Chapter 6
Nodes as Routers or Computer Hosts

This chapter focuses on a basic network component, *Node*. In NS2, a Node acts as a computer host (e.g., a source or a destination) and a router (e.g., an intermediate node). It receives packets from an attached application or an upstream object, and forwards them to the attached links specified in the routing table (as a router) or delivers them from/to transport layer agents (as a host).

In the following, we first give an overview of routing mechanism and Nodes in Sect. 6.1. Sections 6.2–6.4 discuss three main routing components: classifiers, routing modules, and route logic, respectively. In Sect. 6.5, we show how the aforementioned Node components are assembled to compose a Node. Finally, the chapter summary is provided in Sect. 6.6.

6.1 An Overview of Nodes in NS2

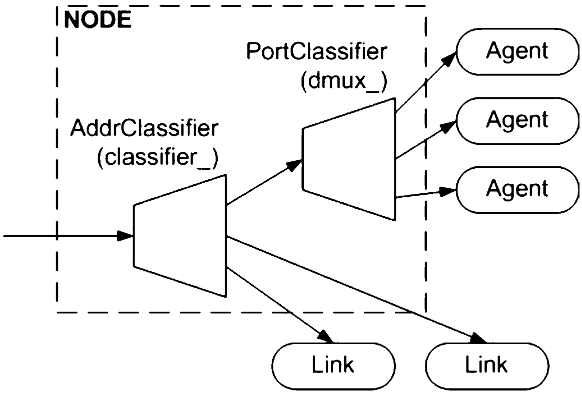
6.1.1 Routing Concept and Terminology

In NS2, routing has a broader definition than that usually used in practice. Routing usually refers to a network layer operation which determines the route along which a packet should be forwarded to its destination. In NS2, routing is an act of forwarding a packet from one NsObject to another. It can occur within a Node (i.e., no communication), from a Node to a link (i.e., network layer), between a Node and an agent (i.e., transport layer), and so on. In order to avoid confusion, let us define the following terminologies:

- *Routing mechanism*: An act of determining and passing packets according to predefined routing rules
- *Routing rule or route entry*: A rule which determines where a packet should be forwarded to; it is usually expressed in the form of `(dst,target)` – meaning that packets destined for "`dst`" should be forwarded to "`target`."
- *Routing table*: A collection of routing rules

T. Issariyakul and E. Hossain, *Introduction to Network Simulator NS2*,
DOI 10.1007/978-1-4614-1406-3_6, © Springer Science+Business Media, LLC 2012

Fig. 6.1 Node architecture



- *Routing algorithm*: An algorithm which computes routing rules (e.g., Dijkstra algorithm [19])
- *Routing protocol*: A communication protocol designed to update the routing rules according to dynamic environment (e.g., Ad hoc On-demand Distance Vector (AODV) [24])
- *Routing agent*: An entity which gathers parameters (e.g., network topology) necessary to compute routing rules.
- *Route logic*: An NS2 component which runs the routing algorithm (i.e., computing routing rules)
- *Router*: An entity which run routing mechanism; in NS2, this entity is an *address classifier*.
- *Routing module*: A single point of management, which manages a group of classifiers

This chapter focuses on static routing, which involves the following main NS2 components: Nodes, classifiers, routing modules, route logic.

6.1.2 Architecture of a Node

A Node is an OTcl composite object whose architecture is shown in Fig. 6.1. Nodes are defined in an OTcl class `Node`, which is bound to C++ class with the same name. A Node consists of two main components: an address classifier (instvar `classifier_`) and a port classifier (instvar `dmux_`). These two components have one entry point and multiple forwarding targets. An address classifier acts as a router which receives a packet from an upstream object and forwards the packet to one of its connecting links based on the address embedded in packet header. A port classifier acts as a transport layer bridge – taking a packet from the address classifier (in case that the packet is destined to this particular node), and forwarding the packet to one of the attached transport layer agents.

6.1.3 Default Nodes and Node Configuration Interface

A default NS2 Node is based on flat-addressing and static routing. With flat-addressing, an address of every new node is incremented by one from that of the previously created node. Static routing assumes no change in topology. The routing table is computed once at the beginning of the Simulation phase and does not change thereafter. By default, NS2 uses the Dijkstra's shortest path algorithm [19] to compute optimal routes for all pairs of Nodes. Again, this chapter focuses on Nodes with flat-addressing and static routing only. The details about other routing protocols as well as hierarchical addressing can be found in the NS manual [17].

To provide a default Node with more functionalities such as link layer or Medium Access Control (MAC) protocol functionalities, we may use the instproc `node-config` of class `Simulator` whose syntax is as follows:

```
$ns node-config -<option> [<value>]
```

where `$ns` is the `Simulator` object.

An example use of the instproc `node-config{args}` for the default setting is shown below:

```
$ns_ node-config -addressType   flat
                 -adhocRouting
                 -llType
                 -macType
                 -propType
                 -ifqType
                 -ifqLen
                 -phyType
                 -antType
                 -channel
                 -channelType
                 -topologyInstance
```

By default, almost every option is specified as `NULL` with the exception of `addressType`, which is set to be flat addressing. The instproc `node-config` has an option `reset`, i.e.,

```
$ns node-config -reset
```

which is used to restore default parameter setting. The details of instproc `node-config` (e.g., other options) can be found in the file ˜*ns*/tcl/lib/ns-lib.tcl and [17].

Note that this instproc does not immediately configure the Nodes as specified in the `<option>`. Instead, it stores `<value>` in the instvars of the `Simulator` corresponding to `<option>`. This stored configuration will be used during a Node construction process. As a result, the instproc `node-config` must be executed before Node construction.

6.2 Classifiers: Multi-Target Packet Forwarders

A classifier is a packet forwarding object with multiple connecting targets. It classifies incoming packets according to a predefined criterion (e.g., destination address or transport layer port). Packets with the same category are forwarded to the same NsObject.

NS2 implements classifiers using the concept of *slots*. A slot is a placeholder for a pointer to an NsObject. It is associated with a packet category. When a packet arrives, a classifier determines the packet category and forwards the packet to the NsObject whose pointer was installed in the associated slot.

In the following, we shall discuss the details of two main processes of classifiers: configuration and internal mechanism. Configuration defines what the users ask a classifier to perform. It includes the following main steps:

1. Define the categories
2. Identify a corresponding slot as well as a forwarding NsObject for each category
3. Install the NsObject pointer in the selected slot

Internal mechanism is what a classifier does to carry out the requirement provided by users. It usually begins with the C++ function `recv(p,h)`.

For example, suppose we would like to attach a node to a transport layer agent at the port number 50. In the configuration, we install the agent in slot number 50. The internal mechanism is to tell the classifier the following: send all the packets whose port number is 50 to the NsObject whose pointer is in the slot number 50.

6.2.1 *Class `Classifier` and Its Main Components*

NS2 implements classifiers in a C++ class `Classifier` (see the declaration in Program 6.1), which is bound to an OTcl class with the same name. The main components of a classifier include the following.

6.2.1.1 C++ Variables

The C++ class `Classifier` has two key variables: `slot_` and `default_target_` (Lines 13 and 14 in Program 6.1). The variable `slot_` is a link list whose entries are pointers to downstream NsObjects. Each of these NsObjects corresponds to a predefined criterion. Packets matching with a certain criterion are forwarded to the corresponding NsObject. The variable `default_target_` points to a downstream NsObject for packets which do not match with any predefined criterion.

Program 6.1 Declaration of class `Classifier`

```
    //~/ns/classifier/classifier.h
 1  class Classifier : public NsObject {
 2  public:
 3      Classifier();
 4      virtual ~Classifier();
 5      virtual void recv(Packet* p, Handler* h);
 6      virtual NsObject* find(Packet*);
 7      virtual int classify(Packet *);
 8      virtual void clear(int slot);
 9      virtual void install(int slot, NsObject*);
10      inline int mshift(int val) {return((val >> shift_) &
          mask_);}
11  protected:
12      virtual int command(int argc, const char*const* argv);
13      NsObject** slot_;
14      NsObject *default_target_;
15      int shift_;
16      int mask_;
17 };
```

The class `Classifier` also have two supplementary variables: `shift_` (Line 15) and `mask_` (Line 16). These two variables are used in function `mshift (val)` (Line 10) to reformat the address (see also Sect. 15.4).

6.2.1.2 C++ Functions

The main C++ functions of class `Classifier` are shown below:

Configuration Functions

`install(slot,p)`	Store the input `NsObject` pointer "p" in the slot number "`slot`'.
`install_next(node)`	Install the `NsObject` pointer "node" in the next available slot.
`do_install (dst,target)`	Similar to `install{slot,p}` but the input parameter `dst` is a string instead of an integer.
`clear(slot)`	Remove the NsObject pointer installed in the slot number "`slot`."
`mshift(val)`	Shift `val` to the left by "`shift_`" bits. Masks the shifted value using a logical AND (`&`) operation with "`mask_`."

Packet Forwarding (i.e., Internal) Functions

`recv(p,h)`	Receive a packet `*p` and handler `*h`.
`find(p)`	Return a forwarding NsObject pointer for an incoming packet `*p`.
`classify(p)`	Return a slot number whose associated criterion matches with the header of an incoming packet `*p`.

6.2.1.3 Main Configuring Interface

C++ Functions

Program 6.2 shows the details of key C++ configuration functions. Function `install(slot,p)` stores the input `NsObject` pointer "`p`" in the slot number "`slot`" of the variable "`slot_`" (Line 5). Function `install_next(node)` installs the input `NsObject` pointer "`node`" in the next available slot (Lines 10 and 11). Function `do_install(dst,target)` converts "`dst`" to be an integer variable (Line 21), and installs the NsObject pointer "`target`" in the slot corresponding to "`dst`" (Line 22). Finally, function `clear(slot)` removes the installed `NsObject` pointer from the slot number "`slot`" of the variable "`slot_`" (Line 16).

OTcl Commands

Class `Classifier` also defines the following key OTcl commands in a C++ function `command(...)` of class `Classifier` (in the file ~*ns*/classifier/classifier.cc).

`slot{index}`	Return the NsObject stored in the slot number `index`
`clear{slot}`	Clear the NsObject pointer installed in the slot number `slot`.
`install{index object}`	Install `object` in the slot number `index`.
`installNext{object}`	Install `object` in the next available slot.
`defaulttarget{object}`	Store `object` in the C++ variable `default_target_`.

6.2.1.4 Main Internal Mechanism

As an NsObject, a classifier receives a packet by having its upstream object invoke its function `recv(p,h)`, passing a packet pointer "`p`" and a handler pointer "`h`" as

Program 6.2 Functions `install`, `install_next`, `clear`, and `do_install` of class `Classifier`

```
//~ns/classifier/classifier.cc
void Classifier::install(int slot, NsObject* p)
{
    if (slot >= nslot_)
        alloc(slot);
    slot_[slot] = p;
    if (slot >= maxslot_)
        maxslot_ = slot;
}

int Classifier::install_next(NsObject *node) {
    int slot = maxslot_ + 1;
    install(slot, node);
    return (slot);
}

void Classifier::clear(int slot)
{
    slot_[slot] = 0;
    if (slot == maxslot_)
        while (--maxslot_ >= 0 && slot_[maxslot_] == 0);
}

//~ns/classifier/classifier.h
virtual void do_install(char* dst, NsObject *target) {
    int slot = atoi(dst);
    install(slot, target);
}
```

input arguments. In Program 6.3, Line 3 determines a forwarding NsObject "`node`" for an incoming packet `*p`, by invoking function `find(*p)`. Then, Line 8 passes the packet pointer "`p`" and the handler pointer "`h`" to its forwarding NsObject `*node` by executing `node->recv(p,h)`.

Function `find(p)` (Lines 10–18 in Program 6.3) examines the incoming packet `*p` and retrieves the matched `NsObject` pointer installed in the variable `slot_`. Line 13 invokes function `classify(p)` to retrieve the slot number (i.e., the variable `cl`) corresponding to the packet `*p`. Then, Lines 14 and 17 return the `NsObject` pointer (i.e., `node`) stored in the slot number `cl` of the variable `slot_`.

Function `classify(p)` is perhaps the most important function of a classifier. This is the place where the classification criterion is defined. Function `classify(p)` returns the slot number which matches with the input packet `*p` under the predefined criterion. Since the classification criteria could be different for different types of classifiers, function `classify(p)` is usually

Program 6.3 Functions `recv` and `find` of class `Classifier`

```
//~/ns/classifier/classifier.cc
void Classifier::recv(Packet* p, Handler* h)
{
   NsObject* node = find(p);
   if (node == NULL) {
       Packet::free(p);
       return;
   }
   node->recv(p,h);
}

NsObject* Classifier::find(Packet* p)
{
   NsObject* node = NULL;
   int cl = classify(p);
   if (cl < 0 || cl >= nslot_ || (node = slot_[cl]) == 0) {
       /*There is no potential target in the slot;*/
   }
   return (node);
 }
```

Program 6.4 Function `classify` of class `PortClassifier`

```
  //~ns/classifier/classifier-port.cc
1 int PortClassifier::classify(Packet *p)
2 {
3   hdr_ip* iph = hdr_ip::access(p);
4   return iph->dport();
5 }
```

overridden in the derived classes of class `Classifier`. In Sects. 6.2.2 and 6.2.3, we show two example implementations of function `classify(p)` in classes `PortClassifier` and `DestHashClassifier`, respectively.

6.2.2 *Port Classifiers*

Derived from class `Classifier`, class `PortClassifier` classifies packets based on the destination port. From Lines 3 and 4 in Program 6.4, function `classify(p)` returns the destination port number of the IP header of the incoming packet `*p`.

A port classifier is used as a demultiplexer which bridges a node to receiving transport layer agents. It determines the transport layer port number stored in the header of the received packet `*p`. Suppose the port number is `cl`. Then the

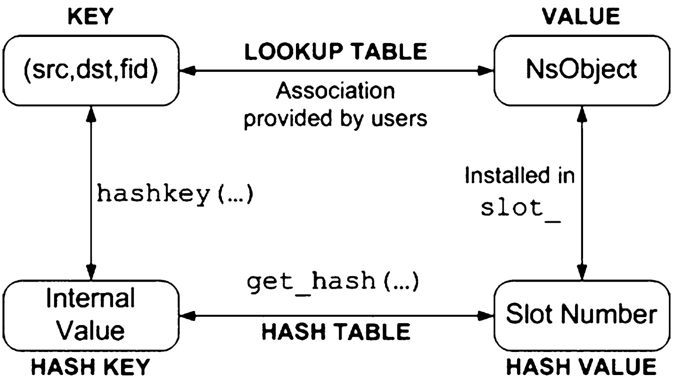


Fig. 6.2 Hash terminology and relevant functions of class `HashClassifier`

packet is forwarded to the NsObject associated with `slot_[cl]`. By installing a pointer to a receiving transport layer agent in `slot_[cl]`, the classifier forwards packets whose destination port is "`cl`" to the associated agent. The details of how a port classifier bridges a Node to a transport layer agent will be discussed later in Sect. 6.5.3.

6.2.3 Hash Classifiers

From Fig. 6.1, another important classifier in a Node is address classifier. In NS2, address classifiers are implemented in so-called hash classifiers.

6.2.3.1 An Overview of Hash Classifiers

Hash table is a data structure which facilitates a key-value lookup process.[1] The lookup process is facilitated by hashing the key into a readily manageable form. The results are stored in a so-called hash-table. The lookup is carried out over the hash table instead of the original table to expedite the lookup process.

Before proceeding further, let us introduce the following hashing terminologies. In this respect, consider, as an example, a hash classifier which classifies packets based on three input parameters: flow ID, source address, and destination address in Fig. 6.2.

[1]Suppose we have a table which associates keys and values. Given a key, the lookup process searches in the table for the matched key, and returns the corresponding value.

- *A key*: Keywords we would like to find (e.g., flow ID, source address, and destination address)
- *A value*: An entry paired with a key (e.g., a pointer to an NsObject)
- *A hash function*: A function which hashes (i.e., transforms) a key into a hash key
- *A hash key*: A transformed key; a lookup process will search over hash keys, rather than the original keys.
- *A hash value*: An entry paired with a hash key (e.g., index of the variable `slot_`)
- *A lookup table*: A table consists of (key,value) pairs.
- *A hash table*: A table consists of (hash-key, hash-value) pairs.
- *A record (or an entry)*: A pair of (key,value)
- *A hash record (or a hash entry)*: A pair of (hash-value, hash-value)[2]

Address classifiers classify packets based on the destination address. In this respect, an address and an NsObject are viewed as a key and a value, respectively. A hash classifier hashes an address into a hash key (internal to NS2), which is associated with a hash value (i.e., the slot number in which the NsObject is installed) by the underlying hash table. When receiving a packet, an address classifier looks up the slot number from the hash table, rather than the original lookup table. This eliminates the need to compare records one by one and greatly expedites the lookup process.

6.2.3.2 C++ Implementation of Class `HashClassifier`

The hash classifiers classify packets based on one or more of the following criteria: flow ID, source address, and destination address. NS2 defines a C++ class `HashClassifier` as a template. All the helper functions are defined here, but the key function `classify(p)`, which defines packet classification criteria, is defined by its derived classes.

Program 6.5 shows the details of a C++ class `HashClassifier` which is mapped to an OTcl class `Classifier/Hash`. Class `HashClassifier` has three main variables. First, variable `default_` (Line 15) contains the default slot for a packet which does not match with any entry in the table. Second, variable `ht_` (Line 16) is the hash table. Finally, variable `keylen_` (Line 17) is the number of components in a key. By default, a key consists of flow ID, source address, and destination address, and the value of `keylen_` is 3.

The key functions of class `HashClassifier` are shown below (see also Fig. 6.2):

[2]Since a record and a hash record have one-to-one relationship, we shall use these two terms interchangeably.

Program 6.5 Declaration of class `HashClassifier`

```
//~ns/classifier/classifier-hash.h
class HashClassifier : public Classifier {
public:
    HashClassifier(int keylen): default_(-1),
      keylen_(keylen);
    ~HashClassifier();
    virtual int classify(Packet *p);
    virtual long lookup(Packet* p) ;
    void set_default(int slot) { default_ = slot; }
protected:
    long lookup(nsaddr_t src, nsaddr_t dst, int fid);
    void reset();
    int set_hash(nsaddr_t src, nsaddr_t dst, int fid, long
      slot);
    long get_hash(nsaddr_t src, nsaddr_t dst, int fid);
    virtual int command(int argc, const char*const* argv);
    virtual const char* hashkey(nsaddr_t, nsaddr_t, int)=0;
    int default_;
    Tcl_HashTable ht_;
    int keylen_;
};
```

`lookup(p)`	Return the slot number which matches with the incoming packet `p`.
`lookup(src,... dst,fid)`	Return the slot number whose corresponding source address, destination address, and flow ID are `src`, `dst`, and `fid`, respectively.
`set_hash(src,... dst,fid,slot)`	Hash the key (`src,dst,fid`) into a hash key, and associates the hash key with the slot number `slot`.
`get_hash(src,... dst,fid)`	Return the slot number which matches with the key (`src,dst,fid`).
`hashkey(src,...) dst,fid)`	Return a hash key for the input key (`src,dst,fid`). This function is pure virtual and should be overridden by child classes of class `HashClassifier`.

Program 6.6 shows the details of functions `lookup(p)` and `get_hash(src, dst,fid)` of class `HashClassifier`. Function `lookup(p)` retrieves a key associated with the packet `*p`. It then asks the function `get_hash(...)` for the corresponding hash value (i.e., slot number).

In Line 6, function `get_hash(...)` invokes function `hashkey(...)` to determine the hash key corresponding to the input key (`src,dst,fid`). Then, function `Tcl_FindHashEntry` (...) locates the hash record in the hash table which matches with the hash key. If the record was found, function `Tcl_GetHash Value(ep)` will retrieve and return the corresponding hash value (i.e., slot

Program 6.6 Functions `lookup` and `get_hash` of class `HashClassifier`

```
//~ns/classifier/classifier-hash.cc
long HashClassifier::lookup(Packet* p) {
    hdr_ip* h = hdr_ip::access(p);
    return get_hash(mshift(h->saddr()),mshift(h->daddr()),
                                                h->flowid());
}

long HashClassifier::get_hash(nsaddr_t src,
                            nsaddr_t dst, int fid) {
    Tcl_HashEntry *ep= Tcl_FindHashEntry(&ht_,
                                 hashkey(src, dst, fid));
    if (ep)
        return (long)Tcl_GetHashValue(ep);
    return -1;
}
```

number) to the caller. Note that the function `hashkey(...)` is declared as pure virtual in class `HashClassifier` and must be overridden by the child classes of class `HashClassifier`.

6.2.3.3 Child Classes of Class `HashClassifier`

Class `HashClassifier` has four major child classes (class names on the left and right are compiled and interpreted classes, respectively):

- `DestHashClassifier` ⇔ `Classifier/Hash/Dest`: classifies packets based on the destination address.
- `SrcDestHashClassifier` ⇔ `Classifier/Hash/SrcDest`: classifies packets based on source and destination addresses.
- `FidHashClassifier` ⇔ `Classifier/Hash/Fid`: classifies packets based on a flow ID.
- `SrcDestFidHashClassifier` ⇔ `Classifier/Hash/SrcDestFid`: classifies packets based on source address, destination address, and flow ID.

6.2.3.4 C++ Class `DestHashClassifier`

As an example, consider class `DestHashClassifier` (Program 6.7), a child class of class `HashClassifier`, which classifies incoming packets by the destination address only. Class `DestHashClassifier` overrides functions `classify(p)`, `do_install(dst,target)`, and `hashkey(...)`, and uses other functions (e.g., `lookup(p)`) of class `HashClassifier` (i.e., its parent class).

Program 6.7 Declaration of class `DestHashClassifier`

```
//~ns/classifier/classifier-hash.h
class DestHashClassifier : public HashClassifier {
public:
    DestHashClassifier() : HashClassifier(TCL_ONE_WORD_KEYS)
      {}
    virtual int command(int argc, const char*const* argv);
    int classify(Packet *p);
    virtual void do_install(char *dst, NsObject *target);
protected:
    const char* hashkey(nsaddr_t, nsaddr_t dst, int) {
        long key = mshift(dst);
        return (const char*) key;
    }
};
```

Program 6.8 Functions `classify` and `do_install` of class `DestHashClassifier`

```
    //~ns/classifier/classifier-hash.cc
1   int DestHashClassifier::classify(Packet * p) {
2       int slot = lookup(p);
3       if (slot >= 0 && slot <=maxslot_)
4           return (slot);
5       else if (default_ >= 0)
6           return (default_);
7       else return (-1);
8   }

9   void DestHashClassifier::do_install(char* dst, NsObject
      *target) {
10      nsaddr_t d = atoi(dst);
11      int slot = getnxt(target);
12      install(slot, target);
13      if (set_hash(0, d, 0, slot) < 0)
14         /* show error */
15  }
```

Program 6.8 shows the implementation of function `classify(p)` of class `DestHashClassifier`. This function obtains a matching slot number "`slot`" by invoking `lookup(p)` (Line 2; See also Fig. 6.3), and returns "`slot`" if it is valid (Line 4). Otherwise, Line 6 will return the variable "`default_`."[3] If neither `slot` nor `default_` is valid, Line 7 will return –1, indicating no matching entry in the hash table.

[3] The variable "`default_`" contains the default slot number. It is defined on Line 15 of Program 6.5.

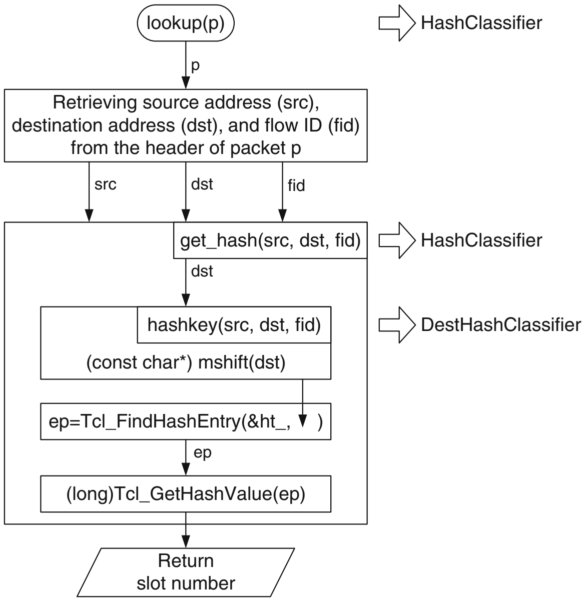


Fig. 6.3 Flowchart of function `lookup(p)` invoked from class `DestHashClassifier`

Function `do_install(dst,target)` installs (Line 12) an `NsObject` pointer `target` in the next available slot, and registers this installation in the hash table (Line 13). Defined in class `Classifier`, function `getnxt(target)` returns the available slot where `target` will be installed (see file ~*ns*/classifier/classifier.cc). Again, the statement `set_hash(0,d,0,slot)` hashes the key with source address "`0`," destination address "d," and flow ID "`0`," and associates the result with the slot number "`slot`." Finally function `hashkey(...)` in Lines 8–11 of Program 6.7 returns the destination address, reformatted by function `mshift(...)`.

Figure 6.3 shows a process when a `DestHashClassifier` object invokes function `lookup(p)`. In this figure, the function name is indicated at the top of each box, while the corresponding class is shown in the right of a block arrow. The process follows what we have discussed earlier. The important point here is that the only function defined in class `DestHashClassifier` is the function `hashkey(...)`. Functions `lookup(p)` and `get_hash(...)` belong to class `HashClassifier`. This is a beauty of OOP, since we only need to override one function for a derived class (e.g., class `DestHashClassifier`), and are able to reuse the rest of the code from the parent class (e.g., class `HashClassifier`).

Later in Sect. 6.5.4, we shall discuss how a destination hash classifier is used to perform routing functionality.

6.2.4 *Creating Your Own Classifiers*

Here are the key steps for defining your own classifiers.

1. *Design*: Define criteria with tuples `(criterion,slot,NsObject)`. If a packet matches with the `criterion`, send the packet to the `NsObject` installed in `slot_[slot]`.
2. *Class construction*: Derive your C++ classifier class, for example, class `YourClassifier` from class `Classifier`. Create a shadow OTcl class.
3. *Internal mechanism*: Override function `classify(p)` according to the design in Step 1.
4. *Configuration*: In the OTcl domain, install the `NsObject` in the slot number `slot` of the `YourClassifier` object. For example, let `$clsfr` be a `YourClassifier` object and and `$obj` be an NsObject in the OTcl domain. You can install `$obj` in the slot number 10 of `$clsfr` by executing the following statement: `$clsfr install 10 $obj`.

6.3 Routing Modules

6.3.1 *An Overview of Routing Modules*

The main functionality of routing modules is to facilitate classifier management. For example, consider Fig. 6.4, where ten address classifiers are connected to each other. It would be rather inconvenient to configure all these ten classifiers using ten OTcl statements.

The configuration process can be facilitated by maintaining a linear topology. Even if the topology of classifiers is as complicated as a full mesh, the topology of routing modules is always linear. We can feed a configuration command to the first routing module in line, and let the routing modules propagate the configuration command toward the end of the line. Since every classifier is connected to one of these routing modules, the configuration command will eventually reach all the classifiers.

Based on the above idea, NS2 uses the following route configuration principles:

1. Assign a routing module for a classifier and connect all related routing modules in a linear topology.
2. Configure classifiers through the head routing module only.
3. Disallow direct classifier configuration.

These principles are implemented in various NS2 components such as routing agents, the route logic, and Nodes. As we shall see later on, class `Node` makes no

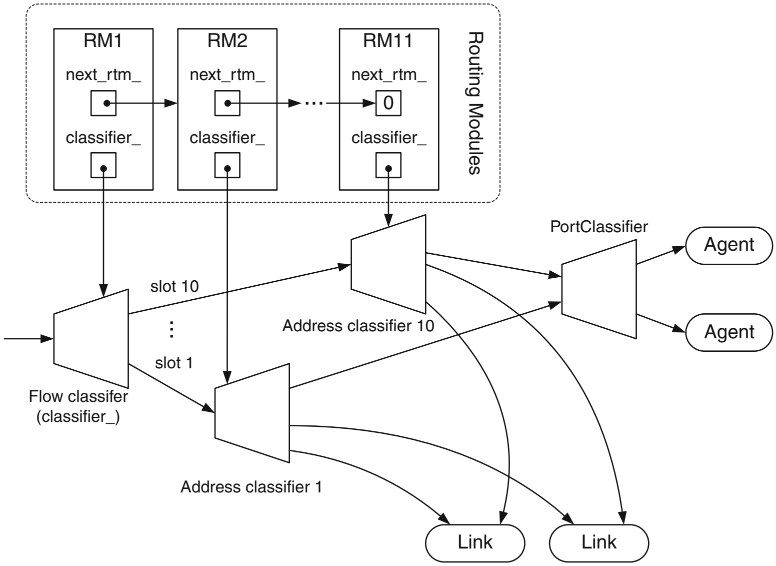


Fig. 6.4 The relationship among routing modules and classifiers in a node

attempt to directly modify its classifiers (e.g., instvars `classifier_` and `dmux_` in Fig. 6.1). Instead, it provides instprocs `add-route{...}` and `attach{...}`, which ask the related routing modules to propagate the configuration commands on its behalf.

6.3.2 *C++ Class* `RoutingModule`

Program 6.9 shows the declaration of class `RoutingModule`, which has three main variables. Variable `classifier_` in Line 15 is a pointer to a `Classifier` object. This variable is bound to an OTcl instvar with the same name (Line 26).

A linear topology of routing modules is created using of a pointer `next_rtm_` (Line 12), which points to another `RoutingModule` object. Finally, variable "`n_`" in Line 14 is a pointer to the associated `Node` object. These three variables are initialized to `NULL` in the constructor (Line 15).

The key functions of class `RoutingModule` include the followings (see Program 6.10):

Program 6.9 Declaration and the constructor of a C++ class `RoutingModule` which is bound to an OTcl class `RtModule`

```
//~ns/routing/rtmodule.h
class RoutingModule : public TclObject {
public:
  RoutingModule();
  inline Node* node() { return n_; }
  virtual int attach(Node *n) { n_ = n; return TCL_OK; }
  virtual int command(int argc, const char*const* argv);
  virtual const char* module_name() const { return NULL; }
  void route_notify(RoutingModule *rtm);
  void unreg_route_notify(RoutingModule *rtm);
  virtual void add_route(char *dst, NsObject *target);
  virtual void delete_route(char *dst, NsObject *nullagent);
  RoutingModule *next_rtm_;
protected:
  Node *n_;
  Classifier *classifier_;
};

static class RoutingModuleClass : public TclClass {
public:
  RoutingModuleClass() : TclClass("RtModule") {}
  TclObject* create(int, const char*const*) {
    return (new RoutingModule);
  }
} class_routing_module;

RoutingModule::RoutingModule() :
        next_rtm_(NULL), n_(NULL), classifier_(NULL) {
  bind("classifier_", (TclObject**)&classifier_);
}
```

`node()`	Return the attached `Node` object `n_`.
`attach(n)`	Store an input `Node` object "`n`" in the variable `n_`.
`module_name()`	Return the name of the routing module.
`route_notify(rtm)`	Add an input `RoutingModule *rtm` to the end of the link list.
`unreg_route_notify(rtm)`	Remove an input `RoutingModule` pointer `*rtm` from the link list.
`add_route(dst,target)`	Inform every classifier associated with the link list to add a routing rule `(dst,target)`.
`delete_route(... dst,nullagent)`	Inform every classifier in the link list to delete a routing rule with destination `dst`.

Program 6.10 Functions `route_notify`, `unreg_route_notify`, `add_route`, and `delete_route` of class `RoutingModule`

```
   //~ns/routing/rtmodule.cc
1  void RoutingModule::route_notify(RoutingModule *rtm) {
2      if (next_rtm_ != NULL)
3          next_rtm_->route_notify(rtm);
4      else
5          next_rtm_ = rtm;
6  }

7  void RoutingModule::unreg_route_notify(RoutingModule *rtm) {
8      if (next_rtm_) {
9          if (next_rtm_ == rtm) {
10             next_rtm_ = next_rtm_->next_rtm_;
11         }
12         else {
13             next_rtm_->unreg_route_notify(rtm);
14         }
15     }
16 }

17 void RoutingModule::add_route(char *dst, NsObject *target)
18 {
19     if (classifier_)
20         classifier_->do_install(dst,target);
21     if (next_rtm_ != NULL)
22         next_rtm_->add_route(dst,target);
23 }

24 void RoutingModule::delete_route(char *dst, NsObject
     *nullagent)
25 {
26     if (classifier_)
27         classifier_->do_install(dst,nullagent);
28     if (next_rtm_)
29         next_rtm_->add_route(dst,nullagent);
30 }
```

Consider Program 6.10. Lines 1–16 show the details of functions `route_notify(rtm)` and `unreg_route_notify(rtm)`. Function `route_notify (rtm)` recursively invokes itself (Line 3) until it reaches the last routing module in the link list, where `next_rtm_` is `NULL`. Then, it attaches the input routing module `*rtm` as the last component of the link list (Line 5). Function `unreg_route_notify(rtm)` recursively searches down the link list (Line 13) until it finds and removes the input routing module pointer "`rtm`" (Lines 9 and 10).

Lines 17–30 show the details of functions `add_route(dst, target)` and `delete_route(dst,nullagent)`. Function `add_route(dst,target)` takes a destination node "`dst`" and a forwarding `NsObject` pointer "`target`" as input arguments. It installs the pointer "`target`" in all the associated classifiers

(Line 20). Again, this routing rule is propagated down the link list (Line 22), until reaching the last element of the link list. Function `delete_route(dst, nullagent)` does the opposite. It recursively installs a null agent "`nullagent`" (i.e., a packet dropping point) as the target for packets destined for a destination node "`dst`" in all the classifiers, essentially removing the routing rule with the destination "`dst`" from all the classifiers.

6.3.3 OTcl Class `RtModule`

In the OTcl domain, the routing module is defined in class `RtModule` bound to the C++ class `RoutingModule`. Class `RtModule` has two instvars: `classifier_` and `next_rtm_`. The instvar `classifier_` is bound to the class variable in the C++ domain with the same name, while the instvar `next_rtm_` is not.[4]

The OTcl class `RtModule` also defines the following instprocs and OTcl commands. For brevity, we show the details of some instprocs in Program 6.11. The details of other instprocs and OTcl command can be found in file ~*ns*/tcl/lib/ns-rtmodule.tcl and ~*ns*/routing/rtmodule.cc, respectively.

6.3.3.1 Initialization Instprocs

`register{node}`	Create two-way connection to the input `node` (Lines 1–13).
`unregister{}`	Remove itself from the associated node and the chain of routing modules (see the file).
`attach-node{node}`	Set the C++ variable `n_` to point to the input `node` (OTcl command; see the file).
`route-notify{module}`	Store the incoming `module` as the last element in the OTcl chain of routing modules (Lines 14–21).
`unreg-route-notify{module}`	Remove the incoming `module` from the OTcl chain of routing modules (see the file).

[4]Caution: When creating a chain of routing modules, use instproc `route_notify{...}`. If you directly configure the instvar `next_rtm_`, the C++ variable `next_rtm_` will not be automatically configured.

Program 6.11 Related Instprocs of OTcl classes `RtModule` and `RtModule/Base`

```
   //~/ns/tcl/lib/ns-rtmodule.tcl
1  RtModule instproc register { node } {
2      $self attach-node $node
3      $node route-notify $self
4      $node port-notify $self
5  }

6  RtModule/Base instproc register { node } {
7      $self next $node
8      $self instvar classifier_
9      set classifier_ [new Classifier/Hash/Dest 32]
10     $classifier_ set mask_ [AddrParams NodeMask 1]
11     $classifier_ set shift_ [AddrParams NodeShift 1]
12     $node install-entry $self $classifier_
13 }

14 RtModule instproc route-notify { module } {
15     $self instvar next_rtm_
16     if {$next_rtm_ == ""} {
17         set next_rtm_ $module
18     } else {
19         $next_rtm_ route-notify $module
20     }
21 }

22 RtModule instproc add-route { dst target } {
23     $self instvar next_rtm_
24     [$self set classifier_] install $dst $target
25     if {$next_rtm_ != ""} {
26         $next_rtm_ add-route $dst $target
27     }
28 }

29 RtModule instproc attach { agent port } {
30     $agent target [[$self node] entry]
31     [[$self node] demux] install $port $agent
32 }
```

6.3.3.2 Instprocs for Configuring Classifiers

`add-route{dst target}`	Propagate a routing rule (`dst`, `target`) to all the attached classifiers (Lines 22–28).
`delete-route{dst nullagent}`	Remove a routing rule whose destination is "`dst`" (see the file).

`attach{agent port}`	Install the "`agent`" in the slot number "`port`" of the demultiplexer "`dmux_`" of the associated Node (Lines 29–32). We shall discuss the details of transport layer agent attachment in Sect. 6.5.3.

6.3.4 *Built-in Routing Modules*

6.3.4.1 The List of Built-in Routing Modules

The C++ class `RoutingModule` and the OTcl class `RtModule` are not actually in use. They are just the base classes from which the following routing module classes derive.

Routing module	C++ class	OTcl class
Routing module	`RoutingModule`	`RtModule`
Base routing module (default)	`BaseRoutingModule`	`RtModule/Base`
Multicast routing module	`McastRoutingModule`	`RtModule/Mcast`
Hierarchical routing module	`HierRoutingModule`	`RtModule/Hier`
Manual routing module	`ManualRoutingModule`	`RtModule/Manual`
Source routing module	`SourceRoutingModule`	`RtModule/Source`
Quick start for TCP/IP routing module (determine initial congestion window)	`QSRoutingModule`	`RtModule/QS`
Virtual classifier routing module	`VCRoutingModule`	`RtModule/VC`
Pragmatic general multicast routing module (reliable multicast)	`PgmRoutingModule`	`RtModule/PGM`
Light-weight multicast services routing module (reliable multicast)	`LmsRoutingModule`	`RtModule/LMS`

Among these classes, the base routing module are the most widely used. As an example, we shall discuss the details of the base routing module.

6.3.4.2 C++ Class `BaseRoutingModule` and OTcl Class `RtModule/Base`

Base routing modules are the default routing modules used for static routing. Again, they are represented in the C++ class `BaseRoutingModule` bound to the OTcl class `RtModule/Base`. From Program 6.12, class `BaseRoutingModule` derives from class `RoutingModule`. It overrides function `module_name()`, by setting its name to be "`Base`" (Line 4). A base routing module classifies packets based on its destination address only. Therefore, the type of the variable `classifier_` is defined as a `DestHashClassifier` pointer (Line 7).

Program 6.12 Declaration of class `BaseRoutingModule` which is bound to the OTcl class `RtModule/Base`

```
   //~ns/routing/rtmodule.h
1  class BaseRoutingModule : public RoutingModule {
2  public:
3      BaseRoutingModule() : RoutingModule() {}
4      virtual const char* module_name() const { return "Base";
           }
5      virtual int command(int argc, const char*const* argv);
6  protected:
7      DestHashClassifier *classifier_;
8  };

   //~ns/routing/rtmodule.cc
9  static class BaseRoutingModuleClass : public TclClass {
10 public:
11     BaseRoutingModuleClass() : TclClass("RtModule/Base") {}
12     TclObject* create(int, const char*const*) {
13         return (new BaseRoutingModule);
14     }
15 } class_base_routing_module;
```

In the OTcl domain, class `RtModule/Base` also overrides instproc `register{node}` of class `RtModule` (Lines 6–13 in Program 6.11). In addition to creating a two-way connection to the input `Node` object `node` (performed by its base class), the base routing module creates (Line 9) and installs (Line 12) a destination hash classifier inside the `node`. We shall discuss the details of the instproc `install-entry{...}` later in Sect. 6.5.2.

6.4 Route Logic

The main responsibility of a route logic object is to compute the routing table. Route logic is implemented in a C++ class `RouteLogic` which is bound to the OTcl class with the same name (see Program 6.13).

6.4.1 C++ Implementation

The C++ Class `RouteLogic` has two key variables: "`adj_`" (Line 14), which is the adjacency matrix used to compute the routing table, and "`route_`" (Line 15), which is the routing table. It has the following three main functions:

Program 6.13 Declaration of class `RouteLogic` and the corresponding OTcl mapping class

```
//~/ns/routing/route.h
class RouteLogic : public TclObject {
public:
    RouteLogic();
    ~RouteLogic();
    int command(int argc, const char*const* argv);
    virtual int lookup_flat(int sid, int did);
protected:
    void reset(int src, int dst);
    void reset_all();
    void compute_routes();
    void insert(int src, int dst, double cost);
    void insert(int src, int dst, double cost, void* entry);
    adj_entry *adj_;
    route_entry *route_;
};

//~/ns/routing/route.cc
class RouteLogicClass : public TclClass {
public:
    RouteLogicClass() : TclClass("RouteLogic") {}
    TclObject* create(int, const char*const*) {
        return (new RouteLogic());
    }
} routelogic_class;
```

`insert(src,... dst,cost)`	Inform the route logic of the `cost` to go from the Node `src` to the Node `dst`
`compute_routes()`	Compute the optimal routes for all source-destination pairs and store the computed routes in the variable `route_`.
`lookup_flat(... sid,did)`	Search within the variable `route_` for an entry with matching source ID (`sid`) and destination ID (`did`), and returns the index of the forwarding object (e.g., connecting link).

6.4.2 OTcl Implementation

In the interpreted hierarchy, the OTcl class `RouteLogic` has one key instvar `rtprotos_`. The instvar `rtprotos_` is an associative array whose index is the name of the routing protocol and value is the routing agent object. Again, we are dealing with static routing. Therefore, the instvar `rtprotos_` does not exist.

Program 6.14 Instprocs `configure` and `lookup` of class `RouteLogic`

```
    //~/ns/tcl/lib/ns-route.tcl
1  RouteLogic instproc configure {} {
2      $self instvar rtprotos_
3      if [info exists rtprotos_] {
4          foreach proto [array names rtprotos_] {
5              eval Agent/rtProto/$proto init-all $rtprotos_
                 ($proto)
6          }
7      } else {
8          Agent/rtProto/Static init-all
9      }
10 }

11 RouteLogic instproc lookup { nodeid destid } {
12     if { $nodeid == $destid } {
13         return $nodeid
14     }
15     set ns [Simulator instance]
16     set node [$ns get-node-by-id $nodeid]
17     $self cmd lookup $nodeid $destid
18 }
```

The OTcl class `RouteLogic` also has two major instprocs as shown in Program 6.14).

`configure{}`	Initialize all the routing protocols (Lines 1–10).
`lookup{sid,did}`	Return the forwarding object for packets going from Node `sid` to Node `did` (Lines 11–18).

6.5 Node Construction and Configuration

So far in this chapter, we have discussed major components of a Node – classifiers, routing modules, and route logic. We now present how NS2 creates and puts together these main components.

In the following, we first show the key instvars of the OTcl class `Node` and their relationships in Sect. 6.5.1. Then, we show an approach to put classifiers into a `Node` object in Sect. 6.5.2. Sections 6.5.3 and 6.5.4 show how a `Node` is bridged to the transport (i.e., upper) layer and to the routing (i.e., lower) layer, respectively. Finally, Sect. 6.5.5 discusses the key steps to create and configure a `Node` object.

6.5.1 *Key Variables of the OTcl Class* `Node` *and Their Relationship*

The list of major instvars of the OTcl class `Node` is given below.

`id_`	Node ID
`agents_`	List of attached transport layer agents
`nn_`	Total number of Nodes
`neighbor_`	List of neighboring nodes
`nodetype_`	Node type (e.g., regular node or mobile node)
`ns_`	The Simulator
`classifier_`	Address classifier, which is the default node entry
`dmux_`	The demultiplexer or port classifier
`module_list_`	List of enabled routing modules
`reg_module_`	List of registered routing modules
`rtnotif_`	The head of the chain of routing modules which will be notified of route updates
`ptnotif_`	List of routing modules which will be notified of port attachment/detachment
`hook_assoc_`	Sequence of the chain of classifiers
`mod_assoc_`	Association of classifiers and routing modules

6.5.1.1 Routing-Related Instvars

The following five instvars of an OTcl Node plays a major role in packet routing: `module_list_`, `reg_module_`, `rtnotif_`, `ptnotif_`, and `mod_assoc_`. The instvar `module_list_` is a `list` of strings, each of which represents the name of enabled routing module. The instvar `reg_module_` is an associative array whose index and value are the name of the routing module and the routing module instance.

The instvars `rtnotif_` and `ptnotif_` contain the objects which should be notified of a route change and an agent attachment/detachment, respectively. While `rtnotif_` is the head of the link list of the routing modules, `ptnotif_` is simply an OTcl `list` whose elements are the routing modules. Finally, instvar `mod_assoc_` is an associative array whose indexes and values are classifiers and the associated routing modules, respectively.

Figure 6.5 shows an example of routing-related variable setting both in C++ and OTcl domain. Here, we assume that there are two classifiers. The first, `switch_`, classifies the geometry (i.e., circle/triangle/square). The second classifies, `classifier_`, color (i.e., black/white).

The above two classifiers are controlled by routing modules `RtModule/Mcast` and `RtModule/Base`, respectively. Since there are two routing modules, the instvar `reg_modules_` has two entries.

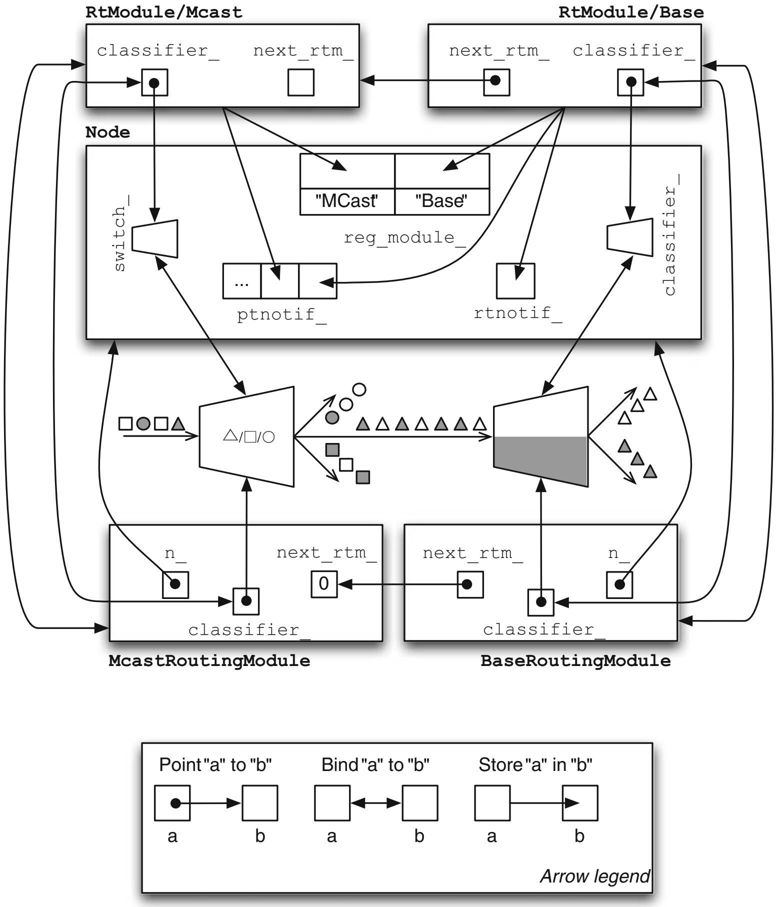


Fig. 6.5 An example of node configuration with two classifiers

Suppose further that both the classifiers need to be informed of routing change and agent attachment/detachment. We need to put both the associated routing modules in the `list` instvar `ptnotif_`. On the other hand, we only set one routing module (i.e., `RtModule/Base` associated with `classifier_` in this case) as the instvar `rtnotif_`. The route configuration command can be propagated to `RtModule/Mcast` via the variable `next_rtm_` of the head (i.e., Base) routing module.

6.5.1.2 Classifier-Related Instvars

Class `Node` has three instvars related to classifiers: `classifier_`, `hook_assoc_`, and `mod_assoc_`. Instvar `classifier_` is the default Node entry as well as the head of the chain of classifiers. Instvar `hook_assoc_` is an associative

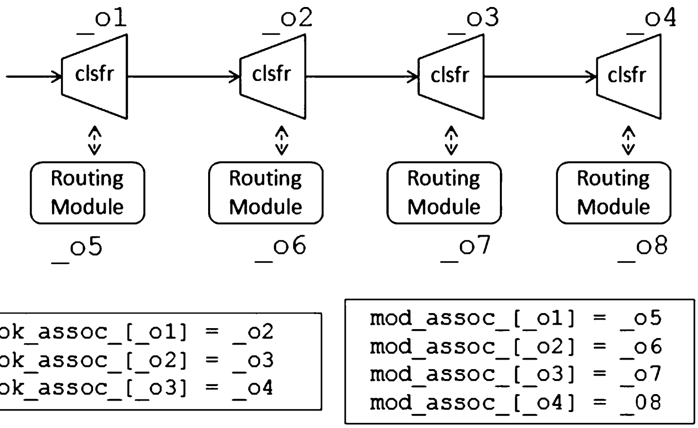


Fig. 6.6 An example of values stored in variables hook_assoc_ and mod_assoc_

array whose index is a classifier and value is the downstream classifier in the chain. The index and value of the associative array mod_assoc_ are classifiers and the associated routing modules, respectively.

Consider Fig. 6.6 for example. Here, we install classifiers _o1, _o2, _o3, and _o4 into a Node, and associate them with routing modules _o5, _o6, _o7, and _o8, respectively. Then, the instvar classifier_ would be _o1. The indexes and values of hook_assoc_ and mod_assoc_ would be as shown in the figure.

6.5.2 *Installing Classifiers in a Node*

Class Node provides three instprocs to configure classifiers. First, as shown in Program 6.15, the instproc insert-entry{module clsfr hook} takes three input arguments: a routing module "module", a classifier "clsfr", and an optional argument "hook." It installs the current head classifier in the slot number "hook" of the input classifier clsfr (Line 8), and replaces the head classifier with the input classifier clsfr (Line 12). The instvars hook_assoc_ and mod_assoc_ are updated in Lines 4 and 11, respectively.

The input argument "hook" can have one of the three following values:

- A number: The input "clsfr" will be configured as explained above.
- A string "target": The existing head classifier will be configured as a target of the input NsObject clsfr.[5]
- Null: The input "clsfr" will not be configured. We might have to configure it later.

[5]Note that the input clsfr needs not be a classifier.

Program 6.15 Instprocs `insert-entry` and `install-demux` of class `Node`

```
    //~ns/tcl/lib/ns-node.tcl
 1  Node instproc insert-entry { module clsfr {hook ""} } {
 2      $self instvar classifier_ mod_assoc_ hook_assoc_
 3      if { $hook != "" } {
 4          set hook_assoc_($clsfr) $classifier_
 5          if { $hook == "target" } {
 6              $clsfr target $classifier_
 7          } elseif { $hook != "" } {
 8              $clsfr install $hook $classifier_
 9          }
10      }
11      set mod_assoc_($clsfr) $module
12      set classifier_ $clsfr
13  }

14  Node instproc install-demux {demux {port ""} } {
15      $self instvar dmux_ address_
16      if { $dmux_ != "" } {
17           $self delete-route $dmux_
18           if { $port != "" } {
19               $demux install $port $dmux_
20           }
21       }
22       set dmux_ $demux
23      $self add-route $address_ $dmux_
24  }
```

The second classifier configuration is the instproc `install-entry{module clsfr hook}`, which is very similar to instproc `insert-entry{...}`. The only difference is that it also destroys the existing head classifier, if any. The details of the instproc `install-entry{...}` can be found in the file ~*ns*/tcl/lib/ns-node.tcl.

The last classifier configuration is the instproc `install-demux{demux port}`, whose details are shown in Lines 14–24 of Program 6.15. This instproc takes two input arguments: `demux` (mandatory) and `port` (optional). It replaces the existing demultiplexer[6] "`dmux_`" with the input demultiplexer `demux` (Line 22). If "`port`" exists, the current demultiplexer "`dmux_`" will be installed in the slot number "`port`" of the input demultiplexer "`demux`" (Lines 18–20).

6.5.3 Bridging a Node to a Transport Layer Protocol

To attach an agent to a Node, we use instproc `attach-agent{node agent}` of class `Simulator`, where "`node`" and "`agent`" are `Node` and `Agent` objects,

[6] A demultiplexer classifies packets based on the port number specified in the packet header (see Sect. 6.2.2 for more details).

Program 6.16 Agent attachment instprocs

```
    //~ns/tcl/lib/ns-lib.tcl
 1  Simulator instproc attach-agent { node agent } {
 2      $node attach $agent
 3  }

    //~ns/tcl/lib/ns-node.tcl
 4  Node instproc attach { agent { port "" } } {
 5      $self instvar agents_ address_ dmux_
 6      lappend agents_ $agent
 7      $agent set node_ $self
 8      $agent set agent_addr_ [AddrParams addr2id $address_]
 9      if { $dmux_ == "" } {
10          set dmux_ [new Classifier/Port]
11          $self add-route $address_ $dmux_
12      }
13      if { $port == "" } {
14          set port [$dmux_ alloc-port [[Simulator
                                        instance] nullagent]]
15      }
16      $agent set agent_port_ $port
17      $self add-target $agent $port
18 }

19 Node instproc add-target { agent port } {
20      $self instvar ptnotif_
21      foreach m [$self set ptnotif_] {
22          $m attach $agent $port
23      }
24 }
```

respectively. Program 6.16 shows the instprocs related to an agent attachment process. The process proceeds as follows:

1. `Simulator::attach-agent{node agent}`: Invoke "`$node attach $agent`" (Line 2).
2. `Node::attach{agent port}`: Update instvar "`agent`" (Lines 6–8 and Line 16), create "`dmux_`" if necessary (Lines 9–15), and invoke "`$self add-target $agent $port`" (Line 17).
3. `Node::add-target{agent port}`: From Sect. 6.5.1, routing modules related to port attachment are stored in the list instvar `ptnotif_`. Therefore, Lines 22 and 23 execute instproc `attach{agent port}` of all the routing modules stored in the instvars `ptnotif_`.[7]
4. `RtModule::attach{agent port}`: Consider Lines 29–32 in Program 6.11. As a sending agent, the input "`agent`" is set as an upstream object of the node entry (Line 30). As a receiving agent, it is installed in the slot number "`port`" of demultiplexer "`dmux_`" (Line 31).

[7]Again, Nodes do not directly configure port classifiers. It asks routing modules stored in `ptnotif_` to do so on its behalf.

Program 6.17 Instprocs `add-route` of class `Node`

```
   //~ns/tcl/lib/ns-node.tcl
1  Node instproc add-route { dst target } {
2      $self instvar rtnotif_
3      if {$rtnotif_ != ""} {
4          $rtnotif_ add-route $dst $target
5      }
6      $self incr-rtgtable-size
7  }
```

Note that although an agent can be *either* a sending agent or a receiving agent, this instproc assigns both roles to every agent. This does not cause any problem at runtime due to the following reasons. A sending agent is attached to a source node, and always transmits packets destined to a destination node. It takes no action when receiving a packet from a demultiplexer. A receiving agent, on the other hand, does not generate a packet. Therefore, it can never send a packet to the node entry.

6.5.4 Adding/Deleting a Routing Rule

Class `Node` provides an instproc `add-route{dst target}` to add a routing rule (`dst,target`) to the routing table. In Program 6.17, instproc `add-route {dst target}` of class `Node` invokes the instproc `add-route{...}` of the routing module `rtnotif_` which is of class `RtModule`[8] (Line 4). From Lines 22–29 of Program 6.11, the instproc `add-route{...}` of class `RtModule` installs the routing rule (`dst,target`) in the `classifier_` of all the related routing module.

The mechanism for deleting a routing rule is similar to that for adding a routing rule, and is omitted for brevity. The readers may find the details of route entry deletion in the instproc `delete-route{dst nullagent}` of classes `Node` and `RtModule` (see file ~*ns*/tcl/lib/ns-node.tcl and file ~*ns*/tcl/lib/ns-rtmodule.tcl).

6.5.5 Node Construction and Configuration

There are two key steps to put together a Node (e.g., as shown in Figs. 6.1 and 6.5). We now discuss the details of node construction and configuration in sequence.

[8]Again, class `Node` makes no attempt to directly modify the classifiers. It asks the routing modules in the chain, whose head is `rtnotif_`, to do so on its behalf.

Program 6.18 Default value of instvar node_factory_ and instproc node of class Simulator

```
   //~ns/tcl/lib/ns-default.tcl
1  Simulator set node_factory_ Node

   //~ns/tcl/lib/ns-lib.tcl
2  Simulator instproc node args {
3      $self instvar Node_ routingAgent_
4      set node [eval new [Simulator set node_factory_] $args]
5      set Node_([$node id]) $node
6      $self add-node $node [$node id]
7      $node nodeid [$node id]
8      $node set ns_ $self
9      return $node
10 }
```

6.5.5.1 Node Construction

A Node object is created using an OTcl statement "$ns node," where $ns is the Simulator instance. The Instproc "node" of class Simulator uses instproc "new{...}" to create a Node object (Line 4 where node_factory_ is set to Node in Line 1 of Program 6.18). It also updates instvars of the Simulator so that they can later be used by other simulation objects throughout the simulation.

The construction of an OTcl Node object (using new{...}) consists of seven main steps (see also Program 6.19).

Step 1: Constructor of the OTcl Class Node

Instproc init{...} sets up instvars of class Node, and invokes instproc mk-default-classifier{} of the Node object (Line 22 in Program 6.19).

Step 2: Instproc **mk-default-classifier**{}

The instproc mk-default-classifier{} creates (using new{...}) and registers (using register-module{mod}) routing modules whose names are stored in the instvar module_list_ (Lines 27–29 in Program 6.19). By default, only "Base" routing module is stored in the instvar module_list_ (Line 1 in Program 6.19).

To enable/disable other routing modules, the following two instprocs of class RtModule must be invoked before the execution of "$ns node":

```
enable-module{<name>}
disable-module{<name>}
```

where <name> is the name of the routing module, which is to be enabled/disabled.

Program 6.19 Instprocs related to the Node Construction Process

```
   //~/ns/tcl/lib/ns-node.tcl
1  Node set module_list_ { Base }

2  Node instproc init args {
3      eval $self next $args
4      $self instvar id_ agents_ dmux_ neighbor_ rtsize_
5         address_ \ nodetype_ multiPath_ ns_ rtnotif_ ptnotif_
6      set ns_ [Simulator instance]
7      set id_ [Node getid]
8      $self nodeid $id_   ;# Propagate id_ into c++ space
9      if {[llength $args] != 0} {
10         set address_ [lindex $args 0]
11     } else {
12         set address_ $id_
13     }
14     $self cmd addr $address_; # Propagate address_ into
          C++ space
15     set neighbor_ ""
16     set agents_ ""
17     set dmux_ ""
18     set rtsize_ 0
19     set ptnotif_ {}
20     set rtnotif_ {}
21     set nodetype_ [$ns_ get-nodetype]
22     $self mk-default-classifier
23     set multiPath_ [$class set multiPath_]
24 }

25 Node instproc mk-default-classifier {} {
26     Node instvar module_list_
27     foreach modname [Node set module_list_] {
28         $self register-module [new RtModule/$modname]
29     }
30 }

31 Node instproc register-module { mod } {
32     $self instvar reg_module_
33     $mod register $self
34     set reg_module_([$mod module-name]) $mod
35 }
```

Step 3: Instproc `register-module{mod}` of Class `Node`

This instproc invokes the instproc `register{node}` of the input routing module "`mod`" and updates the instvar "`reg_module_`" (see Lines 31–35).

Step 4: Instproc `register{node}` of Class `RtModule/Base`

This instproc first invokes instproc `register{node}` of its parent class (by the statement `$self next $node` in Line 7 of Program 6.11). Then, Lines 9–12 create (using `new{...}`) and configure (using `install-entry{...}`) the head classifier (i.e., `classifier_`) of the Node.

Step 5: Instproc `register{node}` of Class `RtModule`

From Program 6.11, this instproc attaches input `Node` object "`node`"to the routing module (Line 2). It also invokes instproc `route-notify{module}` (Line 3) and `port-notify{module}` (Line 4) of the associated Node to include the routing module into the route notification list `rtnotif_` and port notification list `ptnotif_` of the associated Node.

Step 6: Instproc `route-notify{module}` of Class `Node`

As shown in Program 6.20, the instproc `route-notify{module}` (Lines 1–9) stores the input routing module "`module`" as the last element of the link list of routing modules. It also invokes the OTcl command `route-notify` of the input routing module (Line 8). The OTcl command `route-notify` invokes the C++ function `route_notify(rtm)` associated with the attached Node (see Lines 10–16) to store the routing module as the last routing module in the link list (see Lines 17–22).

Step 7: Instproc `port-notify{module}`of Class `Node`

As shown in Lines 23–26 of Program 6.20, the instproc `port-notify{module}` appends the input argument "`module`" to the end of the list instvar `ptnotif_`.

6.5.5.2 Agent and Route Configuration

We have discussed how NS2 creates and puts main components within a `Node` object. The final step is to instruct these components what to do when receiving a packet.

From Fig. 6.1, a `Node` object contains two key components: an address classifier `classifier_`, and a port classifier/demultiplexer, `dmux_`. Class `Simulator` provides two instprocs to configure these two classifiers.[9]

[9]Since the `Simulator` object is accessible to the Tcl simulation script, users generally use these two instprocs to configure Nodes.

Program 6.20 Instprocs and functions which are related to instprocs `route-notify` and `port-notify` of the OTcl class `Node`

```
//~/ns/tcl/lib/ns-node.tcl
Node instproc route-notify { module } {
    $self instvar rtnotif_
    if {$rtnotif_ == ""} {
        set rtnotif_ $module
    } else {
        $rtnotif_ route-notify $module
    }
    $module cmd route-notify $self
}

//~ns/routing/rtmodule.cc
int BaseRoutingModule::command(int argc, const char*const*
  argv) {
    Tcl& tcl = Tcl::instance();
    if (argc == 3) {
        if (strcmp(argv[1] , "route-notify") == 0) {
            n_->route_notify(this);
        }
}

//~ns/common/node.cc
void Node::route_notify(RoutingModule *rtm) {
    if (rtnotif_ == NULL)
        rtnotif_ = rtm;
    else
        rtnotif_->route_notify(rtm);
}

//~/ns/tcl/lib/ns-node.tcl
Node instproc port-notify { module } {
    $self instvar ptnotif_
    lappend ptnotif_ $module
}
```

- **Instproc** `attach-agent{...}`: Connect a Node to a transport layer agent (see the details in Sect. 6.5.3).
- **Instproc** `run{}`: Create, compute, and install routing tables in classifiers of all the Nodes. Again, by default, NS2 uses the Dijkstra's algorithm to compute routing tables for all pairs of Nodes. The key steps in the instproc `run{}` are shown below:

Step 1: Instproc `run{}` of Class `Simulator`

Shown in Line 2 of Program 4.12, instproc `run{}` of class `Simulator` executes the instproc `configure{}` of the `RouteLogic` object.

Program 6.21 Instprocs related to the route configuration process

```
   //~ns/tcl/rtglib/route-proto.tcl
1  Agent/rtProto/Static proc init-all args {
2      [Simulator instance] compute-routes
3  }

   //~ns/tcl/lib/ns-route.tcl
4  Simulator instproc compute-routes {} {
5          $self compute-flat-routes
6  }

7  Simulator instproc compute-flat-routes {} {
8      $self instvar Node_ link_
9      set r [$self get-routelogic]
10     $self cmd get-routelogic $r
11     foreach ln [array names link_] {
12         set L [split $ln :]
13         set srcID [lindex $L 0]
14         set dstID [lindex $L 1]
15         if { [$link_($ln) up?] == "up" } {
16             $r insert $srcID $dstID [$link_($ln) cost?]
17         } else {
18             $r reset $srcID $dstID
19         }
20     }
21     $r compute
22     set n [Node set nn_]
23     $self populate-flat-classifiers $n
24 }
```

Step 2: Instproc `configure{}` of Class `RouteLogic`

Defined in Lines 1–10 of Program 6.14, the instproc `configure{}` of class `Route Logic` configures the routing table for all the Nodes by invoking instproc `init-all{}` of class `Agent/rtProto/Static`.

Step 3: Instproc `init-all{}` of Class `Agent/rtProto/Static`

Defined in Lines 1–3 of Program 6.21, the instproc `init-all{}` of class `Agent /rtProto/Static` invokes the instproc `compute-routes{}` of the Simulator.

Step 4: Instproc `compute-routes{}` of Class `Simulator`

By default, this instproc invokes the instproc `compute-flat-routes{}` to compute and setup the routing table (see Lines 4–6 in Program 6.21).

Program 6.22 An OTcl command `populate-flat-classifiers`, a function `populate_flat_classifiers` of class `Simulator`, and a function `add_route` of class `Node`

```
//~ns/common/simulator.cc
int Simulator::command(int argc, const char*const* argv) {
      ...
   if (strcmp(argv[1], "populate-flat-classifiers") == 0) {
      nn_ = atoi(argv[2]);
      populate_flat_classifiers();
      return TCL_OK;
   }
   ...
}

void Simulator::populate_flat_classifiers() {
   ...
   for (int i=0; i<nn_; i++) {
      for (int j=0; j<nn_; j++) {
         if (i != j) {
            int nh = -1;
            nh = rtobject_->lookup_flat(i, j);
            if (nh >= 0) {
               NsObject *l_head=get_link_head(nodelist_[i],
                 nh);
               sprintf(tmp, "%d", j);
               nodelist_[i]->add_route(tmp, l_head);
            }
         }
      }
   }
}

//~ns/common/node.cc
void Node::add_route(char *dst, NsObject *target) {
    if (rtnotif_)
        rtnotif_->add_route(dst, target);
}
```

Step 5: Instproc `compute-flat-routes{}` of Class `Simulator`

Defined in Lines 7–24 of Program 6.21, this instproc computes and installs the routing table in all related address classifiers.

- Retrieve and store the route logic in a local variable `$r` (Lines 9–10)
- Collect and insert topology information into the retrieved route logic `$r` (Lines 11–20)
- Compute the optimal route (Line 21)
- Add routing rules into all related classifiers (Lines 22 and 23)

Program 6.22 shows the details of how the computed routing rules are propagated to all the nodes. Lines 1–9 show the details of OTcl command `populate-flat-classifiers{n}`. This OTcl command stores the input number of nodes "n" in the variable `nn_` (Line 4), and invokes the function `populate_flat_classifiers()` (Line 5) to install the computed routing rules in all the classifiers.

Function `populate_flat_classifiers()` adds the routing rules for all pairs `(i,j)` of `nn_` nodes (Lines 10–25). For each pair, Line 16 retrieves the next hop (i.e., forwarding) referencing point "nh" of a forwarding object for a packet traveling from Node "i" to Node "j." Line 18 retrieves the link entry point "l_head" corresponding to the variable "nh." Lines 19 and 20 add a new routing rule for the node `i` (i.e., `nodelist_[i]`). The rule specifies the link entry "l_head" as a forwarding target for packet destined for a destination node `j`. The rule is added to the Node "i" via its function `add_route(dst,target)`.

Function `add_route(dst,target)` simply invokes function `add_route (dst,target)` of the associated `RoutingModule` object `rtnotif_` (Lines 26–29). Defined in Program 6.10, function `add_route(dst,target)` of class `RoutingModule` recursively installs the input routing rule `(dst,target)` down the link list of routing modules.

6.6 Chapter Summary

A Node is a basic component which acts as a router and a computer host. Its main responsibilities are to forward packets according to a routing table and to bridge the high-layer protocols to a low-level network. A Node consists of three key components: classifiers, routing modules, and the route logic. A classifier is a multi-target packet forwarder. It forwards packets in the same category to the same forwarding NsObject. In a Node, an address classifier and a port classifier act as a router and a bridge to the transport layer, respectively.

Routing modules are responsible for managing classifiers. By convention, all the configuration commands must go though routing modules only. Finally, the route logic collects network topology, computes the optimal routing rules, and install the resulting rules in all the Nodes.

During the Network Configuration Phase, a Node is created by executing `$ns node` where `$ns` is the `Simulator` object. Here, address classifiers and routing modules are installed in the Node. The instruction of what to do when receiving a packet is provided later when the following two OTcl statements are executed. First, the transport layer connections are created using the instproc `attach-agent{...}` of class `Simulator`. Second, the instproc `run{}` of class `Simulator` computes the optimal routes for all pairs of nodes and installs the computed routing tables in relevant classifiers.

6.7 Exercises

1. What is a Classifier? What are the similarities/differences between a Connector and a Classifier?
2. Explain and give example for the following terminologies:

a. Routing mechanism,	b. Routing rules,	c. Routing table,
d. Routing algorithm,	e. Routing protocol,	f. Routing agent,
g. Route logic,	h. Routers,	i. Routing module.

3. What are routing modules? Explain their roles and necessities.
4. Explain how classifiers work.
   a. What are slots? What does "installing an NsObject in a slot" mean?
   b. How does a packet enter a classifier? Explain the packet flow mechanism since a packet enters a classifier until it leaves the classifier. Give an example and draw a diagram to support your answer.
5. What is a hash function? Explain your answer and show few applications which use hash functions.
6. What are the components in NS2 which
   a. Find the optimal route,
   b. Propagate topology and routing information,
   c. Forward packet based on the routing information.
7. Consider a packet size classifier which classifies packet into small (smaller than 40 bytes), medium (not smaller than 40 bytes but smaller than 1,000 bytes), and large (not smaller than 1,000 bytes) packets.
   a. Create a packet size classifier.
   b. Configure the classifier such that small, medium, and large packets are sent to the NsObject whose addresses are stored in variables "`sm`," "`md`," and "`lg`," respectively.
   c. Explain the packet flow mechanism and run an NS2 program to test your answer.

   Hint: Packet size can be obtained by C++ statements "`hdr_cmn* ch = hdr_cmn::access(p); ch->size_`"; see Chap. 8.
8. Draw a Node diagram. What OTcl commands do you use to create a Node like in the diagram. Explain, step-by-step, how NS2 creates the Node.
9. What is the default routing algorithm in NS2? How does NS2 setup the routing in a Node?
10. What is a node entry? Show an OTcl statement to retrieve a reference to the node entry.

11. How does NS2 inform a Node of

    a. New route,
    b. New agent which shall be attached to a certain port?

    Show the step-by-step processes in NS2 via examples.