTcl:

```
set newtcp [new Agent/TCP]                ;# create new object (and C++ shadow object)
$newtcp set window_ 20                    ;# sets the tcp agent's window to 20
$newtcp target $dest                      ;# target is implemented in Connector class
$newtcp set portID_ 1                     ;# exists only in OTcl, not in C++
```

8.4.2 Default Values

Default values for member variables, those visible in OTcl only and those linked between OTcl and C++ with `bind` are initialized in the ~*ns*/tcl/lib/ns-default.tcl file. For example, `Agent` is initialized as follows:

```
Agent set fid_ 0
Agent set prio_ 0
Agent set addr_ 0
```

```
Agent set dst_ 0
Agent set flags_ 0
```

Generally these initializations are placed in the OTcl namespace before any objects of these types are created. Thus, when an `Agent` object is created, the calls to `bind` in the objects' constructors will causes the corresponding member variables to be set to these specified defaults.

8.4.3 OTcl Methods

The instance procedures defined for the OTcl `Agent` class are currently found in ~*ns*/tcl/lib/ns-agent.tcl. They are as follows:

`port`	the agent's port identifier
`dst-port`	the destination's port identifier
`attach-source` ⟨`stype`⟩	create and attach a Source object to an agent

8.5 Examples: Tcp, TCP Sink Agents

The `class TCP` represents a simplified TCP sender. It sends data to a `TCPSink` agent and processes its acknowledgments. It has a separate object associated with it which represents an application's demand. By looking at the `class TCPAgent` and `class TCPSinkAgent`, we may see how relatively complex agents are constructed. An example from the Tahoe TCP agent `TCPAgent` is also given to illustrate the use of timers.

8.5.1 Creating the Agent

The following OTcl code fragment creates a `TCP` agent and sets it up:

```
set tcp [new Agent/TCP]                    ;# create sender agent
$tcp set fid_ 2                            ;# set IP-layer flow ID
set sink [new Agent/TCPSink]               ;# create receiver agent
$ns attach-agent $n0 $tcp                  ;# put sender on node $n0
$ns attach-agent $n3 $sink                 ;# put receiver on node $n3
$ns connect $tcp $sink                     ;# establish TCP connection
set ftp [new Application/FTP]              ;# create an FTP source "application"
$ftp attach-agent $tcp                     ;# associate FTP with the TCP sender
$ns at 1.2 "$ftp start"                    ;#arrange for FTP to start at time 1.2 sec
```

The OTcl instruction `new Agent/TCP` results in the creation of a C++ `TcpAgent` class object. It's constructor performs first invokes the constructor of the `Agent` base class and then performs its own bindings. These two constructors appear as follows:

The TcpSimpleAgent constructor (~*ns*/tcp.cc):

```
TcpAgent::TcpAgent() : Agent(PT_TCP), rtt_active_(0), rtt_seq_(-1),
                rtx_timer_(this), delsnd_timer_(this)
{
        bind("window_", &wnd_);
```

```
        bind("windowInit_", &wnd_init_);
        bind("windowOption_", &wnd_option_);
        bind("windowConstant_", &wnd_const_);
        ...
        bind("off_ip_", &off_ip_);
        bind("off_tcp_", &off_tcp_);
        ...
}
```

The Agent constructor (~*ns*/agent.cc):

```
Agent::Agent(int pkttype) :
        addr_(-1), dst_(-1), size_(0), type_(pkttype), fid_(-1),
        prio_(-1), flags_(0)
{
        memset(pending_, 0, sizeof(pending_));                    /* timers */
        // this is really an IP agent, so set up
        // for generating the appropriate IP fields...
        bind("addr_", (int*)&addr_);
        bind("dst_", (int*)&dst_);
        bind("fid_", (int*)&fid_);
        bind("prio_", (int*)&prio_);
        bind("flags_", (int*)&flags_);
        ...
}
```

These code fragments illustrate the common case where an agent's constructor passes a packet type identifier to the `Agent` constructor. The values for the various packet types are used by the packet tracing facility (Section 14.5) and are defined in ~*ns*/trace.h. The variables which are bound in the `TcpAgent` constructor are ordinary instance/member variables for the class with the exception of the special integer values `off_tcp_` and `off_ip_`. These are needed in order to access a TCP header and IP header, respectively. Additional details are in the section on packet headers (Section 10.1).

Note that the `TcpAgent` constructor contains initializations for two timers, `rtx_timer_` and `delsnd_timer_`.

`TimerHandler` objects are initialized by providing a pointer (the `this` pointer) to the relevant agent.

8.5.2 Starting the Agent

The `TcpAgent` agent is started in the example when its FTP source receives the `start` directive at time 1.2. The `start` operation is an instance procedure defined on the class Application/FTP (Section 21.4). It is defined in ~*ns*/tcl/lib/ns-source.tcl as follows:

```
Application/FTP instproc start {} {
        [$self agent] send -1
}
```

In this case, `agent` refers to our simple TCP agent and `send -1` is analogous to sending an arbitrarily large file.

The call to `send` eventually results in the simple TCP sender generating packets. The following function `output` performs this:

```
void TcpAgent::output(int seqno, int reason)
{
        Packet* p = allocpkt();
        hdr_tcp *tcph = (hdr_tcp*)p->access(off_tcp_);
        double now = Scheduler::instance().clock();
        tcph->seqno() = seqno;
        tcph->ts() = now;
        tcph->reason() = reason;
        Connector::send(p, 0);
        ...
        if (!(rtx_timer_.status() == TIMER_PENDING))
                /* No timer pending. Schedule one. */
                set_rtx_timer();
}
```

Here we see an illustration of the use of the `Agent::allocpkt()` method. This output routine first allocates a new packet (with its common and IP headers already filled in), but then must fill in the appropriate TCP-layer header fields. To find the TCP header in a packet (assuming it has been enabled (Section 10.2.3)) the `off_tcp_` must be properly initialized, as illustrated in the constructor. The packet `access()` method returns a pointer to the TCP header, its sequence number and time stamp fields are filled in, and the `send()` method of the class Connector is called to send the packet downstream one hop. Note that the C++ `::` scoping operator is used here to avoid calling `TcpSimpleAgent::send()` (which is also defined). The check for a pending timer uses the timer method `status()` which is defined in the base class TimerHandler. It is used here to set a retransmission timer if one is not already set (a TCP sender only sets one timer per window of packets on each connection).

8.5.3 Processing Input at Receiver

Many of the TCP agents can be used with the `class TCPSink` as the peer. This class defines the `recv()` and `ack()` methods as follows:

```
void TcpSink::recv(Packet* pkt, Handler*)
{
        hdr_tcp *th = (hdr_tcp*)pkt->access(off_tcp_);
        acker_->update(th->seqno());
        ack(pkt);
        Packet::free(pkt);
}

void TcpSink::ack(Packet* opkt)
{
        Packet* npkt = allocpkt();

        hdr_tcp *otcp = (hdr_tcp*)opkt->access(off_tcp_);
        hdr_tcp *ntcp = (hdr_tcp*)npkt->access(off_tcp_);
        ntcp->seqno() = acker_->Seqno();
        ntcp->ts() = otcp->ts();

        hdr_ip* oip = (hdr_ip*)opkt->access(off_ip_);
        hdr_ip* nip = (hdr_ip*)npkt->access(off_ip_);
        nip->flowid() = oip->flowid();
```

```
        hdr_flags* of = (hdr_flags*)opkt->access(off_flags_);
        hdr_flags* nf = (hdr_flags*)npkt->access(off_flags_);
        nf->ecn_ = of->ecn_;

        acker_->append_ack((hdr_cmn*)npkt->access(off_cmn_),
                           ntcp, otcp->seqno());
        send(npkt, 0);
}
```

The `recv()` method overrides the `Agent::recv()` method (which merely discards the received packet). It updates some internal state with the sequence number of the received packet (and therefore requires the `off_tcp_` variable to be properly initialized. It then generates an acknowledgment for the received packet. The `ack()` method makes liberal use of access to packet header fields including separate accesses to the TCP header, IP header, Flags header, and common header. The call to `send()` invokes the `Connector::send()` method.

8.5.4 Processing Responses at the Sender

Once the simple TCP's peer receives data and generates an ACK, the sender must (usually) process the ACK. In the `TcpAgent` agent, this is done as follows:

```
/*
 * main reception path - should only see acks, otherwise the
 * network connections are misconfigured
 */
void TcpAgent::recv(Packet *pkt, Handler*)
{
        hdr_tcp *tcph = (hdr_tcp*)pkt->access(off_tcp_);
        hdr_ip* iph = (hdr_ip*)pkt->access(off_ip_);
        ...
        if (((hdr_flags*)pkt->access(off_flags_))->ecn_)
                quench(1);
        if (tcph->seqno() > last_ack_) {
                newack(pkt);
                opencwnd();
        } else if (tcph->seqno() == last_ack_) {
                if (++dupacks_ == NUMDUPACKS) {
                        ...
                }
        }
        Packet::free(pkt);
        send(0, 0, maxburst_);
}
```

This routine is invoked when an ACK arrives at the sender. In this case, once the information in the ACK is processed (by `newack`) the packet is no longer needed and is returned to the packet memory allocator. In addition, the receipt of the ACK indicates the possibility of sending additional data, so the `TcpSimpleAgent::send()` method is invoked which attempts to send more data if the TCP window allows.

8.5.5 Implementing Timers

As described in the following chapter (Chapter 9), specific timer classes must be derived from an abstract base `class TimerHandler` defined in ~*ns*/timer-handler.h. Instances of these subclasses can then be used as various agent timers. An agent may wish to override the `Agent::timeout()` method (which does nothing). In the case of the Tahoe TCP agent, two timers are used: a delayed send timer `delsnd_timer_` and a retransmission timer `rtx_timer_`. We describe the retransmission timer in TCP (Section 9.1.2) as an example of timer usage.

8.6 Creating a New Agent

To create a new agent, one has to do the following:

1. decide its inheritance structure (Section 8.6.1), and create the appropriate class definitions,
2. define the `recv()` and `timeout()` methods (Section 8.6.2),
3. define any necessary timer classes,
4. define OTcl linkage functions (Section 8.6.3),
5. write the necessary OTcl code to access the agent (Section 8.6.4).

The action required to create and agent can be illustrated by means of a very simple example. Suppose we wish to construct an agent which performs the ICMP ECHO REQUEST/REPLY (or "ping") operations.

8.6.1 Example: A "ping" requestor (Inheritance Structure)

Deciding on the inheritance structure is a matter of personal choice, but is likely to be related to the layer at which the agent will operate and its assumptions on lower layer functionality. The simplest type of Agent, connectionless datagram-oriented transport, is the `Agent/UDP` base class. Traffic generators can easily be connected to UDP Agents. For protocols wishing to use a connection-oriented stream transport (like TCP), the various TCP Agents could be used. Finally, if a new transport or "sub-transport" protocol is to be developed, using `Agent` as the base class would likely be the best choice. In our example, we'll use Agent as the base class, given that we are constructing an agent logically belonging to the IP layer (or just above it).

We may use the following class definitions:

```
class ECHO_Timer;

class ECHO_Agent : public Agent {
 public:
        ECHO_Agent();
        int command(int argc, const char*const* argv);
 protected:
        void timeout(int);
        void sendit();
        double interval_;
        ECHO_Timer echo_timer_;
};

class ECHO_Timer : public TimerHandler {
```

```
public:
        ECHO_Timer(ECHO_Agent *a) : TimerHandler() { a_ = a; }
protected:
        virtual void expire(Event *e);
        ECHO_Agent *a_;
};
```

8.6.2 The **recv**() and **timeout**() Methods

The `recv`() method is not defined here, as this agent represents a request function and will generally not be receiving events or packets[1]. By not defining the `recv`() method, the base class version of `recv`() (*i.e.*, `Connector::recv`()) is used. The `timeout`() method is used to periodically send request packets. The following `timeout`() method is used, along with a helper method, `sendit`():

```
void ECHO_Agent::timeout(int)
{
        sendit();
        echo_timer_.resched(interval_);
}

void ECHO_Agent::sendit()
{
        Packet* p = allocpkt();
        ECHOHeader *eh = ECHOHeader::access(p->bits());
        eh->timestamp() = Scheduler::instance().clock();
        send(p, 0);     // Connector::send()
}

void ECHO_Timer::expire(Event *e)
{
        a_->timeout(0);
}
```

The `timeout`() method simply arranges for `sendit`() to be executed every `interval_` seconds. The `sendit`() method creates a new packet with most of its header fields already set up by `allocpkt`(). The packet is only lacks the current time stamp. The call to `access`() provides for a structured interface to the packet header fields, and is used to set the timestamp field. Note that this agent uses its own special header ("ECHOHeader"). The creation and use of packet headers is described in later chapter (Chapter 10); to send the packet to the next downstream node, `Connector::send`() is invoked without a handler.

8.6.3 Linking the "ping" Agent with OTcl

We have the methods and mechanisms for establishing OTcl Linkage earlier (Chapter 2). This section is a brief review of the essential features of that earlier chapter, and describes the minimum functionality required to create the ping agent.

There are three items we must handle to properly link our agent with Otcl. First we need to establish a mapping between the OTcl name for our class and the actual object created when an instantiation of the class is requested in OTcl. This is done as follows:

[1]This is perhaps unrealistically simple. An ICMP ECHO REQUEST agent would likely wish to process ECHO REPLY messages.

```
static class ECHOClass : public TclClass {
public:
        ECHOClass() : TclClass("Agent/ECHO") {}
        TclObject* create(int argc, const char*const* argv) {
                return (new ECHO_Agent());
        }
} class_echo;
```

Here, a *static* object "class_echo" is created. It's constructor (executed immediately when the simulator is executed) places the class name "Agent/ECHO" into the OTcl name space. The mixing of case is by convention; recall from Section 2.3 in the earlier chapters that the "/" character is a hierarchy delimiter for the interpreted hierarchy. The definition of the `create()` method specifies how a C++ shadow object should be created when the OTcl interpreter is instructed to create an object of class "Agent/ECHO". In this case, a dynamically-allocated object is returned. This is the normal way new C++ shadow objects are created.

Once we have the object creation set up, we will want to link C++ member variables with corresponding variables in the OTcl nname space, so that accesses to OTcl variables are actually backed by member variables in C++. Assume we would like OTcl to be able to adjust the sending interval and the packet size. This is accomplished in the class's constructor:

```
ECHO_Agent::ECHO_Agent() : Agent(PT_ECHO)
{
        bind_time("interval_", &interval_);
        bind("packetSize_", &size_);
}
```

Here, the C++ variables `interval_` and `size_` are linked to the OTcl instance variables `interval_` and `packetSize_`, respectively. Any read or modify operation to the Otcl variables will result in a corresponding access to the underlying C++ variables. The details of the `bind()` methods are described elsewhere (Section 2.2.2). The defined constant `PT_ECHO` is passed to the `Agent()` constuctor so that the `Agent::allocpkt()` method may set the packet type field used by the trace support (Section 14.5). In this case, `PT_ECHO` represents a new packet type and must be defined in ~*ns*/trace.h (Section 14.4).

Once object creation and variable binding is set up, we may want to create methods implemented in C++ but which can be invoked from OTcl (Section 2.2.4). These are often control functions that initiate, terminate or modify behavior. In our present example, we may wish to be able to start the ping query agent from OTcl using a "start" directive. This may be implemented as follows:

```
int ECHO_Agent::command(int argc, const char*const* argv)
{
        if (argc == 2) {
                if (strcmp(argv[1], "start") == 0) {
                        timeout(0);
                        return (TCL_OK);
                }
        }
        return (Agent::command(argc, argv));
}
```

Here, the `start()` method available to OTcl simply calls the C++ member function `timeout()` which initiates the first packet generation and schedules the next. Note this class is so simple it does not even include a way to be stopped.

8.6.4 Using the agent through OTcl

The agent we have created will have to be instantiated and attached to a node. Note that a node and simulator object is assumed to have already been created. The following OTcl code performs these functions:

```
set echoagent [new Agent/ECHO]
$simulator attach-agent $node $echoagent
```

To set the interval and packet size, and start packet generation, the following OTcl code is executed:

```
$echoagent set dst_ $dest
$echoagent set fid_ 0
$echoagent set prio_ 0
$echoagent set flags_ 0
$echoagent set interval_ 1.5
$echoagent set packetSize_ 1024
$echoagent start
```

This will cause our agent to generate one 1024-byte packet destined for node `$dest` every 1.5 seconds.

8.7 The Agent API

Simulated applications may be implemented on top of protocol agents. Chapter 21 describes the API used by applications to access the services provided by the protocol agent.

Chapter 9

Timers

Timers may be implemented in C++ or OTcl. In C++, timers are based on an abstract base class defined in ~*ns*/timer-handler.h. They are most often used in agents, but the framework is general enough to be used by other objects. The discussion below is oriented towards the use of timers in agents.

The procedures and functions described in this chapter can be found in ~*ns*/tcl/ex/timer.tcl, and ~*ns*/timer-handler.{cc, h}.

In OTcl, a simple timer class is defined in ~*ns*/tcl/ex/timer.tcl. Subclasses can be derived to provide a simple mechanism for scheduling events at the OTcl level.

9.1 C++ abstract base class TimerHandler

The abstract base class `TimerHandler` contains the following public member functions:

`void sched`(double delay)	schedule a timer to expire delay seconds in the future
`void resched`(double delay)	reschedule a timer (similar to `sched`(), but timer may be pending)
`void cancel`()	cancel a pending timer
`int status`()	returns timer status (either IDLE, PENDING, or HANDLING)

The abstract base class `TimerHandler` contains the following protected members:

`virtual void expire`(Event* e)	`=0`	this method must be filled in by the timer client
`virtual void handle`(Event* e)	`=0`	consumes an event
`int status_`		keeps track of the current timer status
`Event event_`		event to be consumed upon timer expiration

The pure virtual functions must be defined by the timer classes deriving from this abstract base class.

Finally, two private inline functions are defined:

```
inline void _sched(double delay) {
    (void)Scheduler::instance().schedule(this, &event_, delay);
}
```

```
inline void _cancel() {
    (void)Scheduler::instance().cancel(&event_);
}
```

From this code we can see that timers make use of methods of the `Scheduler` class.

9.1.1 Definition of a new timer

To define a new timer, subclass this function and define `handle`() if needed (`handle`() is not always required):

```
class MyTimer : public TimerHandler {
public:
  MyTimer(MyAgentClass *a) : TimerHandler() { a_ = a; }
  virtual double expire(Event *e);
protected:
  MyAgentClass *a_;
};
```

Then define expire:

```
double
MyTimer::expire(Event *e)
{
  //  do the work
  // return TIMER_HANDLED;      //  => do not reschedule timer
  // return delay;              //  => reschedule timer after delay
}
```

Note that `expire`() can return either the flag TIMER_HANDLED or a delay value, depending on the requirements for this timer.

Often `MyTimer` will be a friend of `MyAgentClass`, or `expire`() will only call a public function of `MyAgentClass`.

Timers are not directly accessible from the OTcl level, although users are free to establish method bindings if they so desire.

9.1.2 Example: Tcp retransmission timer

TCP is an example of an agent which requires timers. There are three timers defined in the basic Tahoe TCP agent defined in `tcp.cc`:

```
rtx_timer_;                                       /* Retransmission timer */
delsnd_timer_;      /* Delays sending of packets by a small random amount of time, */
                                                  /* to avoid phase effects */
burstsnd_timer_;        /* Helps TCP to stagger the transmission of a large window */
                                                  /* into several smaller bursts */
```

In ~*ns*/tcp.h, three classes are derived from the base class `class TimerHandler`:

```
class RtxTimer : public TimerHandler {
public:
    RtxTimer(TcpAgent *a) : TimerHandler() { a_ = a; }
protected:
    virtual void expire(Event *e);
    TcpAgent *a_;
};

class DelSndTimer : public TimerHandler {
public:
    DelSndTimer(TcpAgent *a) : TimerHandler() { a_ = a; }
protected:
    virtual void expire(Event *e);
    TcpAgent *a_;
};

class BurstSndTimer : public TimerHandler {
public:
    BurstSndTimer(TcpAgent *a) : TimerHandler() { a_ = a; }
protected:
    virtual void expire(Event *e);
    TcpAgent *a_;
};
```

In the constructor for `TcpAgent` in `tcp.cc`, each of these timers is initialized with the `this` pointer, which is assigned to the pointer `a_`.

```
TcpAgent::TcpAgent() : Agent(PT_TCP), rtt_active_(0), rtt_seq_(-1),
    ...
    rtx_timer_(this), delsnd_timer_(this), burstsnd_timer_(this)
{
    ...
}
```

In the following, we will focus only on the retransmission timer. Various helper methods may be defined to schedule timer events; *e.g.*,

```
/*
 * Set retransmit timer using current rtt estimate. By calling resched()
 * it does not matter whether the timer was already running.
 */
void TcpAgent::set_rtx_timer()
{
    rtx_timer_.resched(rtt_timeout());
}

/*
 *   Set new retransmission timer if not all outstanding
 *   data has been acked. Otherwise, if a timer is still
 *   outstanding, cancel it.
 */
void TcpAgent::newtimer(Packet* pkt)
```

```
{
    hdr_tcp *tcph = (hdr_tcp*)pkt->access(off_tcp_);
    if (t_seqno_ > tcph->seqno())
        set_rtx_timer();
    else if (rtx_timer_.status() == TIMER_PENDING)
        rtx_timer_.cancel();
}
```

In the above code, the `set_rtx_timer`() method reschedules the retransmission timer by calling `rtx_timer_.resched`(). Note that if it is unclear whether or not the timer is already running, calling `resched`() eliminates the need to explicitly cancel the timer. In the second function, examples are given of the use of the `status`() and `cancel`(void) methods.

Finally, the `expire`(void) method for class `RtxTimer` must be defined. In this case, `expire`(void) calls the `timeout`(void) method for `TcpAgent`. This is possible because `timeout`() is a public member function; if it were not, then `RtxTimer` would have had to have been declared a friend class of `TcpAgent`.

```
void TcpAgent::timeout(int tno)
{
    /* retransmit timer */
    if (tno == TCP_TIMER_RTX) {
        if (highest_ack_ == maxseq_ && !slow_start_restart_) {
            /*
             * TCP option:
             * If no outstanding data, then don't do anything.
             */
            return;
        };
        recover_ = maxseq_;
        recover_cause_ = 2;
        closecwnd(0);
        reset_rtx_timer(0,1);
        send_much(0, TCP_REASON_TIMEOUT, maxburst_);
    } else {
        /*
         * delayed-send timer, with random overhead
         * to avoid phase effects
         */
        send_much(1, TCP_REASON_TIMEOUT, maxburst_);
    }
}

void RtxTimer::expire(Event *e) {
    a_->timeout(TCP_TIMER_RTX);
}
```

The various TCP agents contain additional examples of timers.

9.2 OTcl Timer class

A simple timer class is defined in ~*ns*/tcl/ex/timer.tcl. Subclasses of `Timer` can be defined as needed. Unlike the C++ timer API, where a `sched()` aborts if the timer is already set, `sched()` and `resched()` are the same; i.e., no state is kept for the OTcl timers. The following methods are defined in the `Timer` base class:

```
$self sched $delay      ;# causes "$self timeout" to be called $delay seconds in the future
$self resched $delay    ;# same as "$self sched $delay"
$self cancel            ;# cancels any pending scheduled callback
$self destroy           ;# same as "$self cancel"
$self expire            ;# calls "$self timeout" immediately
```

Chapter 10

Packet Headers and Formats

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

Objects in the `class Packet` are the fundamental unit of exchange between objects in the simulation. The class `Packet` provides enough information to link a packet on to a list (*i.e.*, in a `PacketQueue` or on a free list of packets), refer to a buffer containing packet headers that are defined on a per-protocol basis, and to refer to a buffer of packet data. New protocols may define their own packet headers or may extend existing headers with additional fields.

New packet headers are introduced into the simulator by defining a C++ structure with the needed fields, defining a static class to provide OTcl linkage, and then modifying some of the simulator initialization code to assign a byte offset in each packet where the new header is to be located relative to others.

When the simulator is initialized through OTcl, a user may choose to enable only a subset of the compiled-in packet formats, resulting in a modest savings of memory during the execution of the simulation. Presently, most configured-in packet formats are enabled. The management of which packet formats are currently enabled in a simulation is handled by a special packet header manager object described below. This object supports an OTcl method used to specify which packet headers will be used in a simulation. If an object in the simulator makes use of a field in a header which has not been enabled, a run-time fatal program abort occurs.

10.1 A Protocol-Specific Packet Header

Protocol developers will often wish to provide a specific header type to be used in packets. Doing so allows a new protocol implementation to avoid overloading already-existing header fields. We consider a simplified version of RTP as an example. The RTP header will require a sequence number fields and a source identifier field. The following classes create the needed header (see ~*ns*/rtp.h and ~*ns*/rtp.cc):

From rtp.h:

```
        /* rtp packet. For now, just have srcid + seqno. */
        struct hdr_rtp {
                u_int32_t srcid_;
                int seqno_;
                /* per-field member functions */
                u_int32_t& srcid() { return (srcid_); }
```

```
        int& seqno() { return (seqno_); }
};
```

From rtp.cc:

```
class RTPAgent: public Agent {
        ...
        int off_rtp_;
};

class RTPHeaderClass : public PacketHeaderClass {
public:
        RTPHeaderClass() : PacketHeaderClass("PacketHeader/RTP",
                                             sizeof(hdr_rtp)) {}
} class_rtphdr;

void RTPAgent::sendpkt()
{
        Packet* p = allocpkt();
        hdr_rtp *rh = (hdr_rtp*)p->access(off_rtp_);
        lastpkttime_ = Scheduler::instance().clock();

        /* Fill in srcid_ and seqno */
        rh->seqno() = seqno_++;
        rh->srcid() = session_->srcid();
        target_->recv(p, 0);
}

RTPAgent::RTPAgent()
        : session_(0), lastpkttime_(-1e6)
{
        type_ = PT_RTP;
        bind("seqno_", &seqno_);
        bind("off_rtp_", &off_rtp_);
}
```

The first structure defines the layout (in terms of words and their placement): which fields are needed and how big they are. This structure definition is only used by the compiler to compute byte offsets of fields; no objects of this structure type are ever directly allocated. The structure also provides member functions which in turn provide a layer of data hiding for objects wishing to read or modify header fields of packets. Note that the variable `off_rtp_` is used to find the byte offset at which the rtp header is located in an arbitrary packet. To access any packet header other than the "common" header (see below, Section 10.2.2), the accessing code must obtain the appropriate header offset. This is accomplished by declaring and binding the integer variable `off_`⟨`hdrname`⟩`_` where ⟨`hdrname`⟩ refers to a shorthand name of the header of interest which must match the name assigned in ~*ns*/tcl/lib/ns-packet.tcl. This is performed above by the RTPAgent's constructor. Generally, one header object for each type of header in the simulation is instantiated at simulator run-time. A particular header is enabled via OTcl in the simulation during simulator configuration time (see Section 10.2.3).

The static object `class_rtphdr` of `class RTPHeaderClass` is used to provide linkage to OTcl when the RTP header is enabled at configuration time. When the simulator executes, this static object calls the `PacketHeaderClass` constructor with arguments `"PacketHeader/RTP"` and `sizeof(hdr_rtp)`. This causes the size of the RTP header to be stored and made available to the packet header manager at configuration time (see below, Section 10.2.3).

The sample member function `sendpkt()` method of `RTPAgent` creates a new packet to send by calling `allocpkt()`, which handles assignment of all the network-layer packet header fields (in this case, IP). Headers other than IP are handled separately. In this case, the agent uses the `RTPHeader` defined above. The `Packet::access`(void) member function returns the address of the first byte in a buffer used to hold header information (see below). Its return value is cast as a pointer to the header of interest, after which member functions of the `RTPHeader` object are used to access individual fields.

10.1.1 Adding a New Packet Header Type

Assuming we wish to create a new header called `newhdr` the following steps are performed:

1. create a new structure defining the raw fields (called `hdr_newhdr`)
2. define member functions for needed fields
3. create a static class to perform OTcl linkage (defines `PacketHeader/Newhdr`)
4. edit ~*ns*/tcl/lib/ns-packet.tcl to enable new packet format (see 10.2.3)

10.2 Packet Classes

There are three C++ classes relevant to the handling of packets and packet headers in general: `Packet`, `PacketHeader`, and `PacketHeaderManager`. The `class Packet` defines the type for all packets in the simulation; it is a subclass of `Event` so that packets may be scheduled (e.g. for later arrival at some queue). The `class PacketHeader` provides a base class for any packet header configured into the simulation. It essentially provides enough internal state to locate any particular packet header in the collection of packet headers present in any given packet. The `class PacketHeaderManager` defines a class used to collect and manage currently-configured headers. It is invoked by a method available to OTcl at simulation configuration time to enable some subset of the compiled-in packet headers.

10.2.1 The Packet Class

The class Packet defines the structure of a packet and provides member functions to handle a free list for objects of this type. It is illustrated in Figure 10.1 and defined as follows in `packet.h`:

```
class Packet : public Event {
private:
        friend class PacketQueue;
        u_char* bits_;
        u_char* data_;                          /* variable size buffer for 'data' */
        u_int datalen_;                         /* length of variable size buffer */
protected:
        static Packet* free_;
public:
        Packet* next_;                          /* for queues and the free list */
        static int hdrlen_;
        Packet() : bits_(0), datalen_(0), next_(0) {}
        u_char* const bits() { return (bits_); }
        Packet* copy() const;
        static Packet* alloc();
        static Packet* alloc(int);
        inline void allocdata(int);
```

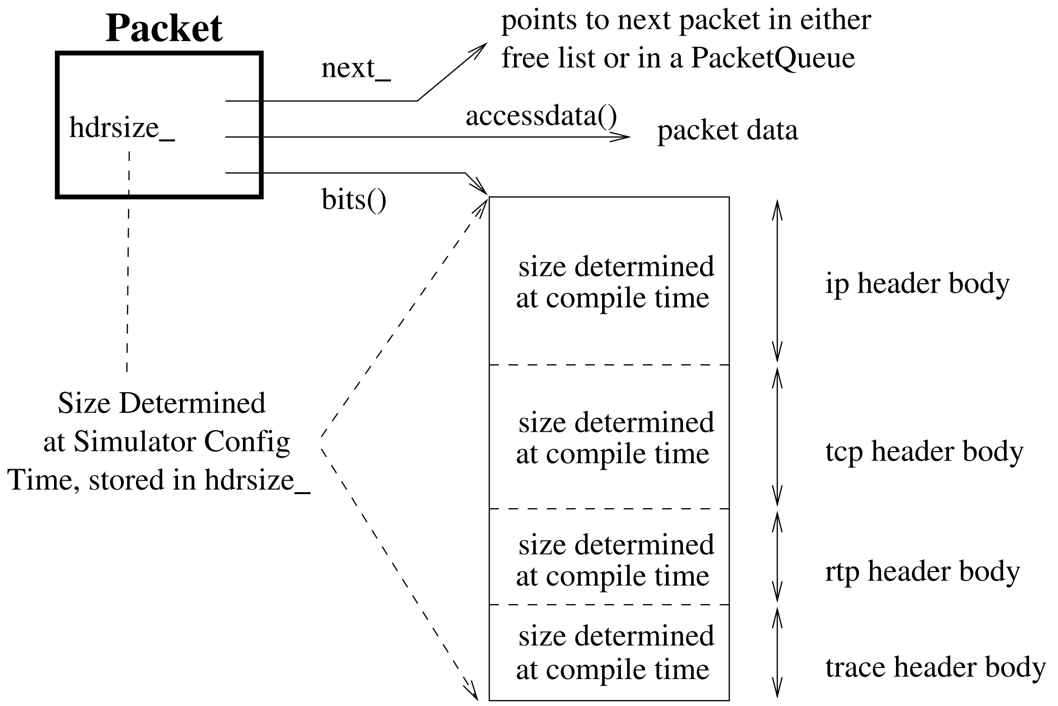


Figure 10.1: A Packet Object

```
        static void free(Packet*);
        inline u_char* access(int off) {
                if (off < 0)
                        abort();
                return (&bits_[off]);
        }
        inline u_char* accessdata() { return data_; }
};
```

This class holds a pointer to a generic array of unsigned characters (commonly called the "bag of bits" or BOB for short) where packet header fields are stored. It also holds a pointer to packet "data" (which is often not used in simulations). The `bits_` variable contains the address of the first byte of the BOB. Effectively BOB is (currently implemented as) a concatenation of all the structures defined for each packet header (by convention, the structures with names beginning `hdr_`⟨`something`⟩) that have been configured in. BOB generally remains a fixed size throughout a simulation, and the size is recorded in the `Packet::hdrlen_` member variable. This size is updated during simulator configuration by OTcl[1].

The other methods of the class Packet are for creating new packets and storing old (unused) ones on a private free list. Such allocation and deallocation is performed by the following code (in ~*ns*/packet.h):

```
inline Packet* Packet::alloc()
{
        Packet* p = free_;
        if (p != 0)
                free_ = p->next_;
        else {
                p = new Packet;
```

[1] It is not intended to be updated after configuration time. Doing so *should* be possible, but is currently untested.

```
                p->bits_ = new u_char[hdrsize_];
                if (p == 0 || p->bits_ == 0)
                        abort();
        }
        return (p);
}

/* allocate a packet with an n byte data buffer */
inline Packet* Packet::alloc(int n)
{
        Packet* p = alloc();
        if (n > 0)
               p->allocdata(n);
        return (p);
}

/* allocate an n byte data buffer to an existing packet */
inline void Packet::allocdata(int n)
{
        datalen_ = n;
        data_ = new u_char[n];
        if (data_ == 0)
                abort();

}

inline void Packet::free(Packet* p)
{
        p->next_ = free_;
        free_ = p;
        if (p->datalen_) {
                delete p->data_;
                p->datalen_ = 0;
        }
}

inline Packet* Packet::copy() const
{
        Packet* p = alloc();
        memcpy(p->bits(), bits_, hdrlen_);
        if (datalen_) {
                p->datalen_ = datalen_;
                p->data_ = new u_char[datalen_];
                memcpy(p->data_, data_, datalen_);
        }
        return (p);
}
```

The `alloc()` method is a support function commonly used to create new packets. It is called by `Agent::allocpkt()` method on behalf of agents and is thus not normally invoked directly by most objects. It first attempts to locate an old packet on the free list and if this fails allocates a new one using the C++ `new` operator. Note that `Packet` class objects and BOBs are allocated separately. The `free()` method frees a packet by returning it to the free list. Note that *packets are never returned to the system's memory allocator*. Instead, they are stored on a free list when `Packet::free()` is called. The `copy()` member creates a new, identical copy of a packet with the exception of the `uid_` field, which is unique. This function is used by

Replicator objects to support multicast distribution and LANs.

10.2.2 The hdr_cmn Class

Each packet in the simulator has a "common" header which is defined in ~*ns*/packet.h as follows:

```
struct hdr_cmn {
        double  ts_;                 /* timestamp: for q-delay measurement */
        int     ptype_;              /* packet type (see above) */
        int     uid_;                /* unique id */
        int     size_;               /* simulated packet size */
        int     iface_;              /* receiving interface (label) */

        /*  per-field member functions  */
        int& ptype() { return (ptype_); }
        int& uid() { return (uid_); }
        int& size() { return (size_); }
        int& iface() { return (iface_); }
        double& timestamp() { return (ts_); }
};
```

This structure primarily defines fields used for tracing the flow of packets or measuring other quantities. The time stamp field is used to measure queuing delay at switch nodes. The ptype_ field is used to identify the type of packets, which makes reading traces simpler. The uid_ field is used by the scheduler in scheduling packet arrivals. The size_ field is of general use and gives the simulated packet's size. Note that the actual number of bytes consumed in the simulation may not relate to the value of this field. Rather, it is used most often in computing the time required for a packet to be delivered along a network link. The iface_ field is used by the simulator when performing multicast distribution tree computations. It is a label indicating (typically) on which link a packet was received.

10.2.3 The PacketHeaderManager Class

An object of the class PacketHeaderManager is used to manage the set of currently-active packet header types and assign each of them unique offsets in the BOB. It is defined in both the C++ and OTcl code:

From tcl/lib/ns-packet.tcl:

```
PacketHeaderManager set hdrlen_ 0

#XXX could potentially get rid of this by searching having a more
# uniform offset concept...
foreach pair {
                { Common off_cmn_ }
                { Mac off_mac_ }
                { LL off_ll_ }
                { Snoop off_snoop_ }
                { IP off_ip_ }
                { TCP off_tcp_ }
                { TCPA off_tcpasym_ }
```

```
                { Flags off_flags_ }
                { RTP off_rtp_ }
                { Message off_msg_ }
                { IVS off_ivs_ }
                { rtProtoDV off_DV_ }
                { CtrMcast off_CtrMcast_ }
                { Prune off_prune_ }
                { Tap off_tap_ }
                { aSRM off_asrm_ }
                { SRM off_srm_ }} {
        set cl [lindex $pair 0]
        set var [lindex $pair 1]
        PacketHeaderManager set vartab_($cl) $var
}

Simulator instproc create_packetformat { } {
        set pm [new PacketHeaderManager]
        foreach oclass [PacketHeader info subclass] {
                set L [split $oclass /]
                set cl [lindex $L 1]
                set var [PacketHeaderManager set vartab_($cl)]
                set off [$pm allochdr $cl]
                TclObject set $var $off
        }
        $self set packetManager_ $pm
}
PacketHeaderManager instproc allochdr cl {
        set size [PacketHeader/$cl set hdrlen_]

        $self instvar hdrlen_
        set NS_ALIGN 8
        # round up to nearest NS_ALIGN bytes
        set incr [expr ($size + ($NS_ALIGN-1)) & ~($NS_ALIGN-1)]
        set base $hdrlen_
        incr hdrlen_ $incr

        return $base
}
```

From packet.cc:

```
/* manages active packet header types */
class PacketHeaderManager : public TclObject {
public:
        PacketHeaderManager() {
                bind("hdrlen_", &Packet::hdrlen_);
        }
};
```

The code in ~*ns*/tcl/lib/ns-packet.tcl is executed when the simulator initializes. Thus, the `foreach` statement is executed before the simulation begins, and initializes the OTcl class array `vartab_` to contain the mapping between class the name and the name of the variable used to contain that class's header in a packet (which is initialized later). For example, the value of `vartab_(IP)` is set to `off_ip_`.

The `create_packetformat{}` instance procedure is part of the basic Simulator class and is called one time during simulator configuration. It first creates a single `PacketHeaderManager` object. The C++ constructor links the OTcl instance variable `hdrlen_` (of class `PacketHeaderManager`) to the C++ variable `Packet::hdrlen_` (a static member of the `Packet` class). This has the effect of setting `Packet::hdrlen_` to zero. Note that binding across class types in this fashion is unusual.

After creating the packet manager, the `foreach` loop enables each of the packet headers of interest. This loop iterates through the list of defined packet headers of the form (h_i, o_i) where h_i is the name of the ith header and o_i is the name of the variable containing the location of the h_i header in BOB. The placement of headers is performed by the `allochdr` instproc of the `PacketHeaderManager` OTcl class. The procedure keeps a running variable `hdrlen_` with the current length of BOB as new packet headers are enabled. It also arranges for 8-byte alignment for any newly-enabled packet header. This is needed to ensure that when double-world length quantities are used in packet headers on machines where double-word alignment is required, access faults are not produced.[2].

[2] In some processer architectures, including the Sparc and HP-PA, double-word access must be performed on a double-word boundary (i.e. addresses ending in 0 mod 8). Attempting to perform unaligned accesses result in an abnormal program termination.

Chapter 11

Error Model

The procedures and functions described in this chapter can be found in ~*ns*/errmodel.{cc, h}.

Error model simulates link-level errors or loss by either marking the packet's error flag or dumping the packet to a drop target. In simulations, errors can be generated from a simple model such as the packet error rate, or from more complicated statistical and empirical models. To support a wide variety of models, the unit of error can be specified in term of packet, bits, or time-based.

The `ErrorModel` class is derived from the `Connector` base class. As the result, it inherits some methods for hooking up objects such as `target` and `drop-target`. If the drop target exists, it will received corrupted packets from `ErrorModel`. Otherwise, `ErrorModel` just marks the `error_` flag of the packet's common header, thereby, allowing agents to handle the loss. The `ErrorModel` also defines additional Tcl method `unit` to specify the unit of error and `ranvar` to specify the random variable for generating errors. If not specified, the unit of error will be in packets, and the random variable will be uniform distributed from 0 to 1. Below is a simple example of creating an error model with the packet error rate of 1 percent (0.01):

```
# create a loss_module and set its packet error rate to 1 percent
set loss_module [new ErrorModel]
$loss_module set rate_ 0.01

# optional: set the unit and random variable
$loss_module unit pkt                                ;# error unit: packets (the default)
$loss_module ranvar [new RandomVariable/Uniform]

# set target for dropped packets
$loss_module drop-target [new Agent/Null]
```

In C++, the `ErrorModel` contains both the mechanism and policy for dropping packets. The packet dropping mechanism is handled by the `recv` method, and packet corrupting policy is handled by the `corrupt` method.

```
enum ErrorUnit { EU_PKT=0, EU_BIT, EU_TIME };

class ErrorModel : public Connector {
public:
        ErrorModel();
        void recv(Packet*, Handler*);
```

```
        virtual int corrupt(Packet*);
        inline double rate() { return rate_; }
protected:
        int command(int argc, const char*const* argv);
        ErrorUnit eu_;                              /* error unit in pkt, bit, or time */
        RandomVariable* ranvar_;
        double rate_;
};
```

The ErrorModel only implements a simple policy based on a single error rate, either in packets of bits. More sophisticated dropping policy can be implemented in C++ by deriving from ErrorModel and redefining its corrupt method.

Chapter 12

Local Area Networks

The characteristics of the wireless and local area networks (LAN) are inherently different from those of point-to-point links. A network consisting of multiple point-to-point links cannot capture the sharing and contention properties of a LAN. To simulate these properties, we created a new type of a Node, called `LanNode`. The OTcl configurations and interfaces for `LanNode` reside in the following two files in the main *ns* directory:

```
tcl/lan/vlan.tcl
tcl/lan/ns-ll.tcl
tcl/lan/ns-mac.tcl
```

12.1 Tcl configuration

The interface for creating and configuring a LAN slightly differs from those of point-to-point link. At the top level, the OTcl class `Simulator` exports a new method called `make-lan`. The parameters to this method are similar to the method `duplex-link`, except that `make-lan` only accepts a list of nodes as a single parameter instead of 2 parameters as in `duplex-link`:

```
Simulator instproc make-lan {nodes bw delay lltype ifqtype mactype chantype}
```

The optional parameters to `make-lan` specify the type of objects to be created for the link layer (`LL`), the interface queue, the MAC layer (`Mac`), and the physical layer (`Channel`). Below is an example of how a new CSMA/CD (Ethernet) LAN is created.

Example:

```
$ns make-lan "$n1 $n2" $bw $delay LL Queue/DropTail Mac/Csma/Cd
```

creates a LAN with basic link-layer, drop-tail queue, and CSMA/CD MAC.

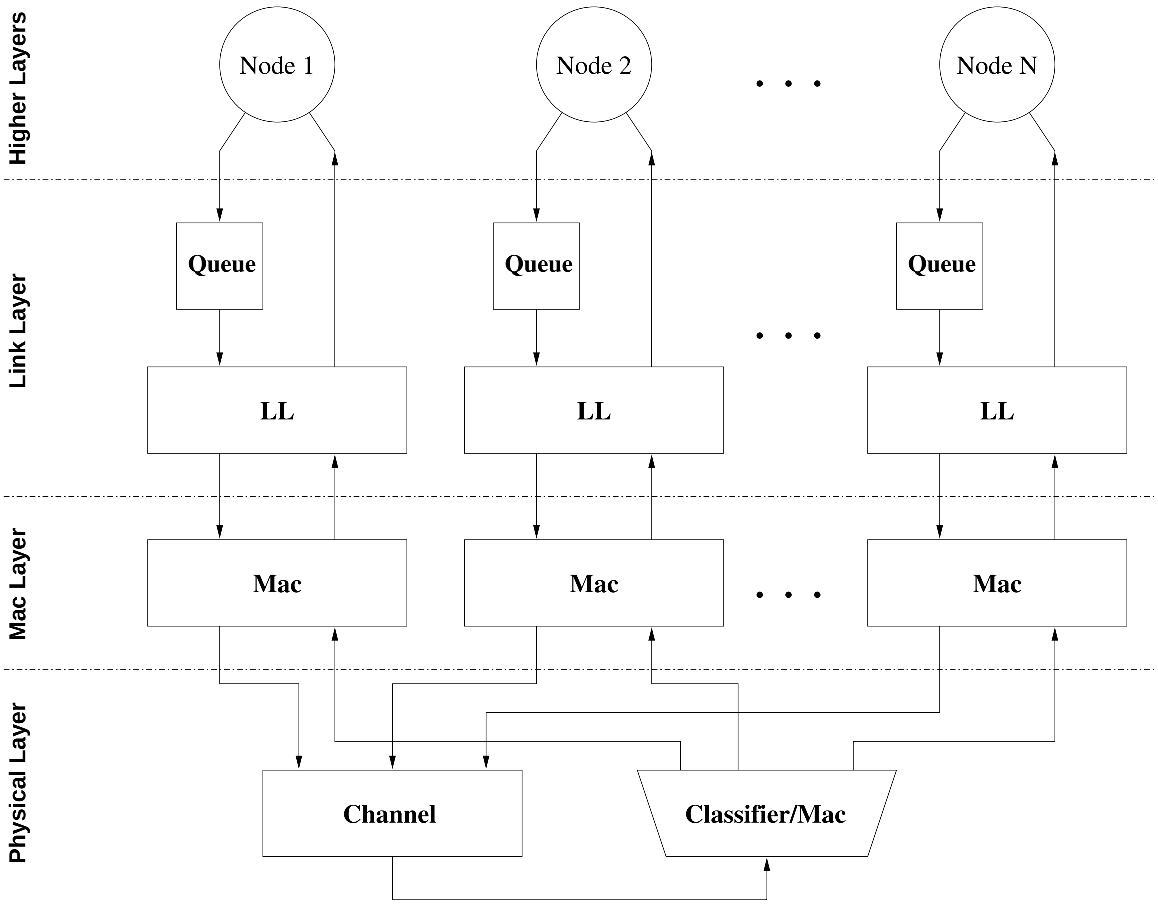


Figure 12.1: Connectivity within a LAN

12.2 Components of a LAN

LanLink captures the functionality of the three lowest layers in the network stack:

1. Link Layer (LL)
2. Medium Access Control (MAC) Layer
3. Physical (PHY) Layer

Figure 12.1 illustrates the extended network stack that makes simulations of local area network possible in *ns*. A packet sent down the stack flows through the link layer (`Queue` and `LL`), the MAC layer (`Mac`), and the physical layer (`Channel` to `Classifier/Mac`). The packet then makes its way up the stack through the `Mac`, and the `LL`.

At the bottom of the stack, the physical layer is composed of two simulation objects: the `Channel` and `Classifier/Mac`. The `Channel` object simulates the shared medium and supports the medium access mechanisms of the MAC objects on the sending side of the transmission. On the receiving side, the `Classifier/Mac` is responsible for delivering and optionally replicating packets to the receiving MAC objects.

Depending on the type of physical layer, the MAC layer must contain a certain set of functionalities such as: carrier sense, collision detection, collision avoidance, etc. Since these functionalities affect both the sending and receiving sides, they are implemented in a single Mac object. For sending, the Mac object must follow a certain medium access protocol before transmitting the packet on the channel. For receiving, the MAC layer is responsible for delivering the packet to the link layer.

Above the MAC layer, the link layer can potentially have many functionalities such as queuing and link-level retransmission. The need of having a wide variety of link-level schemes leads to the division of functionality into two components: Queue and LL (link-layer). The Queue object, simulating the interface queue, belongs to the same Queue class that is described in Chapter 6. The LL object implements a particular data link protocol, such as ARQ. By combining both the sending and receiving functionalities into one module, the LL object can also support other mechanisms such as piggybacking.

12.3 Channel Class

The Channel class simulates the actual transmission of the packet at the physical layer. The basic Channel implements a shared medium with support for contention mechanisms. It allows the MAC to carry out carrier sense, contention, and collision detection. If more than one transmissions overlaps in time, a channel raises the collision flag. By checking this flag, the MAC object can implement collision detection and handling.

Since the transmission time is a function of the number of bits in the packet and the modulation speed of each individual interface (MAC), the Channel object only sets its busy signal for the duration requested by the MAC object. It also schedules the packets to be delivered to the destination MAC objects after the transmission time plus the propagation delay.

12.3.1 Channel State

The C++ class Channel includes enough internal state to schedule packet delivery and detect collisions. It exports the following OTcl configuration parameter:

delay_ propagation delay on the channel

12.3.2 Example: Channel and classifier of the physical layer

```
set channel_ [new Channel]
$channel_ set delay_ 4us        # propagation delay

set mcl_ [new Classifier/Mac]
$channel_ target $mcl_
$mcl_ install $mac_DA $recv_iface
        . . .
```

12.3.3 Channel Class in C++

In C++, the class Channel extends the Connector object with several new methods to support a variety of MAC protocols. The class is defined as follow in ~*ns*/channel.h:

```
class Channel : public Connector {
public:
     Channel();
```

```
        void recv(Packet* p, Handler*);
        virtual int send(Packet* p, double txtime);
        virtual void contention(Packet*, Handler*);
        int hold(double txtime);
        virtual int collision() { return numtx_ > 1; }
        virtual double txstop() { return txstop_; }
                . . .
};
```

The important methods of the class `Channel` are:

- `txstop()` method returns the time when the channel will become idle, which can be used by the MAC to implement carrier sense.
- `contention()` method allows the MAC to contend for the channel before sending a packet. The channel then use this packet to signal the corresponding `Mac` object at the end of each contention period.
- `collision()` method indicates whether a collision occurs during the contention period. When the `Channel` signal the end of the contention period, the MAC can use the `collision()` method to detect collision.
- `send()` method allows the MAC object to transmit a packet on the channel for a specified duration of time.
- `hold()` method allows the MAC object to hold the channel for a specified duration of time without actually transmitting any packets. This is useful in simulating the jamming mechanism of some MAC protocols.

12.4 MacClassifier Class

The `MacClassifier` class extends the `Classifier` class to implement a simple broadcasting mechanism. It modifies the `recv()` method in the following way: since the replication of a packet is expensive, normally a unicast packet will be classified by the MAC destination address `macDA_` and delivered directly to the MAC object with such an address. However, if the destination object cannot be found or if the MAC destination address is explicitly set to the broadcast address `BCAST_ADDR`, the packet will be replicated and sent to all MACs on the lan excluding the one that is the source of the packet. Finally, by setting the bound variable `MacClassifier::bcast_` to a non–zero value, will cause `MacClassifier` always to replicate packets.

```
class MacClassifier : public Classifier {
public:
    void recv(Packet*, Handler*);
};

void MacClassifier::recv(Packet* p, Handler*)
{
    Mac* mac;
    hdr_mac* mh = hdr_mac::access(p);

    if (bcast_ || mh->macDA() == BCAST_ADDR || (mac = (Mac *)find(p)) == 0) {
            // Replicate packets to all slots (broadcast)
            . . .
            return;
    }
    mac->recv(p);
}
```

12.5 MAC Class

The Mac object simulates the medium access protocols that are necessary in the shared medium environment such as the wireless and local area networks. Since the sending and receiving mechanisms are tightly coupled in most types of MAC layers, it is essential for the Mac object to be duplex.

On the sending side, the Mac object is responsible for adding the MAC header and transmitting the packet onto the channel. On the receiving side, the Mac object asynchronously receives packets from the classifier of the physical layer. After MAC protocol processing, it passes the data packet to the link layer.

12.5.1 Mac State

The C++ class Mac class contains enough internal state to simulate the particular MAC protocol. It also exports the following OTcl configuration parameter:

bandwidth_	modulation rate of the MAC
hlen_	additional bytes added to packet for MAC header
label_	MAC address

12.5.2 Mac Methods

The class Mac class added several Tcl methods for configuration, in particular, linking with other simulation objects:

channel	specify the channel for transmission
classifier	the classifier that deliver packets to receiving MAC
maclist	a link list of MAC interfaces on the same node

12.5.3 Mac Class in C++

In C++, the Mac class derives from Connector. When the recv() method gets a packet, it identifies the direction of the packet based on the presence of a callback handler. If there is a callback handler, the packet is outgoing, otherwise, it is incoming.

```
class Mac : public Connector {
public:
     Mac();
     virtual void recv(Packet* p, Handler* h);
     virtual void send(Packet* p);
     virtual void resume(Packet* p = 0);
               . . .
 };
```

When a Mac object receives a packet via its recv() method, it checks whether the packet is outgoing or incoming. For an outgoing packet, it assumes that the link-layer of the sender has obtained the destination MAC address and filled in the macDA_ field of the MAC header, hdr_mac. The Mac object fills in the rest of the MAC header with the source MAC address and the frame type. It then passes the packet to its `send()` method, which carries out the medium access protocol. For the basic `Mac` object, the `send` method calls `txtime()` to compute the transmission time, then invokes `Channel::send` to transmit the packet. Finally, it schedules itself to resume after the transmission time has elapsed.

For an incoming packet, the MAC object does its protocol processing and passes the packet to the link-layer.

12.5.4 CSMA-based MAC

The `class CsmaMac` extends the `Mac` class with new methods that implements carrier sense and backoff mechanisms. The `CsmaMac::send()` method detects when the channel becomes idle using `Channel::txtime()`. If the channel is busy, the MAC schedules the next carrier sense at the moment the channel turns idle. Once the channel is idle, the `CsmaMac` object initiates the contention period with `Channel::contention()`. At the end of the contention period, the `endofContention()` method is invoked. At this time, the basic `CsmaMac` just transmits the packet using `Channel::send`.

```
class CsmaMac : public Mac {
public:
    CsmaMac();
    void send(Packet* p);
    void resume(Packet* p = 0);
    virtual void endofContention(Packet* p);
    virtual void backoff(Handler* h, Packet* p, double delay=0);
            . . .
};

class CsmaCdMac : public CsmaMac {
public:
    CsmaCdMac();
    void endofContention(Packet*);
};

class CsmaCaMac : public CsmaMac {
public:
    CsmaCaMac();
    virtual void send(Packet*);
};
```

The `CsmaCdMac` extends `CsmaMac` to carry out collision detection procedure of the CSMA/CD (Ethernet) protocol. When the channel signals the end of contention period, the `endofContention` method checks for collision using the `Channel::collision` method. If there is a collision, the MAC invokes its `backoff` method to schedule the next carrier sense to retransmit the packet.

The `CsmaCaMac` extends the `send` method of `CsmaMac` to carry out the collision avoidance (CSMA/CA) procedure. Instead of transmitting immediately when the channel is idle, the `CsmaCaMac` object backs off a random number of slots, then transmits if the channel remains idle until the end of the backoff period.

12.6 LL (link-layer) Class

The link-layer object is responsible for simulating the data link protocols. Many protocols can be implemented within this layer such as packet fragmentation and reassembly, and reliable link protocol.

Another important function of the link layer is setting the MAC destination address in the MAC header of the packet. In the current implementation this task involves two separate issues: finding the next–hop–node's IP address (routing) and resolving this IP address into the correct MAC address (ARP). For simplicity, the default mapping between MAC and IP addresses is one–to–one, which means that IP addresses are re–used at the MAC layer.

12.6.1 LL Class in C++

The C++ class `LL` derives from the `LinkDelay` class. Since it is a duplex object, it keeps a separate pointer for the send target, `sendtarget`, and the receive target, `recvtarget`. It also defines the methods `recvfrom()` and `sendto()` to handle the incoming and outgoing packets respectively.

```
class LL : public LinkDelay {
public:
    LL();
    virtual void recv(Packet* p, Handler* h);
    virtual Packet* sendto(Packet* p, Handler* h = 0);
    virtual Packet* recvfrom(Packet* p);

    inline int seqno()  return seqno_;
    inline int ackno()  return ackno_;
    inline int macDA()  return macDA_;
    inline Queue *ifq()  return ifq_;
    inline NsObject* sendtarget()  return sendtarget_;
    inline NsObject* recvtarget()  return recvtarget_;

protected:
    int command(int argc, const char*const* argv);
    void handle(Event* e)  recv((Packet*)e, 0);
    inline virtual int arp (int ip_addr)  return ip_addr;
    int seqno_; // link-layer sequence number
    int ackno_; // ACK received so far
    int macDA_; // destination MAC address
    Queue* ifq_; // interface queue
    NsObject* sendtarget_; // for outgoing packet
    NsObject* recvtarget_; // for incoming packet

    LanRouter* lanrouter_; // for lookups of the next hop
};
```

12.6.2 Example: Link Layer configuration

```
set ll_  [new LL]
set ifq_ [new Queue/DropTail]
$ll_ lanrouter  [new LanRouter $ns $lan] # LanRouter is one object
```

```
                                        # per LAN
    $ll_ set delay_ $delay        # link-level overhead
    $ll_ set bandwidth_ $bw       # bandwidth
    $ll_ sendtarget $mac          # interface queue at the sender side
    $ll_ recvtarget $iif          # input interface of the receiver
            . . .
```

12.7 LanRouter class

By default, there is just one `LanRouter` object per LAN, which is created when a new `LanNode` is initialized. For every node on the LAN, the link layer object (`LL`) has a pointer to the `LanRouter`, so it is able to find the next hop for the packet that is sent on the LAN:

```
Packet* LL::sendto(Packet* p, Handler* h)
{
        int nh = (lanrouter_) ? lanrouter_->next_hop(p) : -1;
        . . .
}
```

`LanRouter` is able to find the next hop by querying the current `RouteLogic`:

```
int LanRouter::next_hop(Packet *p) {
        int next_hopIP;
        if (enableHrouting_) {
                routelogic_->lookup_hier(lanaddr_, adst, next_hopIP);
        } else {
                routelogic_->lookup_flat(lanaddr_, adst, next_hopIP);
        }
```

One limitation of this is that `RouteLogic` may not be aware of dynamic changes to the routing. But it is always possible to derive a new class from `LanRouter` so that to re–define its `next_hop` method to handle dynamic changes appopriately.

12.8 Other Components

In addition to the C++ components described above, simulating local area networks also requires a number of existing components in *ns* such as `Classifier`, `Queue`, and `Trace`, `networkinterface`, etc. Configuring these objects requires knowledge of what the user wants to simulate. The default configuration is implemented in the two Tcl files mentioned at the beginning of this chapter. To obtain more realistic simulations of wireless networks, one can use the `ErrorModel` described in Chapter 11.

12.9 LANs and *ns* routing

When a LAN is created using either `make-lan` or `newLan`, a "*virtual LAN node*" `LanNode` is created. `LanNode` keeps together all shared objects on the LAN: `Channel`, `Classifier/Mac`, and `LanRouter`. Then for each node on the LAN, a LanIface object is created. LanIface contains all other objects that are needed on the per–node basis: a Queue, a link layer (LL), Mac, etc. It should be emphasized that LanNode is a node only for routing algorithms: Node and LanNode have very little in common. One of few things that they share is an identifier taken from the Node ID–space. If *hierarchical routing* is used, LanNode *has to be assigned a hierarchical address* just like any other node. From the point of view of *ns* (static) routing, LanNode is just another node connected to every node on the LAN. Links connecting the LanNode with

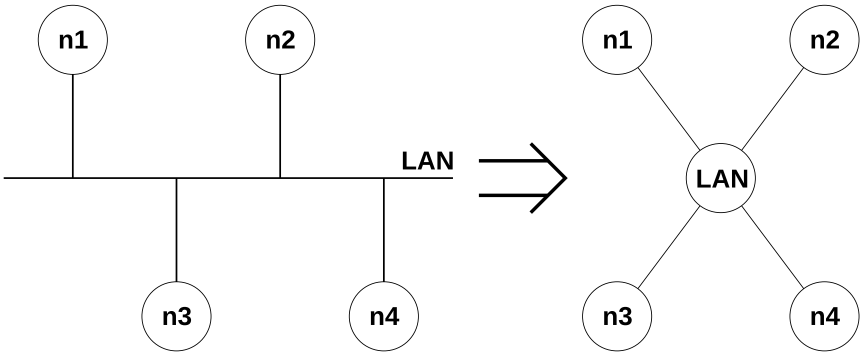


Figure 12.2: Actual LAN configuration (left) and as seen by *ns* routing (right)

the nodes on the LAN are also "virtual" (Vlink). The default routing cost of such a link is $1/2$, so the cost of traversing two Vlinks (e.g. **n1** → **LAN** → **n2**) is counted as just one hop.

Most important method of Vlink is the one that gives the head of the link:

```
Vlink instproc head {} {
    $self instvar lan_ dst_ src_
    if {$src_ == [$lan_ set id_]} {
        # if this is a link FROM the lan vnode,
        # it doesn't matter what we return, because
        # it's only used by $lan add-route (empty)
        return ""
    } else {
        # if this is a link TO the lan vnode,
        # return the entry to the lanIface object
        set src_lif [$lan_ set lanIface_($src_)]
        return [$src_lif entry]
    }
}
```

This method is used by static (default) routing to install correct routes at a node (see Simulator methods compute-flat-routes and compute-hier-routes in tcl/lib/ns-route.tcl, as well as Node methods add-route and add-hroute in tcl/lib/ns-node.tcl).

From the code fragment above it can be seen that it returns LAN interface of the node as a head of the link to be installed in the appropriate classifier.

Thus, Vlink *does not impose any delay on the packet* and serves the only purpose to install LAN interfaces instead of normal links at nodes' classifiers.

Note, that this design allows to have nodes connected by parallel LANs, while in the current implementation it is impossible to have nodes connected by parallel simple links and use them both (the array Simulator instvar link_ holds the link object for each connected pair of source and destination, and it can be only one object per source/destination pair).

Part III

Support

Chapter 13

Mathematical Support

The simulator includes a small collection of mathematical functions used to implement random variate generation and integration. This area of the simulator is currently undergoing some changes.

The procedures and functions described in this chapter can be found in ~*ns*/rng.{cc, h}, ~*ns*/random.{cc, h}, ~*ns*/ranvar.{cc, h}, ~*ns*/pareto.{cc, h}, ~*ns*/expoo.{cc, h}, ~*ns*/tcl/lib/ns-random.cc, and ~*ns*/integrator.{cc, h}.

13.1 Random Number Generation

The `RNG` class contains an implementation of the minimal standard multiplicative linear congruential generator of Park and Miller [13].

Multiple instances of the RNG class can be created to allow a simulation to draw random numbers from independent random number streams. For instance, a user who wants to generate the same traffic (based on some random process) in 2 different simulation experiments that compare different dropping algorithms that are themselves based on random processes may choose to base the traffic generation on one random number stream and the dropping algorithms on another stream. However, when using multiple RNG objects in a simulation care should be taken to insure that they are seeded in such a way as to guarantee that they produce independent, high-quality streams of random numbers. We describe approaches to seed the RNG below.

Most users will be satisfied with a single instance of the RNG. Hence, a default RNG, created at simulator initialization time, is provided.

C++ Support This random number generator is implemented by the RNG class, defined in ~*ns*/rng.h:

```
class RNG : public TclObject {
enum RNGSources { RAW_SEED_SOURCE, PREDEF_SEED_SOURCE, HEURISTIC_SEED_SOURCE };
        ...
        // These are primitive but maybe useful.
        inline int uniform_positive_int() {  // range [0, MAXINT]
                return (int)(stream_.next());
        }
        inline double uniform_double() { // range [0.0, 1.0)
```

```
                return stream_.next_double();
        }

        inline int uniform(int k)
                { return (uniform_positive_int() % (unsigned)k); }
        inline double uniform(double r)
                { return (r * uniform_double());}
        inline double uniform(double a, double b)
                { return (a + uniform(b - a)); }
        inline double exponential()
                { return (-log(uniform_double())); }
        inline double exponential(double r)
                { return (r * exponential());}
        inline double pareto(double scale, double shape)
                { return (scale * (1.0/pow(uniform_double(), 1.0/shape)));}
        ...
};
```

The `uniform_positive_int` method generates random integers in the range $[0, 2^{31} - 1]$. In particular, Additional member functions provide the following random variate generation:

`uniform`(double r)	generate a floating-point number uniformly distributed on $[0, r]$
`uniform`(double a, double b)	generate a floating-point number uniformly distributed on $[a, b]$
`exponential`()	generate a floating-point number exponentially distributed (with parameter 1) on $[0, \infty)$
`integer`(int k)	generate an integer uniformly distributed on $[0, (k - 1)]$

The `Random` class (in random.h) is an older interface to the standard random number stream.

Here's a sample use of RNG modeled on RED. rng_ is an instance of class RNG:

```
        ...
        //  drop probability is computed, pick random number and act
        double u = rng_->uniform_double();
        if (u <= edv_.v_prob) {
                edv_.count = 0;
                if (edp_.setbit)
                        iph->flags() |= IP_ECN;                              /* ip ecn bit */
                else
                        return (1);
        }
        ...
```

Seeding the random number generator When doing simulations often you will either want to get absolutely repeatable (deterministic) results or different results each time. Each approach requires a different seeding method.

To get deterministic behavior, invoke the `set_seed()` method with the first parameter either RAW_SEED_SOURCE or PREDEF_SEED_SOURCE. With a raw seed, the second parameter specifies the numeric seed that will be used (any integer). The alternative is predefined seeds, where the second selects seeds from a table of 64 known good seeds (which happen to be about 33 million steps apart in the default random number generation stream). Predefined seeds are know to provide a good random number stream, but there are limited numbers of them. Raw seeds may not provide a statistically good random number stream but are easy to generate.

To get non-deterministic behavior, pick a seed with either RAW_SEED_SOURCE or HEURISTIC_SEED_SOURCE. For raw streams the second argument specifies the stream. For heuristic streams the second argument is ignored and a seed is generated based on the current time of day and a counter. Both have the caveat that they may not provide a statistically good random number stream. It is very unlikely that any two heuristic seeds will be identical.

OTcl support The RNG class can be accessed from OTcl. For example, a new RNG is created and seeded with:

```
set rng [new RNG]
$rng seed 0                    ;# seeds the RNG heuristically
$rng seed n                    ;# seeds the RNG with value n
$rng next-random               ;# return the next random number
$rng uniform a b               ;# return a number uniformly distributed on [a, b]
$rng integer k                 ;# return an integer uniformly distributed on [0, (k-1)]
$rng exponential               ;# return a number from an exponential distribution with average 1.
```

Currently there is no way to select predefined seeds from OTcl.

13.2 Random Variables

The `class RandomVariable` provides a thin layer of functionality on top of the base random number generator and the default random number stream. It is defined in ~*ns*/ranvar.h:

```
class RandomVariable : public TclObject {
public:
      virtual double value() = 0;
      int command(int argc, const char*const* argv);
      RandomVariable();
protected:
      RNG* rng_;
};
```

Classes derived from this abstract class implement specific distributions. Each distribution is parameterized with the values of appropriate parameters. The value method is used to return a value from the distribution.

The currently defined distributions, and their associated parameters are:

`class UniformRandomVariable`	`min_, max_`
`class ExponentialRandomVariable`	`avg_`
`class ParetoRandomVariable`	`avg_, shape_`
`class ConstantRandomVariable`	`val_`
`class HyperExponentialRandomVariable`	`avg_, cov_`

The RandomVariable class is available in OTcl. For instance, to create a random variable that generates number uniformly on [10, 20]:

```
set u [new RandomVariable/Uniform]
```

```
$u set min_ 10
$u set max_ 20
$u value
```

By default, RandomVariable objects use the default random number generator described in the previous section. The use-rng method can be used to associate a RandomVariable with a non-default RNG:

```
set rng [new RNG]
$rng seed 0

set e [new RandomVariable/Exponential]
$e use-rng $rng
```

13.3 Integrals

The `class Integrator` supports the approximation of (continuous) integration by (discrete) sums; it is defined in ~*ns*/integrator.h as

From integrator.h:

```
class Integrator : public TclObject {
public:
        Integrator();
        void set(double x, double y);
        void newPoint(double x, double y);
        int command(int argc, const char*const* argv);
protected:
        double lastx_;
        double lasty_;
        double sum_;
};
```

From integrator.cc:

```
Integrator::Integrator() : lastx_(0.), lasty_(0.), sum_(0.)
{
        bind("lastx_", &lastx_);
        bind("lasty_", &lasty_);
        bind("sum_", &sum_);
}

void Integrator::set(double x, double y)
{
        lastx_ = x;
        lasty_ = y;
        sum_ = 0.;
}

void Integrator::newPoint(double x, double y)
{
        sum_ += (x - lastx_) * lasty_;
        lastx_ = x;
```

```
        lasty_ = y;
}

int Integrator::command(int argc, const char*const* argv)
{
        if (argc == 4) {
                if (strcmp(argv[1], "newpoint") == 0) {
                        double x = atof(argv[2]);
                        double y = atof(argv[3]);
                        newPoint(x, y);
                        return (TCL_OK);
                }
        }
        return (TclObject::command(argc, argv));
}
```

This class provides a base class used by other classes such as `QueueMonitor` that keep running sums. Each new element of the running sum is added by the `newPoint`(x, y) function. After the kth execution of `newPoint`, the running sum is equal to $\sum_{i=1}^{k} y_{i-1}(x_i - x_{i-1})$ where $x_0 = y_0 = 0$ unless `lastx_`, `lasty_`, or `sum_` are reset via OTcl. Note that a new point in the sum can be added either by the C++ member `newPoint` or the OTcl member `newpoint`. The use of integrals to compute certain types of averages (e.g. mean queue lengths) is given in (pp. 429–430, [8]).

Chapter 14

Trace and Monitoring Support

The procedures and functions described in this chapter can be found in ~*ns*/trace.{cc, h}, ~*ns*/tcl/lib/ns-trace.tcl, ~*ns*/queue-monitor.{cc, h}, ~*ns*/tcl/lib/ns-link.tcl, ~*ns*/packet.h, ~*ns*/flowmon.cc, and ~*ns*/classifier-hash.cc.

There are a number of ways of collecting output or trace data on a simulation. Generally, trace data is either displayed directly during execution of the simulation, or (more commonly) stored in a file to be post-processed and analyzed. There are two primary but distinct types of monitoring capabilities currently supported by the simulator. The first, called *traces*, record each individual packet as it arrives, departs, or is dropped at a link or queue. Trace objects are configured into a simulation as nodes in the network topology, usually with a Tcl "Channel" object hooked to them, representing the destination of collected data (typically a trace file in the current directory). The other types of objects, called *monitors*, record counts of various interesting quantities such as packet and byte arrivals, departures, etc. Monitors can monitor counts associated with all packets, or on a per-flow basis using a *flow monitor* below (Section 14.7).

To support traces, there is a special *common* header included in each packet (this format is defined in ~*ns*/packet.h as `hdr_cmn`). It presently includes a unique identifier on each packet, a packet type field (set by agents when they generate packets), a packet size field (in bytes, used to determine the transmission time for packets), and an interface label (used for computing multicast distribution trees).

Monitors are supported by a separate set of objects that are created and inserted into the network topology around queues. They provide a place where arrival statistics and times are gathered and make use of the `class Integrator` (Section 13.3) to compute statistics over time intervals.

14.1 Trace Support

The trace support in OTcl consists of a number of specialized classes visible in OTcl but implemented in C++, combined with a set of Tcl helper procedures and classes defined in the *ns* library.

All following OTcl classes are supported by underlying C++ classes defined in ~*ns*/trace.cc. Objects of the following types are inserted directly in-line in the network topology:

Trace/Hop	trace a "hop" (XXX what does this mean exactly; it is not really used XXX)
Trace/Enque	a packet arrival (usually at a queue)
Trace/Deque	a packet departure (usually at a queue)
Trace/Drop	packet drop (packet delivered to drop-target)
Trace/Recv	packet receive event at the destination node of a link
SnoopQueue/In	on input, collect a time/size sample (pass packet on)
SnoopQueue/Out	on output, collect a time/size sample (pass packet on)
SnoopQueue/Drop	on drop, collect a time/size sample (pass packet on)
SnoopQueue/EDrop	on an "early" drop, collect a time/size sample (pass packet on)

Objects of the following types are added in the simulation and a referenced by the objects listed above. They are used to aggregate statistics collected by the SnoopQueue objects:

QueueMonitor	receive and aggregate collected samples from snoopers
QueueMonitor/ED	queue-monitor capable of distinguishing between "early" and standard packet drops
QueueMonitor/ED/Flowmon	per-flow statistics monitor (manager)
QueueMonitor/ED/Flow	per-flow statistics container
QueueMonitor/Compat	a replacement for a standard QueueMonitor when *ns* v1 compatibility is in use

14.1.1 OTcl Helper Functions

The following helper functions may be used within simulation scripts to help in attaching trace elements (see ~*ns*/tcl/lib/ns-lib.tcl); they are instance procedures of the class Simulator:

`flush-trace {}`	flush buffers for all trace objects in simulation
`create-trace { type file src dst }`	create a trace object of type *type* between the given src and dest nodes. If *file* is non-null, it is interpreted as a Tcl channel and is attached to the newly-created trace object. The procedure returns the handle to the newly created trace object.
`trace-queue { n1 n2 file }`	arrange for tracing on the link between nodes *n1* and *n2*. This function calls create-trace, so the same rules apply with respect to the *file* argument.
`trace-callback{ ns command }`	arranges to call `command` when a line is to be traced. The procedure treats `command` as a string and evaluates it for every line traced. See ~*ns*/tcl/ex/callback_demo.tcl for additional details on usage.
`monitor-queue { n1 n2 }`	this function calls the `init-monitor` function on the link between nodes *n1* and *n2*.
`drop-trace { n1 n2 trace }`	the given *trace* object is made the drop-target of the queue associated with the link between nodes *n1* and *n2*.

The `create-trace{}` procedure is used to create a new `Trace` object of the appropriate kind and attach an Tcl I/O channel to it (typically a file handle). The `src_` and `dst_` fields are are used by the underlying C++ object for producing the trace output file so that trace output can include the node addresses defining the endpoints of the link which is being traced. Note that they are not used for *matching*. Specifically, these values in no way relate to the packet header `src` and `dst` fields, which are also displayed when tracing. See the description of the `Trace` class below (Section 14.3).

The `trace-queue` function enables `Enque`, `Deque`, and `Drop` tracing on the link between nodes `n1` and `n2`. The Link `trace` procedure is described below (Section 14.2).

The monitor-queue function is constructed similarly to trace-queue. By calling the link's init-monitor procedure, it arranges for the creation of objects (SnoopQueue and QueueMonitor objects) which can, in turn, be used to ascertain time-aggregated queue statistics.

The drop-trace function provides a way to specify a Queue's drop target without having a direct handle of the queue.

14.2 Library support and examples

The Simulator procedures described above require the trace and init-monitor methods associated with the OTcl Link class. Several subclasses of link are defined, the most common of which is called SimpleLink. Thus, the trace and init-monitor methods are actually part of the SimpleLink class rather than the Link base class. The trace function is defined as follows (in ns-link.tcl):

```
#
# Build trace objects for this link and
# update the object linkage
#
SimpleLink instproc trace { ns f } {
        $self instvar enqT_ deqT_ drpT_ queue_ link_ head_ fromNode_ toNode_
        $self instvar drophead_

        set enqT_ [$ns create-trace Enque $f $fromNode_ $toNode_]
        set deqT_ [$ns create-trace Deque $f $fromNode_ $toNode_]
        set drpT_ [$ns create-trace Drop $f $fromNode_ $toNode_]

        $drpT_ target [$drophead_ target]
        $drophead_ target $drpT_
        $queue_ drop-target $drpT_

        $deqT_ target [$queue_ target]
        $queue_ target $deqT_

        if { [$head_ info class] == "networkinterface" } {
            $enqT_ target [$head_ target]
            $head_ target $enqT_
            # puts "head is i/f"
        } else {
            $enqT_ target $head_
            set head_ $enqT_
            # puts "head is not i/f"
        }
}
```

This function establishes Enque, Deque, and Drop traces in the simulator $ns and directs their output to I/O handle $f. The function assumes a queue has been associated with the link. It operates by first creating three new trace objects and inserting the Enque object before the queue, the Deque object after the queue, and the Drop object between the queue and its previous drop target. Note that all trace output is directed to the same I/O handle.

This function performs one other additional tasks. It checks to see if a link contains a network interface, and if so, leaves it as the first object in the chain of objects in the link, but otherwise inserts the Enque object as the first one.

The following functions, `init-monitor` and `attach-monitor`, are used to create a set of objects used to monitor queue sizes of a queue associated with a link. They are defined as follows:

```
SimpleLink instproc attach-monitors { insnoop outsnoop dropsnoop qmon } {
        $self instvar queue_ head_ snoopIn_ snoopOut_ snoopDrop_
        $self instvar drophead_ qMonitor_

        set snoopIn_ $insnoop
        set snoopOut_ $outsnoop
        set snoopDrop_ $dropsnoop

        $snoopIn_ target $head_
        set head_ $snoopIn_

        $snoopOut_ target [$queue_ target]
        $queue_ target $snoopOut_

        $snoopDrop_ target [$drophead_ target]
        $drophead_ target $snoopDrop_

        $snoopIn_ set-monitor $qmon
        $snoopOut_ set-monitor $qmon
        $snoopDrop_ set-monitor $qmon
        set qMonitor_ $qmon
}

#
# 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 instproc init-monitor { ns qtrace sampleInterval} {
        $self instvar qMonitor_ ns_ qtrace_ sampleInterval_

        set ns_ $ns
        set qtrace_ $qtrace
        set sampleInterval_ $sampleInterval
        set qMonitor_ [new QueueMonitor]

        $self attach-monitors [new SnoopQueue/In] \
                [new SnoopQueue/Out] [new SnoopQueue/Drop] $qMonitor_

        set bytesInt_ [new Integrator]
        $qMonitor_ set-bytes-integrator $bytesInt_
        set pktsInt_ [new Integrator]
        $qMonitor_ set-pkts-integrator $pktsInt_
        return $qMonitor_
}
```

These functions establish queue monitoring on the `SimpleLink` object in the simulator `ns`. Queue monitoring is implemented by constructing three `SnoopQueue` objects and one `QueueMonitor` object. The `SnoopQueue` objects are linked in around a `Queue` in a way similar to `Trace` objects. The `SnoopQueue/In(Out)` object monitors packet arrivals(departures) and reports them to an associated `QueueMonitor` agent. In addition, a `SnoopQueue/Out` object is also used to accumulate packet drop statistics to an associated QueueMonitor object. For init-monitor the same QueueMonitor object is used in all cases. The C++ definitions of the SnoopQueue and QueueMonitor classes are described below.

14.3 The C++ Trace Class

Underlying C++ objects are created in support of the interface specified in Section 14.3 and are linked into the network topology as network elements. The single C++ Trace class is used to implement the OTcl classes Trace/Hop, Trace/Enque, Trace/Deque, and Trace/Drop. The type_ field is used to differentiate among the various types of traces any particular Trace object might implement. Currently, this field may contain one of the following symbolic characters: **+** for enque, **-** for deque, **h** for hop, and **d** for drop. The overall class is defined as follows in ~*ns*/trace.cc:

```
class Trace : public Connector {
 protected:
        int type_;
        nsaddr_t src_;
        nsaddr_t dst_;
        Tcl_Channel channel_;
        int callback_;
        char wrk_[256];
        void format(int tt, int s, int d, Packet* p);
        void annotate(const char* s);
        int show_tcphdr_;  // bool flags; backward compat
 public:
        Trace(int type);
        ~Trace();
        int command(int argc, const char*const* argv);
        void recv(Packet* p, Handler*);
        void dump();
        inline char* buffer() { return (wrk_); }
};
```

The src_, and dst_ internal state is used to label trace output and is independent of the corresponding field names in packet headers. The main recv() method is defined as follows:

```
void Trace::recv(Packet* p, Handler* h)
{
        format(type_, src_, dst_, p);
        dump();
        /* hack: if trace object not attached to anything free packet */
        if (target_ == 0)
                Packet::free(p);
        else
                send(p, h); /* Connector::send() */
}
```

The function merely formats a trace entry using the source, destination, and particular trace type character. The dump function writes the formatted entry out to the I/O handle associated with channel_. The format function, in effect, dictates the trace file format.

14.4 Trace File Format

The `Trace::format()` method defines the trace file format used in trace files produced by the `Trace` class. It is constructed to maintain backward compatibility with output files in earlier versions of the simulator (*i.e.*, *ns* v1) so that *ns* v1 post-processing scripts continue to operate. The important pieces of its implementation are as follows:

```
//  this function should retain some backward-compatibility, so that
//  scripts don't break.
void Trace::format(int tt, int s, int d, Packet* p)
{
        hdr_cmn *th = (hdr_cmn*)p->access(off_cmn_);
        hdr_ip *iph = (hdr_ip*)p->access(off_ip_);
        hdr_tcp *tcph = (hdr_tcp*)p->access(off_tcp_);
        hdr_rtp *rh = (hdr_rtp*)p->access(off_rtp_);
        int t = th->ptype();
        const char* name = pt_names[t];

        if (name == 0)
                abort();

        int seqno;
        /* XXX */
        /*  CBR's now have seqno's too  */
        if (t == PT_RTP || t == PT_CBR)
                seqno = rh->seqno();
        else if (t == PT_TCP || t == PT_ACK)
                seqno = tcph->seqno();
        else
                seqno = -1;

        ...

        if (!show_tcphdr_) {
                sprintf(wrk_, "%c %g %d %d %s %d %s %d %d.%d %d.%d %d %d",
                        tt,
                        Scheduler::instance().clock(),
                        s,
                        d,
                        name,
                        th->size(),
                        flags,
                        iph->flowid() /* was p->class_ */,
                        iph->src() >> 8, iph->src() & 0xff,      // XXX
                        iph->dst() >> 8, iph->dst() & 0xff,      // XXX
                        seqno,
                        th->uid() /* was p->uid_ */);
        } else {
                sprintf(wrk_,
                "%c %g %d %d %s %d %s %d %d.%d %d.%d %d %d %d 0x%x %d",
                        tt,
                        Scheduler::instance().clock(),
                        s,
                        d,
```

```
                                name,
                                th->size(),
                                flags,
                                iph->flowid() /* was p->class_ */,
                                iph->src() >> 8, iph->src() & 0xff,      // XXX
                                iph->dst() >> 8, iph->dst() & 0xff,      // XXX
                                seqno,
                                th->uid(), /* was p->uid_ */
                                tcph->ackno(),
                                tcph->flags(),
                                tcph->hlen());
        }
```

This function is somewhat unelegant, primarily due to the desire to maintain backward compatibility. It formats the source, destination, and type fields defined in the trace object (*not in the packet headers*), the current time, along with various packet header fields including, type of packet (as a name), size, flags (symbolically), flow identifier, source and destination packet header fields, sequence number (if present), and unique identifier. The `show_tcphdr_` variable indicates whether the trace output should append tcp header information (ack number, flags, header length) at the end of each output line. This is especially useful for simulations using FullTCP agents (Section 19.3). An example of a trace file (without the tcp header fields) migh appear as follows:

```
+ 1.84375 0 2 cbr 210 ------- 0 0.0 3.1 225 610
- 1.84375 0 2 cbr 210 ------- 0 0.0 3.1 225 610
r 1.84471 2 1 cbr 210 ------- 1 3.0 1.0 195 600
r 1.84566 2 0 ack 40 ------- 2 3.2 0.1 82 602
+ 1.84566 0 2 tcp 1000 ------- 2 0.1 3.2 102 611
- 1.84566 0 2 tcp 1000 ------- 2 0.1 3.2 102 611
r 1.84609 0 2 cbr 210 ------- 0 0.0 3.1 225 610
+ 1.84609 2 3 cbr 210 ------- 0 0.0 3.1 225 610
d 1.84609 2 3 cbr 210 ------- 0 0.0 3.1 225 610
- 1.8461 2 3 cbr 210 ------- 0 0.0 3.1 192 511
r 1.84612 3 2 cbr 210 ------- 1 3.0 1.0 196 603
+ 1.84612 2 1 cbr 210 ------- 1 3.0 1.0 196 603
- 1.84612 2 1 cbr 210 ------- 1 3.0 1.0 196 603
+ 1.84625 3 2 cbr 210 ------- 1 3.0 1.0 199 612
```

Here we see 14 trace entries, five enque operations (indicated by "+" in the first column), four deque operations (indicated by "-"), four receive events (indicated by "r"), and one drop event. (this had better be a trace fragment, or some packets would have just vanished!). The simulated time (in seconds) at which each event occurred is listed in the second column. The next two fields indicate between which two nodes tracing is happening. The next field is a descriptive name for the the type of packet seen (Section 14.5). The next field is the packet's size, as encoded in its IP header. The next four characters represent special flag bits which may be enabled. Presently only one such bit exists (explicit congestion notification, or ECN). In this example, ECN is not used. The next field gives the IP *flow identifier* field as defined for IP version 6.[1]. The subsequent two fields indicate the packet's source and destination node addresses, respectively. The following field indicates the sequence number.[2] The last field is a unique packet identifier. Each new packet created in the simulation is assigned a new, unique identifier.

[1]In *ns* v1, each packet included a `class` field, which was used by CBQ to classify packets. It then found additional use to differentiate between "flows" at one trace point. In *ns* v2, the flow ID field is available for this purpose, but any additional information (which was commonly overloaded into the class field in *ns* v1) should be placed in its own separate field, possibly in some other header

[2]In *ns* v1, all packets contained a sequence number, whereas in *ns* v2 only those Agents interested in providing sequencing will generate sequence numbers. Thus, this field may not be useful in *ns* v2 for packets generated by agents that have not filled in a sequence number. It is used here to remain backward compatible with *ns* v1.

14.5 Packet Types

Each packet contains a packet type field used by `Trace::format` to print out the type of packet encountered. The type field is defined in the `TraceHeader` class, and is considered to be part of the trace support; it is not interpreted elsewhere in the simulator. Initialization of the type field in packets is performed by the `Agent::allocpkt`(void) method. The type field is set to integer values associated with the definition passed to the `Agent` constructor (Section 8.6.3). The currently-supported definitions, their values, and their associated symblic names are as follows (defined in ~*ns*/packet.h):

```
#define PT_TCP          0
#define PT_TELNET       1
#define PT_CBR          2
#define PT_AUDIO        3
#define PT_VIDEO        4
#define PT_ACK          5
#define PT_START        6
#define PT_STOP         7
#define PT_PRUNE        8
#define PT_GRAFT        9
#define PT_MESSAGE      10
#define PT_RTCP         11
#define PT_RTP          12
#define PT_RTPROTO_DV   13
#define PT_CtrMcast_Encap 14
#define PT_CtrMcast_Decap 15
#define PT_SRM          16
#define PT_NTYPE        17

#define PT_NAMES "tcp", "telnet", "cbr", "audio", "video", "ack", \
        "start", "stop", "prune", "graft", "message", "rtcp", "rtp", \
        "rtProtoDV", "CtrMcast_Encap", "CtrMcast_Decap", "SRM"
```

The definition of `PT_NAMES` is used to initialize the `npt_names` array as indicated above; `PT_NTYPE` is not presently used.

14.6 Queue Monitoring

Queue monitoring refers to the capability of tracking the dynamics of packets at a queue (or other object). A queue monitor tracks packet arrival/departure/drop statistics, and may optionally compute averages of these values. Monitoring may be applied all packets (aggregate statistics), or per-flow statistics (using a Flow Monitor).

Several classes are used in supporting queue monitoring. When a packet arrives at a link where queue monitoring is enabled, it generally passes through a `SnoopQueue` object when it arrives and leaves (or is dropped). These objects contain a reference to a `QueueMonitor` object.

A `QueueMonitor` is defined as follows (~*ns*/queue-monitor.cc):

```
class QueueMonitor : public TclObject {
 public:
        QueueMonitor() : bytesInt_(NULL), pktsInt_(NULL), delaySamp_(NULL),
          size_(0), pkts_(0),
```

```
            parrivals_(0), barrivals_(0),
            pdepartures_(0), bdepartures_(0),
            pdrops_(0), bdrops_(0),
            srcId_(0), dstId_(0), channel_(0) {

                  bind("size_", &size_);
                  bind("pkts_", &pkts_);
                  bind("parrivals_", &parrivals_);
                  bind("barrivals_", &barrivals_);
                  bind("pdepartures_", &pdepartures_);
                  bind("bdepartures_", &bdepartures_);
                  bind("pdrops_", &pdrops_);
                  bind("bdrops_", &bdrops_);
                  bind("off_cmn_", &off_cmn_);
          };

          int size() const { return (size_); }
          int pkts() const { return (pkts_); }
          int parrivals() const { return (parrivals_); }
          int barrivals() const { return (barrivals_); }
          int pdepartures() const { return (pdepartures_); }
          int bdepartures() const { return (bdepartures_); }
          int pdrops() const { return (pdrops_); }
          int bdrops() const { return (bdrops_); }
          void printStats();
          virtual void in(Packet*);
          virtual void out(Packet*);
          virtual void drop(Packet*);
          virtual void edrop(Packet*) { abort(); }; // not here
          virtual int command(int argc, const char*const* argv);
          ...

// packet arrival to a queue
void QueueMonitor::in(Packet* p)
{
          hdr_cmn* hdr = (hdr_cmn*)p->access(off_cmn_);
          double now = Scheduler::instance().clock();
          int pktsz = hdr->size();

          barrivals_ += pktsz;
          parrivals_++;
          size_ += pktsz;
          pkts_++;
          if (bytesInt_)
                  bytesInt_->newPoint(now, double(size_));
          if (pktsInt_)
                  pktsInt_->newPoint(now, double(pkts_));
          if (delaySamp_)
                  hdr->timestamp() = now;
          if (channel_)
                  printStats();
}

... in(), out(), drop() are all defined similarly ...
```

It addition to the packet and byte counters, a queue monitor may optionally refer to objects that keep an integral of the queue size over time using `Integrator` objects, which are defined in Section13.3. The `Integrator` class provides a simple implementation of integral approximation by discrete sums.

All bound variables beginning with **p** refer to packet counts, and all variables beginning with **b** refer to byte counts. The variable `size_` records the instantaneous queue size in bytes, and the variable `pkts_` records the same value in packets. When a `QueueMonitor` is configured to include the integral functions (on bytes or packets or both), it computes the approximate integral of the queue size (in bytes) with respect to time over the interval $[t_0, now]$, where t_0 is either the start of the simulation or the last time the `sum_` field of the underlying `Integrator` class was reset.

The `QueueMonitor` class is not derived from `Connector`, and is not linked directly into the network topology. Rather, objects of the `SnoopQueue` class (or its derived classes) are inserted into the network topology, and these objects contain references to an associated queue monitor. Ordinarily, multiple `SnoopQueue` objects will refer to the same queue monitor. Objects constructed out of these classes are linked in the simulation topology as described above and call `QueueMonitor out`, `in`, or `drop` procedures, depending on the particular type of snoopy queue.

14.7 Per-Flow Monitoring

A collection of specialized classes are used to to implement per-flow statistics gathering. These classes include: `QueueMonitor/ED/Flowmon`, `QueueMonitor/ED/Flow`, and `Classifier/Hash`. Typically, an arriving packet is inspected to determine to which flow it belongs. This inspection and flow mapping is performed by a *classifier* object (described in section 14.7.1). Once the correct flow is determined, the packet is passed to a *flow monitor*, which is responsible for collecting per-flow state. Per-flow state is contained in *flow* objects in a one-to-one relationship to the flows known by the flow monitor. Typically, a flow monitor will create flow objects on-demand when packets arrive that cannot be mapped to an already-known flow.

14.7.1 The Flow Monitor

The `QueueMonitor/ED/Flowmon` class is responsible for managing the creation of new flow objects when packets arrive on previously unknown flows and for updating existing flow objects. Because it is a subclass of `QueueMonitor`, each flow monitor contains an aggregate count of packet and byte arrivals, departures, and drops. Thus, it is not necessary to create a separate queue monitor to record aggregate statistics. It provides the following OTcl interface:

classifier	get(set) classifier to map packets to flows
attach	attach a Tcl I/O channel to this monitor
dump	dump contents of flow monitor to Tcl channel
flows	return string of flow object names known to this monitor

The `classifier` function sets or gets the name of the previously-allocated object which will perform packet-to-flow mapping for the flow monitor. Typically, the type of classifier used will have to do with the notion of "flow" held by the user. One of the hash based classifiers that inspect various IP-level header fields is typically used here (e.g. fid, src/dst, src/dst/fid). Note that while classifiers usually receive packets and forward them on to downstream objects, the flow monitor uses the classifier only for its packet mapping capability, so the flow monitor acts as a passive monitor only and does not actively forward packets.

The `attach` and `dump` functions are used to associate a Tcl I/O stream with the flow monitor, and dump its contents on-demand. The file format used by the `dump` command is described below.

The `flows` function returns a list of the names of flows known by the flow monitor in a way understandable to Tcl. This allows tcl code to interrogate a flow monitor in order to obtain handles to the individual flows it maintains.

14.7.2 Flow Monitor Trace Format

The flow monitor defines a trace format which may be used by post-processing scripts to determine various counts on a per-flow basis. The format is defined by the folling code in ~*ns*/flowmon.cc:

```
void
FlowMon::fformat(Flow* f)
{
        double now = Scheduler::instance().clock();
        sprintf(wrk_, "%8.3f %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d
%d",
                now,
                f->flowid(),     // flowid
                0,               // category
                f->ptype(),      // type (from common header)
                f->flowid(),     // flowid (formerly class)
                f->src(),
                f->dst(),
                f->parrivals(), // arrivals this flow (pkts)
                f->barrivals(), // arrivals this flow (bytes)
                f->epdrops(),   // early drops this flow (pkts)
                f->ebdrops(),   // early drops this flow (bytes)
                parrivals(),    // all arrivals (pkts)
                barrivals(),    // all arrivals (bytes)
                epdrops(),      // total early drops (pkts)
                ebdrops(),      // total early drops (bytes)
                pdrops(),       // total drops (pkts)
                bdrops(),       // total drops (bytes)
                f->pdrops(),    // drops this flow (pkts) [includes edrops]
                f->bdrops()     // drops this flow (bytes) [includes edrops]
        );
};
```

Most of the fields are explained in the code comments. The “category” is historical, but is used to maintain loose backward-compatibility with the flow manager format in *ns* version 1.

14.7.3 The Flow Class

The class `QueueMonitor/ED/Flow` is used by the flow monitor for containing per-flow counters. As a subclass of `QueueMonitor`, it inherits the standard counters for arrivals, departures, and drops, both in packets and bytes. In addition, because each flow is typically identified by some combination of the packet source, destination, and flow identifier fields, these objects contain such fields. It’s OTcl interface contains only bound variables:

`src_`	source address on packets for this flow
`dst_`	destination address on packets for this flow
`flowid_`	flow id on packets for this flow

Note that packets may be mapped to flows (by classifiers) using criteria other than a src/dst/flowid triple. In such circumstances, only those fields actually used by the classifier in performing the packet-flow mapping should be considered reliable.

Part IV

Routing

Chapter 15

Unicast Routing

This section describes the structure of unicast routing in *ns*. We begin by describing the interface to the user (Section 15.1), through methods in the `class Simulator` and the `class RouteLogic`. We then describe configuration mechanisms for specialised routing (Section 15.2) such as asymmetric routing, or equal cost multipath routing The next section describes the the configuration mechanisms for individual routing strategies and protocols (Section 15.3). We conclude with a comprehensive look at the internal architecture (Section 15.4) of routing in *ns*.

The procedures and functions described in this chapter can be found in ~*ns*/tcl/lib/ns-route.tcl, ~*ns*/tcl/rtglib/route-proto.tcl, ~*ns*/tcl/mcast/McastProto.tcl, and ~*ns*/rtProtoDV.{cc, h}.

15.1 The Interface to the Simulation Operator (The API)

The user level simulation script requires one command: to specify the unicast routing strategy or protocols for the simulation. A routing strategy is a general mechanism by which *ns* will compute routes for the simulation. There are three routing strategies in *ns*: Static, Session, and Dynamic. Conversely, a routing protocol is a realisation of a specific algorithm. Currently, Static and Session routing use the Dijkstra's all-pairs SPF algorithm []; one type of dynamic routing strategy is currently implemented: the Distributed Bellman-Ford algorithm []. In *ns*, we blur the distinction between strategy and protocol for static and session routing, considering them simply as protocols[1].

`rtproto{}` is the instance procedure in the `class Simulator` that specifies the unicast routing protocol to be used in the simulation. It takes multiple arguments, the first of which is mandatory; this first argument identifies the routing protocol to be used. Subsequent arguments specify the nodes that will run the instance of this protocol. The default is to run the same routing protocol on all the nodes in the topology. As an example, the following commands illustrate the use of the `rtproto{}` command.

```
$ns rtproto Static                 ;# Enable static route strategy for the simulation
$ns rtproto Session                ;# Enable session routing for this simulation
$ns rtproto DV $n1 $n2 $n3         ;# Run DV agents on nodes $n1, $n2, and $n3
```

If a simulation script does not specify any `rtproto{}` command, then *ns* will run Static routing on all the nodes in the topology.

[1] The consideration is that static and session routing strategies/protocols are implemented as agents derived from the `class Agent/rtProto` similar to how the different dynamic routing protocols are implemented; hence the blurred distinctions.

Multiple `rtproto{}` lines for the same or different routing protocols can occur in a simulation script. However, a simulation cannot use both centralised routing mechanisms such as static or session routing and detailed dynamic routing protocols such as DV.

In dynamic routing, each node can be running more than one routing protocol. In such situations, more than one routing protocol can have a route to the same destination. Therefore, each protocol affixes a preference value to each of its routes. These values are non-negative integers in the range 0...255. The lower the value, the more preferred the route. When multiple routing protocol agents have a route to the same destination, the most preferred route is chosen and installed in the node's forwarding tables. If more than one agent has the most preferred routes, the ones with the lowest metric is chosen. We call the least cost route from the most preferred protocol the "candidate" route. If there are multiple candidate routes from the same or different protocols, then, currently, one of the agent's routes is randomly chosen[2].

Preference Assignment and Control Each protocol agent stores an array of route preferences, `rtpref_`. There is one element per destination, indexed by the node handle. The default preference values used by each protocol are derived from a class variable, `preference_`, for that protocol. The current defaults are:

```
Agent/rtProto set preference_ 200                    ;# global default preference
Agent/rtProto/Direct3 set preference_ 100
Agent/rtProto/DV set preference_ 120
```

A simulation script can control routing by altering the preference for routes in one of three ways: alter the preference for a specific route learned *via* a particular protocol agent, alter the preference for all routes learned by the agent, or alter the class variables for the agent before the agent is created.

Link Cost Assignment and Control In the currently implemented route protocols, the metric of a route to a destination, at a node, is the cost to reach the destination from that node. It is possible to change the link costs at each of the links. The instance procedure `cost{}` is invoked as `$ns cost ⟨node1⟩ ⟨node2⟩ ⟨cost⟩`, and sets the cost of the link from ⟨node1⟩ to ⟨node2⟩ to ⟨cost⟩.

```
$ns cost $n1 $n2 10              ;# set cost of link from $n1 to $n2 to 10
$ns cost $n2 $n1  5              ;# set cost of link in reverse direction to 5
[$ns link $n1 $n2] cost?         ;# query cost of link from $n1 to $n2
[$ns link $n2 $n1] cost?         ;# query cost of link in reverse direction
```

Notice that the procedure sets the cost along one direction only. Similarly, the procedure `cost?{}` returns the cost of traversing the specified unidirectional link. The default cost of a link is 1.

15.2 Other Configuration Mechanisms for Specialised Routing

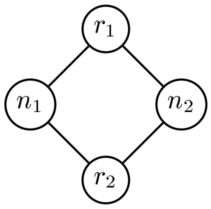
It is possible to adjust preference and cost mechanisms to get two special types of route configurations: asymmetric routing, and multipath routing.

[2] This really is undesirable, and may be fixed at some point. The fix will probably be to favor the agents in class preference order. A user level simulation relying on this behavior, or getting into this situation in specific topologies is not recommended.

[3] Direct is a special routing strategy that is used in conjunction with Dynamic routing. We will describe this in greater detail as part of the route architecture description.

Asymetric Routing Asymetric routing occurs when the path from node n_1 to node n_2 is different from the path from n_2 to n_1. The following shows a simple topology, and cost configuration that can achieve such a result:

Nodes n_1 and n_2 use different paths to reach each other. All other pairs of nodes use symmetric paths to reach each other.




```
$ns cost $n1 $r1 2
$ns cost $n2 $r2 2
$ns cost $r1 $n2 3
```

Any routing protocol that uses link costs as the metric can observe such asymmetric routing if the link costs are appropriately configured[4].

MultiPath Routing Each node can be individually configured to use multiple separate paths to a particular destination. The instance variable `multiPath_` determines whether or not that node will use multiple paths to any destination. Each node initialises its instance variable from a class variable of the same name. If multiple candidate routes to a destination are available, all of which are learned through the same protocol, then that node can use all of the different routes to the destination simultaneously. A typical configuration is as shown below:

```
Node set multiPath_ 1          ;# All new nodes in the simulation use multiPaths where applicable
```

or alternately

```
set n1 [$ns Node]              ;# only enable $n1 to use multiPaths where applicable
$n1 set multiPath_ 1
```

Currently, only DV routing can generate multipath routes.

15.3 Protocol Specific Configuration Parameters

Static Routing The static route computation strategy is the default route computation mechanism in *ns*. This strategy uses the Dijkstra's all-pairs SPF algorithm []. The route computation algorithm is run exactly once prior to the start of the simulation. The routes are computed using an adjacency matrix and link costs of all the links in the topology.

Session Routing The static routing strategy described earlier only computes routes for the topology once in the course of a simulation. If the topology changes while the simulation is in progress, Then some sources and destinations may become unreachable from each other.

Session routing strategy is almost identical to static routing, in that it runs the Dijkstra all-pairs SPF algorithm prior to the start of the simulation, using the adjacency matrix and link costs of the links in the topology. However, it will also run the same algorithm to recompute routes in the event that the topology changes during the course of a simulation.

Note that session routing leads to complete and instantaneous change in the routes of the topology, whenever that topology changes. If the topology is always connected, then there is end-to-end connectivity at all times during the course of the simulation. However, the user should note that the instantaneous recompute of the routes in the topology can lead to temporary violations of causality around the instant that the topology changes.

DV Routing DV routing is the implementation of Distributed Bellman-Ford (or Distance Vector) routing in *ns*. The implementation sends periodic route updates every `advertInterval`. This variable is a class variable in the `class`

[4]Link costs can also be used to favour or disregard specific links in order to achieve particular topology configurations.

Agent/rtProto/DV. Its default value is 2 seconds.

In addition to periodic updates, each agent also sends triggered updates; it does this whenever the forwarding tables in the node change. This occurs either due to changes in the topology, or because an agent at the node received a route update, and recomputed and installed new routes.

Each agent employs the split horizon with poisoned reverse mechanisms to advertise its routes to adjacent peers. "Split horizon" is the mechanism by which an agent will not advertise the route to a destination out of the interface that it is using to reach that destination. In a "Split horizon with poisoned reverse" mechanism, the agent will advertise that route out of that interface with a metric of infinity.

Each DV agent uses a default `preference_` of 120. The value is determined by the class variable of the same name.

Each agent uses the class variable `INFINITY` (set at 32) to determine the validity of a route.

15.4 Internals and Architecture of Routing

We start with a discussion of the classes associated with unicast routing, and the code path used to configure and execute each of the different routing protocols. We conclude with a description of the interface between unicast routing and network dynamics, and that between unicast and multicast routing.

15.4.1 The classes

There are four main classes, the class RouteLogic, the class rtObject, the class rtPeer, and the base class Agent/rtProto for all protocols. In addition, the routing architecture extends the classes Simulator, Link, Node and Classifier.

`class RouteLogic` This class defines two methods to configure unicast routing, and one method to query it for route information. It also defines an instance procedure that is applicable when the topology is dynamic. We discuss this last procedure in conjunction with the interface to network dynamics.

- The instance procedure `register{}` is invoked by `Simulator::rtproto{}`. It takes the protocol and a list of nodes as arguments, and constructs an instance variable, `rtprotos_`, as an array; the array index is the name of the protocol, and the value is the list of nodes that will run this protocol.
- The `configure{}` reads the `rtprotos_` instance variable, and for each element in the array, invokes route protocol methods to perform the appropriate initializations. It is invoked by the simulator run procedure.

  For each protocol ⟨rt-proto⟩ indexed in the `rtprotos_` array, this routine invokes `Agent/rtProto/⟨rt-proto⟩ init-all rtprotos_(⟨rt-proto⟩)`.

  If there are no elements in `rtprotos_`, the routine invokes Static routing, as `Agent/rtProto/Static init-all`.
- The instance procedure `lookup{}` takes two node numbers, $nodeId_1$ and $nodeId_2$, as argument; it returns the id of the neighbor node that $nodeId_1$ uses to reach $nodeId_2$.

  The procedure is used by the static route computation procedure to query the computed routes and populate the routes at each of the nodes. It is also used by the multicast routing protocols to perform the appropriate RPF check.

  Note that this procedure overloads an instproc-like of the same name. The procedure queries the appropriate `rtObject` entities if they exist (which they will if dynamic routing strategies are used in the simulation); otherwise, the procedure invokes the instproc-like to obtain the relevant information.

class rtObject is used in simulations that use dynamic routing. Each node has a rtObject associated with it, that acts as a co-ordinator for the different routing protocols that operate at a node. At any node, the rtObject at that node tracks each of the protocols operating at that node; it computes and installs the nest route to each destination available via each of the protocols. In the event that the routing tables change, or the topology changes, the rtObject will alert the protocols to take the appropriate action.

The class defines the procedure `init-all{}`; this procedure takes a list of nodes as arguments, and creates a rtObject at each of the nodes in its argument list. It subsequently invokes its `compute-routes`.

The assumption is that the constructor for each of the new objects will instantiate the "Direct" route protocol at each of these nodes. This route protocol is responsible for computing the routes to immediately adjacent neighbors. When `compute-routes{}` is run by the `init-all{}` procedure, these direct routes are installed in the node by the appropriate route object.

The other instance procedures in this class are:

- `init{}` The constructor sets up pointers from itself to the node, in its instance variable `node_`, and from the node to itself, through the Node instance procedure `init-routing{}` and the Node instance variable `rtObject_`. It then initializes an array of `nextHop_`, `rtpref_`, `metric_`, `rtVia_`. The index of each of these arrays is the handle of the destination node.

  The `nextHop_` contains the link that will be used to reach the particular destination; `rtpref_` and `metric_` are the preference and metric for the route installed in the node; `rtVia_` is the name of the agent whose route is installed in the node.

  The constructor also creates the instance of the Direct route protocol, and invokes `compute-routes{}` for that protocol.
- `add-proto{}` creates an instance of the protocol, stores a reference to it in its array of protocols, `rtProtos_`. The index of the array is the name of the protocol. It also attaches the protocol object to the node, and returns the handle of the protocol object.
- `lookup{}` takes a destination node handle, and returns the id of the neighbor node that is used to reach the destination.

  If multiple paths are in use, then it returns a list of the neighbor nodes that will be used.

  If the node does not have a route to the destination, the procedure will return -1.
- `compute-routes{}` is the core procedure in this class. It first checks to see if any of the routing protocols at the node have computed any new routes. If they have, it will determine the best route to each destination from among all the protocols. If any routes have changed, the procedure will notify each of the protocols of the number of such changes, in case any of these protocols wants to send a fresh update. Finally, it will also notify any multicast protocol that new unicast route tables have been computed.

  The routine checks the protocol agent's instance variable, `rtsChanged_` to see if any of the routes in that protocol have changed since the protocol was last examined. It then uses the protocol's instance variable arrays, `nextHop_`, `rtpref_`, and `metric_` to compute its own arrays. The rtObject will install or modify any of the routes as the changes are found.

  If any of the routes at the node have changed, the rtObject will invoke the protocol agent's instance procedures, `send-updates{}` with the number of changes as argument. It will then invoke the multicast route object, if it exists.

The next set of routines are used to query the rtObject for various state information.

- `dump-routes{}` takes a output file descriptor as argument, and writes out the routing table at that node on that file descriptor.

  A typical dump output is:

- `rtProto?{}` takes a route protocol as argument, and returns a handle to the instance of the protocol running at the node.
- `nextHop?{}` takes a destination node handle, and returns the link that is used to reach that destination.
- Similarly, `rtpref?{}` and `metric?{}` take a destination node handle as argument, and return the preference and metric of the route to the destination installed at the node.

The `class rtPeer` is a container class used by the protocol agents. Each object stores the address of the peer agent, and the metric and preference for each route advertised by that peer. A protocol agent will store one object per peer. The class maintains the instance variable `addr_`, and the instance variable arrays, `metric_` and `rtpref_`; the array indices are the destination node handles.

The class instance procedures, `metric{}` and `preference{}`, take one destination and value, and set the respective array variable. The procedures, `metric?{}` and `preference?{}`, take a destination and return the current value for that destination. The instance procedure `addr?{}` returns the address of the peer agent.

`class Agent/rtProto` This class is the base class from which all routing protocol agents are derived. Each protocol agent must define the procedure`init-all{}` to initialize the complete protocol, and possibly instance procedures `init{}`, `compute-routes{}`, and `send-updates{}`. In addition, if the topology is dynamic, and the protocol supports route computation to react to changes in the topology, then the protocol should define the procedure `compute-all{}`, and possibly the instance procedure `intf-changed{}`. In this section, we will briefly describe the interface for the basic procedures. We will defer the description of `compute-all{}` and `intf-changed{}` to the section on network dynamics. We also defer the description of the details of each of the protocols to their separate section at the end of the chapter.

— The procedure `init-all{}` is a global initialization procedure for the class. It may be given a list of the nodes as an argument. This the list of nodes that should run this routing protocol. However, centralised routing protocols such as static and session routing will ignore this argument; detailed dynamic routing protocols such as DV will use this argument list to instantiate protocols agents at each of the nodes specified.

  Note that derived classes in OTcl do not inherit the procedures defined in the base class. Therefore, every derived routing protocol class must define its own procedures explicitly.

— The instance procedure `init{}` is the constructor for protocol agents that are created. The base class constructor initializes the default preference for objects in this class, identifies the interfaces incident on the node and their current status. The interfaces are indexed by the neighbor handle and stored in the instance variable array, `ifs_`; the corresponding status instance variable array is `ifstat_`.

  Centralized routing protocols such as static and session routing do not create separate agents per node, and therefore do not access any of these instance procedures.

— The instance procedure `compute-routes{}` computes the actual routes for the protocol. The computation is based on the routes learned by the protocol, and varies from protocol to protocol.

  This routine is invoked by the rtObject whenever the topology changes. It is also invoked when the node receives an update for the protocol.

  If the routine computes new routes, `rtObject::compute-routes{}` needs to be invoked to recompute and possibly install new routes at the node. The actual invoking of the rtObject is done by the procedure that invoked this routine in the first place.

— The instance procedure `send-updates{}` is invoked by the rtObject whenever the node routing tables have changed, and fresh updates have to be sent to all peers. The rtObject passes as argument the number of changes that were done. This procedure may also be invoked when there are no changes to the routes, but the topology incident on the node changes state. The number of changes is used to determine the list of peers to which a route update must be sent.

Other procedures relate to responding to topology changes and are described later (Section 15.4.2).

Other Extensions to the Simulator, Node, Link, and Classifier

— We have discussed the methods `rtproto{}` and `cost{}` in the class Simulator earlier (Section 15.1). The one other method used internally is `get-routelogic{}`; this procedure returns the instance of routelogic in the simulation.

  The method is used by the class Simulator, and unicast and multicast routing.

— The class Node contains these additional instance procedures to support dynamic unicast routing: `init-routing{}`, `add-routes{}`, `delete-routes{}`, and `rtObject?{}`.

  The instance procedure `init-routing{}` is invoked by the `rtObject` at the node. It stores a pointer to the rtObject, in its instance variable `rtObject_`, for later manipulation or retrieval. It also checks its class variable to see if it should use multiPath routing, and sets up an instance variable to that effect. If multiPath routing could be used, the instance variable array `routes_` stores a count of the number of paths installed for each destination. This is the only array in unicast routing that is indexed by the node id, rather than the node handle.

  The instance procedure `rtObject?{}` returns the rtObject handle for that node.

  The instance procedure `add-routes{}` takes a node id, and a list of links. It will add the list of links as the routes to reach the destination identified by the node id. The realization of multiPath routing is done by using a separate Classifier/multiPath. For any given destination id d, if this node has multiple paths to d, then the main classifier points to this multipath classifier instead of the link to reach the destination. Each of the multiple paths identified by the interfaces being used is installed in the multipath classifier. The multipath classifier will use each of the links installed in it for succeeding packets forwarded to it.

  The instance procedure `delete-routes{}` takes a node id, a list of interfaces, and a nullAgent. It removes each of the interfaces in the list from the installed list of interfaces. If the entry did not previously use a multipath classifier, then it must have had only one route, and the route entry is set to point to the nullAgent specified.

  Q: WHY DOES IT NOT POINT TO NULLAGENT IF THE ENTRIES IN THE MPATHCLASSIFIER GOES TO ZERO?

— The main extension to the class Link for unicast routing is to support the notion of link costs. The instance variable `cost_` contains the cost of the unidirectional link. The instance procedures `cost{}` and `cost?{}` set and get the cost on the link.

  Note that `cost{}` takes the cost as argument. It is preferable to use the simulator method to set the cost variable, similar to the simulator instance procedures to set the queue or delay on a link.

— The `class Classifier` contains three new procedures, two of which overloads an existing instproc-like, and the other two provide new functionality.

  The instance procedure `install{}` overloads the existing instproc-like of the same name. The procedure stores the entry being installed in the instance variable array, `elements_`, and then invokes the instproc-like.

  The instance procedure `installNext{}` also overloads the existing instproc-like of the same name. This instproc-like simply installs the entry into the next available slot.

  The instance procedure `adjacents{}` returns a list of ⟨key, value⟩ pairs of all elements installed in the classifier.

15.4.2 Interface to Network Dynamics and Multicast

This section describes the methods applied in unicast routing to respond to changes in the topology. The complete sequence of actions that cause the changes in the topology, and fire the appropriate actions is described in a different section. The response to topology changes falls into two categories: actions taken by individual agents at each of the nodes, and actions to be taken globally for the entire protocol.

Detailed routing protocols such as the DV implementation require actions to be performed by individual protocol agents at the affected nodes. Centralized routing protocols such as static and session routing fall into the latter category exclusively. Detailed routing protocols could use such techniques to gather statistics related to the operation of the routing protocol; however, no such code is currently implemented in *ns*.

Actions at the individual nodes Following any change in the topology, the network dynamics models will first invoke `rtObject::intf-changed{}` at each of the affected nodes. For each of the unicast routing protocols operating at that node, `rtObject::intf-changed{}` will invoke each individual protocol's instance procedure, `intf-changed{}`, followed by that protocol's `compute-routes{}`.

After each protocol has computed its individual routes `rtObject::intf-changed{}` invokes `compute-routes{}` to possibly install new routes. If new routes were installed in the node, `rtObject::compute-routes{}` will invoke `send-updates{}` for each of the protocols operating at the node. The procedure will also flag the multicast route object of the route changes at the node, indicating the number of changes that have been executed. `rtObject::flag-multicast{}` will, in turn, notify the multicast route object to take appropriate action.

The one exception to the interface between unicast and multicast routing is the interaction between dynamic dense mode multicast and detailed unicast routing. This dynamicDM implementation in *ns* assumes neighbor nodes will send an implicit update whenever their routes change, without actually sending the update. It then uses this implicit information to compute appropriate parent-child relationships for the multicast spanning trees. Therefore, detailed unicast routing will invoke `rtObject_ flag-multicast 1` whenever it receives a route update as well, even if that update does not result in any change in its own routing tables.

Global Actions Once the detailed actions at each of the affected nodes is completed, the network dynamics models will notify the RouteLogic instance (`RouteLogic::notify{}`) of changes to topology. This procedure invokes the procedure `compute-all{}` for each of the protocols that were ever installed at any of the nodes. Centralized routing protocols such as session routing use this signal to recompute the routes to the topology. Finally, the `RouteLogic::notify{}` procedure notifies any instances of centralised multicast that are operating at the node.

15.5 Protocol Internals

In this section, we describe any leftover details of each of the routing protocol agents. Note that this is the only place where we describe the internal route protocol agent, "Direct" routing.

Direct Routing This protocol tracks the state of the incident links, and maintains routes to immediately adjacent neighbors only. As with the other protocols, it maintains instance variable arrays of `nextHop_`, `rtpref_`, and `metric_`, indexed by the handle of each of the possible destinations in the topology.

The instance procedure `compute-routes{}` computes routes based on the current state of the link, and the previously known state of the incident links.

No other procedures or instance procedures are defined for this protocol.

Static Routing The procedure `compute-routes{}` in the `class RouteLogic` first creates the adjacency matrix, and then invokes the C++ method, `compute_routes()` of the shadow object. Finally, the procedure retrieves the result of the route computation, and inserts the appropriate routes at each of the nodes in the topology.

The class only defines the procedure `init-all{}` that invokes `compute-routes{}`.

Session Routing The class defines the procedure `init-all{}` to compute the routes at the start of the simulation. It also defines the procedure `compute-all{}` to compute the routes when the topology changes. Each of these procedures directly invokes `compute-routes{}`.

DV Routing In a dynamic routing strategy, nodes send and receive messages, and compute the routes in the topology based on the messages exchanged. The procedure `init-all{}` takes a list of nodes as the argument; the default is the list of nodes in the topology. At each of the nodes in the argument, the procedure starts the `class rtObject` and a `class Agent/rtProto/DV` agents. It then determines the DV peers for each of the newly created DV agents, and creates the relevant `rtPeer` objects.

The constructor for the DV agent initializes a number of instance variables; each agent stores an array, indexed by the destination node handle, of the preference and metric, the interface (or link) to the next hop, and the remote peer incident on the interface, for the best route to each destination computed by the agent. The agent creates these instance variables, and then schedules sending its first update within the first 0.5 seconds of simulation start.

Each agent stores the list of its peers indexed by the handle of the peer node. Each peer is a separate peer structure that holds the address of the peer agent, the metric and preference of the route to each destination advertised by that peer. We discuss the rtPeer structure later when discuss the route architecture. The peer structures are initialized by the procedure `add-peer{}` invoked by `init-all{}`.

The routine `send-periodic-update{}` invokes `send-updates{}` to send the actual updates. It then reschedules sending the next periodic update after `advertInterval` jitterred slightly to avoid possible synchronization effects.

`send-updates{}` will send updates to a select set of peers. If any of the routes at that node have changed, or for periodic updates, the procedure will send updates to all peers. Otherwise, if some incident links have just recovered, the procedure will send updates to the adjacent peers on those incident links only.

`send-updates{}` uses the procedure `send-to-peer{}` to send the actual updates. This procedure packages the update, taking the split-horizon and poison reverse mechanisms into account. It invokes the instproc-like, `send-update{}` (Note the singular case) to send the actual update. The actual route update is stored in the class variable `msg_` indexed by a non-decreasing integer as index. The instproc-like only sends the index to `msg_` to the remote peer. This eliminates the need to convert from OTcl strings to alternate formats and back.

When a peer receives a route update it first checks to determine if the update from differs from the previous ones. The agent will compute new routes if the update contains new information.

Chapter 16

Multicast Routing

This section describes the usage and the internals of multicast routing implementation in *ns*. We first describe the user interface to enable multicast routing (Section 16.1), specify the multicast routing protocol to be used and the various methods and configuration parameters specific to the protocols currently supported in *ns*. We then describe in detail the internals and the architecture of the multicast routing implementation in *ns* (Section 16.2).

The procedures and functions described in this chapter can be found in various files in the directories ~*ns*/tcl/mcast, ~*ns*/tcl/ctr-mcast; additional support routines are in ~*ns*/mcast_ctrl.{cc,h}, ~*ns*/tcl/lib/ns-lib.tcl, and ~*ns*/tcl/lib/ns-node.tcl.

16.1 Multicast API

Multicast forwarding requires enhancements to the nodes and links in the topology. Therefore, the user must specify multicast requirements to the Simulator class before creating the topology. This is done in one of two ways:

```
    set ns [new Simulator -multicast ignored_flag]
or
    set ns [new Simulator]
    $ns multicast
```

With this, nodes contain additional classifiers and replicators for multicast forwarding, and links contain elements to assign incoming interface labels to all packets entering a node.

A multicast routing strategy is the mechanism by which the multicast distribution tree is computed in the simulation. *ns* supports two multiast route computation strategies: centralised or detailed. The method `mrtproto{}` in the Class Simulator specifies either the route computation strategy, for centralised multicast routing, or the specific detailed multicast routing protocol that should be used. For detailed multicast routing, `mrtproto{}` will accept, as additional arguments, a list of nodes that will run an instance of that routing protocol. Thus, the following are examples of valid invocations of multicast routing in *ns*:

```
set cmc [$ns mrtproto CtrMcast {}]        ;# specify centralized multicast for all nodes
                                          ;# cmc is the handle for multicast protocol object
$ns mrtproto DM $n1 $n2 $n3               ;# specify dense mode multicast for nodes n1, n2 and n3
$ns mrtproto detailedDM                   ;# specify dynamic dense mode to run on all nodes
```

Notice in the above examples that CtrMcast returns a handle that can be used for additional configuration of centralised multicast routing. The other routing protocols will return a null string. If no additional nodes are specified for detailed multicast routing, all the nodes in the topology will run instances of the same protocol.

New/unused multicast address are allocated using the procedure `allocaddr{}`. The default configuration in *ns* only provides for 128 node topologies when multicast routing is enabled. The procedure `expandaddr{}` expands the address space to support $2^{30} - 1$ node topologies. `allocaddr{}` and `expandaddr{}` are class procedures in the class Node.

The agents use the instance procedures `join-group{}` and `leave-group{}`, in the class Node to join and leave multicast groups. These procedures take two mandatory arguments. The first argument identifies the corresponding agent and second argument specifies the group address.

An example of a relatively simple multicast configuration is:

```
set ns [new Simulator -multicast on]                ;# enable multicast routing
Node expandaddr                    ;# expand address space, required if more than 128 nodes
set group [Node allocaddr]                          ;# allocate a multicast address
set node0 [$ns node]                                ;# create multicast capable nodes
set node1 [$ns node]
$ns duplex-link $n0 $n1 1.5Mb 10ms DropTail

set mproto detailedDM                               ;# configure multicast protocol
set mrthandle [$ns mrtproto $mproto {}]             ;# if an empty list is give,
                                                    ;# all nodes will contain multicast protocol agents
set udp [new Agent/UDP]                             ;# create a source agent at node 0
$ns attach-agent $node0 $udp
set src [new Application/Traffic/CBR]
$src attach-agent $udp
$udp set dst_ $group

set rcvr [new Agent/LossMonitor]                    ;# create a receiver agent at node 1
$ns attach-agent $node1 $rcvr
$ns at 0.3 "$node1 join-group $rcvr $group"   ;# join the group at simulation time 0.3 (sec)
```

16.1.1 Multicast Behavior Monitor Configuration

ns supports a multicast monitor module that can trace some types of packet activity about a specific source group pair. The module counts the number of packets in transit periodically and prints the results to stdout. `trace-topo{}` counts all links. `trace-tree{}` counts only links on the tree. Currently, it only traces prune, join, register, and data packets.

```
set mcastmonitor [$ns McastMonitor]
$mcastmonitor set period_ 0.02                      ;# default 0.03 (sec)
$mcastmonitor trace-topo $src $group                ;# trace the entire topology
```

The following sample output illustrates the output file format (time, prune, join, register, data, source, group):

```
0.14999999999999999 0 0 4 0 0 32770
0.17999999999999999 0 0 4 0 0 32770
0.20999999999999999 0 0 4 2 0 32770
0.23999999999999999 0 0 4 7 0 32770
0.27000000000000002 0 0 4 8 0 32770
```

16.1.2 Protocol Specific configuration

In this section, we briefly illustrate the protocol specific configuration mechanisms for all the protocols implemented in *ns*.

Centralized Multicast The centralized multicast is a sparse mode implementation of multicast similar to PIM-SM [5]. A Rendezvous Point (RP) rooted shared tree is built for a multicast group. The actual sending of prune, join messages etc. to set up state at the nodes is not simulated. A centralized computation agent is used to compute the forwarding trees and set up multicast forwarding state, $\langle$S, G$\rangle$ at the relevant nodes as new receivers join a group. Data packets from the senders to a group are unicast to the RP. Note that data packets from the senders are unicast to the RP even if there are no receivers for the group.

The method of enabling centralised multicast routing in a simulation is:

```
set mproto CtrMcast                                  ;# set multicast protocol
set mrthandle [$ns mrtproto $mproto {}]
```

The command procedure `mrtproto{}` returns a handle to the multicast protocol object. This handle can be used to control the RP and the boot-strap-router (BSR), switch tree-types for a particular group, from shared trees to source specific trees or vice-versa, and recompute multicast routes.

```
$mrthandle set_c_rp {$node0 $node1}                  ;# set the RPs
$mrthandle set_c_bsr {$node0:0 $node1:1}    ;# set the BSR, specified as list of node:priority

$mrthandle get_c_rp $node0 $group                    ;# get the current RP ???
$mrthandle get_c_bsr $node0                          ;# get the current BSR

$mrthandle switch-treetype $group           ;# to source specific or shared tree

$mrthandle compute-mroutes        ;# recompute routes. usually invoked automatically as needed
```

Note that whenever network dynamics occur or unicast routing changes, `compute-mroutes{}` could be invoked to re-compute the multicast routes. The instantaneous re-computation feature of centralised algorithms may result in causality violations during the transient periods.

Static Dense Mode The Static Dense Mode protocol is loosely based on DVMRP [14]. It computes parent-child relationships as in DVMRP to reduce the number of links over which data packets are broadcast. At each node, the implementation infers poison-reverse information and computes the parent-child relationships for that node by looking at the unicast routing tables at each of the adjacent nodes. The implementation only works on point-to-point links, *i.e.*, no LANs; it does not also adapt to network dynamics.

Any node that receives data for a particular group for which it has no downstream receivers, send a prune upstream. A prune message causes the upstream node to initiate prune state at that node. The prune state prevents that node from sending data for that group downstream to the node that sent the original prune message while the state is active. The time duration for which a prune state is active is configured through the DM class variable, `PruneTimeout`. A typical DM configuration is shown below:

```
DM set PruneTimeout 0.3                              ;# default 0.5 (sec)
set mproto DM
set mrthandle [$ns mrtproto $mproto {}]
```

Detailed dense mode This protocol implements a basic dense mode protocol, similar to DM described earlier. However, it does not compute parent-child relationships; this implementation also works over LANs, and is capable of adapting to changes in the topology. The timers have been implemented in a separate class, with more fine grained control. The three timers are:

Timer/GraftRtx **Graft Retransmit Timer** default = 0.05s.
set by a downstream node when it has active receivers downstream and needs to remain grafted onto the multicast tree at all times, even in the face of topology or group membership dynamics.

Timer/Iface/Prune **Prune Timer** default = 0.5s.
set by an upstream node on specific outgoing links to avoid sending data on those links.

Timer/Iface/Deletion **Interface Deletion Timer** default = 0.1s.
set by a node on a LAN interface to mark that interface as non-forwarding, possibly because another node on that LAN has "assert"ed.

Each timer has a class variable, `timeout_`, that controls the firing interval of that timer, and hence can be individually set. A typical detailedDM script might be:

```
Timer/Iface/Prune set timeout_ 0.3s.                      ;# default = 0.5s.
set mproto dynamicDM
set mrthandle [$ns mrtproto $mproto {}]
```

16.2 Internals of Multicast Routing

We describe the internals in three parts: first the classes to implement and support multicast routing; second, the specific protocol implementation details; and finally, provide a list of the variables that are used in the implementations.

16.2.1 The classes

The main classes in the implementation are the `class mrtObject` and the `class McastProtocol`. There are also extensions to the base classes: Simulator, Node, Classifier, *etc*. We describe these classes and extensions in this subsection. The specific protocol implementations also use adjunct data structures for specific tasks, such as timer mechanisms by detailed dense mode, encapsulation/decapsulation agents for centralised multicast *etc*.; we defer the description of these objects to the section on the description of the particular protocol itself.

mrtObject class There is one mrtObject object per multicast capable node. Each mrtObject object maintain a list of multicast protocols. This arbiter supports the ability for a node to run multiple multicast routing protocols. The node uses the arbiter to perform protocol actions, either to a specific protocol instance active at that node, or to all protocol instances at that node.

`addproto`{instance}	adds the handle for a protocol instance to its list of protocols.
`getType`{protocol}	returns the handle to the protocol instance active at that node, and the NULL string if none exists.
`all-mprotos`{op, args}	internal routine to execute "`op`" with "`args`" on all protocol instances active at that node.
`start`{}	starts execution of all protocols.
`stop`{}	stop execution of all protocols.
`notify`{}	force route recompute of all protocols following a topology change.
`dump-mroutes`{file-handle}	dump multicast routes to specified file-handle.
`join-group`{G, [S]}	signals all protocol instances to join ⟨S, G⟩.
`leave-group`{G, [S]}	signals all protocol instances to leave ⟨S, G⟩.
`upcall`{code, s, g, iif}	signalled by node on forwarding errors in classifier; this routine in turn signals all protocol instances that will then invoke the appropriate handle function for that particular code.
`drop`{rep, s, g, iif}	Called on packet drop, possibly to prune an interface.

In addition, the mrtObject class supports the concept of well known groups, *i.e.*, those groups that do not require explicit protocol support. Two well known groups, `ALL_ROUTERS` and `ALL_PIM_ROUTERS` are predefined in *ns*. The `class mrtObject` defines two class procedures to set and get information about these well known groups.

`registerWellKnownGroups`{name}	assigns `name` a well known group address.
`getWellKnownGroup`{name}	returns the address allocated to well known group, `name`. If `name` is not registered as a well known group, then it returns the address for `ALL_ROUTERS`.

McastProtocol class This is the base class for the implementation of all the multicast protocols. It contains basic multicast functions:

`start`{}, `stop`{}	Set the `status_` of execution of this protocol instance.
`getStatus`{}	return the status of execution of this protocol instance.
`getType`{}	return the type of protocol executed by this instance.
`upcall`{code args}	invoked when the node classifier signals an error, either due to a cache-miss or a wrong-iif for incoming packet. This routine invokes the protocol specific handle, `handle-`⟨`code`⟩{} with specified `args` to handle the signal.

The class also defines wrappers to access interface related information. These directly translate into calls to the node or link; however, these wrappers also perform some additional minimal checking as noted below:

`rpf-nbr`{src}	return the RPF neighbour towards the `src` at that node.
`iface2link`{ifid}	return the link that tags the specified ⟨if id⟩.
`link2iif`{link}	return the label affixed by the specified link. Check if the destination of the link is this node itself. (However, at this time, do nothing if this is not true.)
`link2oif`{link}	return the output object at the head of the specifed link. Check if the link is incident to it. (However, at this time, do nothing if this is not true.)

16.2.2 Extensions to other classes in *ns*

In the earlier chapter describing nodes in *ns* (Chapter 4), we described the internal structure of the node in *ns*. To briefly recap that description, the node entry for a multicast node is the `switch_`. It looks at the highest bit to decide if the destination is a multicast or unicast packet. Multicast packets are forwarded to a multicast classifier which maintains a list of replicators; there is one replicator per ⟨source, group, incoming interface⟩ tuple. Replicators copy the incoming packet and forward to all outgoing interfaces.

Class Node Node support for multicast in two primary ways: as a focal point for access to the multicast protocols, in the areas of address allocation, control and management, and group membership dynamics; and secondly, primitives to access and control interfaces on links incident on that node.

`expandaddr{}`, `allocaddr{}`	Class procedures for address management. `expandaddr{}` increases the address space from 128 multicast capable nodes to $2^{30} - 1$. `allocaddr{}` allocates the next available multicast address.
`start-mcast{}`, `stop-mcast{}`	To start and stop multicast routing at that node.
`notify-mcast{}`	`notify-mcast{}` signals the mrtObject at that node to recompute multicastroutes following a topology change or unicast route update from a neighbour.
`getArbiter{}`	returns a handle to mrtObject operating at that node.
`dump-routes{file-handle}`	to dump the multicast forwarding tables at that node.
`new-group{s g iif code}`	When a multicast data packet is received, and the multicast classifier cannot find the slot corresponding to that data packet, it invokes `Node::new-group{}` to establish the appropriate entry. The code indicates the reason for not finding the slot. Currently there are two possibilities, cache-miss and wrong-iif. This procedure notifies the arbiter instance to establish the new group.
`join-group{a g}`	An ⟨agent⟩ at a node that joins a particular group invokes "`$node join-group ⟨agent⟩ ⟨group⟩`". The node signals the mrtObject to join the particular ⟨group⟩, and adds ⟨agent⟩ to its list of agents at that ⟨group⟩. It then adds ⟨agent⟩ to all replicators associated with ⟨group⟩.
`leave-group{a g}`	`Node::leave-group{}` reverses the process described earlier. It disables the outgoing interfaces to the receiver agents for all the replicators of the group, deletes the receiver agents from the local `Agents_` list; it then invokes the arbiter instance's `leave-group{}`.
`add-mfc{s g iif oiflist}`	`Node::add-mfc{}` adds a multicast forwarding cache entry for a particular ⟨source, group, iif⟩. The mechanism is: (1) create a new replicator (if one does not already exist), (2) update the `replicator_` and `repByGroup_` instance variable arrays at the node, (3) adds all outgoing interfaces and local agents to the appropriate replicator, and finally, (4) invoke the multicast classifier's `add-rep{}` to create a slot for the replicator in the multicast classifier.
`del-mfc{s g oiflist}`	disables each oif in ⟨oiflist⟩ from the replicator for ⟨s g⟩.

The list of primitives accessible at the node to control its interfaces are listed below.

`add-iif`{ifid link},	
`add-oif`{link if}	Invoked during link creation to prep the node about its incoming interface label and outgoing interface object.
`get-oif`{link},	
`get-all-oifs`{}	Return the oif for a specific link, or the oifs for all links incident on this node.
`link2oif`{link}	Returns the nsObject on link that is incident to the node.
`rpf-nbr`{src}	Returns a handle to the neighbour node that is its next hop to the specified ⟨src⟩.
`getReps`{s g}	Returns a handle to the replicator that matches ⟨s, g⟩. If either is a wildcard, then the match is also a wildcard, returning the list of handles to such replicators that match the wildcard.
`getReps-raw`{s g}	As above, but returns a list of ⟨key, handle⟩ pairs.
`clearReps`{s g}	Removes all replicators associated with ⟨s, g⟩.

Class Link and SimpleLink This class provides methods to check the type of link, and the label it affixes on individual packets that traverse it. There is one additional method to actually place the interface objects on this link. These methods are:

`isLan?`{}	
`if-label?`{}	returns the interface label affixed by this link to packets that traverse it.
`enable-mcast`{}	Internal procedure called by the SimpleLink constructor to add appropriate objects and state for multicast. By default, (and for the point-to-point link case) it places a networkInterface object on the link, and signals the nodes on incident on the link about this link.

Class networkInterface This is a simple connector that is placed on each link. It affixes its label id to each packet that traverses it. The packet id is used by the destination node incident on that link to identify the link by which the packet reached it. The label id is configured by the Link constructor. This object is an internal object, and is not designed to be manipulated by user level simulation scripts. The object only supports two methods:

`label`{ifid}	assigns ⟨if id⟩ that this object will affix to each packet.
`label`{}	returns the label that this object affixes to each packet.

The global class variable, `ifnum_`, specifies the next available ⟨ifid⟩ number.

Class Classifier There is one multicast classifier per node. The node stores a reference to this classifier in its instance variable `multiclassifier_`. When this classifier receives a packet, it looks at the ⟨source, group⟩ information in the packet headers, and the interface that the packet arrived from (the incoming interface or iif); using that information, the classifier will identify the slot that should be used to forward that packet. The slot will point to the appropriate replicator.

However, if the slot is invalid, or the classifier does not have an entry for this ⟨source, group, iif⟩, then it will invoke `new-group`{} for the classifier, with one of two codes to identify the problem: 1) `cache-miss` indicates that the classifier did not find any ⟨source, group⟩ entries; 2) `wrong-iif` indicates that the classifier found ⟨source, group⟩ entries, but none matching the interface that this packet arrived over. In this latter situation, this particular data is dropped.

`add-rep`{} creates a slot in the classifier and adds a replicator for ⟨source, group, iif⟩ to that slot.

Class Replicator When a replicator receives a packet, it copies the packet to all of its slots. Each slot points to an outgoing interface for a particular ⟨source, group, iif⟩. If no slot is found, the C++ replicator invokes the class instance procedure drop{} to trigger protocol specific actions. We will describe the protocol specific actions in the next section, when we describe the internal procedures of each of the multicast routing protocols.

There are instance procedures to control the elements in each slot:

insert{oif}	inserting a new outgoing interface to the next available slot.
disable{oif}	disable the slot pointing to the specified oif.
enable{oif}	enable the slot pointing to the specified oif.
is-active{}	returns true if the replicator has at least one active slot.
exists{oif}	returns true if the slot pointing to the specified oif is active.
change-iface{source, group, oldiif, newiif}	modified the iif entry for the particular replicator.

16.2.3 Protocol Internals

We now describe the implementation of the different multicast routing protocol agents.

Centralized Multicast

CtrMcast is inherits from McastProtocol. One CtrMcast agent needs to be created for each node. There is a central CtrMcastComp agent to compute and install multicast routes for the entire topology. Each CtrMcast agent processes membership dynamic commands, and redirects the CtrMcastComp agent to recompute the appropriate routes.

join-group{}	registers the new member with the CtrMCastComp agent, and invokes that agent to recompute routes for that member.
leave-group{}	is the inverse of join-group{}.
handle-cache-miss{}	called when no proper forwarding entry is found for a particular packet source. In case of centralized multicast, it means a new source has started sending data packets. Thus, the CtrMcast agent registers this new source with the CtrMcastComp agent. If there are any members in that group, compute the new multicast tree. If the group is in RPT (shared tree) mode, then 1) create an encapsulation agent at the source; 2) a corresponding decapsulation agent is created at the RP; 3) the two agents are connected by unicast; and 4) the $\langle$S,G$\rangle$ entry points its outgoing interface to the encapsulation agent.

CtrMcastComp is the centralised multicast route computation agent.

`reset-mroutes{}`	resets all multicast forwarding entries.
`compute-mroutes{}`	(re)computes all multicast forwarding entries.
`compute-tree{source, group}`	computes a multicast tree for one source to reach all the receivers in a specific group.
`compute-branch{source, group, member}`	is executed when a receiver joins a multicast group. It could also be invoked by `compute-tree{}` when it itself is recomputing the multicast tree, and has to reparent all receivers. The algorithm starts at the receiver, recursively finding successive next hops, until it either reaches the source or RP, or it reaches a node that is already a part of the relevant multicast tree. During the process, several new replicators and an outgoing interface will be installed.
`prune-branch{source, group, member}`	is similar to `compute-branch{}` except the outgoing interface is disabled; if the outgoing interface list is empty at that node, it will walk up the multicast tree, pruning at each of the intermediate nodes, until it reaches a node that has a non-empty outgoing interface list for the particular multicast tree.

Dense Mode

`join-group{group}`	sends graft messages upstream if ⟨S,G⟩ does not contain any active outgoing slots (*i.e.*, no downstream receivers).
`leave-group{group}`	does not do anything.
`handle-cache-miss{group}`	creates ⟨S,G⟩ with only the outgoing interfaces to each of the child nodes. For basic dense mode, parent-child relationship is computed only once at the start-up. However, for dynamic dense mode, the incoming interface must be set properly so packets from the wrong incoming interfaces will be dropped. Finally, in the case of PIM dense mode, `handle-cache-miss{}` is the same as in dynamic dense mode, except that the outgoing interface list is set to all neighbors excluding the incoming interface.
`drop{replicator, source, group}`	sends prune messages upstream. Recall that the packet is only dropped when there are no downstream receivers for the ⟨S, G⟩ tuple.
`recv-prune{from, source, group}`	resets the prune timer if the interface had been pruned previously; otherwise, it starts the prune timer and disables the interface; furthermore, if the outgoing interface list becomes empty, it forwards the prune message upstream.
`recv-graft{from, source, group}`	cancels any existing prune timer, and re-enables the pruned interface. If the outgoing interface list was previously empty, it forwards the graft upstream.
`periodic-check{}`	This procedure is only used in dynamic dense mode. Each node periodically updates its parent-child relationships with respect to each of its neighbors.
`handle-wrong-iif{}`	This procedure is only used by PIM dense mode. This is invoked when the multicast classifier drops a packet because it arrived on the wrong interface, and invoked `new-group{}`. This routine is invoked by `mrtObject::new-group{}`. When invoked, the agent checks if the incoming interface in the forwarding entry has become outdated. If so, this procedure will update the forwarding entry to the new incoming interface; otherwise, it will send a prune message back towards the wrong incoming interface to stop packets reaching from the wrong path.
`dump-routes{file-handle}`	Dumps the state of replicators installed at the node on the specified file handle.

16.2.4 The internal variables

Class mrtObject	
`protocols_`	A list of handles of protocol instances active at the node at which this protocol operates.
`mask-wkgroups`	Class variable—defines the mask used to identify well known groups.
`wkgroups`	Class array variable—array of allocated well known groups addresses, indexed by the group name. `wkgroups`(Allocd) is a special variable indicating the highest currently allocated well known group.
McastProtocol	
`status_`	takes values "up" or "down", to indicate the status of execution of the protocol instance.
`type_`	returns the type of protocol executed by this instance, *e.g.*, DM, or detailedDM.
Simulator	
`multiSim_`	1 if multicast simulation is enabled, 0 otherwise.
`MrtHandle_`	handle to the centralised multicast simulation object.
Node	
`switch_`	handle for classifier that looks at the high bit of the destination address in each packet to determine whether it is a multicast packet (bit = 1) or a unicast packet (bit = 0).
`multiclassifier_`	handle to classifier that performs the ⟨s, g, iif⟩ match.
`replicator_`	array indexed by ⟨s:g⟩ of handles that replicate a multicast packet on to the required links.
`Agents_`	array indexed by multicast group of the list of agents at the local node that listen to the specific group.
`outLink_`	Cached list of outgoing interfaces at this node.
Link and **SimpleLink**	
`iif_`	handle for the networkInterface object placed on this link.
`head_`	first object on the link, a no-op constructor. However, this object contains the instance variable, `link_`, that points to the container Link object.
networkInterface	
`ifnum_`	Class variable—holds the next available interface number.

Chapter 17

Network Dynamics

This chapter describes the capabilities in *ns* to make the simulation topologies dynamic. We start with the instance procedures to the class Simulator that are useful to a simulation script (Section 17.1). The next section describes the internal architecture (Section 17.2), including the different classes and instance variables and procedures; the following section describes the interaction with unicast routing (Section 17.3). This aspect of network dynamics is still somewhat experimental in *ns*. The last section of this chapter outlines some of the deficiencies in the current realization (Section 17.4) of network dynamics, some one or which may be fixed in the future.

The procedures and functions described in this chapter can be found in ~*ns*/tcl/rtglib/dynamics.tcl and ~*ns*/tcl/lib/route-proto.tcl.

17.1 The user level API

The user level interface to network dynamics is a collection of instance procedures in the class Simulator, and one procedure to trace and log the dynamics activity. Reflecting a rather poor choice of names, these procedures are `rtmodel`, `rtmodel-delete`, and `rtmodel-at`. There is one other procedure, `rtmodel-configure`, that is used internally by the class Simulator to configure the rtmodels just prior to simulation start. We describe this method later (Section 17.2).

— The instance procedure `rtmodel{}` defines a model to be applied to the nodes and links in the topology. Some examples of this command as it would be used in a simulation script are:

```
$ns rtmodel Exponential 0.8 1.0 1.0 $n1
$ns rtmodel Trace dynamics.trc  $n2 $n3
$ns rtmodel Deterministic 20.0 20.0 $node(1) $node(5)
```

The procedure requires at least three arguments:

- The first two arguments define the model that will be used, and the parameters to configure the model.
  The currently implemented models in *ns* are Exponential (On/Off), Deterministic (On/Off), Trace (driven), or Manual (one-shot) models.
- The number, format, and interpretation of the configuration parameters is specific to the particular model.
  1. The exponential on/off model takes four parameters: ⟨[start time], up interval, down interval, [finish time]⟩. ⟨start time⟩ defaults to $0.5s$. from the start of the simulation, ⟨finish time⟩ defaults to the end of the simulation. ⟨up interval⟩ and ⟨down interval⟩ specify the mean of the exponential distribution defining the time that the

node or link will be up and down respectively. The default up and down interval values are $10s$. and $1s$. respectively. Any of these values can be specified as "$-$" to default to the original value.

The following are example specifications of parameters to this model:

```
0.8 1.0 1.0        ; # start at 0.8s., up/down = 1.0s., finish is default
5.0 0.5            ; # start is default, up/down = 5.0s, 0.5s., finish is default
- 0.7              ; # start, up interval are default, down = 0.7s., finish is default
- - - 10           ; # start, up, down are default, finish at 10s.
```

2. The deterministic on/off model is similar to the exponential model above, and takes four parameters: $\langle$[start time], up interval, down interval, [finish time]$\rangle$. $\langle$start time$\rangle$ defaults to the start of the simulation, $\langle$finish time$\rangle$ defaults to the end of the simulation. Only the interpretation of the up and down interval is different; $\langle$up interval$\rangle$ and $\langle$down interval$\rangle$ specify the exact duration that the node or link will be up and down respectively. The default values for these parameters are: $\langle$start time$\rangle$ is $0.5s$. from start of simulation, $\langle$up interval$\rangle$ is $2.0s$., $\langle$down interval$\rangle$ is $1.0s$., and $\langle$finish time$\rangle$ is the duration of the simulation.
3. The trace driven model takes one parameter: the name of the trace file. The format of the input trace file is identical to that output by the dynamics trace modules, *viz.*, `v` $\langle$`time`$\rangle$ `link-`$\langle$`operation`$\rangle$ $\langle$`node1`$\rangle$ $\langle$`node2`$\rangle$. Lines that do not correspond to the node or link specified are ignored.

```
v 0.8123 link-up 3 5
v 3.5124 link-down 3 5
```

4. The manual one-shot model takes two parameters: the operation to be performed, and the time that it is to be performed.

- The rest of the arguments to the `rtmodel{}` procedure define the node or link that the model will be applied to. If only one node is specified, it is assumed that the node will fail. This is modeled by making the links incident on the node fail. If two nodes are specified, then the command assumes that the two are adjacent to each other, and the model is applied to the link incident on the two nodes. If more than two nodes are specified, only the first is considered, the subsequent arguments are ignored.
- instance variable, `traceAllFile_` is set.

The command returns the handle to the model that was created in this call.

Internally, `rtmodel{}` stores the list of route models created in the class Simulator instance variable, `rtModel_`.

— The instance procedure `rtmodel-delete{}` takes the handle of a route model as argument, removes it from the `rtModel_` list, and deletes the route model.

— The instance procedure `rtmodel-at{}` is a special interface to the Manual model of network dynamics.

The command takes the time, operation, and node or link as arguments, and applies the operation to the node or link at the specified time. Example uses of this command are:

```
$ns rtmodel-at 3.5 up $n0
$ns rtmodel-at 3.9 up $n(3) $n(5)
$ns rtmodel-at 40  down  $n4
```

Finally, the instance procedure `trace-dynamics{}` of the class rtModel enables tracing of the dynamics effected by this model. It is used as:

```
set fh [open "dyn.tr" w]
$rtmodel1 trace-dynamics $fh
$rtmodel2 trace-dynamics $fh
$rtmodel1 trace-dynamics stdout
```

In this example, `$rtmodel1` writes out trace entries to both dyn.tr and stdout; `$rtmodel2` only writes out trace entries to dyn.tr. A typical sequence of trace entries written out by either model might be:

```
v 0.8123 link-up 3 5
v 0.8123 link-up 5 3
v 3.5124 link-down 3 5
v 3.5124 link-down 5 3
```

These lines above indicate that Link $\langle 3, 5\rangle$ failed at $0.8123s.$, and recovered at time $3.5124s$.

17.2 The Internal Architecture

Each model of network dynamics is implemented as a separate class, derived from the base `class rtModel`. We begin by describing the base class rtModel and the derived classes (Section 17.2.1). The network dynamics models use an internal queuing structure to ensure that simultaneous events are correctly handled, the `class rtQueue`. The next subsection (Section 17.2.2) describes the internals of this structure. Finally, we describe the extensions to the existing classes (Section 17.3.1): the Node, Link, and others.

17.2.1 The class rtModel

To use a new route model, the routine `rtmodel{}` creates an instance of the appropriate type, defines the node or link that the model will operate upon, configures the model, and possibly enables tracing; The individual instance procedures that accomplish this in pieces are:

The constructor for the base class stores a reference to the Simulator in its instance variable, `ns_`. It also initializes the `startTime_` and `finishTime_` from the class variables of the same name.

The instance procedure set-elements identifies the node or link that the model will operate upon. The command stores two arrays: `links_`, of the links that the model will act upon; `nodes_`, of the incident nodes that will be affected by the link failure or recovery caused by the model.

The default procedure in the base class to set the model configuration parameters is set-parms. It assumes a well defined start time, up interval, down interval, and a finish time, and sets up configuration parameters for some class of models. It stores these values in the instance variables: `startTime_`, `upInterval_`, `downInterval_`, `finishTime_`. The exponential and deterministic models use this default routine, the trace based and manual models define their own procedures.

The instance procedure `trace{}` enables `trace-dynamics{}` on each of the links that it affects. Additional details on `trace-dynamics{}` is discussed in the section on extensions to the class Link (Section 17.3.1).

The next sequence of configuration steps are taken just prior to the start of the simulator. *ns* invokes `rtmodel-configure{}` just before starting the simulation. This instance procedure first acquires an instance of the class rtQueue, and then invokes `configure{}` for each route model in its list, `rtModel_`.

The instance procedure `configure{}` makes each link that is is applied to dynamic; this is the set of links stored in its instance variable array, `links_`. Then the procedure schedules its first event.

The default instance procedure `set-first-event{}` schedules the first event to take all the links "down" at `$startTime_` + `upInterval_`. Individual types of route models derived from this base class should redefine tihs function.

Two instance procedures in the base class , set-event{} and set-event-exact{}, can be used to schedule events in the route queue.

set-event{interval, operation} schedules operation after interval seconds from the current time; it uses the procedure set-event-exact{} below.

set-event-exact{fireTime, operation} schedules operation to execute at fireTime.

If the time for execution is greater than the finishTime_, then the only possible action is to take a failed link "up".

Finally, the base class provides the methods to take the links up{} or down{}. Each method invokes the appropriate procedure on each of the links in the instance variable, links_.

Exponential The model schedules its first event to take the links down at startTime_ + E(upInterval_);

It also defines the procedures, up{} and down{}; each procedure invokes the base class procedure to perform the actual operation. This routine then reschedules the next event at E(upInterval) or E(downInterval_) respectively.

Deterministic The model defines the procedures, up{} and down{}; each procedure invokes the base class procedure to perform the actual operation. This routine then reschedules the next event at upInterval or downInterval_ respectively.

Trace The model redefines the instance procedure set-parms{} to operan a trace file, and set events based on that input.

The instance procedure get-next-event{} returns the next valid event from the trace file. A valid event is an event that is applicable to one of the links in this object's links_ variable.

The instance procedure set-trace-events{} uses get-next-event{} to schedule the next valid event.

The model redefines set-first-event{}, up{}, and down{} to use set-trace-events{}.

Manual The model is designed to fire exactly once. The instance procedure set-parms{} takes an operation and the time to execute that operation as arguments. set-first-event{} will schedule the event at the appropriate moment.

This routine also redefines notify{} to delete the object instance when the operation is completed. This notion of the object deleting itself is fragile code.

Since the object only fires once and does nto have to be rescheduled, it does not overload the procedures up{} or down{}.

17.2.2 class rtQueue

The simulator needs to co-ordinate multiple simultaneous network dynamics events, especially to ensure the right coherent behaviour. Hence, the network dynamics models use their own internal route queue to schedule dynamics events. There is one instance of this object in the simulator, in the class Simulator instance variable rtq_.

The queue object stores an array of queued operations in its instance variable, rtq_. The index is the time at which the event will execute. Each element is the list of operations that will execute at that time.

The instance procedures insq{} and insq-i{} can insert an element into the queue.

The first argument is the time at which this operation will execute. `insq{}` takes the exact time as argument; `insq-i{}` takes the interval as argument, and schedules the operation `interval` seconds after the current time.

The following arguments specify the object, `$obj`, the instance procedure of that object, `$iproc`, and the arguments to that procedure, `$args`.

These arguments are placed into the route queue for execution at the appropriate time.

The instance procedure `runq{}` executes `eval $obj $iproc $args` at the appropriate instant. After all the events for that instance are executed, `runq{}` will `notify{}` each object about the execution.

Finally, the instance procedure `delq{}` can remove a queued action with the time and the name of the object.

17.3 Interaction with Unicast Routing

In an earlier section, we had described how unicast routing reacts (Section 15.4.2) to changes to the topology. This section details the steps by which the network dynamics code will notify the nodes and routing about the changes to the topology.

1. `rtQueue::runq{}` will invoke the procedures specified by each of the route model instances. After all of the actions are completed, `runq{}` will notify each of the models.
2. `notify{}` will then invoke instance procedures at all of the nodes that were incident to the affected links. Each route model stores the list of nodes in its instance variable array, `nodes_`.

   It will then notify the RouteLogic instance of topology changes.
3. The rtModel object invokes the class Node instance procedure `intf-changed{}` for each of the affected nodes.
4. `Node::intf-changed{}` will notify any `rtObject` at the node of the possible changes to the topology.

   Recall that these route objects are created when the simulation uses detailed dynamic unicast routing.

17.3.1 Extensions to Other Classes

The existing classes assume that the topology is static by default. In this section, we document the necessary changes to these classes to support dynamic topologies.

We have already described the instance procedures in the `class Simulator` to create or manipulate route models, *i.e.*, `rtmodel{}`, `rtmodel-at{}`, `rtmodel-delete{}`, and `rtmodel-configure{}` in earlier sections (Section 17.2.1). Similarly, the `class Node` contains the instance procedure `intf-changed{}` that we described in the previous section (Section 17.3).

The network dynamics code operates on individual links. Each model currently translates its specification into operations on the appropriate links. The following paragraphs describe the class Link and related classes.

`class DynamicLink` This class is the only TclObject in the network dynamics code. The shadow class is called `class DynaLink`. The class supports one bound variable, `status_`. `status_` is 1 when the link is up, and 0 when the link is down. The shadow object's `recv()` method checks the `status_` variable, to decide whether or not a packet should be forwarded.

`class Link` This class supports the primitives: up and down, and up? to set and query `status_`. These primitives are instance procedures of the class.

The instance procedures `up{}` and `down{}` set `status_` to 1 and 0 respectively.

In addition, when the link fails, `down{}` will reset all connectors that make up the link. Each connector, including all queues and the delay object will flush and drop any packets that it currently stores. This emulates the packet drop due to link failure.

Both procedures then write trace entries to each file handle in the list, `dynT_`.

The instance procedure `up?{}` returns the current value of `status_`.

In addition, the class contains the instance procedure `all-connectors{}`. This procedure takes an operation as argument, and applies the operation uniformly to all of the class instance variables that are handles for TclObjects.

`class SimpleLink` The class supports two instance procedures `dynamic{}` and `trace-dynamics{}`. We have already described the latter procedure when describing the `trace{}` procedure in the class rtModel.

The instance procedure `dynamic{}` inserts a DynamicLink object (Section 5.2) at the head of the queue. It points the down-target of the object to the drop target of the link, `drpT_`, if the object is defined, or to the `nullAgent_` in the simulator. It also signals each connector in the link that the link is now dynamic.

Most connectors ignore this signal to be become dynamic; the exception is `DelayLink` object. This object will normally schedule each packet it receives for reception by the destination node at the appropriate time. When the link is dynamic, the object will queue each packet internally; it schedules only one event for the next packet that will be delivered, instead of one event per packet normally. If the link fails, the route model will signal a `reset`, at which point, the shadow object will execute its reset instproc-like, and flush all packets in its internal queue. Additional details about the DelayLink can be found in another chapter (Chapter 7).

17.4 Deficencies in the Current Network Dynamics API

There are a number of deficencies in the current API that should be changed in the next iteration:

1. There is no way to specify a cluster of nodes or links that behave in lock-step dynamic synchrony.
2. Node failure should be dealt with as its own mechanism, rather than a second grade citizen of link failure. This shows up in a number of situations, such as:
   (a) The method of emulating node failure as the failure of the incident links is broken. Ideally, node failure should cause all agents incident on the node to be reset.
   (b) There is no tracing associated with node failure.
3. If two distinct route models are applied to two separate links incident on a common node, and the two links experience a topology change at the same instant, then the node will be notified more than once.

Part V

Transport

Chapter 18

UDP Agents

18.1 UDP Agents

UDP agents are implemented in `udp.{cc, h}`. A UDP agent accepts data in variable size chunks from an application, and segments the data if needed. UDP packets also contain a monotonically increasing sequence number and an RTP timestamp. Although real UDP packets do not contain sequence numbers or timestamps, this sequence number does not incur any simulated overhead, and can be useful for tracefile analysis or for simulating UDP-based applications.

The default maximum segment size (MSS) for UDP agents is 1000 byte:

```
Agent/UDP set packetSize_   1000                ;# max segment size
```

This OTcl instvar is bound to the C++ agent variable `size_`.

Applications can access UDP agents via the `sendmsg`() function in C++, or via the `send` or `sendmsg` methods in OTcl, as described in section 21.2.3.

The following is a simple example of how a UDP agent may be used in a program. In the example, the CBR traffic generator is started at time 1.0, at which time the generator begins to periodically call the UDP agent `sendmsg`() function.

```
set ns [new Simulator]
set n0 [$ns node]
set n1 [$ns node]
$ns duplex-link $n0 $n1 5Mb 2ms DropTail

set udp0 [new Agent/UDP]
$ns attach-agent $n0 $udp0
set cbr0 [new Application/Traffic/CBR]
$cbr0 attach-agent $udp0
$udp0 set packetSize_ 536                    ;# set MSS to 536 bytes

set null0 [new Agent/Null]
$ns attach-agent $n1 $null0
$ns connect $udp0 $null0
$ns at 1.0 "$cbr0 start"
```

Chapter 19

TCP Agents

This section describes the operation of the TCP agents in *ns*. There are two major types of TCP agents: one-way agents and a two-way agent. One-way agents are further subdivided into a set of TCP senders (which obey different congestion and error control techniques) and receivers ("sinks"). The two-way agent is symmetric in the sense that it represents both a sender and receiver. It is still under development.

The files described in this section are too numerous to enumerate here. Basically it covers most files matching the regular expression ~*ns*/tcp*.{cc, h}.

The one-way TCP sending agents currently supported are:

- Agent/TCP - a "tahoe" TCP sender
- Agent/TCP/Reno - a "Reno" TCP sender
- Agent/TCP/NewReno - Reno with a modification
- Agent/TCP/Sack1 - TCP with selective repeat (follows RFC2018)
- Agent/TCP/Vegas - TCP Vegas
- Agent/TCP/Fack - Reno TCP with "forward acknowledgment"

The one-way TCP receiving agents currently supported are:

- Agent/TCPSink - TCP sink with one ACK per packet
- Agent/TCPSink/DelAck - TCP sink with configurable delay per ACK
- Agent/TCPSink/Sack1 - selective ACK sink (follows RFC2018)
- Agent/TCPSink/Sack1/DelAck - Sack1 with DelAck

The two-way experimental sender currently supports only a Reno form of TCP:

- Agent/TCP/FullTcp

The section comprises three parts: the first part is a simple overview and example of configuring the base TCP send/sink agents (the sink requires no configuration). The second part describes the internals of the base send agent, and last part is a description of the extensions for the other types of agents that have been included in the simulator.

19.1 One-Way TCP Senders

The simulator supports several versions of an abstracted TCP sender. These objects attempt to capture the essence of the TCP congestion and error control behaviors, but are not intended to be faithful replicas of real-world TCP implementations. They do not contain a dynamic window advertisement, they do segment number and ACK number computations entirely in packet units, there is no SYN/FIN connection establishment/teardown, and no data is ever transferred (e.g. no checksums or urgent data).

19.1.1 The Base TCP Sender (Tahoe TCP)

The "Tahoe" TCP agent `Agent/TCP` performs congestion control and round-trip-time estimation in a way similar to the version of TCP released with the 4.3BSD "Tahoe" UN'X system release from UC Berkeley. The congestion window is increased by one packet per new ACK received during slow-start (when $cwnd\_ < ssthresh\_$) and is increased by $\frac{1}{cwnd\_}$ for each new ACK received during congestion avoidance (when $cwnd\_ \geq ssthresh\_$).

Responses to Congestion Tahoe TCP assumes a packet has been lost (due to congestion) when it observes `NUMDUPACKS` (defined in `tcp.h`, currently 3) duplicate ACKs, or when a retransmission timer expires. In either case, Tahoe TCP reacts by setting `ssthresh_` to half of the current window size (the minimum of `cwnd_` and `window_`) or 2, whichever is larger. It then initializes `cwnd_` back to the value of `windowInit_`. This will typically cause the TCP to enter slow-start.

Round-Trip Time Estimation and RTO Timeout Selection Four variables are used to estimate the round-trip time and set the retransmission timer: `rtt_, srtt_, rttvar_, tcpTick_, and backoff_`. TCP initializes rttvar to $3/tcpTick\_$ and backoff to 1. When any future retransmission timer is set, it's timeout is set to the current time plus $\max(bt(a+4v+1), 64)$ seconds, where b is the current backoff value, t is the value of tcpTick, a is the value of srtt, and v is the value of rttvar.

Round-trip time samples arrive with new ACKs. The RTT sample is computed as the difference between the current time and a "time echo" field in the ACK packet. When the first sample is taken, its value is used as the initial value for `srtt_`. Half the first sample is used as the initial value for `rttvar_`. For subsequent samples, the values are updated as follows:

$$srtt = \frac{7}{8} \times srtt + \frac{1}{8} \times sample$$

$$rttvar = \frac{3}{4} \times rttvar + \frac{1}{4} \times |sample - srtt|$$

19.1.2 Configuration

Running an TCP simulation requires creating and configuring the agent, attaching an application-level data source (a traffic generator), and starting the agent and the traffic generator.

19.1.3 Simple Configuration

Creating the Agent

```
set ns [new Simulator]                                          ;# preamble initialization
set node1 [$ns node]                                            ;# agent to reside on this node
set node2 [$ns node]                                            ;# agent to reside on this node

set tcp1 [$ns create-connection TCP $node1 TCPSink $node2 42]
$tcp  set window_ 50                                            ;# configure the TCP agent

set ftp1 [new Application/FTP]
$ftp1 attach-agent $tcp1

$ns at 0.0 "$ftp start"
```

This example illustrates the use of the simulator built-in function `create-connection`. The arguments to this function are: the source agent to create, the source node, the target agent to create, the target node, and the flow ID to be used on the connection. The function operates by creating the two agents, setting the flow ID fields in the agents, attaching the source and target agents to their respective nodes, and finally connecting the agents (i.e. setting appropriate source and destination addresses and ports). The return value of the function is the name of the source agent created.

TCP Data Source The TCP agent does not generate any application data on its own; instead, the simulation user can connect any traffic generation module to the TCP agent to generate data. Two applications are commonly used for TCP: FTP and Telnet. FTP represents a bulk data transfer of large size, and telnet chooses its transfer sizes randomly from tcplib (see the file `tcplib-telnet.cc`. Details on configuring these application source objects are in Section 21.4.

19.1.4 Other Configuration Parameters

In addition to the `window_` parameter listed above, the TCP agent supports additional configuration variables. Each of the variables described in this subsection is both a class variable and an instance variable. Changing the class variable changes the default value for all agents that are created subsequently. Changing the instance variable of a particular agent only affects the values used by that agent. For example,

```
  Agent/TCP set window_ 100                                     ;# Changes the class variable
  $tcp set window_ 2.0                                          ;# Changes window_ for the $tcp object only
```

The default parameters for each TCP agent are:

```
Agent/TCP set window_   20                  ;# max bound on window size
Agent/TCP set windowInit_ 1                 ;# initial/reset value of cwnd
Agent/TCP set windowOption_ 1               ;# cong avoid algorithm (1: standard)
Agent/TCP set windowConstant_ 4             ;# used only when windowOption != 1
Agent/TCP set windowThresh_ 0.002           ;# used in computing averaged window
Agent/TCP set overhead_ 0                   ;# !=0 adds random time between sends
Agent/TCP set ecn_ 0                        ;# TCP should react to ecn bit
Agent/TCP set packetSize_ 1000              ;# packet size used by sender (bytes)
Agent/TCP set bugFix_ true                  ;# see explanation
Agent/TCP set slow_start_restart_ true      ;# see explanation
Agent/TCP set tcpTick_ 0.1                  ;# timer granulatiry in sec (.1 is NONSTANDARD)
Agent/TCP set maxrto_ 64                    ;# bound on RTO (seconds)
Agent/TCP set dupacks_ 0                    ;# duplicate ACK counter
Agent/TCP set ack_ 0                        ;# highest ACK received
```

```
Agent/TCP set cwnd_ 0             ;# congestion window (packets)
Agent/TCP set awnd_ 0             ;# averaged cwnd (experimental)
Agent/TCP set ssthresh_ 0         ;# slow-stat threshold (packets)
Agent/TCP set rtt_ 0              ;# rtt sample
Agent/TCP set srtt_ 0             ;# smoothed (averaged) rtt
Agent/TCP set rttvar_ 0           ;# mean deviation of rtt samples
Agent/TCP set backoff_ 0          ;# current RTO backoff factor
Agent/TCP set maxseq_ 0           ;# max (packet) seq number sent
```

For many simulations, few of the configuration parameters are likely to require modification. The more commonly modified parameters include: `window_` and `packetSize_`. The first of these bounds the window TCP uses, and is considered to play the role of the receiver's advertised window in real-world TCP (although it remains constant). The packet size essentially functions like the MSS size in real-world TCP. Changes to these parameters can have a profound effect on the behavior of TCP. Generally, those TCPs with larger packet sizes, bigger windows, and smaller round trip times (a result of the topology and congestion) are more agressive in acquiring network bandwidth.

19.1.5 Other One-Way TCP Senders

Reno TCP The Reno TCP agent is very similar to the Tahoe TCP agent, except it also includes *fast recovery*, where the current congestion window is "inflated" by the number of duplicate ACKs the TCP sender has received before receiving a new ACK. A "new ACK" refers to any ACK with a value higher than the higest seen so far. In addition, the Reno TCP agent does not return to slow-start during a fast retransmit. Rather, it reduces sets the congestion window to half the current window and resets `ssthresh_` to match this value.

NewReno TCP This agent is based on the Reno TCP agent, but which modifies the action taken when receiving new ACKS. In order to exit fast recovery, the sender must receive an ACK for the highest sequence number sent. Thus, new "partial ACKs" (those which represent new ACKs but do not represent an ACK for all outstanding data) do not deflate the window (and possibly lead to a stall, characteristic of Reno).

Vegas TCP This agent implements "Vegas" TCP ([2, 3]). It was contributed by Ted Kuo.

Sack TCP This agent implements selective repeat, based on selective ACKs provided by the receiver. It follows the ACK scheme described in [10], and was developed with Matt Mathis and Jamshid Mahdavi.

Fack TCP This agent implements "forward ACK" TCP, a modification of Sack TCP described in [9].

19.2 TCP Receivers (sinks)

The TCP senders described above represent one-way data senders. They must peer with a "TCP sink" object.

19.2.1 The Base TCP Sink

The base TCP sink object (`Agent/TCPSink`) is responsible for returning ACKs to a peer TCP source object. It generates one ACK per packet received. The size of the ACKs may be configured. The creation and configuration of the TCP sink object is generally performed automatically by a library call (see `create-connection` above).

configuration parameters

```
Agent/TCPSink set packetSize_ 40
```

19.2.2 Delayed-ACK TCP Sink

A delayed-ACK sink object (`Agent/Agent/TCPSink/DelAck`) is available for simulating a TCP receiver that ACKs less than once per packet received. This object contains a bound variable `interval_` which gives the number of seconds to wait between ACKs. The delayed ACK sink implements an agressive ACK policy whereby only ACKs for in-order packets are delayed. Out-of-order packets cause immediate ACK generation.

configuration parameters

```
Agent/TCPSink/DelAck set interval_ 100ms
```

19.2.3 Sack TCP Sink

The selective-acknowledgment TCP sink (`Agent/TCPSink/Sack1`) implements SACK generation modeled after the description of SACK in RFC 2018. This object includes a bound variable `maxSackBlocks_` which gives the maximum number of blocks of information in an ACK available for holding SACK information. The default value for this variable is 3, in accordance with the expected use of SACK with RTTM (see RFC 2018, section 3). Delayed and selective ACKs together are implemented by an object of type `Agent/TCPSink/Sack1/DelAck`.

configuration parameters

```
Agent/TCPSink set maxSackBlocks_ 3
```

19.3 Two-Way TCP Agents (FullTcp)

The `Agent/TCP/FullTcp` object is a new addition to the suite of TCP agents supported in the simulator and is still under development. It is different from (and incompatible with) the other agents, but does use some of the same architecture. It differs from these agents in the following ways: following ways:

- connections may be establised and town down (SYN/FIN packets are exchanged)
- bidirectional data transfer is supported

- sequence numbers are in bytes rather than packets

The generation of SYN packets (and their ACKs) can be of critical importance in trying to model real-world behavior when using many very short data transfers. This version of TCP currently defaults to sending data on the 3rd segment of an initial 3-way handshake, a behavior somewhat different than common real-world TCP implementations. A "typical" TCP connection proceeds with an active opener sending a SYN, the passive opener responding with a SYN+ACK, the active opener responding with an ACK, and then some time later sending the first segment with data (corresponding to the first application write). Thus, this version of TCP sends data at a time somewhat earlier than typical implementations. This TCP can also be configured to send data on the initial SYN segment. Future changes to FullTCP may include a modification to send the first data segment later, and possibly to implement T/TCP functionality.

Currently FullTCP is only implemented with Reno congestion control, but ultimately it should be available with the full range of congestion control algorithms (e.g., Tahoe, SACK, Vegas, etc.).

19.3.1 Simple Configuration

Running an Full TCP simulation requires creating and configuring the agent, attaching an application-level data source (a traffic generator), and starting the agent and the traffic generator.

Creating the Agent

```
# set up connection (do not use "create-connection" method because
# we need a handle on the sink object)
set src [new Agent/TCP/FullTcp]                    ;# create agent
set sink [new Agent/TCP/FullTcp]                   ;# create agent
$ns_ attach-agent $node_(s1) $src                  ;# bind src to node
$ns_ attach-agent $node_(k1) $sink                 ;# bind sink to node
$src set fid_ 0                                    ;# set flow ID field
$sink set fid_ 0                                   ;# set flow ID field
$ns_ connect $src $sink                            ;# active connection src to sink

# set up TCP-level connections
$sink listen                                       ;# will figure out who its peer is
$src set window_ 100;
```

The creation of the FullTcp agent is similar to the other agents, but the sink is placed in a listening state by the `listen` method. Because a handle to the receiving side is required in order to make this call, the `create-connection` call used above cannot be used.

Configuration Parameters The following configuration parameters are available through Tcl for the FullTcp agent:

```
Agent/TCP/FullTcp set segsperack_ 1          ;# segs received before generating ACK
Agent/TCP/FullTcp set segsize_ 536           ;# segment size (MSS size for bulk xfers)
Agent/TCP/FullTcp set tcprexmtthresh_ 3      ;# dupACKs thresh to trigger fast rexmt
Agent/TCP/FullTcp set iss_ 0                 ;# initial send sequence number
Agent/TCP/FullTcp set nodelay_ false         ;# disable sender-side Nagle algorithm
Agent/TCP/FullTcp set data_on_syn_ false     ;# send data on initial SYN?
```

```
Agent/TCP/FullTcp set dupseg_fix_ true                ;# avoid fast rxt due to dup segs+acks
Agent/TCP/FullTcp set dupack_reset_ false ;# reset dupACK ctr on !0 len data segs containing dup ACKs
Agent/TCP/FullTcp set interval_ 0.1                   ;# as in TCP above, (100ms is non-std)
```

19.4 Architecture and Internals

The base TCP agent (class `Agent/TCP`) is constructed as a collection of routines for sending packets, processing ACKs, managing the send window, and handling timeouts. Generally, each of these routines may be over-ridden by a function with the same name in a derived class (this is how many of the TCP sender variants are implemented).

The TCP header The TCP header is defined by the `hdr_tcp` structure in the file ~*ns*/tcp.h. The base agent only makes use of the following subset of the fields:

```
ts_              /* current time packet was sent from source */
ts_echo_         /* for ACKs: timestamp field from packet associated with this ACK */
seqno_           /* sequence number for this data segment or ACK (Note: overloading!) */
reason_          /* set by sender when (re)transmitting to trace reason for send */
```

Functions for Sending Data Note that generally the sending TCP never actually sends data (it only sets the packet size).

send_much(force, reason, maxburst) - this function attempts to send as many packets as the current sent window allows. It also keeps track of how many packets it has sent, and limits to the total to *maxburst*.
The function `output(seqno, reason)` sends one packet with the given sequence number and updates the maximum sent sequence number variable (`maxseq_`) to hold the given sequence number if it is the greatest sent so far. This function also assigns the various fields in the TCP header (sequence number, timestamp, reason for transmission). This function also sets a retransmission timer if one is not already pending.

Functions for Window Management The usable send window at any time is given by the function **window()**. It returns the minimum of the congestion window and the variable `wnd_`, which represents the receiver's advertised window.

opencwnd() - this function opens the congestion window. It is invoked when a new ACK arrives. When in slow-start, the function merely increments `cwnd_` by each ACK received. When in congestion avoidance, the standard configuration increments `cwnd_` by its reciprocal. Other window growth options are supported during congestion avoidance, but they are experimental (and not documented; contact Sally Floyd for details).

closecwnd(int how) - this function reduces the congestion window. It may be invoked in several ways: when entering fast retransmit, due to a timer expiration, or due to a congestion notification (ECN bit set). Its argument `how` indicates how the congestion window should be reduced. The value **0** is used for retransmission timeouts and fast retransmit in Tahoe TCP. It typically causes the TCP to enter slow-start and reduce `ssthresh_` to half the current window. The value **1** is used by Reno TCP for implementing fast recovery (which avoids returning to slow-start). The value **2** is used for reducing the window due to an ECN indication. It resets the congestion window to its initial value (usually causing slow-start), but does not alter `ssthresh_`.

Functions for Processing ACKs **recv()** - this function is the main reception path for ACKs. Note that because only one direction of data flow is in use, this function should only ever be invoked with a pure ACK packet (i.e. no data). The function stores the timestamp from the ACK in `ts_peer_`, and checks for the presence of the ECN bit (reducing the send window if appropriate). If the ACK is a new ACK, it calls **newack()**, and otherwise checks to see if it is a duplicate of the last ACK seen. If so, it enters fast retransmit by closing the window, resetting the retransmission timer, and sending a packet by calling `send_much`.

newack() - this function processes a "new" ACK (one that contains an ACK number higher than any seen so far). The function sets a new retransmission timer by calling **newtimer()**, updates the RTT estimation by calling **rtt_update**, and updates the highest and last ACK variables.

Functions for Managing the Retransmission Timer These functions serve two purposes: estimating the round-trip time and setting the actual retransmission timer. **rtt_init** - this function initializes `srtt_` and `rtt_` to zero, sets `rttvar_` to $3/tcp\_tick\_$, and sets the backoff multiplier to one.

rtt_timeout - this function gives the timeout value in seconds that should be used to schedule the next retransmission timer. It computes this based on the current estimates of the mean and deviation of the round-trip time. In addition, it implements Karn's exponential timer backoff for multiple consecutive retransmission timeouts.

rtt_update - this function takes as argument the measured RTT and averages it in to the running mean and deviation estimators according to the description above. Note that `t_srtt_` and `t_rttvar` are both stored in fixed-point (integers). They have 3 and 2 bits, respectively, to the right of the binary point.

reset_rtx_timer - This function is invoked during fast retransmit or during a timeout. It sets a retransmission timer by calling `set_rtx_timer` and if invoked by a timeout also calls `rtt_backoff`.

rtt_backoff - this function backs off the retransmission timer (by doubling it).

newtimer - this function called only when a new ACK arrives. If the sender's left window edge is beyond the ACK, then `set_rtx_timer` is called, otherwise if a retransmission timer is pending it is cancelled.

19.5 Tracing TCP Dynamics

The behavior of TCP is often observed by constructing a sequence number-vs-time plot. Typically, a trace is performed by enabling tracing on a link over which the TCP packets will pass. Two trace methods are supported: the default one (used for tracing TCP agents), and an extension used only for FullTcP.

19.6 One-Way Trace TCP Trace Dynamics

TCP packets generated by one of the one-way TCP agents and destined for a TCP sink agent passing over a traced link (see section 14) will generate a trace file lines of the form:

```
+ 0.94176 2 3 tcp 1000 ------ 0 0.0 3.0 25 40
+ 0.94276 2 3 tcp 1000 ------ 0 0.0 3.0 26 41
d 0.94276 2 3 tcp 1000 ------ 0 0.0 3.0 26 41
+ 0.95072 2 0 ack 40 ------ 0 3.0 0.0 14 29
- 0.95072 2 0 ack 40 ------ 0 3.0 0.0 14 29
- 0.95176 2 3 tcp 1000 ------ 0 0.0 3.0 21 36
+ 0.95176 2 3 tcp 1000 ------ 0 0.0 3.0 27 42
```

The exact format of this trace file is given in section 14.4. When tracing TCP, packets of type tcp or ack are relevant. Their type, size, sequence number (ack number for ack packets), and arrival/depart/drop time are given by field positions 5, 6, 11, and 2, respectively. The + indicates a packet arrival, d a drop, and - a departure. A number of scripts process this file to produce graphical output or statistical summaries (see, for example, ~*ns*/test-suite.tcl, the `finish` procedure.

19.7 One-Way Trace TCP Trace Dynamics

TCP packets generated by FullTcp and passing over a traced link contain additional information not displayed by default using the regular trace object. By enabling the flag `show_tcphdr_` on the trace object (see section refsec:traceformat), 3 additional header fields are written to the trace file: ack number, tcp-specific flags, and header length.

Chapter 20

Agent/SRM

This chapter describes the internals of the SRM implementation in *ns*. The chapter is in three parts: the first part is an overview of a minimal SRM configuration, and a "complete" description of the configuration parameters of the base SRM agent. The second part describes the architecture, internals, and the code path of the base SRM agent. The last part of the chapter is a description of the extensions for other types of SRM agents that have been attempted to date.

The procedures and functions described in this chapter can be found in ~*ns*/tcl/mcast/srm.tcl, ~*ns*/tcl/mcast/srm-adaptive.tcl, ~*ns*/tcl/mcast/srm-nam.tcl, ~*ns*/tcl/mcast/srm-debug.tcl, and ~*ns*/srm.{cc, h}.

20.1 Configuration

Running an SRM simulation requires creating and configuring the agent, attaching an application-level data source (a traffic generator), and starting the agent and the traffic generator.

20.1.1 Trivial Configuration

Creating the Agent

```
set ns [new Simulator]                              ;# preamble initialization
$ns enableMcast
set node [$ns node]                                 ;# agent to reside on this node
set group [$ns allocaddr]                           ;# multicast group for this agent

set srm [new Agent/SRM]
$srm  set dst_ $group                               ;# configure the SRM agent
$ns attach-agent $node $srm

$srm set fid_ 1                                     ;# optional configuration
$srm log [open srmStats.tr w]                       ;# log statistics in this file
$srm trace [open srmEvents.tr w]                    ;# trace events for this agent
```

The key steps in configuring a virgin SRM agent are to assign its multicast group, and attach it to a node.

Other useful configuration parameters are to assign a separate flow id to traffic originating from this agent, to open a log file for statistics, and a trace file for trace data[1].

The file `tcl/mcast/srm-nam.tcl` contains definitions that overload the agent's `send` methods; this separates control traffic originating from the agent by type. Each type is allocated a separate flowID. The traffic is separated into session messages (flowid = 40), requests (flowid = 41), and repair messages (flowid = 42). The base flowid can be changed by setting global variable `ctrlFid` to one less than the desired flowid before sourcing `srm-nam.tcl`. To do this, the simulation script must source `srm-nam.tcl` before creating any SRM agents. This is useful for analysis of traffic traces, or for visualization in nam.

Application Data Handling The agent does not generate any application data on its own; instead, the simulation user can connect any traffic generation module to any SRM agent to generate data. The following code demonstrates how a traffic generation agent can be attached to an SRM agent:

```
set packetSize 210
set exp0 [new Application/Traffic/Exponential]          ;# configure traffic generator
$exp0 set packet_size_ $packetSize
$exp0 set burst_time_ 500ms
$exp0 set idle_time_ 500ms
$exp0 set rate_ 100k

$exp0 attach-agent $srm0                    ;# attach application to SRM agent
$srm0 set packetSize_ $packetSize           ;# to generate repair packets of appropriate size
$srm0 set tg_ $exp0                         ;# pointer to traffic generator object
$srm0 set app_fid_ 0                        ;# fid value for packets generated by traffic generator
```

The user can attach any traffic generator to an SRM agent. The SRM agent will add the SRM headers, set the destination address to the multicast group, and deliver the packet to its target. The SRM header contains the type of the message, the identity of the sender, the sequence number of the message, and (for control messages), the round for which this message is being sent. Each data unit in SRM is identified as ⟨sender's id, message sequence number⟩.

The SRM agent does not generate its own data; it does not also keep track of the data sent, except to record the sequence numbers of messages received in the event that it has to do error recovery. Since the agent has no actual record of past data, it needs to know what packet size to use for each repair message. Hence, the instance variable `packetSize_` specifies the size of repair messages generated by the agent.

Starting the Agent and Traffic Generator The agent and the traffic generator must be started separately.

```
$srm start
$exp0 start
```

Alternatively, the traffic generator can be started from the SRM Agent:

```
$srm0 start-source
```

At `start`, the agent joins the multicast group, and starts generating session messages. The `start-source` triggers the traffic generator to start sending data.

[1]Note that the trace data can also be used to gather certain kinds of trace data. We will illustrate this later.

20.1.2 Other Configuration Parameters

In addition to the above parameters, the SRM agent supports additional configuration variables. Each of the variables described in this section is both an OTcl class variable and an OTcl object's instance variable. Changing the class variable changes the default value for all agents that are created subsequently. Changing the instance variable of a particular agent only affects the values used by that agent. For example,

```
Agent/SRM set D1_ 2.0                    ;# Changes the class variable
$srm set D1_ 2.0           ;# Changes D1_ for the particular $srm object only
```

The default request and repair timer parameters [6] for each SRM agent are:

```
Agent/SRM set C1_      2.0                      ;# request parameters
Agent/SRM set C2_      2.0
Agent/SRM set D1_      1.0                      ;# repair parameters
Agent/SRM set D2_      1.0
```

It is thus possible to trivially obtain two flavors of SRM agents based on whether the agents use probabilistic or deterministic suppression by using the following definitions:

```
Class Agent/SRM/Deterministic -superclass Agent/SRM
Agent/SRM/Deterministic set C2_ 0.0
Agent/SRM/Deterministic set D2_ 0.0

Class Agent/SRM/Probabilistic -superclass Agent/SRM
Agent/SRM/Probabilistic set C1_ 0.0
Agent/SRM/Probabilistic set D1_ 0.0
```

In a later section (Section 20.7), we will discuss other ways of extending the SRM agent.

Timer related functions are handled by separate objects belonging to the class SRM. Timers are required for loss recovery and sending periodic session messages. There are loss recovery objects to send request and repair messages. The agent creates a separate request or repair object to handle each loss. In contrast, the agent only creates one session object to send periodic session messages. The default classes the express each of these functions are:

```
Agent/SRM set requestFunction_  "SRM/request"
Agent/SRM set repairFunction_   "SRM/repair"
Agent/SRM set sessionFunction_  "SRM/session"

Agent/SRM set requestBackoffLimit_     5        ;# parameter to requestFunction_
Agent/SRM set sessionDelay_            1.0      ;# parameter to sessionFunction_
```

The instance procedures `requestFunction{}`, `repairFunction{}`, and `sessionFunction{}` can be used to change the default function for individual agents. The last two lines are specific parameters used by the request and session objects. The following section (Section 20.2) describes the implementation of theses objects in greater detail.

20.1.3 Statistics

Each agent tracks two sets of statistics: statistics to measure the response to data loss, and overall statistics for each request/repair. In addition, there are methods to access other information from the agent.

Data Loss The statistics to measure the response to data losses tracks the duplicate requests (and repairs), and the average request (and repair) delay. The algorithm used is documented in Floyd *et al*[6]. In this algorithm, each new request (or repair) starts a new request (or repair) period. During the request (or repair) period, the agent measures the number of first round duplicate requests (or repairs) until the round terminates either due to receiving a request (or repair), or due to the agent sending one. The following code illustrates how the user can simple retrieve the current values in an agent:

```
set statsList [$srm array get statistics_]
array set statsArray [$srm array get statistics_]
```

The first form returns a list of key-value pairs. The second form loads the list into the `statsArray` for further manipulation. The keys of the array are `dup-req`, `ave-dup-req`, `req-delay`, `ave-req-delay`, `dup-rep`, `ave-dup-rep`, `rep-delay`, and `ave-rep-delay`.

Overall Statistics In addition, each loss recovery and session object keeps track of times and statistics. In particular, each object records its `startTime`, `serviceTime`, `distance`, as are relevant to that object; startTime is the time that this object was created, serviceTime is the time for this object to complete its task, and the distance is the one-way time to reach the remote peer.

For request objects, startTime is the time a packet loss is detected, serviceTime is the time to finally receive that packet, and distance is the distance to the original sender of the packet. For repair objects, startTime is the time that a request for retransmission is received, serviceTime is the time send a repair, and the distance is the distance to the original requester. For both types of objects, the serviceTime is normalized by the distance. For the session object, startTime is the time that the agent joins the multicast group. serviceTime and distance are not relevant.

Each object also maintains statistics particular to that type of object. Request objects track the number of duplicate requests and repairs received, the number of requests sent, and the number of times this object had to backoff before finally receiving the data. Repair objects track the number of duplicate requests and repairs, as well as whether or not this object for this agent sent the repair. Session objects simply record the number of session messages sent.

The values of the timers and the statistics for each object are written to the log file every time an object completes the error recovery function it was tasked to do. The format of this trace file is:

```
                ⟨prefix⟩ ⟨id⟩ ⟨times⟩ ⟨stats⟩
where
⟨prefix⟩ is        ⟨time⟩ n ⟨node id⟩ m ⟨msg id⟩ r ⟨round⟩
                ⟨msg id⟩ is expressed as ⟨source id:sequence number⟩
⟨id⟩ is            type ⟨of object⟩
⟨times⟩ is         list of key-value pairs of startTime, serviceTime, distance
⟨stats⟩ is         list of key-value pairs of per object statistics
                dupRQST, dupREPR, #sent, backoff              for request objects
                dupRQST, dupREPR, #sent                       for repair objects
                #sent                                         for session objects
```

The following sample output illustrates the output file format (the lines have been folded to fit on the page):

```
3.6274 n 0 m <1:1> r 1 type repair serviceTime 0.500222 \
       startTime 3.5853553333333332 distance 0.0105 #sent 1 dupREPR 0 dupRQST 0
3.6417 n 1 m <1:1> r 2 type request serviceTime 2.66406 \
       startTime 3.5542666666666665 distance 0.0105 backoff 1 #sent 1 dupREPR 0 dupRQST 0
3.6876 n 2 m <1:1> r 2 type request serviceTime 1.33406 \
       startTime 3.5685333333333333 distance 0.021 backoff 1 #sent 0 dupREPR 0 dupRQST 0
3.7349 n 3 m <1:1> r 2 type request serviceTime 0.876812 \
       startTime 3.5828000000000002 distance 0.032 backoff 1 #sent 0 dupREPR 0 dupRQST 0
3.7793 n 5 m <1:1> r 2 type request serviceTime 0.669063 \
       startTime 3.5970666666666671 distance 0.042 backoff 1 #sent 0 dupREPR 0 dupRQST 0
3.7808 n 4 m <1:1> r 2 type request serviceTime 0.661192 \
       startTime 3.5970666666666671 distance 0.0425 backoff 1 #sent 0 dupREPR 0 dupRQST 0
```

Miscellaneous Information Finally, the user can use the following methods to gather additional information about the agent:

- `groupSize?{}` returns the agent's current estimate of the multicast group size.
- `distances?{}` returns a list of key-value pairs of distances; the key is the address of the agent, the value is the estimate of the distance to that agent. The first element is the address of this agent, and the distance of 0.
- `distance?{}` returns the distance to the particular agent specified as argument.

  The default distance at the start of any simulation is 1.

```
$srm(i) groupSize?            ;# returns $srm(i)'s estimate of the group size
$srm(i) distances?            ;# returns list of ⟨address, distance⟩ tuples
$srm(i) distance? 257         ;# returns the distance to agent at address 257
```

20.1.4 Tracing

Each object writes out trace information that can be used to track the progress of the object in its error recovery. Each trace entry is of the form:

`⟨prefix⟩ ⟨tag⟩ ⟨type of entry⟩ ⟨values⟩`

The prefix is as describe in the previous section for statistics. The tag is **Q** for request objects, **P** for repair objects, and **S** for session objects. The following types of trace entries and parameters are written by each object:

Tag	Type of Object	Other values	Comments
Q	DETECT		
Q	INTERVALS	C1 ⟨C1_⟩ C2 ⟨C2_⟩ dist ⟨distance⟩ i ⟨backoff_⟩	
Q	NTIMER	at ⟨time⟩	Time the request timer will fire
Q	SENDNACK		
Q	NACK	IGNORE-BACKOFF ⟨time⟩	Receive NACK, ignore other NACKs until ⟨time⟩
Q	REPAIR	IGNORES ⟨time⟩	Receive REPAIR, ignore NACKs until ⟨time⟩
Q	DATA		Agent receives data instead of repair. Possibly indicates out of order arrival of data.
P	NACK	from ⟨requester⟩	Receive NACK, initiate repair
P	INTERVALS	D1 ⟨D1_⟩ D2 ⟨D2_⟩ dist ⟨distance⟩	
P	RTIMER	at ⟨time⟩	Time the repair timer will fire
P	SENDREP		
P	REPAIR	IGNORES ⟨time⟩	Receive REPAIR, ignore NACKs until ⟨time⟩
P	DATA		Agent receives data instead of repair. Indicates premature request by an agent.
S	SESSION		logs session message sent

The following illustrates a typical trace for a single loss and recovery.

```
3.5543 n 1 m <1:1> r 0 Q DETECT
3.5543 n 1 m <1:1> r 1 Q INTERVALS C1 2.0 C2 0.0 d 0.0105 i 1
3.5543 n 1 m <1:1> r 1 Q NTIMER at 3.57527
3.5685 n 2 m <1:1> r 0 Q DETECT
3.5685 n 2 m <1:1> r 1 Q INTERVALS C1 2.0 C2 0.0 d 0.021 i 1
3.5685 n 2 m <1:1> r 1 Q NTIMER at 3.61053
3.5753 n 1 m <1:1> r 1 Q SENDNACK
3.5753 n 1 m <1:1> r 2 Q INTERVALS C1 2.0 C2 0.0 d 0.0105 i 2
3.5753 n 1 m <1:1> r 2 Q NTIMER at 3.61727
3.5753 n 1 m <1:1> r 2 Q NACK IGNORE-BACKOFF 3.59627
3.5828 n 3 m <1:1> r 0 Q DETECT
3.5828 n 3 m <1:1> r 1 Q INTERVALS C1 2.0 C2 0.0 d 0.032 i 1
3.5828 n 3 m <1:1> r 1 Q NTIMER at 3.6468
3.5854 n 0 m <1:1> r 0 P NACK from 257
3.5854 n 0 m <1:1> r 1 P INTERVALS D1 1.0 D2 0.0 d 0.0105
3.5854 n 0 m <1:1> r 1 P RTIMER at 3.59586
3.5886 n 2 m <1:1> r 2 Q INTERVALS C1 2.0 C2 0.0 d 0.021 i 2
3.5886 n 2 m <1:1> r 2 Q NTIMER at 3.67262
3.5886 n 2 m <1:1> r 2 Q NACK IGNORE-BACKOFF 3.63062
3.5959 n 0 m <1:1> r 1 P SENDREP
3.5959 n 0 m <1:1> r 1 P REPAIR IGNORES 3.62736
3.5971 n 4 m <1:1> r 0 Q DETECT
3.5971 n 4 m <1:1> r 1 Q INTERVALS C1 2.0 C2 0.0 d 0.0425 i 1
3.5971 n 4 m <1:1> r 1 Q NTIMER at 3.68207
3.5971 n 5 m <1:1> r 0 Q DETECT
3.5971 n 5 m <1:1> r 1 Q INTERVALS C1 2.0 C2 0.0 d 0.042 i 1
3.5971 n 5 m <1:1> r 1 Q NTIMER at 3.68107
3.6029 n 3 m <1:1> r 2 Q INTERVALS C1 2.0 C2 0.0 d 0.032 i 2
```

```
3.6029 n 3 m <1:1> r 2 Q NTIMER at 3.73089
3.6029 n 3 m <1:1> r 2 Q NACK IGNORE-BACKOFF 3.66689
3.6102 n 1 m <1:1> r 2 Q REPAIR IGNORES 3.64171
3.6172 n 4 m <1:1> r 2 Q INTERVALS C1 2.0 C2 0.0 d 0.0425 i 2
3.6172 n 4 m <1:1> r 2 Q NTIMER at 3.78715
3.6172 n 4 m <1:1> r 2 Q NACK IGNORE-BACKOFF 3.70215
3.6172 n 5 m <1:1> r 2 Q INTERVALS C1 2.0 C2 0.0 d 0.042 i 2
3.6172 n 5 m <1:1> r 2 Q NTIMER at 3.78515
3.6172 n 5 m <1:1> r 2 Q NACK IGNORE-BACKOFF 3.70115
3.6246 n 2 m <1:1> r 2 Q REPAIR IGNORES 3.68756
3.6389 n 3 m <1:1> r 2 Q REPAIR IGNORES 3.73492
3.6533 n 4 m <1:1> r 2 Q REPAIR IGNORES 3.78077
3.6533 n 5 m <1:1> r 2 Q REPAIR IGNORES 3.77927
```

The logging of request and repair traces is done by `SRM::evTrace{}`. However, the routine `SRM/Session::evTrace{}`, overrides the base class definition of `srm::evTrace{}`, and writes out nothing. Individual simulation scripts can override these methods for greater flexibility in logging options. One possible reason to override these methods might to reduce the amount of data generated; the new procedure could then generate compressed and processed output.

Notice that the trace filoe contains sufficient information and details to derive most of the statistics written out in the log file, or is stored in the statistics arrays.

20.2 Architecture and Internals

The SRM agent implementation splits the protocol functions into packet handling, loss recovery, and session message activity.

> Packet handling consists of forwarding application data messages, sending and receipt of control messages. These activities are executed by C++ methods.
>
> Error detection is done in C++ due to receipt of messages. However, the loss recovery is entirely done through instance procedures in OTcl.
>
> The sending and processing of messages is accomplished in C++; the policy about when these messages should be sent is decided by instance procedures in OTcl.

We first describe the C++ processing due to receipt of messages (Section 20.3). Loss recovery and the sending of session messages involves timer based processing. The agent uses a separate `class SRM` to perform the timer based functions. For each loss, an agent may do either request or repair processing. Each agent will instantiate a separate loss recovery object for every loss, as is appropriate for the processing that it has to do. In the following section we describe the basic timer based functions and the loss recovery mechanisms (Section 20.5). Finally, each agent uses one timer based function for sending periodic session messages (Section 20.6).

20.3 Packet Handling: Processing received messages

The `recv()` method can receive four type of messages: data, request, repair, and session messages.

Data Packets The agent does not generate any data messages. The user has to specify an external agent to generate traffic. The `recv()` method must distinguish between locally originated data that must be sent to the multicast group, and data received from multicast group that must be processed. Therefore, the application agent must set the packet's destination address to zero.

For locally originated data, the agent adds the appropriate SRM headers, sets the destination address to the multicast group, and forwards the packet to its target.

On receiving a data message from the group, `recv_data`(sender, msgid) will update its state marking message ⟨sender, msgid⟩ received, and possibly trigger requests if it detects losses. In addition, if the message was an older message received out of order, then there must be a pending request or repair that must be cleared. In that case, the compiled object invokes the OTcl instance procedure, `recv-data`{sender, msgid}[2].

Currently, there is no provision for the receivers to actually receive any application data. The agent does not also store any of the user data. It only generates repair messages of the appropriate size, defined by the instance variable `packetSize_`. However, the agent assumes that any application data is placed in the data portion of the packet, pointed to by `packet->accessdata()`.

Request Packets On receiving a request, `recv_rqst`(sender, msgid) will check whether it needs to schedule requests for other missing data. If it has received this request before it was aware that the source had generated this data message (*i.e.*, the sequence number of the request is higher than the last known sequence number of data from this source), then the agent can infer that it is missing this, as well as data from the last known sequence number onwards; it schedules requests for all of the missing data and returns. On the other hand, if the sequence number of the request is less than the last known sequence number from the source, then the agent can be in one of three states: (1) it does not have this data, and has a request pending for it, (2) it has the data, and has seen an earlier request, upon which it has a repair pending for it, or (3) it has the data, and it should instantiate a repair. All of these error recovery mechanisms are done in OTcl; `recv_rqst`() invokes the instance procedure `recv-rqst`{sender, msgid, requester} for further processing.

Repair Packets On receiving a repair, `recv_repr`(sender, msgid) will check whether it needs to schedule requests for other missing data. If it has received this repair before it was aware that the source had generated this data message (*i.e.*, the sequence number of the repair is higher than the last known sequence number of data from this source), then the agent can infer that it is missing all data between the last known sequence number and that on the repair; it schedules requests for all of this data, marks this message as received, and returns. On the other hand, if the sequence number of the request is less than the last known sequence number from the source, then the agent can be in one of three states: (1) it does not have this data, and has a request pending for it, (2) it has the data, and has seen an earlier request, upon which it has a repair pending for it, or (3) it has the data, and probably scheduled a repair for it at some time; after error recovery, its hold down timer (equal to three times its distance to some requester) expired, at which time the pending object was cleared. In this last situation, the agent will simply ignore the repair, for lack of being able to do anything meaningful. All of these error recovery mechanisms are done in OTcl; `recv_repr`() invokes the instance procedure `recv-repr`{sender, msgid} to complete the loss recovery phase for the particular message.

Session Packets On receiving a session message, the agent updates its sequence numbers for all active sources, and computes its instantaneous distance to the sending agent if possible. The agent will ignore earlier session messages from a group member, if it has received a later one out of order.

Session message processing is done in `recv_sess`(). The format of the session message is: ⟨count of tuples in this message, list of tuples⟩, where each tuple indicates the ⟨sender id, last sequence number from the source, time the last session message was received from this sender, time that that message was sent⟩. The first tuple is the information about the local agent[3].

[2]Technically, `recv_data`() invokes the instance procedure `recv data ⟨sender⟩ ⟨msgid⟩`, that then invokes `recv-data{}`. The indirection allows individual simulation scripts to override the `recv{}` as needed.

[3]Note that this implementation of session message handling is subtly different from that used in *wb* or described in [6]. In principle, an agent disseminates a list of the data it has actually received. Our implementation, on the other hand, only disseminates a count of the last message sequence number per source that the agent knows that that the source has sent. This is a constraint when studying aspects of loss recovery during partition and healing. It is reasonable to expect that the maintainer of this code will fix this problem during one of his numerous intervals of copious spare time.

20.4 Loss Detection—The Class SRMinfo

A very small encapsulating class, entirely in C++, tracks a number of assorted state information. Each member of the group, n_i, uses one SRMinfo block for every other member of the group.

An SRMinfo object about group member n_j at n_i, contains information about the session messages received by n_i from n_j. n_i can use this information to compute its distance to n_j.

If n_j sends is active in sending data traffic, then the SRMinfo object will also contain information about the received data, including a bit vector indicating all packets received from n_j.

The agent keeps a list of SRMinfo objects, one per group member, in its member variable, `sip_`. Its method, `get_state`(int sender) will return the object corresponding to that sender, possibly creating that object, if it did not already exist. The `class SRMinfo` has two methods to access and set the bit vector, *i.e.*,

`ifReceived`(int id)	indicates whether the particular message from the appropriate sender, with id `id` was received at n_i,
`setReceived`(int id)	to set the bit to indicate that the particular message from the appropriate sender, with id `id` was received at n_i.

The session message variables to access timing information are public; no encapsulating methods are provided. These are:

```
int     lsess_;                  /* # of last session msg received */
int     sendTime_;               /* Time sess. msg. # sent */
int     recvTime_;               /* Time sess. msg. # received */
double  distance_;

/*  Data messages  */
int     ldata_;                  /* # of last data msg sent */
```

20.5 Loss Recovery Objects

In the last section, we described the agent behavior when it receives a message. Timers are used to control when any particular control message is to be sent. The SRM agent uses a separate `class SRM` to do the timer based processing. In this section, we describe the basics if the class SRM, and the loss recovery objects. The following section will describe how the class SRM is used for sending periodic session messages. An SRM agent will instantiate one object to recover from one lost data packet. Agents that detect the loss will instantiate an object in the `class SRM/request`; agents that receive a request and have the required data will instantiate an object in the `class SRM/repair`.

Request Mechanisms SRM agents detect loss when they receive a message, and infer the loss based on the sequence number on the message received. Since packet reception is handled entirely by the compiled object, loss detection occurs in the C++ methods. Loss recovery, however, is handled entirely by instance procedures of the corresponding interpreted object in OTcl.

When any of the methods detects new losses, it invokes `Agent/SRM::request{}` with a list of the message sequence numbers that are missing. `request{}` will create a new `requestFunction_` object for each message that is missing. The agent stores the object handle in its array of `pending_` objects. The key to the array is the message identifier ⟨sender⟩:⟨msgid⟩.

The default requestFunction_ is class SRM/request The constructor for the class SRM/request calls the base class constructor to initialize the simulator instance (ns_), the SRM agent (agent_), trace file (trace_), and the times_ array. It then initializes its statistics_ array with the pertinent elements.

A separate call to set-params{} sets the sender_, msgid_, round_ instance variables for the request object. The object determines C1_ and C2_ by querying its agent_. It sets its distance to the sender (times_(distance)) and fixes other scheduling parameters: the backoff constant (backoff_), the current number of backoffs (backoffCtr_), and the limit (backoffLimit_) fixed by the agent. set-params{} writes the trace entry "Q DETECT".

The final step in request{} is to schedule the timer to send the actual request at the appropriate moment. The instance procedure SRM/request::schedule{} uses compute-delay{} and its current backoff constant to determine the delay. The object schedules send-request{} to be executed after delay_ seconds. The instance variable eventID_ stores a handle to the scheduled event. The default compute-delay{} function returns a value uniformly distributed in the interval $[C_1 d_s, (C_1 + C_2) d_s]$, where d_s is twice $times_(distance). The schedule{} schedules an event to send a request after the computed delay. The routine writes a trace entry "Q NTIMER at ⟨time⟩".

When the scheduled timer fires, the routine send-request{} sends the appropriate message. It invokes "$agent_ send request ⟨args⟩" to send the request. Note that send{} is an instproc-like, executed by the command() method of the compiled object. However, it is possible to overload the instproc-like with a specific instance procedure send{} for specific configurations. As an example, recall that the file tcl/mcast/srm-nam.tcl overloads the send{} command to set the flowid based on type of message that is sent. send-request{} updates the statistics, and writes the trace entry "Q SENDNACK".

When the agent receives a control message for a packet for which a pending object exists, the agent will hand the message off to the object for processing.

When a request for a particular packet is received, the request object can be in one of two states: it is ignoring requests, considering them to be duplicates, or it will cancel its send event and re-schedule another one, after having backed off its timer. If ignoring requests it will update its statistics, and write the trace entry "Q NACK dup". Otherwise, set a time based on its current estimate of the delay_, until which to ignore further requests. This interval is marked by the instance variable ignore_. If the object reschedules its timer, it will write the trace entry " Q NACK IGNORE-BACKOFF ⟨ignore⟩". Note that this re-scheduling relies on the fact that the agent has joined the multicast group, and will therefore receive a copy of every message it sends out.

When the request object receives a repair for the particular packet, it can be in one of two states: either it is still waiting for the repair, or it has already received an earlier repair. If it is the former, there will be an event pending to send a request, and eventID_ will point to that event. The object will compute its serviceTime, cancel that event, and set a hold-down period during which it will ignore other requests. At the end of the hold-down period, the object will ask its agent to clear it. It will write the trace entry "Q REPAIR IGNORES ⟨ignore⟩". On the other hand, if this is a duplicate repair, the object will update its statistics, and write the trace entry "Q REPAIR dup".

When the loss recovery phase is completed by the object, Agent/SRM::clear{} will remove the object from its array of pending_ objects, and place it in its list of done_ objects. Periodically, the agent will cleanup and delete the done_ objects.

Repair Mechanisms The agent will initiate a repair if it receives a request for a packet, and it does not have a request object pending_ for that packet. The default repair object belongs to the class SRM/repair. Barring minor differences, the sequence of events and the instance procedures in this class are identical to those for SRM/request. Rather than outline every single procedure, we only outline the differences from those described earlier for a request object.

The repair object uses the repair parameters, D1_, D2_. A repair object does not repeatedly reschedule is timers; therefore, it does not use any of the backoff variables such as that used by a request object. The repair object ignores all requests for the same packet. The repair objet does not use the `ignore_` variable that request objects use. The trace entries written by repair objects are marginally different; they are "P NACK from ⟨requester⟩", "P RTIMER at ⟨fireTime⟩", "P SENDREP", "P REPAIR IGNORES ⟨holddown⟩".

Apart from these differences, the calling sequence for events in a repair object is similar to that of a request object.

Mechanisms for Statistics The agent, in concert with the request and repair objects, collect statistics about their response to data loss [6]. Each call to the agent `request{}` procedure marks a new period. At the start of a new period, `mark-period{}` computes the moving average of the number of duplicates in the last period. Whenever the agent receives a first round request from another agent, and it had sent a request in that round, then it considers the request as a duplicate request, and increments the appropriate counters. A request object does not consider duplicate requests if it did not itself send a request in the first round. If the agent has a repair object pending, then it does not consider the arrival of duplicate requests for that packet. The object methods `SRM/request::dup-request?{}` and `SRM/repair::dup-request?{}` encode these policies, and return 0 or 1 as required.

A request object also computes the elapsed time between when the loss is detected to when it receives the first request. The agent computes a moving average of this elapsed time. The object computes the elapsed time (or delay) when it cancels its scheduled event for the first round. The object invokes Agent/SRM::update-ave to compute the moving average of the delay.

The agent keeps similar statistics of the duplicate repairs, and the repair delay.

The agent stores the number of rounds taken for one loss recovery, to ensure that subsequent loss recovery phases for that packet that are not definitely not due to data loss do not account for these statistics. The agent stores the number of routes taken for a phase in the array `old_`. When a new loss recovery object is instantiated, the object will use the agent's instance procedure `round?{}` to determine the number of rounds in a previous loss recovery phase for that packet.

20.6 Session Objects

Session objects, like the loss recovery objects (Section 20.5), are derived from the base `class SRM` Unlike the loss recovery objects though, the agent only creates one session object for the lifetime of the agent. The constructor invokes the base class constructor as before; it then sets its instance variable `sessionDelay_`. The agent creates the session object when it `start{}`s. At that time, it also invokes SRM/session::schedule, to send a session message after `sessionDelay_` seconds.

When the object sends a session message, it will schedule to send the next one after some interval. It will also update its statistics. `send-session{}` writes out the trace entry "S SESSION".

The class overrides the `evTrace{}` routine that writes out the trace entries. SRM/session::evTrace disable writing out the trace entry for session messages.

Two types of session message scheduling strategies are currently available: The function in the base class schedules sending session messages at fixed intervals of `sessionDelay_` jittered around a small value to avoid synchronization among all the agents at all the nodes. `class SRM/session/logScaled`chedules sending messages at intervals of `sessionDelay` times $\log_2$(`groupSize_`) so that the frequency of session messages is inversely proportional to the size of the group.

The base class that sends messages at fixed intervals is the default `sessionFunction_` for the agent.

20.7 Extending the Base Class Agent

In the earlier section on configuration parameters (Section 20.1.2), we had shown how to trivially extend the agent to get deterministic and probabilistic protocol behavior. In this section, we describe how to derive more complex extensions to the protocol for fixed and adaptive timer mechanisms.

20.7.1 Fixed Timers

The fixed timer mechanism are done in the derived `class Agent/SRM/Fixed` The main difference with fixed timers is that the repair parameters are set to log(`groupSize_`). Therefore, the repair procedure of a fixed timer agent will set D_1 and D_2 to be proportional to the group size before scheduling the repair object.

20.7.2 Adaptive Timers

Agents using adaptive timer mechanisms modify their request and repair parameters under three conditions (1) every time a new loss object is created; (2) when sending a message; and (3) when they receive a duplicate, if their relative distance to the loss is less than that of the agent that sends the duplicate. All three changes require extensions to the agent and the loss objects. The `class Agent/SRM/Adaptive` uses `class SRM/request/Adaptive` and `class SRM/repair/Adaptive` as the request and repair functions respectively. In addition, the last item requires extending the packet headers, to advertise their distances to the loss. The corresponding compiled class for the agent is the `class ASRMAgent`.

Recompute for Each New Loss Object Each time a new request object is created, SRM/request/Adaptive::set-params invokes `$agent_ recompute-request-params`. The agent method `recompute-request-params`(). uses the statistics about duplicates and delay to modify C_1 and C_2 for the current and future requests.

Similarly, SRM/request/Adaptive::set-params for a new repair object invokes `$agent_ recompute-repair-params`. The agent method `recompute-repair-params`(). uses the statistics objects to modify D_1 and D_2 for the current and future repairs.

Sending a Message If a loss object sends a request in its first `round_`, then the agent, in the instance procedure `sending-request{}`, will lower C_1, and set its instance variable `closest_(requestor)` to 1.

Similarly, a loss object that sends a repair in its first `round_` will invoke the agent's instance procedure, `sending-repair{}`, to lower D_1 and set `closest_(repairor)` to 1.

Advertising the Distance Each agent must add additional information to each request/repair that it sends out. The base `class SRMAgent` invokes the virtual method `addExtendedHeaders()` for each SRM packet that it sends out. The method is invoked after adding the SRM packet headers, and before the packet is transmitted. The adaptive SRM agent overloads `addExtendedHeaders()` to specify its distances in the additional headers. When sending a request, that agent unequivocally knows the identity of the sender. As an example, the definition of `addExtendedHeaders()` for the adaptive SRM agent is:

```
void addExtendedHeaders(Packet* p) {
        SRMinfo* sp;
        hdr_srm*  sh = (hdr_srm*) p->access(off_srm_);
```

```
        hdr_asrm* seh = (hdr_asrm*) p->access(off_asrm_);
        switch (sh->type()) {
        case SRM_RQST:
                sp = get_state(sh->sender());
                seh->distance() = sp->distance_;
                break;
        ...
        }
}
```

Similarly, the method `parseExtendedHeaders()` is invoked every time an SRM packet is received. It sets the agent member variable `pdistance_` to the distance advertised by the peer that sent the message. The member variable is bound to an instance variable of the same name, so that the peer distance can be accessed by the appropriate instance procedures. The corresponding `parseExtendedHeaders()` method for the Adaptive SRM agent is simply:

```
void parseExtendedHeaders(Packet* p) {
        hdr_asrm* seh = (hdr_asrm*) p->access(off_asrm_);
        pdistance_ = seh->distance();
}
```

Finally, the adaptive SRM agent's extended headers are defined as `struct hdr_asrm`. The header declaration is identical to declaring other packet headers in *ns*. Unlike most other packet headers, these are not automatically available in the packet. The interpreted constructor for the first adaptive agent will add the header to the packet format. For example, the start of the constructor for the `Agent/SRM/Adaptive` agent is:

```
Agent/SRM/Adaptive set done_ 0
Agent/SRM/Adaptive instproc init args {
    if ![$class set done_] {
        set pm [[Simulator instance] set packetManager_]
        TclObject set off_asrm_ [$pm allochdr aSRM]
        $class set done_ 1
    }

    eval $self next $args
    ...
}
```

Part VI

Application

Chapter 21

Applications and transport agent API

Applications sit on top of transport agents in *ns*. There are two basic types of applications: traffic generators and simulated applications. Figure 21.1 illustrates two examples of how applications are composed and attached to transport agents. Transport agents are described in Part V (Transport).

This chapter first describes the base `class Application`. Next, the transport API, through which applications request services from underlying transport agents, is described. Finally, the current implementations of traffic generators and sources are explained.

21.1 The class Application

Application is a C++ class defined as follows:

```
class Application : public TclObject {
public:
        Application();
        virtual void send(int nbytes);
        virtual void recv(int nbytes);
        virtual void resume();
protected:
        int command(int argc, const char*const* argv);
        virtual void start();
        virtual void stop();
        Agent *agent_;
        int enableRecv_;                // call OTcl recv or not
        int enableResume_;              // call OTcl resume or not
};
```

Although objects of `class Application` are not meant to be instantiated, we do not make it an abstract base class so that it is visible from OTcl level. The class provides basic prototypes for application behavior (`send(), recv(), resume(), start(), stop()`), a pointer to the transport agent to which it is connected, and flags that indicate whether a OTcl-level upcall should be made for `recv()` and `resume()` events.

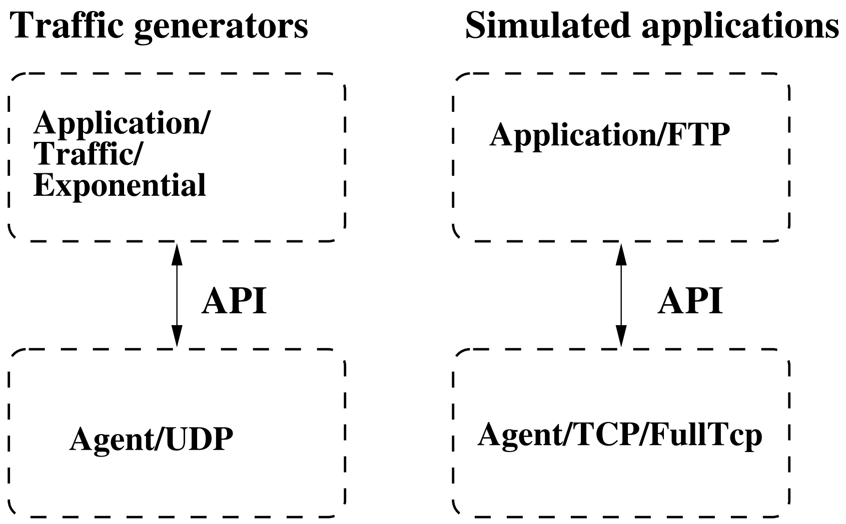


Figure 21.1: Example of Application Composition

21.2 The transport agent API

In real-world systems, applications typically access network services through an applications programming interface (API). The most popular of these APIs is known as "sockets." In *ns*, we mimic the behavior of the sockets API through a set of well-defined API functions. These functions are then mapped to the appropriate internal agent functions (e.g., a call to `send(numBytes)` causes TCP to increment its "send buffer" by a corresponding number of bytes).

This section describes how agents and applications are hooked together and communicate with one another via the API.

21.2.1 Attaching transport agents to nodes

This step is typically done at OTcl level. Agent management was also briefly discussed in Section 4.2.

```
set src [new Agent/TCP/FullTcp]
set sink [new Agent/TCP/FullTcp]
$ns_ attach-agent $node_(s1) $src
$ns_ attach-agent $node_(k1) $sink
$ns_ connect $src $sink
```

The above code illustrates that in *ns*, agents are first attached to a node via `attach-agent`. Next, the `connect` instproc sets each agent's destination target to the other. Note that, in *ns*, `connect()` has different semantics than in regular sockets. In *ns*, `connect()` simply establishes the destination address for an agent, but does not set up the connection. As a result, the overlying application does not need to know its peer's address. For TCPs that exchange SYN segments, the first call to `send()` will trigger the SYN exchange.

To detach an agent from a node, the instproc `detach-agent` can be used; this resets the target for the agent to a null agent.

21.2.2 Attaching applications to agents

After applications are instantiated, they must be connected to a transport agent. The attach-agent method can be used to attach an application to an agent, as follows:

```
set ftp1 [new Application/FTP]
$ftp1 attach-agent $src
```

The following shortcut accomplishes the same result:

```
set ftp1 [$src attach-app FTP]
```

The attach-agent method, which is also used by attach-app, is implemented in C++. It sets the agent_ pointer in class Application to point to the transport agent, and then it calls attachApp() in agent.cc to set the app_ pointer to point back to the application. By maintaining this binding only in C++, OTcl-level instvars pointers are avoided and consistency between OTcl and C++ is guaranteed. The OTcl-level command [$ftp1 agent] can be used by applications to obtain the handler for the transport agent.

21.2.3 Using transport agents via system calls

Once transport agents have been configured and applications attached, applications can use transport services via the following system calls. These calls can be invoked at either OTcl or C++ level, thereby allowing applications to be coded in either C++ or OTcl. These functions have been implemented as virtual functions in the base class Agent, and can be redefined as needed by derived Agents.

- send(int nbytes)—Send nbytes of data to peer. For TCP agents, if nbytes == -1, this corresponds to an "infinite" send; i.e., the TCP agent will act as if its send buffer is continually replenished by the application.
- sendmsg(int nbytes, const char* flags = 0)—Identical to send(int nbytes), except that it passes an additional string flags. Currently one flag value, "MSG_EOF," is defined; MSG_EOF specifies that this is the last batch of data that the application will submit, and serves as an implied close (so that TCP can send FIN with data).
- close()—Requests the agent to close the connection (only applicable for TCP).
- listen()—Requests the agent to listen for new connections (only applicable for Full TCP).
- set_pkttype(int pkttype)—This function sets the type_ variable in the agent to pkttype. Packet types are defined in packet.h. This function is used to override the transport layer packet type for tracing purposes.

Note that certain calls are not applicable for certain agents; e.g., a call to close() a UDP connection results in a no-op. Additional calls can be implemented in specialized agents, provided that they are made public member functions.

21.2.4 Agent upcalls to applications

Since presently in *ns* there is no actual data being passed between applications, agents can instead announce to applications the occurence of certain events at the transport layer through "upcalls." For example, applications can be notified of the arrival of a number of bytes of data; this information may aid the application in modelling real-world application behavior more closely. Two basic "upcalls" have been implemented in base class Application and int the transport agents:

- `recv(int nbytes)`—Announces that `nbytes` of data have been received by the agent. For UDP agents, this signifies the arrival of a single packet. For TCP agents, this signifies the "delivery" of an amount of in-sequence data, which may be larger than that contained in a single packet (due to the possibility of network reordering).
- `resume()`—This indicates to the application that the transport agent has sent out all of the data submitted to it up to that point in time. For TCP, it does not indicate whether the data has been ACKed yet, only that it has been sent out for the first time.

The default behavior is as follows: Depending on whether the application has been implemented in C++ or OTcl, these C++ functions call a similarly named (`recv, resume`) function in the application, if such methods have been defined.

Although strictly not a callback to applications, certain Agents have implemented a callback from C++ to OTcl-level that has been used by applications such as HTTP simulators. This callback method, `done{}`, is used in TCP agents. In TCP, `done{}` is called when a TCP sender has received ACKs for all of its data and is now closed; it therefore can be used to simulate a blocked TCP connection. The `done{}` method was primarily used before this API was completed, but may still be useful for applications that do not want to use `resume()`.

To use `done{}` for FullTcp, for example, you can try:

```
set myagent [new Agent/TCP/FullTcp]
$myagent proc done
    ... code you want ...
```

If you want all the FullTCP's to have the same code you could also do:

```
Agent/TCP/FullTcp instproc done
    ... code you want ...
```

By default, `done{}` does nothing.

21.2.5 An example

Here is an example of how the API is used to implement a simple application (FTP) on top of a FullTCP connection.

```
set src [new Agent/TCP/FullTcp]
set sink [new Agent/TCP/FullTcp]
$ns_ attach-agent $node_(s1) $src
$ns_ attach-agent $node_(k1) $sink
$ns_ connect $src $sink

# set up TCP-level connections
$sink listen;
$src set window_ 100

set ftp1 [new Application/FTP]
$ftp1 attach-agent $src

$ns_ at 0.0 "$ftp1 start"
```

In the configuration script, the first five lines of code allocates two new FullTcp agents, attaches them to the correct nodes, and "connects" them together (assigns the correct destination addresses to each agent). The next two lines configure the TCP agents further, placing one of them in LISTEN mode. Next, `ftp1` is defined as a new FTP Application, and the `attach-agent` method is called in C++ (`app.cc`).

The ftp1 application is started at time 0:

```
Application/FTP instproc start {} {
       [$self agent] send -1;   # Send indefinitely
}
```

Alternatively, the FTP application could have been implemented in C++ as follows:

```
void FTP::start()
{
        agent_->send(-1);    // Send indefinitely
}
```

Since the FTP application does not make use of callbacks, these functions are null in C++ and no OTcl callbacks are made.

21.3 The class TrafficGenerator

TrafficGenerator is an abstract C++ class defined as follows:

```
class TrafficGenerator : public Application {
public:
        TrafficGenerator();
        virtual double next_interval(int &) = 0;
        virtual void init() {}
        virtual double interval() { return 0; }
        virtual int on() { return 0; }
        virtual void timeout();
        virtual void recv() {}
        virtual void resume() {}
protected:
        virtual void start();
        virtual void stop();
        double nextPkttime_;
        int size_;
        int running_;
        TrafficTimer timer_;
};
```

The pure virtual function `next_interval()` returns the time until the next packet is created and also sets the size in bytes of the next packet. The function `start()` calls `init`(void) and starts the timer. The function `timeout()` sends a packet and reschedules the next timeout. The function `stop()` cancels any pending transmissions. Callbacks are typically not used for traffic generators, so these functions (`recv, resume`) are null.

Currently, there are four C++ classes derived from the class TrafficGenerator:

1. `EXPOO_Traffic`—generates traffic according to an Exponential On/Off distribution. Packets are sent at a fixed rate during on periods, and no packets are sent during off periods. Both on and off periods are taken from an exponential distribution. Packets are constant size.
2. `POO_Traffic`—generates traffic according to a Pareto On/Off distribution. This is identical to the Exponential On/Off distribution, except the on and off periods are taken from a pareto distribution. These sources can be used to generate aggregate traffic that exhibits long range dependency.
3. `CBR_Traffic`—generates traffic according to a deterministic rate. Packets are constant size. Optionally, some randomizing dither can be enabled on the interpacket departure intervals.
4. `TrafficTrace`—generates traffic according to a trace file. Each record in the trace file consists of 2 32-bit fields. The first contains the time in microseconds until the next packet is generated. The second contains the length in bytes of the next packet.

These classes can be created from OTcl. The OTcl classes names and associated parameters are given below:

Exponential On/Off An Exponential On/Off object is embodied in the OTcl class Application/Traffic/Exponential. The member variables that parameterize this object are:

`packet_size_`	the constant size of the packets generated
`burst_time_`	the average “on” time for the generator
`idle_time_`	the average “off” time for the generator
`rate_`	the sending rate during “on” times

Hence a new Exponential On/Off traffic generator can be created and parameterized as follows:

```
set e [new Application/Traffic/Exponential]
$e set packet_size_ 210
$e set burst_time_ 500ms
$e set idle_time_ 500ms
$e set rate_ 100k
```

Pareto On/Off A Pareto On/Off object is embodied in the OTcl class Application/Traffic/Pareto. The member variables that parameterize this object are:

`packet_size_`	the constant size of the packets generated
`burst_time_`	the average "on" time for the generator
`idle_time_`	the average "off" time for the generator
`rate_`	the sending rate during "on" times
`shape_`	the "shape" parameter used by the pareto distribution

A new Pareto On/Off traffic generator can be created as follows:

```
set p [new Application/Traffic/Pareto]
$p set packet_size_ 210
$p set burst_time_ 500ms
$p set idle_time_ 500ms
$p set rate_ 200k
$p set shape_ 1.5
```

CBR A CBR object is embodied in the OTcl class Application/Traffic/CBR. The member variables that parameterize this object are:

`rate_`	the sending rate
`packet_size_`	the constant size of the packets generated
`random_`	flag indicating whether or not to introduce random "noise" in the scheduled departure times (default is off)
`maxpkts_`	the maximum number of packets to send (default is (2^28))

Hence a new CBR traffic generator can be created and parameterized as follows:

```
set e [new Application/Traffic/CBR]
$e set packet_size_ 48
$e set rate_ 64Kb
$e set random_ 1
```

Traffic Trace A Traffic Trace object is instantiated by the OTcl class Application/Traffic/Trace. The associated class Tracefile is used to enable multiple Traffic/Trace objects to be associated with a single trace file. The Traffic/Trace class uses the method attach-tracefile to associate a Traffic/Trace object with a particular Tracefile object. The method filename of the Tracefile class associates a trace file with the Tracefile object. The following example shows how to create two Application/Traffic/Trace objects, each associated with the same trace file (called "example-trace" in this example). To avoid synchronization of the traffic generated, random starting places within the trace file are chosen for each Traffic/Trace object.

```
set tfile [new Tracefile]
$tfile filename example-trace

set t1 [new Application/Traffic/Trace]
$t1 attach-tracefile $tfile
set t2 [new Application/Traffic/Trace]
$t2 attach-tracefile $tfile
```

21.3.1 An example

The following code illustrates the basic steps to configure an Exponential traffic source over a UDP agent, for traffic flowing from node `s1` to node `k1`:

```
set src [new Agent/UDP]
set sink [new Agent/UDP]
$ns_ attach-agent $node_(s1) $src
$ns_ attach-agent $node_(k1) $sink
$ns_ connect $src $sink

set e [new Application/Traffic/Exponential]
$e attach-agent $src
$e set packet_size_ 210
$e set burst_time_ 500ms
$e set idle_time_ 500ms
$e set rate_ 100k
```

```
$ns_ at 0.0 "$e start"
```

21.4 Simulated applications

There are currently two "simulate application" classes derived from Application: Application/FTP and Application/Telnet. These classes work by advancing the count of packets available to be sent by a TCP transport agent. The actual transmission of available packets is still controlled by TCP's flow and congestion control algorithm.

Application/FTP Application/FTP, implemented in OTcl, simulates bulk data transfer. The following are methods of the Application/FTP class:

`attach-agent`	attaches an Application/FTP object to an agent.
`start`	start the Application/FTP by calling the TCP agent's `send(-1)` function, which causes TCP to behave as if the application were continuously sending new data.
`stop`	stop sending.
`produce n`	set the counter of packets to be sent to n.
`producemore n`	increase the counter of packets to be sent by n.
`send n`	similar to `producemore`, but sends n bytes instead of packets.

Application/Telnet Application/Telnet objects generate packets in one of two ways. If the member variable `interval_` is non-zero, then inter-packet times are chosen from an exponential distribution with average equal to `interval_`. If `interval_` is zero, then inter-arrival times are chosen according to the tcplib distribution (see tcplib-telnet.cc). The start method starts the packet generation process.

Chapter 22

Web cache as an application

All applications described above are "virtual" applications, in the sense that they do not actually transfer their own data in the simulator; all that matter is the *size* and the *time* when data are transferred. Sometimes we may want applications to transfer their own data in simulations. One such example is web caching. Here we want HTTP servers to send HTTP headers of every page to caches and clients. These headers contain page modification time information and other caching directives, which are important for some cache consistency algorithms.

The following describes the APIs to transport application data over TCP, as well as the internals of the web caching applications.

22.1 Transmitting application data over TCP

As indicated in section 21.2.4, transmission of application data can be implemented via agent upcalls. Assuming we are using TCP agents, all data are delivered in sequence, which means we can view the TCP connection as a FIFO pipe. Because ns doesn't support actual data passing, we have to emulate that behavior. First, we need to provide buffer for application data at the sender. Then we count the bytes received at the receiver. When the receiver has got all bytes of the current data transmission, it then gets the data directly from the sender. Class Application/TcpApp is used to implement this functionality.

A TcpApp object contains a pointer to a transport agent, presumably either a FullTcp or a SimpleTcp. [1] (Currently TcpApp doesn't support asymmetric TCP agents, i.e., sender is separated from receiver). It provides the following OTcl interfaces:

- `connect`: Connecting another TcpApp to this one. This connection is bi-directional, i.e., only one call to `connect` is needed, and data can be sent in either direction.
- `send`: It takes two arguments: `(nbytes, str)`. `nbytes` is the "nominal" size of application data. `str` is application data in string form.

In order to send application data in binary form, TcpApp provides a virtual C++ method `send(int nbytes, int dsize, const char *data)`. In fact, this is the method used to implement the OTcl method `send`. Because it's difficult to deal with binary data in Tcl, no OTcl interface is provided to handle binary data. `nbytes` is the number of bytes to be transmitted, `dsize` is the actual size of the array `data`.

[1] A SimpleTcp agent is used solely for web caching simulations. It is actually an UDP agent. It has neither error recovery nor flow/congestion control. It doesn't do packet segmentation. Assuming a loss-free network and in-order packet delivery, SimpleTcp agent simplifies the trace files and hence aids the debugging of application protocols, which, in our case, is the web cache consistency protocol.

TcpApp provides a C++ virtual method `process_data(int size, char*data)` to handle the received data. The default handling is to treat the data as a tcl script and evaluate the script. But it's easy to derive a class to provide other types of handling.

Here is an example of using Application/TcpApp. A similar example is `Test/TcpApp-2node` in *ns*/tcl/test/test-suite-webcache.tcl. First, we create FullTcp agents and connect them:

```
set tcp1 [new Agent/TCP/FullTcp]
set tcp2 [new Agent/TCP/FullTcp]
#   Set TCP parameters here, e.g., window_, iss_, ...

$ns attach-agent $n1 $tcp1
$ns attach-agent $n2 $tcp2
$ns connect $tcp1 $tcp2
$tcp2 listen
```

Then we Create TcpApps and connect them:

```
set app1 [new Application/TcpApp $tcp1]
set app2 [new Application/TcpApp $tcp2]
$app1 connect $app2
```

Now we let `$app1` be sender and `$app2` be receiver:

```
$ns at 1.0 "$app1 send 100 \"$app2 app-recv 100\""
```

Where `app-recv` is defined as:

```
Application/TcpApp instproc app-recv  size
        global ns
        puts "[$ns now] app2 receives data $size from app1"
```

22.2 Overview of web cache classes

There are three major classes related to web cache, as it is in the real world: client (browser), server, and cache. Because they share a common feature, i.e., the HTTP protocol, they are derived from the same base class `Http` (Name of OTcl class, it's called `HttpApp` in C++). For the following reasons, it's not a real Application. First, an HTTP object (i.e., client/cache/server) may want to maintain multiple concurrent HTTP connections, but an Application contains only one `agent_`. Also, an HTTP object needs to transmit real data (e.g., HTTP header) and that's provided by TcpApp instead of any Agent. Therefore, we choose to use a standalone class derived from TclObject for common features of all HTTP objects, which are managing HTTP connections and a set of pages. In the rest of the section, we'll discuss these functionalities of Http. In the next three sections, we'll in turn describe HTTP client, cache and server.

22.2.1 Managing HTTP connections

Every HTTP connection is embodied as a TcpApp object. Http maintains a hash of TcpApp objects, which are all of its active connections. It assumes that to any other Http, it has only one HTTP connection. It also allows dynamic establishment and teardown of connections. Only OTcl interface is provided for establishing, tearing down a connection and sending data through a connection.

OTcl methods Following is a list of OTcl interfaces related to connection management in Http objects:

id	return the id of the Http object, which is the id of the node the object is attached to.
get-cnc ⟨client⟩	return the TCP agent associated with $client (Http object).
is-connected ⟨server⟩	return 0 if not connected to $server, 1 otherwise.
send ⟨client⟩ ⟨bytes⟩ ⟨callback⟩	send $bytes of data to $client. When it's done, execute $callback (a OTcl command).
connect ⟨client⟩ ⟨TCP⟩	associate a TCP agent with $client (Http object). That agent will be used to send packets *to* $client.
disconnect ⟨client⟩	delete the association of a TCP agent with $client. Note that neither the TCP agent nor $client is not deleted, only the association is deleted.

Configuration parameter By default, Http objects use Agent/SimpleTcp as transport agents (section 22.1). They can also use Agent/FullTcp agents, which allows Http objects to operate in a lossy network. Class variable codeTRANSPORT_ is used for this purpose. E.g., `Http set TRANSPORT_FullTcp` tells all Http objects use FullTcp agents.

This configuration should be done *before* simulation starts, and it should not change during simulation, because FullTcp agents do not inter-operate with SimpleTcp agents.

22.2.2 Managing web pages

Http also provides OTcl interfaces to manage a set of pages. The real management of pages are handled by class `PagePool` and its subclasses. Because different HTTP objects have different requirements for page management, we allow different PagePool subclasses to be attached to different subclasses of Http class. Meanwhile, we export a common set of PagePool interfaces to OTcl through Http. For example, a browser may use a PagePool only to generate a request stream, so its PagePool only needs to contain a list of URLs. But a cache may want to store page size, last modification time of every page instead of a list of URLs. However, this separation is not clearcut in the current implementation.

Page URLs are represented in the form of: ⟨`ServerName`⟩`:`⟨`SequenceNumber`⟩ where the `ServerName` is the name of OTcl object, and every page in every server should have a unique `SequenceNumber`. Page contents are ignored. Instead, every page contains several *attributes*, which are represented in OTcl as a list of the following (⟨name⟩ ⟨value⟩) pairs: "modtime ⟨val⟩" (page modification time), "size ⟨val⟩" (page size), and "age ⟨val⟩"} The ordering of these pairs is not significant.

Following is a list of related OTcl methods.

set-pagepool ⟨pagepool⟩	set page pool
enter-page ⟨pageid⟩ ⟨attributes⟩	add a page with id $pageid into pool. $attributes is the attributes of $pageid, as described above.
get-page ⟨pageid⟩	return page attributes in the format described above.
get-modtime ⟨pageid⟩	return the last modification time of the page $pageid.
exist-page ⟨pageid⟩	return 0 if $pageid doesn't exist in this Http object, 1 otherwise.
get-size ⟨pageid⟩	return the size of $pageid.
get-cachetime ⟨pageid⟩	return the time when page $pageid is entered into the cache.

22.2.3 Debugging

HttpApp also provides two debugging methods. `log` registers a file handle as the trace file for all HttpApp-specific traces. Its trace format is described in section 22.7. `evTrace` logs a particular event into trace file. It concatenates time and the id of the HttpApp to the given string, and writes it out. Details can be found in *ns*/webcache/http.cc.

22.3 Web client

Class Http/Client models behavior of a simple web browser. It generates a sequence of page requests, where request interval and page IDs are randomized. It's a pure OTcl class inherited from Http. Next we'll walk through its functionalities and usage.

Creating a client First of all, we create a client and connect it to a cache and a web server. Currently a client is only allowed to connect to a single cache, but it's allowed to connect to multiple servers. Note that this has to be called *AFTER* the simulation starts (i.e., after `$ns run` is called). This remains true for all of the following methods and code examples of Http and its derived classes, unless explicitly said.

```
# Assuming $server is a configured Http/Server.
set client [new Http/Client $ns $node]          ;# client resides on this node
$client connect $server                         ;# connecting client to server
```

Configuring request generation For every request, Http/Client uses PagePool to generate a random page ID, and use a random variable to generate intervals between two consecutive requests: [2]

```
$client set-page-generator $pgp                 ;# attach a configured PagePool
$client set-interval-generator $ranvar          ;# attach a random variable
```

Here we assume that PagePools of Http/Client share the same set of pages as PagePools of the server. Usually we simplify our simulation by letting all clients and servers share the same PagePool, i.e., they have the same set of pages. When there are multiple servers, or servers' PagePools are separated from those of clients', care must be taken to make sure that every client sees the same set of pages as the servers to which they are attached.

[2] Some PagePool, e.g., PagePool/Math, has only one page and therefore it always returns the same page. Some other PagePool, e.g. PagePool/Trace, has multiple pages and needs a random variable to pick out a random page.

Starting After the above setup, starting requests is very simple:

```
$client start-session $cache $server          ;# assuming $cache is a configured Http/Cache
```

OTcl interfaces Following is a list of its OTcl methods (in addition to those inherited from Http). This is not a complete list. More details can be found in *ns*/tcl/webcache/http-agent.tcl.

send-request ⟨server⟩ ⟨type⟩ ⟨pageid⟩ ⟨args⟩	send a request of page $pageid and type $type to $server. The only request type allowed for a client is GET. $args has a format identical to that of $attributes described in `Http::enter-page`.
start-session ⟨cache⟩ ⟨server⟩	start sending requests of a random page to $server via $cache.
start ⟨cache⟩ ⟨server⟩	before sending requests, populate $cache with all pages in the client's PagePool. This method is useful when assuming infinite-sized caches and we want to observe behaviors of cache consistency algorithms in steady state.
set-page-generator ⟨pagepool⟩	attach a PagePool to generate random page IDs.
set-interval-generator ⟨ranvar⟩	attach a random variable to generate random request intervals.

22.4 Web server

Class Http/Server models behavior of a HTTP server. Its configuration is very simple. All that a user needs to do is to create a server, attach a PagePool and wait:

```
set server [new Http/Server $ns $node]          ;# attach $server to $node
$server set-page-generator $pgp                 ;# attach a page pool
```

An Http/Server object waits for incoming requests after simulation starts. Usually clients and caches initiates connection to an Http/Server. But it still has its own `connect` method, which allows an Http/Server object to actively connect to a certain cache (or client). Sometimes this is useful, as explained in Test/TLC1::set-groups{} in *ns*/tcl/test/test-suite-webcache.tcl.

An Http/Server object accepts two types of requests: GET and IMS. GET request models normal client requests. For every GET request, it returns the attributes of the requested page. IMS request models If-Modified-Since used by TTL algorithms for cache consistency. For every IMS (If-Modified-Since) request, it compares the page modification time given in the request and that of the page in its PagePool. If the time indicated in the request is older, it sends back a response with very small size, otherwise it returns all of the page attributes with response size equal the real page size.

22.5 Web cache

Currently 6 types of web caches are implemented, including the base class Http/Cache. Its five derived subclasses implement 5 types of cache consistency algorithms: Plain old TTL, adaptive TTL, Omniscient TTL, Hierarchical multicast invalidation, and hierarchical multicast invalidation plus direct request.

In the following we'll only describe the base class Http/Cache, because all the subclasses involves discussion of cache consistency algorithms and it does not seem to be appropriate here.

22.5.1 Http/Cache

Class Http/Cache models behavior of a simple HTTP cache with infinite size. It doesn't contain removal algorithm, nor consistency algorithm. It is not intended to be used by itself. Rather, it is meant to be a base class for experimenting with various cache consistency algorithms and other cache algorithms.

Creation and startup Creating an Http/Cache requires the same set of parameters as Http/Client and Http/Server. After creation, a cache needs to connect to a certain server. Note that this creation can also be done dynamically, when a request comes in and the cache finds that it's not connected to the server. However, we do not model this behavior in current code. Following code is an example:

```
set cache [new HttpCache $ns $node]          ;# attach cache to $node
$cache connect $server                        ;# connect to $server
```

Like Http/Server, an Http/Cache object waits for requests (and packets from server) after it's initialized as above. When hierarchical caching is used, the following can be used to create the hierarchy:

```
$cache set-parent $parent                     ;# set parent cache
```

Currently all TTL and multicast invalidation caches support hierarchical caching. However, only the two multicast invalidation caches allows multiple cache hierarchies to inter-operate.

OTcl methods Although Http/Cache is a SplitObject, all of its methods are in OTcl. Most of them are used to process an incoming request. Their relations can be illustrated with the flowchart below, followed by explainations:

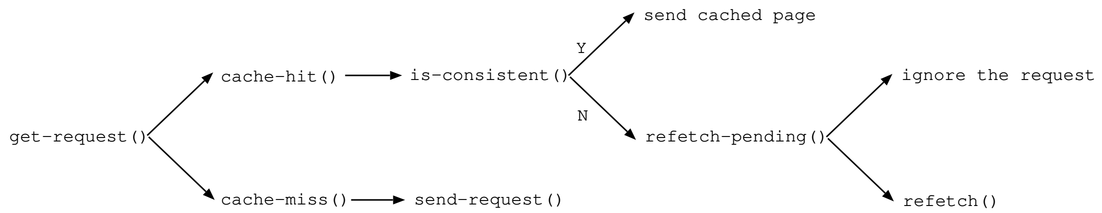


Figure 22.1: Handling of incoming request in Http/Cache

get-request ⟨client⟩ ⟨type⟩ ⟨pageid⟩	The entry point of processing any request. It checks if the requested page $pageid exists in the cache's page pool, then call either `cache-hit` or `cache-miss`.
cache-miss ⟨client⟩ ⟨type⟩ ⟨pageid⟩	This cache doesn't have the page. Send a request to server (or parent cache) to refetch the page if it hasn't already done so. Register $client in a list so that when the cache gets the page, it'll forward the page to all clients who have requested the page.
cache-hit ⟨client⟩ ⟨type⟩ ⟨pageid⟩	Checks the validatity of the cached page. If it's valid, send $client the cached page, otherwise refetch the page.
is-consistent ⟨client⟩ ⟨type⟩ ⟨pageid⟩	Returns 1 if $pageid is valid. This is intended to be overridden by subclasses.
refetch ⟨client⟩ ⟨type⟩ ⟨pageid⟩	Refetch an invalid page from server. This is intended to be overridden by subclasses.

22.6 Putting together: a simple example

We have seen all the pieces, now we present a script which provides a complete view of all pieces together. First, we build topology and other usual initializations:

```
set ns [new Simulator]

# Create topology/routing
set node(c) [$ns node]
set node(e) [$ns node]
set node(s) [$ns node]
$ns duplex-link $node(s) $node(e) 1.5Mb 50ms DropTail
$ns duplex-link $node(e) $node(c) 10Mb 2ms DropTail
$ns rtproto Session
```

Next we create the Http objects:

```
# HTTP logs
set log [open "http.log" w]

# Create page pool as a central page generator. Use PagePool/Math
set pgp [new PagePool/Math]
set tmp [new RandomVariable/Constant]                ;## Page size generator
$tmp set val_ 1024                                   ;## average page size
$pgp ranvar-size $tmp
set tmp [new RandomVariable/Exponential]             ;## Page age generator
$tmp set avg_ 5                                      ;## average page age
$pgp ranvar-age $tmp

set server [new Http/Server $ns $node(s)]    ;## Create a server and link it to the central page pool
$server set-page-generator $pgp
$server log $log

set cache [new Http/Cache $ns $node(e)]              ;## Create a cache
$cache log $log

set client [new Http/Client $ns $node(c)]            ;## Create a client
set tmp [new RandomVariable/Exponential]     ;## Poisson process as request sequence
$tmp set avg_ 5                                      ;## average request interval
$client set-interval-generator $tmp
$client set-page-generator $pgp
$client log $log

set startTime 1                                      ;## simulation start time
set finishTime 50                                    ;## simulation end time
$ns at $startTime "start-connection"
$ns at $finishTime "finish"
```

Then we define a procedure which will be called after simulation starts. The procedure will setup connections among all Http objects.

```
proc start-connection
        global ns server cache client
        $client connect $cache
        $cache connect $server
        $client start-session $cache $server
```

At the end, the usual closing:

```
proc finish
        global ns log
        $ns flush-trace
        flush $log
        close $log
        exit 0

$ns run
```

This script is also available at *ns*/tcl/ex/simple-webcache.tcl. Examining its output `http.log`, one will find that the result of the absense cache consistency algorithm results in a lot of stale hits. This can be easily remedied by replacing "new Http/Cache" line with: `set cache [new Http/Cache/TTL $ns $node(e)]`. For more complicated cache consistency algorithm examples, see *ns*/tcl/test/test-suite-webcache.tcl.

22.7 Http trace format

The trace file of Http agents are constructed in a similar way as the SRM trace files. It consists of multiple entries, each of which occupies one line. The format of each entry is:

Time	ObjectID	Object Values

There are three types of objects: client (**C**), cache (**E**) and server (**S**). Following is a complete enumeration of all possible events and value types associated with these three types of objects.

Object Type	*Event Type*	*Values*
E	HIT	⟨Prefix⟩
E	MISS	⟨Prefix⟩ z ⟨RequestSize⟩
E	IMS	⟨Prefix⟩ z ⟨Size⟩ t ⟨CacheEntryTime⟩
E	REF	p ⟨PageID⟩ s ⟨ServerID⟩ z ⟨Size⟩
E	UPD	p ⟨PageID⟩ m ⟨LastModifiedTime⟩ z ⟨PageSize⟩ s ⟨ServerID⟩
E	GUPD	z ⟨PageSize⟩
E	SINV	p ⟨PageID⟩ m ⟨LastModTime⟩ z ⟨PageSize⟩
E	GINV	p ⟨PageID⟩ m ⟨LastModTime⟩
E	SPF	p ⟨PageID⟩ c ⟨DestCache⟩
E	RPF	p ⟨PageID⟩ c ⟨SrcCache⟩
E	ENT	p ⟨PageID⟩ m ⟨LastModifiedTime⟩ z ⟨PageSize⟩ s ⟨ServerID⟩
C	GET	p ⟨PageID⟩ s ⟨PageServerID⟩ z ⟨RequestSize⟩
C	STA	p ⟨PageID⟩ s ⟨OrigServerID⟩ l ⟨StaleTime⟩
C	RCV	p ⟨PageID⟩ s ⟨PageServerID⟩ l ⟨ResponseTime⟩ z ⟨PageSize⟩
S	INV	p ⟨PageID⟩ m ⟨LastModifiedTime⟩ z ⟨Size⟩
S	UPD	p ⟨PageID⟩ m ⟨LastModifiedTime⟩ z ⟨Size⟩
S	SND	p ⟨PageID⟩ m ⟨LastModifiedTime⟩ z ⟨PageSize⟩ t ⟨Requesttype⟩

⟨Prefix⟩ is the information common to all trace entries. It includes:

p ⟨PageID⟩ | c ⟨RequestClientID⟩ | s ⟨PageServerID⟩

Short Explaination of event operations:

Object Type	*Event Type*	*Explaination*
E	HIT	Cache hit. PageSererID is the id of the "owner" of the page.
E	MISS	Cache miss. In this case the cache will send a request to the server to fetch the page.
E	IMS	If-Modified-Since. Used by TTL procotols to validate an expired page.
E	REF	Page refetch. Used by invalidation protocols to refetch an invalidated page.
E	UPD	Page update. Used by invalidation protocols to "push" updates from parent cache to children caches.
E	SINV	Send invalidation.
E	GINV	Get invalidation.
E	SPF	Send a pro forma
E	RPF	Receive a pro forma
E	ENT	Enter a page into local page cache.
C	GET	Client sends a request for a page.
C	STA	Client gets a stale hit. OrigModTime is the modification time in the web server, CurrModTime is the local page's modification time.
C	RCV	Client receives a page.
S	SND	Server send a response.
S	UPD	Server pushes a page update to its "primary cache". Used by invalidation protocol only.
S	INV	Server sends an invalidation message. Used by invalidation protocol only.

Part VII

Scale

Chapter 23

Session-level Packet Distribution

This section describes the internals of the Session-level Packet Distribution implementation in *ns*. The section is in two parts: the first part is an overview of Session configuration (Section 23.1), and a "complete" description of the configuration parameters of a Session. The second part describes the architecture, internals, and the code path of the Session-level Packet distribution.

The procedures and functions described in this chapter can be found in ~*ns*/tcl/session/session.tcl.

Session-level Packet Distribution is oriented towards performing multicast simulations over large topologies. The memory requirements for some topologies using session level simulations are:

$$\begin{aligned} \text{2048 nodes, degree of connectivity} = 8 &\approx \ \ 40\,\text{MB} \\ \text{2049–4096 nodes} &\approx 167\,\text{MB} \\ \text{4097–8194 nodes} &\approx 671\,\text{MB} \end{aligned}$$

Note however, that session level simulations ignore qeueing delays. Therefore, the accuracy of simulations that use sources with a high data rate, or those that use multiple sources that get aggregated at points within the network is suspect.

23.1 Configuration

Configuration of a session level simulation consists of two parts, configuration of the session level details themselves (Section 23.1.1) and adding loss and error models to the session level abstraction to model specific behaviours (Section 23.1.2).

23.1.1 Basic Configuration

The basic configuration consists of creating and configuring a multicast session. Each Session (*i.e.*, a multicast tree) must be configured strictly in this order: (1) create and configure the session source, (2) create the session helper and attach it to the session source, and finally, (3) have the session members join the session.

```
set ns [new SessionSim]                          ;# preamble initialization
set node [$ns node]
set group [$ns allocaddr]
```

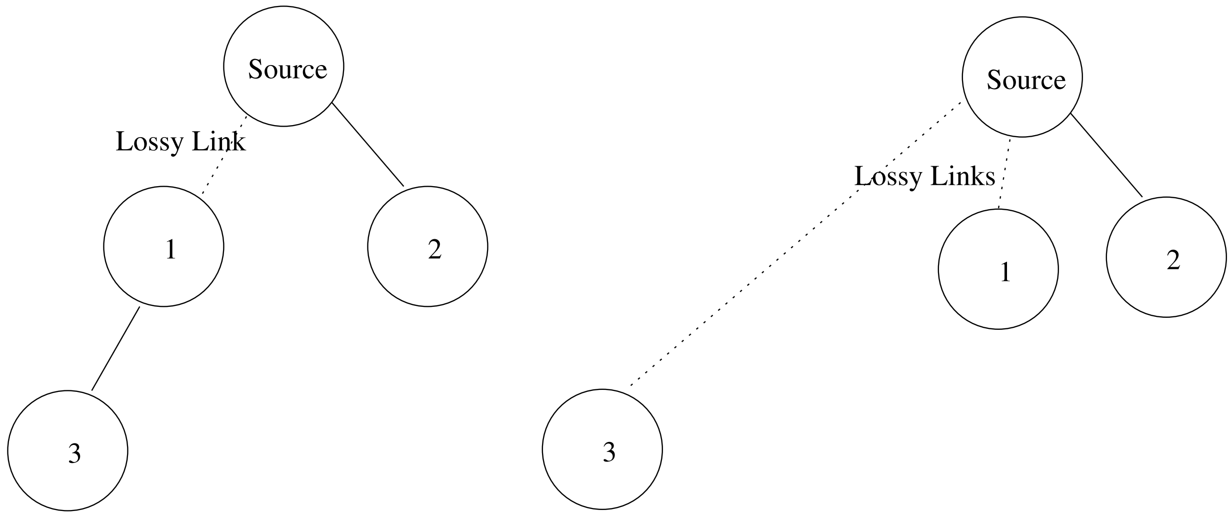


Figure 23.1: Comparison of Multicast Trees for Detailed vs. Session Routing

```
        set udp [new Agent/UDP]                              ;# create and configure the source
        $udp set dst_ $group
set src [new Application/Traffic/CBR]
$src attach-agent $udp
        $ns attach-agent $node $udp

        $ns create-session $node $udp                        ;# create attach session helper to src

        set rcvr [new Agent/NULL]                            ;# configure the receiver
        $ns attach-agent $node $rcvr
        $ns at 0.0 "$node join-group $rcvr $group"           ;# joining the session

        $ns at 0.1 "$src start"
```

A session level simulation scales by translating the topology into a virtual mesh topology. The steps involved in doing this are:

1. All of the classifiers and replicators are eliminated. Each node only stores instance variables to track its node id, and port ids.
2. Links do not consist of multiple components. Each link only stores instance variables to track the bandwidth and delay attributes.
3. The topology, consisting of links is translated into a virtual mesh.

Figure 23.1 shows the difference between a multicast tree in a detailed simulation and one in a session level simulation. Notice that the translation process results in a session level simulation ignoring queuing delays. For most simulations, *ns* already ignores processing delays at all of the nodes.

23.1.2 Inserting a Loss Module

When studying a protocol (*e.g.*, SRM error recovery mechanism), it might be useful to study protocol behavior over lossy links. However, since a session level simulation scales by abstracting out the internal topology, we need additional mechanisms to insert a loss module appropriately. This subsection describes how one can create these loss modules to model error scenarios.

Creating a Loss Module Before we can insert a loss module in between a source-receiver pair, we have to create the loss module. Basically, a loss module compares two values to decide whether to drop a packet. The first value is obtained every time when the loss module receives a packet from a random variable. The second value is fixed and configured when the loss module is created.

The following code gives an example to create a uniform 0.1 loss rate.

```
# creating the uniform distribution random variable
set loss_random_variable [new RandomVariable/Uniform]
$loss_random_variable set min_ 0                 ;# set the range of the random variable
$loss_random_variable set max_ 100

set loss_module [new ErrorModel]                             ;# create the error model
$loss_module drop-target [new Agent/Null]
$loss_module set rate_ 10                  ;# set error rate to 0.1 = 10 / (100 − 0)
$loss_module ranvar $loss_random_variable         ;# attach random var. to loss module
```

A catalogue of the random variable distributions was described earlier (Chapter 13). A more detailed discussion of error models was also described earlier in a different chapter (Chapter 11).

Inserting a Loss Module A loss module can only be inserted after the corresponding receiver has joined the group. The example code below illustrates how a simulation script can introduce a loss module.

```
set sessionhelper [$ns create-session $node $src]  ;# keep a handle to the loss module
$ns at 0.1 "$sessionhelper insert-depended-loss $loss_module $rcvr"
```

23.2 Architecture

The purpose of Session-level packet distribution is to speed up simulations and reduce memory consumption while maintaining reasonable accuracy. The first bottleneck observed is the memory consumption by heavy-weight links and nodes. Therefore, in SessionSim (Simulator for Session-level packet distribution), we keep only minimal amount of states for links and nodes, and connect the higher level source and receiver applications with appropriate delay and loss modules. A particular source in a group sends its data packets to a replicator that is responsible for replicating the packets for all the receivers. Intermediate loss and delay modules between this replicator and the receivers will guarantee the appropriate end-to-end characteristics. To put it another way, a session level simulation abstracts out the topology, routing and queueing delays. Packets in SessionSim do not get routed. They only follow the established Session.

23.3 Internals

This section describes the internals of Session-level Packet Distribution. We first describe the OTcl primitives to configure a session level simulation (Section 23.3.1); we conclude with a brief note on hos packet forwarding is achieved (Section 23.3.2).

23.3.1 Object Linkage

We describe three aspects of constructing a session level simulation in *ns*: the modified topology routines that permit the creation of abstract nodes and links, establish the session helper for each active source, add receivers to the session by inserting the appropriate loss and delay models when that receiver joins the appropriate group.

Nodes and Links The node contains only its node id and the port number for the next agent. A link only contains the values of its bandwidth and delay.

```
SessionNode instproc init {} {
    $self instvar id_ np_
    set id_ [Node getid]
    set np_ 0
}

SessionSim instproc simplex-link { n1 n2 bw delay type } {
    $self instvar bw_ delay_
    set sid [$n1 id]
    set did [$n2 id]

    set bw_($sid:$did) [expr [string trimright $bw Mb] * 1000000]
    set delay_($sid:$did) [expr [string trimright $delay ms] * 0.001]
}
```

Session Helper Each active source in a session requires a "session helper". The session helper in *ns* is realised through a replicator. This session helper is created when the user issues a `create-session{}` to identify the source agent. The simulator itself keeps a reference to the session helper in its instance variable array, `session_`, indexed by the source and destination address of the source.

Note that the destination of source agent must be set before calling `create-session{}`.

```
SessionSim instproc create-session { node agent } {
    $self instvar session_

    set nid [$node id]
    set dst [$agent set dst_]
    set session_($nid:$dst) [new Classifier/Replicator/Demuxer]
    $agent target $session_($nid:$dst)                    ;# attach the replicator to the source
    return $session_($nid:$dst) ;# keep the replicator in the SessionSim instance variable array session_
}
```

Delay and Loss Modules Each receiver in a group requires a delay module that reflects its delay with respect to the particular source. When the receiver joins a group, `join-group{}` identifies all session helpers in `session_`. If the destination index matches the group address the receiver are joining, then the following actions are performed.

1. A new slot of the session helper is created and assigned to the receiver.
2. The routine computes the accumulated bandwidth and delay between the source and receiver using the SessionSim instance procedures `get-bw{}` and `get-delay{}`.
3. A constant random variable is created; it will generate random delivery times using the accumulative delay as an estimate of the average delay.
4. A new delay module is created with the end-to-end bandwidth characteristics, and the random variable generator provides the delay estimates.
5. The delay module in inserted into the session helper and interposed between the helper and the receiver.

See Section 23.1.2 for similarly inserting a loss module for a receiver.

```
SessionSim instproc join-group { agent group } {
    $self instvar session_

    foreach index [array names session_] {
        set pair [split $index :]
        if {[lindex $pair 1] == $group} {
            # Note: must insert the chain of loss, delay,
            # and destination agent in this order:

            $session_($index) insert $agent        ;# insert destination agent into session replicator

            set src [lindex $pair 0]                           ;# find accum. b/w and delay
            set dst [[$agent set node_] id]
            set accu_bw [$self get-bw $dst $src]
            set delay [$self get-delay $dst $src]

            set random_variable [new RandomVariable/Constant]        ;# set delay variable
            $random_variable set avg_ $delay

            set delay_module [new DelayModel]                  ;# configure the delay module
            $delay_module bandwidth $accu_bw
            $delay_module ranvar $random_variable

            $session_($index) insert-module $delay_module $agent ;# insert the delay module
        }
    }
}
```

23.3.2 Packet Forwarding

Packet forwarding activities are executed in C++. A source application generates a packet and forwards to its target which must be a replicator (session helper). The replicator copies the packet and forwards to targets in the active slots which are either delay modules or loss modules. If loss modules, a decision is made whether to drop the packet. If yes, the packet is

Figure 23.2: Architectural Realization of a Session Level Simulation Session

forwarded to the loss modules drop target. If not, the loss module forwards it to its target which must be a delay module. The delay module will forward the packet with a delay to its target which must be a receiver application.

Part VIII

Emulation

Chapter 24

Emulation

This chapter describes the *emulation* facility of *ns*. Emulation refers to the ability to introduce the simulator into a live network. Special objects within the simulator are capable of introducing live traffic into the simulator and injecting traffic from the simulator into the live network.

Note that while the interfaces described below are not expected to change drastically, this facility is still under development and should be considered experimental and subject to change. Note further that the facility described here has been developed under FreeBSD 2.2.5, and use on other systems has not been tested by the author.

24.1 Introduction

The emulation facility can be subdivided into two modes:

1. opaque mode – live data treated as opaque data packets
2. protocol mode – live data may be interpreted/generated by simulator

In opaque mode, the simulator treats network data as uninterpreted packets. In particular, real-world protocol fields are not directly manipulated by the simulator. In opaque mode, live data packets may be dropped, delayed, re-ordered, or duplicated, but because no protocol processing is performed, protocol-specific traffic manipulation scenarios (e.g. "drop the TCP segment containing a retransmission of sequence number 23045") may not be performed. In protocol mode, the simulator is able to interpret and/or generate live network traffic containing arbitrary field assignments. **To date (Mar 1998), only Opaque Mode is currently implemented**.

The interface between the simulator and live network is provided by a collection of objects including *tap agents* and *network objects*. Tap agents embed live network data into simulated packets and vice-versa. Network objects are installed in tap agents and provide an entrypoint for the sending and receipt of live data. Both objects are described in the following sections.

When using the emulation mode, a special version of the system scheduler is used: the `RealTime` scheduler. This scheduler uses the same underlying structure as the standard calendar-queue based scheduler, but ties the execution of events to real-time. It is described below.

24.2 Real-Time Scheduler

The real-time scheduler implements a soft real-time scheduler which ties event execution within the simulator to real time. Provided sufficient CPU horsepower is available to keep up with arriving packets, the simulator virtual time should closely track real-time. If the simulator becomes too slow to keep up with elapsing real time, a warning is continually produced if the skew exceeds a pre-specified constant "slop factor" (currently 10ms).

The main dispatch loop is found in the routine `RealTimeScheduler::run()`, in the file `scheduler.cc`. It follows essentially the following algorithm:

- While simulator is not halted
  - get current real time ("now")
  - dispatch all pending simulator events prior to now
  - fetch next (future) event if there is one
  - delay until the next simulator event is ready or a Tcl event occurs
  - if a tcl event occured, re-insert next event in simulator event queue and continue
  - otherwise, dispatch simulator event, continue
  - if there was no future even, check for Tcl events and continue

The real-time scheduler should always be used with the emulation facility. Failure to do so may easily result in the simulator running faster than real-time. In such cases, traffic passing through the simulated network will not be delayed by the proper amount of time. Enabling the real-time scheduler requires the following specification at the beginning of a simulation script:

```
set ns [new Simulator]
$ns use-scheduler RealTime
```

24.3 Tap Agents

The class `TapAgent` is a simple class derived from the base `Agent` class. As such, it is able to generate simulator packets containing arbitrarily-assigned values within the *ns* common header. The tap agent handles the setting of the common header packet size field and the type field. It uses the packet type `PT_LIVE` for packets injected into the simulator. Each tap agent can have at most one associated network object, although more than one tap agent may be instantiated on a single simulator node.

Configuration Tap agents are able to send and receive packets to/from an associated `Network` object. Assuming a network object `$netobj` refers to a network object, a tap agent is configured using the `network` method:

```
set a0 [new Agent/Tap]
$a0 network $netobj
$a0 set fid_ 26
$a0 set prio_ 2
$ns connect $a0 $a1
```

Note that the configuration of the flow ID and priority are handled through the `Agent` base class. The purpose of setting the flow id field in the common header is to label packets belonging to particular flows of live data. Such packets can be differentially treated with respect to drops, reorderings, etc. The `connect` method instructs agent `$a0` to send its live traffic to the `$a1` agent via the current route through the simulated topology.

24.4 Network Objects

Network objects provide access to a live network. There are several forms of network objects, depending on the protocol layer specified for access to the underlying network, in addition to the facilities provided by the host operating system. Use of some network objects requires special access privileges where noted. Generally, network objects provide an entrypoint into the live network at a particular protocol layer (e.g. link, raw IP, UDP, etc) and with a particular access mode (read-only, write-only, or read-write). Some network objects provide specialized facilities such as filtering or promiscuous access (i.e. the pcap/bpf network object) or group membership (i.e. UDP/IP multicast). The C++ class `Network` is provided as a base class from which specific network objects are derived. Three network objects are currently supported: pcap/bpf, raw IP, and UDP/IP. Each are described below.

24.4.1 Pcap/BPF Network Objects

These objects provide an extended interface to the LBNL packet capture library (libpcap). (See `ftp://ftp.ee.lbl.gov/libpcap.t` for more info). This library provides the ability to capture link-layer frames in a promiscuous fashion from network interface drivers (i.e. a copy is made for those programs making use of libpcap). It also provides the ability to read and write packet trace files in the "tcpdump" format. The extended interface provided by *ns* also allows for writing frames out to the network interface driver, provided the driver itself allows this action. Use of the library to capture or create live traffic may be protected; one generally requires at least read access to the system's packet filter facility which may need to be arranged through a system administrator.

The packet capture library works on several UNIX-based platforms. It is optimized for use with the Berkeley Packet Filter (BPF) [12], and provides a filter compiler for the BPF pseudomachine machine code. On most systems supporting it, a kernel-resident BPF implementation processes the filter code, and applies the resulting pattern matching instructions to received frames. Those frames matching the patterns are received through the BPF machinery; those not matching the pattern are otherwise unaffected. BPF also supports sending link-layer frames. This is generally not suggested, as an entire properly-formatted frame must be created prior to handing it off to BPF. This may be problematic with respect to assigning proper link-layer headers for next-hop destinations. It is generally preferable to use the raw IP network object for sending IP packets, as the system's routing function will be used to determine proper link-layer encapsulating headers.

Configuration Pcap network objects may be configured as either associated with a live network or with a trace file. If associated with a live network, the particular network interface to be used may be specified, as well as an optional promiscuous flag. As with all network objects, they may be opened for reading or writing. Here is an example:

```
set me [exec hostname]
set pf1 [new Network/Pcap/Live]
$pf1 set promisc_ true
set intf [$pf1 open readonly]
puts "pf1 configured on interface $intf"
set filt "(ip src host foobar) and (not ether broadcast)"
set nbytes [$pf1 filter $filt]
puts "filter compiled to $nbytes bytes"
puts "drops: [$pf1 pdrops], pkts: [$pf1 pkts]"
```

This example first determines the name of the local system which will be used in constructing a BPF/libpcap filter predicate. The `new Network/Pcap/Live` call creates an instance of the pcap network object for capturing live traffic. The `promisc_` flag tells the packet filter whether it should configure the undelying interface in promiscuous mode (if it is supported). The `open` call activates the packet filter, and may be specified as `readonly`, `writeonly`, or `readwrite`. It returns the name of the network interface the filter is associated with. The `open` call takes an optional extra parameter (not illustrated) indicating the name of the interface to use in cases where a particular interface should be used on a multi-homed host. The `filter` method is used to create a BPF-compatible packet filter program which is loaded into the underlying BPF machinery. The `filter` method returns the number of bytes used by the filter predicate. The `pdrops` and `pkts` methods are available for statistics collection. They report the number of packets dropped by the filter due to buffer exhaustion and the total number of packets that arrived at the filter, respectively (*not* the number of packets accepted by the filter).

24.4.2 IP Network Objects

These objects provide raw access to the IP protocol, and allow the complete specification of IP packets (including header). The implementation makes use of a *raw socket*. In most UNIX systems, access to such sockets requires super-user privileges. In addition, the interface to raw sockets is somewhat less standard than other types of sockets. The class `Network/IP` provides raw IP functionality plus a base class from which other network objects implementing higher-layer protocols are derived.

Configuration The configuration of a raw IP network object is comparatively simple. The object is not associated with any particular physical network interface; the system's IP routing capability will be used to emit the specified datagram out whichever interface is required to reach the destination address contained in the header. Here is an example of configuring an IP object:

```
set ipnet [new Network/IP]
$ipnet open writeonly
...
$ipnet close
```

The IP network object supports only the `open` and `close` methods.

24.4.3 IP/UDP Network Objects

These objects provide access to the system's UDP implementation along with support for IP multicast group membership operations. **IN PROGRESS**

24.5 An Example

The following code illustrates a small but complete simulation script for setting up an emulation test using BPF and IP network objects. It was run on a multi-homed machine, and the simulator essentially provides routing capability by reading frames from one interface, passing them through the simulated network, and writing them out via the raw IP network object:

```
set me "10.0.1.1"
```

```
set ns [new Simulator]

$ns use-scheduler RealTime

#
# we want the test machine to have ip forwarding disabled, so
# check this (this is how to do so under FreeBSD at least)
#

set ipforw [exec sysctl -n net.inet.ip.forwarding]
if { $ipforw } {
        puts "can not run with ip forwarding enabled"
        exit 1
}

#
# allocate a BPF type network object and a raw-IP object
#
set bpf0 [new Network/Pcap/Live]
set bpf1 [new Network/Pcap/Live]
$bpf0 set promisc_ true
$bpf1 set promisc_ true

set ipnet [new Network/IP]

set nd0 [$bpf0 open readonly fxp0]
set nd1 [$bpf1 open readonly fxp1]
$ipnet open writeonly

#
# try to filter out weird stuff like netbios pkts, arp requests, dns,
# also, don't catch stuff to/from myself or broadcasted
#
set notme "(not ip host $me)"
set notbcast "(not ether broadcast)"
set ftp "and port ftp-data"
set f0len [$bpf0 filter "(ip dst host bit) and $notme and $notbcast"]
set f1len [$bpf1 filter "(ip src host bit) and $notme and $notbcast"]

puts "filter lengths: $f0len (bpf0), $f1len (bpf1)"
puts "dev $nd0 has address [$bpf0 linkaddr]"
puts "dev $nd1 has address [$bpf1 linkaddr]"

set a0 [new Agent/Tap]
set a1 [new Agent/Tap]
set a2 [new Agent/Tap]
puts "install nets into taps..."
$a0 network $bpf0
$a1 network $bpf1
$a2 network $ipnet

set node0 [$ns node]
set node1 [$ns node]
set node2 [$ns node]
```

```
$ns simplex-link $node0 $node2 10Mb 10ms DropTail
$ns simplex-link $node1 $node2 10Mb 10ms DropTail

$ns attach-agent $node0 $a0
$ns attach-agent $node1 $a1
$ns attach-agent $node2 $a2

$ns connect $a0 $a2
$ns connect $a1 $a2

puts "okey"
$ns run
```

Bibliography

[1] C. Alaettinoğlu, A.U. Shankar, K. Dussa-Zeiger, and I. Matta. Design and implementation of MaRS: A routing testbed. *Internetworking: Research and Experience*, 5:17–41, 1994.

[2] L.S. Brakmo, S. O'Malley, and L.L. Peterson. TCP vegas: New techniques for congestion detection and avoidance. In *Proceedings of the ACM SIGCOMM*, pages 24–35, October 1994.

[3] L.S. Brakmo, S. O'Malley, and L.L. Peterson. TCP vegas: New techniques for congestion detection and avoidance. Technical Report TR 94 04, Department of Computer Science, The University of Arizona, Tucson, AZ 85721, February 1994.

[4] R. Brown. Calendar queues: A fast O(1) priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31(10):1220–1227, October 1988.

[5] S. Deering, D. Estrin, D. Farinacci, V. Jacobson, Ching-Gung Liu, and L. Wei. An architecture for wise-area multicast routing. Technical Report USC-SC-94-565, Computer Science Department, University of Southern California, Los Angeles, CA 90089., 1994.

[6] S. Floyd, V. Jacobson, C. Liu, S. McCanne, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. In *Proceedings of the ACM SIGCOMM*, pages 342–356, August 1995.

[7] A. Heybey. *Netsim Manual*. MIT, 1989.

[8] R. Jain. *The Art of Computer Systems Performance Analysis*. John Wiley and Sons, Inc., 1996.

[9] M. Mathis and J. Mahdavi. Forward acknowledgement: Refining TCP congestion control. In *Proceedings of the ACM SIGCOMM*, August 1996.

[10] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanov. *TCP Selective Acknowledgement Options*, RFC 2018 edition, 1996.

[11] S. McCanne and S. Floyd. ns—Network Simulator. `http://www-mash.cs.berkeley.edu/ns/`.

[12] S. McCanne and V. Jacobson. The bsd packet filter: A new architecture for user-level packet capture. pages 259–269, January 1993.

[13] S.K. Park and R.W. Miller. Random number generation: Good ones are hard to find. *Communications of the ACM*, 31(10):1192–1201, October 1988.

[14] D. Waitzman, C. Partridge, and S.E. Deering. *Distance Vector Multicast Routing Protocol*, RFC 1075 edition, 1988.