Chapter 5
Network Objects: Creation, Configuration, and Packet Forwarding

NS2 is a simulation tool designed specifically for communication networks. The main functionalities of NS2 are to set up a network of connecting nodes and to pass packets from one node (which is a network object) to another.

A network object is one of the main NS2 components, which is responsible for packet forwarding. NS2 implements network objects using the polymorphism concept in object-oriented programming (OOP). Polymorphism allows network objects to take different actions ways under different contexts. For example, a `Connector` object immediately passes the received packet to the next network object, while a `Queue`[1] object enqueues the received packets and forwards only the head of the line packet.

This chapter first introduces the NS2 components by showing four major classes of NS2 components, namely, network objects, packet-related objects, simulation-related objects, and helper objects in Sect. 5.1. A part of the C++ class hierarchy, which is related to network objects, is also shown here. Section 5.2 presents class `NsObject` which acts as a template for all network objects. An example of network objects as well as packet forwarding mechanism are illustrated through class `Connector` in Sect. 5.3. Finally, the chapter summary is given in Sect. 5.4. Note that the readers who are not familiar with OOP are recommended to go through a review of the OOP polymorphism concept in Appendix B before proceeding further.

[1]Class `Queue` is a child class of class `Connector`.

T. Issariyakul and E. Hossain, *Introduction to Network Simulator NS2*,
DOI 10.1007/978-1-4614-1406-3_5, © Springer Science+Business Media, LLC 2012

5.1 Overview of NS2 Components

5.1.1 Functionality-Based Classification of NS2 Modules

Based on the functionality, NS2 modules (or objects) can be classified into four following types:

- *Network objects* are responsible for sending, receiving, creating, and destroying packet-related objects. Since these objects are those derived from class `NsObject`, they will be referred to hereafter as NsObjects.
- *Packet-related objects* are various types of packets which are passed around a network.
- *Simulation-related objects* control simulation timing and supervise the entire simulation. As discussed in Chap. 4, examples of simulation-related objects are events, handlers, the Scheduler, and the Simulator.
- *Helper objects* do not explicitly participate in packet forwarding. However, they implicitly help to complete the simulation. For example, a routing module calculates routes from a source to a destination, while network address identifies each of the network objects.

In this chapter, we focus only on network objects. Note that, the simulation-related objects were discussed in Chap. 4. The packet-related objects will be discussed in Chap. 8. The main helper objects will be discussed in Chap. 15.

5.1.2 C++ Class Hierarchy

This section gives an overview of C++ class hierarchies. The entire hierarchy consists of over 100 C++ classes and `struct` data types. Here, we only show a part of the hierarchy (in Fig. 5.1). The readers are referred to [18] for the complete class hierarchy.

As discussed in Chap. 3, all classes deriving from class `TclObject` form the compiled hierarchy. Classes in this hierarchy can be accessed from the OTcl domain. For example, they can be created by the global OTcl procedure "`new{...}`." Classes derived directly from class `TclObject` include network classes (e.g., `NsObject`), packet-related classes (e.g., `PacketQueue`), simulation- related classes (e.g., `Scheduler`), and helper classes (e.g., `Routing- Module`). Again, classes that do not need OTcl counterparts (e.g., classes derived from class `Handler`) form their own standalone hierarchies. These hierarchies are not a part of the compiled hierarchy nor the interpreted hierarchy.

As discussed in Chap. 4, class `Handler` specifies an action associated with an event. Again, class `Handler` contains a pure virtual function `handle(e)` (see Program 4.1). Therefore, its derived classes are responsible for providing

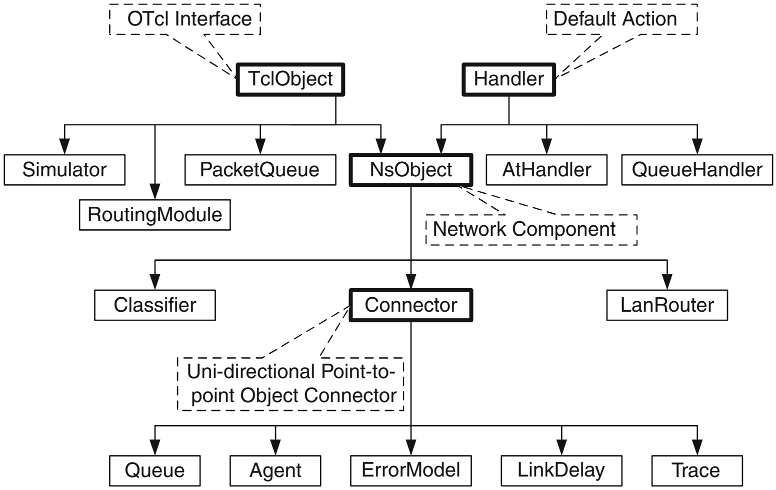

Fig. 5.1 A part of NS2 C++ class hierarchy (this chapter emphasizes on classes in *boxes* with *thick solid lines*)

implementation of the function `handle(e)`. For example, the function `handle(e)` of class `NsObject` tells the NsObject to receive an incoming packet (Program 4.2), while that of class `QueueHandler` invokes function `resume()` of the associated `Queue` object (Lines 1–4 in Program 5.1; also see Sect. 7.3.2).

Program 5.1 Function `handle(e)` of class `QueueHandler`

```
  //~/ns/queue/queue.cc
1 void QueueHandler::handle(Event*)
2 {
3     queue_.resume();
4 }
```

There are three main classes deriving from class `NsObject`: `Connector`, `Classifier`, and `LanRouter`. Connecting two NsObjects, a `Connector` object immediately forwards a received packet to the connecting NsObject (see Sect. 5.3). Connecting an NsObject to several NsObjects, a `Classifier` object classifies packets based on packet header (e.g., destination address, flow ID) and forwards the packets with the same classification to the same connecting NsObject (see Sect. 6.2). Class `LanRouter` also has multiple connecting NsObjects. However, it forwards every received packet to all connecting NsObjects.

5.2 NsObjects: A Network Object Template

5.2.1 Class `NsObject`

Representing NsObjects, class `NsObject` is the base class for all network objects in NS2 (see the declaration in Program 5.2). Again, the main responsibility of an NsObject is to forward packets. Therefore, class `NsObject` defines a pure virtual function `recv(p,h)` (see Line 5 in Program 5.2) as a uniform packet reception interface to force all its derived classes to implement this function.

Program 5.2 Declaration of class `NsObject`

```
   //~/ns/common/object.h
1  class NsObject : public TclObject, public Handler {
2  public:
3      NsObject();
4      virtual ~NsObject();
5      virtual void recv(Packet*, Handler* callback = 0) = 0;
6      virtual int command(int argc, const char*const* argv);
7  protected:
8      virtual void reset();
9      void handle(Event*);
10     int debug_;
11 };
```

Derived directly from class `TclObject` and `Handler` (see Program 5.2), class `NsObject` is the template class for all NS2 network objects. It inherits OTcl interfaces from class `TclObject` and the default action (i.e., function `handle(e)`) from class `Handler`. In addition, it defines a packet reception template and forces all its derived classes to provide packet reception implementation.

Function `recv(p,h)` is the very essence of packet forwarding mechanism in NS2. In NS2, an upstream object maintains a reference to the connecting downstream object. It passes a packet to the downstream object by invoking the function `recv(p,h)` of the downstream object and feeding the packet and optionally a handler as an input argument. Since NS2 focuses mainly on forwarding packets in a downstream direction, NsObjects do not need to have a reference to its upstream objects. In most cases, NsObject configuration involves downstream (not upstream) objects only.

Function `recv(p,h)` takes two input arguments: a packet "`*p`" to be received and a handler "`*h`." Most invocation of function `recv(p,h)` involves only packet "`*p`," not the handler.[2] For example, a `Queue` object (see Sect. 7.3.3) puts the received packet in the buffer and transmits the packet at the head of the buffer.

[2] We will discuss the *callback* mechanism which involves a handler in Sect. 7.3.3.

An ErrorModel object (see Sect. 15.3) imposes error probability on the received packet and forwards the packet to the connecting object if the transmission is not in error.

5.2.2 Packet Forwarding Mechanism of NsObjects

An NsObject forwards packets in two following ways:

- *Immediate packet forwarding*: To forward a packet to a downstream object, an upstream object needs to obtain a reference (e.g., a pointer) to the downstream object and invokes function `recv(p,h)` of the downstream object through the obtained reference. For example, a Connector (see Sect. 5.3) has a pointer "`target_`" to its downstream object. Therefore, it forwards a packet to its downstream object by executing `target_->recv(p,h)`.
- *Delayed packet forwarding*: To delay packet forwarding, a `Packet` object is cast to be an `Event` object, associated with a packet receiving NsObject, and placed on the simulation timeline at a given simulation time. At the firing time, the function `handle(e)` of the NsObject will be invoked, and the packet will be received through function `recv(p,h)` (see an example of delayed packet forwarding in Sect. 5.3).

5.3 Connectors

As shown in Fig. 5.2, a Connector is an NsObject which connects three NsObjects in a unidirectional manner. It receives a packet from an upstream NsObject. By default, a Connector immediately forwards the received packet to its downstream NsObject. Alternatively, it can drop the packet by forwarding the packet to a packet dropping object.[3]

From Fig. 5.2, a Connector is interested in specifying its downstream NsObject and packet dropping NsObject only. The connection from an upstream object to a Connector, on the other hand, is configured by the upstream object, not by the connector.

[3] A packet dropping network object (e.g., a null agent) is responsible for destroying packets.

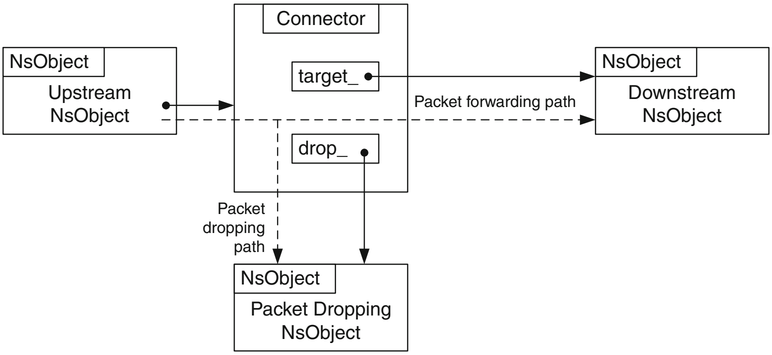

Fig. 5.2 Diagram of a connector: The *solid arrows* represent pointers, while the *dotted arrows* show packet forwarding and dropping paths

Program 5.3 Declaration and function `recv(p,h)` of class `Connector`

```
   //~/ns/common/connector.h
1  class Connector : public NsObject {
2  public:
3      Connector();
4      inline NsObject* target() { return target_; }
5      void target (NsObject *target) { target_ = target; }
6      virtual void drop(Packet* p);
7      void setDropTarget(NsObject *dt) {drop_ = dt; }
8  protected:
9      virtual void drop(Packet* p, const char *s);
10     int command(int argc, const char*const* argv);
11     void recv(Packet*, Handler* callback = 0);
12     inline void send(Packet* p, Handler* h){target_->recv
       (p, h);}
13
14     NsObject* target_;
15     NsObject* drop_;    // drop target for this connector
16 };

   //~/ns/common/connector.cc
17 void Connector::recv(Packet* p, Handler* h){send(p, h);}
```

5.3.1 Class Declaration

Program 5.3 shows the declaration of class `Connector`. Class `Connector` contains two pointers (Lines 14 and 15 in Program 5.3) to `NsObjects`[4]: "`target_`"

[4]Since class `Connector` contains two pointers to abstract object (i.e., class `NsObject`), it can be regarded as an abstract user class for class composition discussed in Sect. B.8. We will discuss the details of how the class composition concept applies to a Connector in the next section.

and “drop_.” From Fig. 5.2, “target_” is the pointer to the connecting downstream NsObject, while “drop_” is the pointer to the packet dropping object.

Class Connector derives from the abstract class NsObject. It overrides the pure virtual function recv(p,h), by simply invoking function send(p,h) (see Line 12 in program 5.3). Function send(p,h) simply forwards the received packet to its downstream object by invoking function recv(p,h) of the downstream object (i.e., target_->recv(p,h) in Line 12).

Program 5.4 Function drop of class connector

```
//~/ns/common/connector.cc
void Connector::drop(Packet* p)
{
    if (drop_ != 0)
        drop_->recv(p);
    else
        Packet::free(p);
}
```

Program 5.4 shows the implementation of function drop(p), which drops or destroys a packet. Function drop(p) takes one input argument, which is a packet to be dropped. If the dropping NsObject exists (i.e., “drop_”$\neq$ 0), this function will forward the packet to the dropping NsObject by invoking drop_->recv(p,h). Otherwise, it will destroy the packet by executing “Packet::free(p)” (see Chap. 8). Note that function drop(p) is declared as virtual (Line 9). Hence, classes derived from class Connector may override this function without any function ambiguity.[5]

5.3.2 *OTcl Configuration Commands*

As discussed in Sect. 4.1, NS2 simulation consists of two phases: Network Configuration Phase and Simulation Phase. In the Network Configuration Phase, a Connector is set up as shown in Fig. 5.2. Again, a Connector configures its downstream and packet dropping NsObjects only.

Suppose OTcl has instantiated three following objects: a Connector object (conn_obj), a downstream object (down_obj), and a dropping object (drop_obj). Then, the Connector is configured using the following two OTcl commands (see Program 5.5):

[5]Function ambiguity is discussed in Appendix B.2.

Program 5.5 OTcl commands `target` and `drop-target` of class `Connector`

```
    //~/ns/common/connector.cc
1   int Connector::command(int argc, const char*const* argv)
2   {
3       Tcl& tcl = Tcl::instance();
4       if (argc == 2) {
5           if (strcmp(argv[1], "target") == 0) {
6               if (target_ != 0)
7                   tcl.result(target_->name());
8               return (TCL_OK);
9           }
10          if (strcmp(argv[1], "drop-target") == 0) {
11              if (drop_ != 0)
12                  tcl.resultf("%s", drop_->name());
13              return (TCL_OK);
14          }
15      }
16      else if (argc == 3) {
17          if (strcmp(argv[1], "target") == 0) {
18              if (*argv[2] == '0') {
19                  target_ = 0;
20                  return (TCL_OK);
21              }
22              target_ = (NsObject*)TclObject::lookup(argv[2]);
23              if (target_ == 0) {
24                  tcl.resultf("no such object %s", argv[2]);
25                  return (TCL_ERROR);
26              }
27              return (TCL_OK);
28          }
29          if (strcmp(argv[1], "drop-target") == 0) {
30              drop_ = (NsObject*)TclObject::lookup(argv[2]);
31              if (drop_ == 0) {
32                  tcl.resultf("no object %s", argv[2]);
33                  return (TCL_ERROR);
34              }
35              return (TCL_OK);
36          }
37      }
38      return (NsObject::command(argc, argv));
39  }
```

- OTcl command `target` with one input argument conforms to the following syntax:

  ```
  $conn_obj target $down_obj
  ```

  This command casts the input argument `down_obj` to be of type `NsObject*` and stores it in the variable "`target_`" (Line 22).

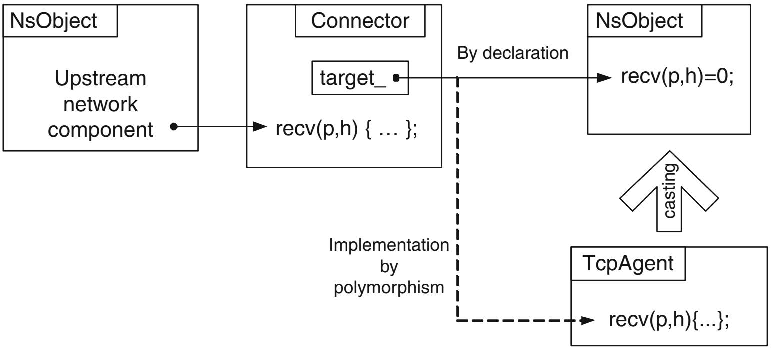

Fig. 5.3 A polymorphism implementation of a connector: A connector declares `target_` as an `NsObject` pointer. In the network configuration phase, the OTcl command `target` is invoked to setup a downstream object of the Connector, and the NsObject `*target_` is cast to a `TcpAgent` object

- OTcl command `target` with no input argument (e.g., `$conn_obj target`) returns OTcl instance corresponding to the C++ variable "`target_`" (Line 5–9). Note that function `name()` of class `TclObject` returns the OTcl reference string associated with the input argument.
- OTcl command `drop-target` with one input argument is very similar to that of the OTcl command `target` but the input argument is cast and stored in the variable "`drop_`" instead of the variable "`target_`."
- OTcl command `drop-target` with no input argument is very similar to that of the OTcl command `target` but it returns the OTcl instance corresponding to the variable "`drop_`" instead of the variable "`target_`."

Example 5.1. Consider the connector configuration in Figs. 5.2–5.3. Let the downstream object be of class `TcpAgent`, which corresponds to class `Agent/Tcp` in the OTcl domain. Also, let a `Agent/Null` object be a packet dropping NsObject. The following program shows how the network is set up from the OTcl domain:

```
set conn_obj [new Connector]
set tcp [new Agent/TCP]
set null [new Agent/Null]

$conn_obj target $tcp
$conn_obj drop-target $null
```

The first three lines create a Connector (`conn`), a TCP object (`tcp`), and a packet dropping object (`null`). The last two lines use the OTcl commands `target` and `drop-target` to set "`tcp`" and "`null`" as the downstream object and the dropping object of the Connector, respectively. □

Connector configuration complies with the class composition programming concept discussed in Appendix B.8. Table 5.1 shows the components in Example 5.1

Table 5.1 Class composition of network components in Example 5.1

Abstract class	NsObject
Derived class	Agent/Tcp and Agent/Null
Abstract user class	Connector
User class	A Tcl simulation script

and the corresponding class composition. Classes Agent/TCP and Agent/Null are OTcl classes whose corresponding C++ classes derive from class NsObject. Class Connector stores pointers (i.e., "target_" and "drop_") to NsObjects, and is therefore considered to be an abstract user class. Finally, as a user class, the Tcl Simulation Script instantiates NsObjects tcp, and null from classes Agent/Tcp, and Agent/Null, respectively, and binds tcp and null to variables target_ and drop_, respectively.

When invoking target and drop-target, tcp and null are first type-cast to NsObject pointers. Then they are assigned to pointers target_ and to drop_, respectively. Functions recv(p,h) of both tcp and null are associated with class Agent/TCP and Agent/Null, respectively, since they both are virtual functions.

5.3.3 Packet Forwarding Mechanism of Connectors

From Sect. 5.2.2, an NsObject forwards a packet in two ways: immediate and delayed packet forwarding. This section demonstrates both the packet forwarding mechanisms through a Connector.

5.3.3.1 Immediate Packet Forwarding

Immediate packet forwarding is carried out by invoking function recv(p,h) of a downstream object. In Example 5.1, the Connector forwards a packet to the TCP object by invoking function recv(p,h) of the TCP object (i.e., target_->recv (p,h), where target_ is configured to point to a TCP object). C++ polymorphism is responsible for associating the function recv(p,h) to class Agent/TCP (i.e., the construction type), not class NsObject (i.e., the declaration type).

5.3.3.2 Delayed Packet Forwarding

Delayed packet forwarding is implemented with the aid of the Scheduler. Here, a packet is cast to an event, associated with a receiving NsObject, and placed on the simulation timeline. For example, to delay packet forwarding in Example 5.1 by "d" seconds, we may invoke the following statement instead of target_-> recv(p,h).

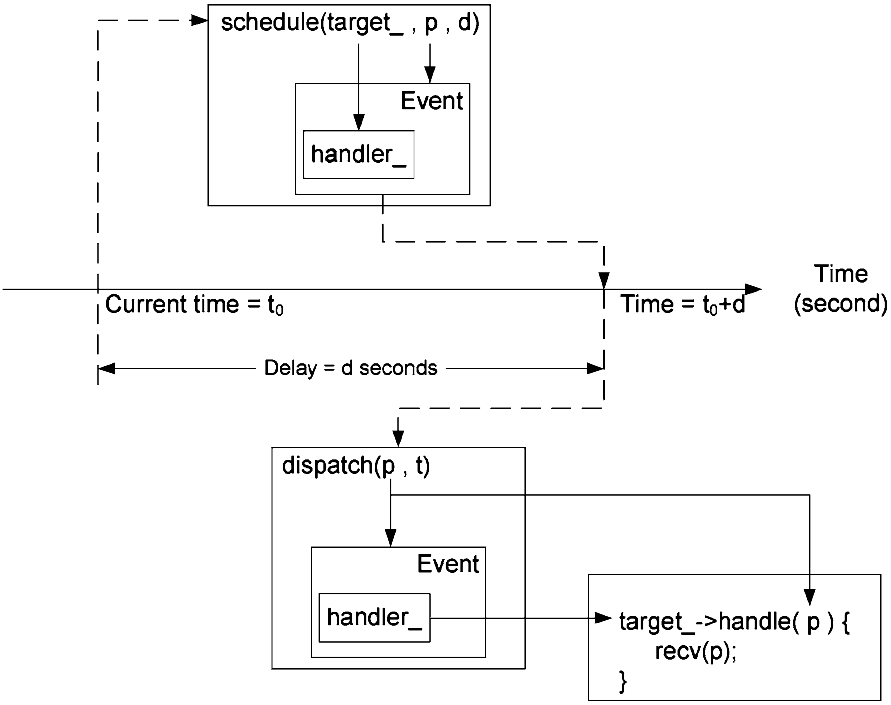

Fig. 5.4 Delayed packet forwarding mechanism

```
Scheduler& s = Scheduler::instance();
s.schedule(target_, p, d);
```

Consider Fig. 5.4 and Program 5.6 altogether. Figure 5.4 shows the diagram of delayed packet forwarding, while Program 5.6 shows the details of functions `schedule(h,e,delay)` as well as `dispatch(p,t)` of class `Scheduler`. The statement "`schedule(target_, p, d)`" casts packet `*p` and the NsObject `*target_` into `Event` and `Handler objects`, respectively (Line 1 of Program 5.6). Line 5 of Program 5.6 associates the packet `*p` with the NsObject `*target_`. Lines 6 and 7 insert the packet `*p` into the simulation timeline at the appropriate time. At the firing time, the event (`*p`) is dispatched (Lines 9–14). The Scheduler invokes function `handle(p)` of the handler associated with event `*p`. In this case, the associated handler is the NsObject `*target_`. Therefore, in Line 13, the default action `handle(p)` of "`target_`", invokes function `recv(p,h)` to receive the scheduled packet (see Program 4.2).

Program 5.6 Functions `schedule` and `dispatch` of class `Scheduler`

```
//~/ns/common/scheduler.cc
void Scheduler::schedule(Handler* h, Event* e, double delay)
{
    ...
    e->uid_ = uid_++;
    e->handler_ = h;
    e->time_ = clock_ + delay;
    insert(e);
}

void Scheduler::dispatch(Event* p, double t)
    ...
    clock_ = t;
    p->uid_ = -p->uid_; // being dispatched
    p->handler_->handle(p); // dispatch
}
```

5.4 Chapter Summary

Referred to as an NsObject, a network object is responsible for sending, receiving, creating, and destroying packets. As an object of class `NsObject`, it derives OTcl interfaces from class `TclObject` and the default action (i.e., function `handle(e)`) from class `Handler`. It defines a pure virtual function `recv(p,h)` as a uniform packet reception interface for the derived classes. Based on the polymorphism concept, the derived classes must provide their own implementation of how to receive a packet.

In NS2, an NsObject needs to create a connection to its downstream object only. Normally, an NsObject forwards a packet to a downstream object by invoking function `recv(p,h)` of its downstream object. In addition, an NsObject can defer packet forwarding by associating a packet to the downstream object and inserting the packet on the simulation timeline. At the firing time, the scheduler dispatches the packet, and the default action of the downstream object is invoked to receive the packet.

As an example, we show the details of class `Connector`, one of the main NsObject classes in NS2. Class `Connector` contains two pointers to NsObjects: "`target_`" pointing to a downstream object and "`drop_`" pointing to a packet dropping object. To configure a Connector, an object whose class derives from class `NsObject` can be set as downstream and dropping objects via OTcl command `target{...}` and `drop-target{...}`, respectively. These two OTcl commands cast the downstream and dropping objects to NsObjects, and assign them to C++ variables `*target_` and `*drop_`, respectively.

5.5 Exercises

1. What are the four types of NS2 objects? Explain their roles and differences among them.
2. Class `NsObject` contains a pure virtual function. What is the name of the function? Give a general description of the function. Why does it have to be declared as pure virtual?
3. What is the function which is central to packet reception mechanism?
4. What are the two packet reception methods? Explain their purposes and how they are implemented in NS2. Formulate an example from class `Connector` to show the process in time sequence.
5. Demonstrate how a packet is dropped in the C++ domain. Can you drop a packet from within any C++ class? Explain your answer via an example C++ class.