Chapter 4
Implementation of Discrete-Event Simulation in NS2

NS2 is a event-driven simulator, where actions are associated with events rather than time. An event in a event-driven simulator consists of execution time, associated actions, and a reference to the next event (Fig. 4.1). These events connect to each other and form a *chain of events* on the *simulation timeline* (e.g., that in Fig. 4.1). Unlike a time-driven simulator, in an event-driven simulator, time between a pair of events does not need to be constant. When the simulation starts, events in the chain are executed from left to right (i.e., chronologically).[1] In the next section, we will discuss the simulation concept of NS2. In Sects. 4.2–4.4, we will explain the details of classes `Event` and `Handler`, class `Scheduler`, and class `Simulator`, respectively. Finally, we summarize this chapter in Sect. 4.5.

4.1 NS2 Simulation Concept

NS2 simulation consists of two major phases.

Phase I: Network Configuration Phase

In this phase, NS2 constructs a network and sets up an initial chain of events. The initial chain of events consists of events that are scheduled to occur at certain times (e.g., start FTP (File Transfer Protocol) traffic at 1 s). These events are called *at-events* (see Sect. 4.2). This phase corresponds to every line in a Tcl simulation script before executing instproc `run{}` of the `Simulator` object.

[1] By execution, we mean taking actions associated with an event.

T. Issariyakul and E. Hossain, *Introduction to Network Simulator NS2*,
DOI 10.1007/978-1-4614-1406-3_4, © Springer Science+Business Media, LLC 2012

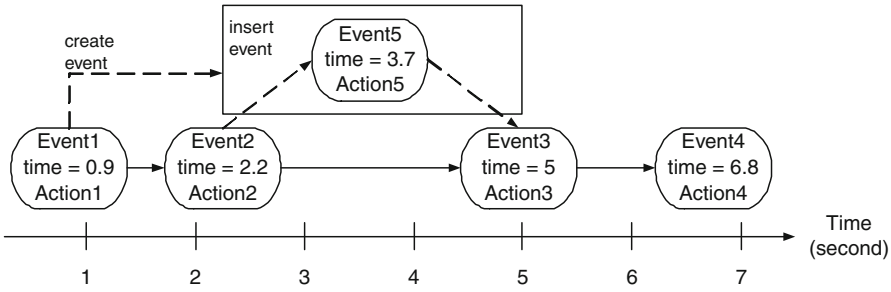

Fig. 4.1 A sample chain of events in a event-driven simulation. Each event contains execution time and a reference to the next event. In this figure, Event 1 creates and inserts Event 5 after Event 2 (the execution time of Event 5 is at 3.7 s)

Phase II: Simulation Phase

This part corresponds to a single line, which invokes the instproc `run{}` of class `Simulator`. Ironically, this single line contributes to most (e.g., 99%) of the simulation.

In this part, NS2 moves along the chain of events and executes events chronologically. Here, the instproc `run{}` starts the simulation by *dispatching* the first event in the chain of events. In NS2, "dispatching an event" or "firing an event" means "taking actions corresponding to that event." An action is, for example, starting FTP traffic or creating another event and inserting the created event into the chain of events. In Fig. 4.1, at 0.9 s, Event1 creates Event5 which will be dispatched at 3.7 s, and inserts Event5 after Event2. After dispatching an event, NS2 moves down the chain and dispatches the next event. This process repeats until the last event in the chain is dispatched, signifying the end of simulation.

4.2 Events and Handlers

4.2.1 An Overview of Events and Handlers

As shown in Fig. 4.1, an event is associated with an action to be taken at a certain time. In NS2, an event contains a *handler* that specifies the action, and the *firing time* or *dispatching time*. Program 4.1 shows declaration of classes `Event` and `Handler`. Class `Event` declares variables "`handler_`" (whose class is `Handler`; Line 5) and "`time_`" (Line 6) as its associated handler and firing time, respectively. To maintain the chain of events, each `Event` object contains pointers "`next_`" (Line 3) and "`prev_`" (Line 4) to the next and previous `Event` objects, respectively. The Variable "`uid_`" (Line 7) is an ID unique to every event.

Program 4.1 Declaration of classes `Event` and `Handler`

```
    //~/ns/common/scheduler.h
1   class Event {
2   public:
3       Event* next_;        /* event list */
4       Event* prev_;
5       Handler* handler_;   /* handler to call when event ready */
6       double time_;        /* time at which event is ready */
7       scheduler_uid_t uid_;    /* unique ID */
8       Event() : time_(0), uid_(0) {}
9   };

10  class Handler {
11  public:
12      virtual ~Handler () {}
13      virtual void handle(Event* e) = 0;
14  };
```

Program 4.2 Function `handle` of class `NsObject`

```
    //~/ns/common/object.cc
1   void NsObject::handle(Event* e)
2   {
3       recv((Packet*)e);
4   }
```

Lines 10–14 in Program 4.1 show the declaration of an abstract class `Handler`. Class `Handler` specifies the *default action* to be taken when an associated event is dispatched in its pure virtual function `handle(e)` (Line 13).[2] This declaration forces all its instantiable derived classes to provide the action in function `handle(e)`. In the followings, we will discuss few classes which derive from classes `Event` and `Handler`. These classes are `NsObject`, `Packet`, `AtEvent`, and `AtHandler`.

4.2.2 Class `NsObject`: A Child Class of Class `Handler`

Derived from class `Handler`, class `NsObject` is one of the main classes in NS2. It is a base class for most of network components. We will discuss the details of this class in Chap. 5. Here, we only show the implementation of the function `handle(e)` of class `NsObject` in Program 4.2. The function `handle(e)` casts an `Event` object associated with the input pointer `(e)` to a `Packet` object.

[2] We call actions specified in the function `handle(e)` *default actions*, since they are taken by default when the associated event is dispatched.

Program 4.3 Declaration of classes `AtEvent` and `AtHandler`, and function `handle` of class `AtHandler`

```
    //~/ns/common/scheduler.cc
 1  class AtEvent : public Event {
 2  public:
 3      AtEvent() : proc_(0) {
 4      }
 5      ~AtEvent() {
 6          if (proc_) delete [] proc_;
 7      }
 8      char* proc_;
 9  };

10  class AtHandler : public Handler {
11  public:
12      void handle(Event* event);
13  } at_handler;

14  void AtHandler::handle(Event* e)
15  {
16      AtEvent* at = (AtEvent*)e;
17      Tcl::instance().eval(at->proc_);
18      delete at;
19  }
```

Then it feeds the casted object to the function `recv(p)` (Line 3). Usually, function `recv(p)`, where "p" is a pointer to a packet, indicates that the `NsObject` has received a packet "p" (see Chap. 5). Unless overridden, by derived classes, the function `handle(e)` of an NsObject simply indicates packet reception.

4.2.3 *Classes `Packet` and `AtEvent`: Child Classes of Class `Event`*

Classes `Packet` and `AtEvent` are among key NS2 classes which derive from class `Event`. These two classes can be placed on the chain of events so that their associated handler will take actions at the firing time. While the details of class `AtEvent` are discussed in this section, that of class `Packet` will be discussed later in Chap. 8.

Declared in Program 4.3, class `AtEvent` represents events whose action is the execution of an OTcl statement. It contains one string variable "`proc_`" (Line 8) which holds an OTcl statement string. At the firing time, the associated handler, whose class is `AtHandler`, will retrieve and execute the OTcl string from this variable.

Program 4.4 Instance procedure `at` of class `Simulator` and command `at` of class `Scheduler`

```
    //~/ns/tcl/lib/ns-lib.tcl
 1  Simulator instproc at args {
 2      $self instvar scheduler_
 3      return [eval $scheduler_ at $args]
 4  }

    //~/ns/common/scheduler.cc
 5  if (strcmp(argv[1], "at") == 0) {
 6      /* t < 0 means relative time: delay = -t */
 7      double delay, t = atof(argv[2]);
 8      const char* proc = argv[3];
 9      AtEvent* e = new AtEvent;
10      int n = strlen(proc);
11      e->proc_ = new char[n + 1];
12      strcpy(e->proc_, proc);
13      delay = (t < 0) ? -t : t - clock();
14      if (delay < 0) {
15          tcl.result("can't schedule command in past");
16          return (TCL_ERROR);
17      }
18      schedule(&at_handler, e, delay);
19      sprintf(tcl.buffer(), UID_PRINTF_FORMAT, e->uid_);
20      tcl.result(tcl.buffer());
21      return (TCL_OK);
22  }
```

Derived from class `Handler`, class `AtHandler` specifies the actions to be taken at firing time in its function `handle(e)` (Lines 14–19). Here, Line 16 casts the input event into an `AtEvent` object. Then Line 17 extracts and executes the OTcl statement from variable "`proc_`" of the cast event.

In the OTcl domain, an `AtEvent` object is placed in a chain of events at a certain firing time by instproc "`at`" of class `Simulator`. Whose syntax is:

```
$ns at <time> <statement>
```

where "`$ns`" is the `Simulator` object (see Sect. 4.4), `<time>` is the firing time, and `<statement>` is an OTcl statement string which will be executed when the simulation time is `<time>` second.

From Lines 1–4 of Program 4.4, the instproc `at{...}` of an OTcl class `Simulator` invokes an OTcl command "`at`" of the `Scheduler` object (Lines 5–22). The OTcl command `at{...}` of class `Scheduler` stores the firing time in a variable "`t`" (Line 7). Line 9 creates an `AtEvent` object. Lines 8 and 10–12 store the input OTcl command in the variable "`proc_`" of the created `AtEvent` object.

Line 13 converts the firing time to the "delay" time from the current time. Finally, Line 18 schedules the created event *e at "delay" seconds in future, feeding address of the variable "at_handler" (see Program 4.3) as an input argument to function schedule(...).

4.3 The Scheduler

The scheduler maintains the chain of events and simulation (virtual) time. At runtime, it moves along the chain and dispatches one event after another. Since there is only one chain of events in a simulation, there is exactly one Scheduler object in a simulation. Hereafter, we will refer to the Scheduler object simply as the Scheduler. Also, NS2 supports the four following types of schedulers: List Scheduler, Heap Scheduler, Calendar Scheduler (default), and Real-time Scheduler. For brevity, we do not discuss the differences among all these schedulers here. The details of these schedulers can be found in [17].

4.3.1 Main Components of the Scheduler

Declared in Program 4.5, class Scheduler consists of a few main variables and functions. Variable "clock_" (Line 19) contains the current simulation time, and function clock() (Line 11) returns the value of the variable "clock_". Variable "halted_" (Line 22) is initialized to 0, and is set to 1 when the simulation is stopped or paused. Variable "instance_" (Line 20) is the reference to the Scheduler, and function instance() (Line 3) returns the variable "instance_". Variable uid_ (Line 21) is the event unique ID. In NS2, the Scheduler acts as a single point of unique ID management. When an event is inserted into the simulation timeline, the Scheduler creates a new unique ID and assigns the ID to the event. Both the variables "instance_" and "uid_" are static, since there is only one Scheduler and unique ID in a simulation.

4.3.2 Data Encapsulation and Polymorphism Concepts

Program 4.5 implements the concepts of *data encapsulation* and *polymorphism* (see Appendix B). It hides the chain of events from the outside world and declares pure virtual functions cancel(e), insert(e), lookup(uid), deque(), and head() in Lines 6–10 to manage the chain. Classes derived from class Scheduler provide implementation of all of the above functions. The beauty of this mechanism is the ease of modifying the scheduler type at runtime.

Program 4.5 Declaration of class `e.g., Scheduler`

```
    //~ns/common/scheduler.h
 1  class Scheduler : public TclObject {
 2  public:
 3      static Scheduler& instance() { return (*instance_); }
 4      void schedule(Handler*, Event*, double delay);
 5      virtual void run();
 6      virtual void cancel(Event*) = 0;
 7      virtual void insert(Event*) = 0;
 8      virtual Event* lookup(scheduler_uid_t uid) = 0;
 9      virtual Event* deque() = 0;
10      virtual const Event* head() = 0;
11      double clock() const { return clock_j}
12      virtual void reset();
13  protected:
14      void dispatch(Event*);
15      void dispatch(Event*, double);
16      Scheduler();
17      virtual ~Scheduler();
18      int command(int argc, const char*const* argv);
19      double clock_;
20      static Scheduler* instance_;
21      static scheduler_uid_t uid_;
22      int halted_;
22  };
```

NS2 implements most of the codes in relation to class `Scheduler`, not its derived classes (e.g., `CalendarScheduler`). At runtime (e.g., in a Tcl simulation script), we can select a scheduler to be of any derived class (e.g., `CalendarScheduler`) of class `Scheduler` without having to modify the codes for the base class (e.g., `Scheduler`).

4.3.3 Main Functions of the Scheduler

Three main functions of class `Scheduler` are `run()` (Program 4.6), `schedule(h,e,delay)` (Program 4.7) and `dispatch(p,t)` (Program 4.8). In Program 4.6, function `run()` first sets variable "`instance_`" to the address of the scheduler (`this`) in Line 3. Then, it keeps dispatching events (Line 6) in the chain until "`halted_`"$\neq 0$ or until all the events are executed (Line 5).

Function `schedule(h,e,delay)` in Program 4.7 takes three input arguments: A `Handler` pointer `(h)`, an `Event` pointer `(e)`, and the delay `(delay)`, respectively. It inserts the input `Event` object `(*e)` into the chain of events. Lines 3–12 check for possible errors. Line 13 increments the unique ID of the Scheduler and assigns it to the input `Event` object. Line 14 associates the input `Handler`

Program 4.6 Function `run` of class `Scheduler`

```
    //~ns/common/scheduler.cc
1   void scheduler::run()
2   {
3       instance_ = this;
4       Event *p;
5       while (!halted_ && (p = deque())) {
6           dispatch(p, p->time_);
7       }
8   }
```

Program 4.7 Function `schedule` of class `Scheduler`

```
    //~ns/common/scheduler.cc
1   void Scheduler::schedule(Handler* h, Event* e, double delay)
2   {
3       if (!h) { /* error: Do not feed in NULL handler */ };
4       if (e->uid_ > 0) {
5           printf("Scheduler: Event UID not valid!\n\n");
6           abort();
7       }
8       if (delay < 0) { /* error: negative delay */ };
9       if (uid_ < 0) {
10          fprintf(stderr, "Scheduler: UID space exhausted!
11          \n"); abort();
12      }
13      e->uid_ = uid_++;
14      e->handler_ = h;
15      double t = clock_ + delay;
16      e->time_ = t;
17      insert(e);
18  }
```

Program 4.8 Function `dispatch` of class `Scheduler`

```
    //~ns/common/scheduler.cc
1   void Scheduler::dispatch(Event* p, double t)
2   {
3       if (t < clock_) { /* error */ };
4       clock_ = t;
5       p->uid_ = -p->uid_;      // being dispatched
6       p->handler_->handle(p); // dispatch
7   }
```

object (*h) with the input Event object (*e). Line 15 converts input delay time (delay) to the firing time (time_) of the Event object "e." Line 17 inserts the configured Event object *e in the chain of events via function insert(e). Since the scheduler increments its unique ID when invoking function schedule(...), every scheduled event will have different unique ID.

Finally, the errors in Lines 3–12 include

1. Null handler (Line 3)
2. Positive Event unique ID (Lines 4–7; See Sect. 4.3.5)
3. Negative delay (Line 8)
4. Negative Scheduler unique ID[3] (Lines 19-12)

Function dispatch(p,t) in Program 4.8 is invoked by function run() at the firing time (Line 6 of Program 4.6). It takes a dispatching event (*p) and firing time (t) as input arguments. Since the scheduler moves forward in simulation time, the firing time (t) cannot be less than the current simulation time (clock_). From Program 4.8, Line 3 will show an error, if t < clock_. Line 4 sets the current simulation virtual time to be the firing time of the event. Line 5 inverts the sign of the "uid_" of the event, indicating that the event is being dispatched. Line 6 invokes function handle(p) of the associated handler "handler_," feeding the event (p) as an input argument.

4.3.4 *Two Auxiliary Functions*

Apart from the above three main function, class Scheduler provides two very useful functions: instance() (Line 3 in Program 4.5) and clock (Line 11 in Program 4.5).

- Function instance() returns *instance_ which is the address of the Scheduler.
- Function clock() returns the current simulation virtual time.

We shall see the use of these two functions throughout NS2 programming.

Example 4.1. In order to obtain the current virtual simulation time in C++, we can resort to the following statements

```
Scheduler& s = Scheduler::instance();
cout << "The current time is " << s.clock() << endl;
```

Here the upper line retrieves the reference to the Scheduler, while the lower line invokes the function clock() of the current simulation time. □

[3]The unique ID of the Scheduler is always positive. Its negative value indicates possible abnormality such as memory overflow or inadvertent memory access violation.

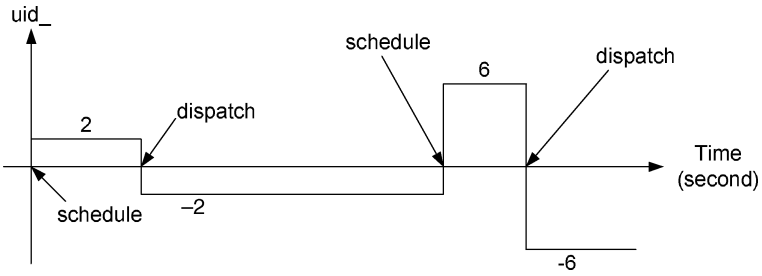

Fig. 4.2 Dynamics of `Event` unique ID `(uid)_`: Take a positive value from `Scheduler ::uid_` when being scheduled, and invert the sign when being dispatched. Increment upon `schedule` and inversion of sign upon `dispatch`

4.3.5 Dynamics of the Unique ID of an Event

The dynamics of the event's unique ID (`uid_`) is fairly subtle. In general, the Scheduler maintains the unique ID and assigns the unique ID to the event being scheduled. To make "`uid_`" unique, the Scheduler increments its "`uid_`" and assigns the incremented "`uid_`" to the scheduling event in its function `schedule(...)` (Line 13 in Program 4.7). When dispatching an event, the scheduler inverts the sign of "`uid_`" of the dispatching event (Line 5 in Program 4.8). Figure 4.2 shows the dynamics of the unique ID caused by the above `schedule(...)` and `dispatch(...)` functions. Unless the associated event is being dispatched, "`uid_`" of an event is always increasing and non-negative. The sign toggling mechanism of unique ID ensures that events will be scheduled and dispatched properly. If a scheduled event is not dispatched, or is dispatched twice, its unique ID will be positive, and an attempt to schedule this undispatched event will cause an error (Lines 5 and 6 in Program 4.7).

4.3.6 Scheduling–Dispatching Mechanism

We conclude this section through an example explaining the scheduling–dispatching mechanism. Consider the following script

```
set ns [new Simulator]
$ns at 10 [puts "An event is dispatched"]
$ns run
```

which prints out the message "`An event is dispatched`" at 10 s after the simulation has started. Figure 4.3 shows the functions (shown in rectangles) and objects (shown in rounded rectangles) related to the scheduling–dispatching mechanism, whose names are shown in boldface font. Again, an `AtEvent` object is scheduled by the OTcl command "`at`" (in the upper-left rectangle), of class `Scheduler`. The Scheduler creates an `AtEvent` object "`e`" and stores

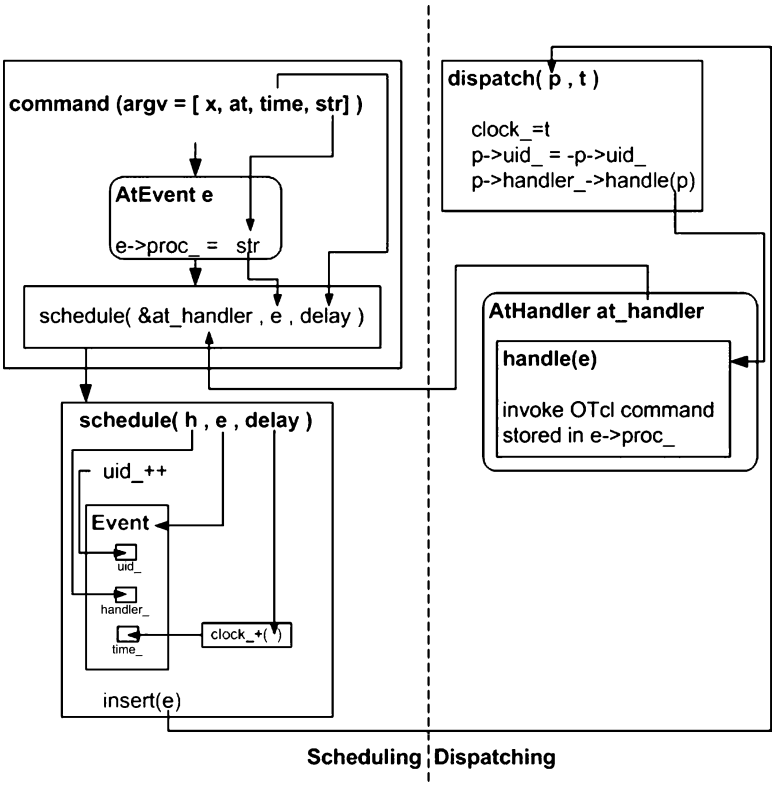

Fig. 4.3 Scheduling and dispatching mechanism of an `AtEvent`

input command (the fourth input argument "`str = puts "An event is dispatched"`)" in `e->proc_`. Then, it schedules the event "e" with `delay` converted from `time = 10` (the third input argument), feeding the address of `AtHandler` object (`at_handler` in the lower right round rectangle) as the corresponding handler.

The lower-left rectangle in Fig. 4.3 shows details of the function `schedule (h,e,delay)` of class `Scheduler`. Before inserting event "e" into the chain of events, function `schedule(...)` configures event "e" as follows: Update "`uid_`" to be the same as that of `Scheduler`, store "`at_handler`" in the handler of event "e," and set firing time to be "`clock_`" (current time) + "`delay`."

At the firing time, the scheduled `AtEvent` object is dispatched through the function `dispatch(p,t)` (the upper-right rectangle in Fig. 4.3). When the scheduled `Event` object "e"[4] is dispatched, function `dispatch(...)` inverts the sign of its variable "`uid_`," and invokes function `handle(e)` of the

[4]In Program 4.8, the first argument of function `dispatch(...)` is "p." Here, we use "e" as the first argument for the sake of explanation.

corresponding handler feeding `Event` object "`e`" as an input argument. Since the handler is "`at_handler`" (see the upper-left rectangle), the OTcl command "`puts "An event is dispatched"`" stored in "`e`" is executed.

4.3.7 Null Event and Dummy Event Scheduling

When being dispatched, an event "`p`" is fed to function `handle(p)` of the associated handler for a certain purpose. For example, the function `handle(p)` of class `NsObject` executes "`recv(p)`", where "`p`" is a packet reception event. Here, the event `*p` must have been created and fed to the function `schedule(...)` before the ongoing dispatching process.

In some cases, an event only indicates the time where the default action is taken but takes no part in such the action. For example, a queue unblocking event informs the associated `Queue` object of the completion of the ongoing transmission (see Sect. 7.3). The function `handle(p)` of the associated handler in this case simply invokes function `resume()` which take no input argument (i.e., action taking). Clearly the queue unblocking event takes no role in the dispatching process. In this case, we do not need to explicitly create an event. Instead, we can use a null event or a dummy event as an input argument to the function `schedule(...)`.

4.3.7.1 Scheduling of a Null Event

Function `schedule(h,e,delay)` takes a pointer to an event as its second input argument. A null event refers to a null pointer which is fed as the second input argument to the function `schedule(...)` (e.g., `schedule(handler, 0,delay)`).

Although simple to use, a null event could lead to runtime error which is difficult to be located. A null event is not an *actual* event. Its unique ID does not follow semantic in Fig. 4.2. The Scheduler ignores the unique ID when scheduling and dispatching a null event, and allows an undispatched event to be rescheduled. This breaks the scheduling–dispatching protection mechanism. Using null events, the users are responsible for ensuring the proper sequence of scheduling–dispatching by themselves.

4.3.7.2 Scheduling of a Dummy Event

This is another approach to schedule and dispatch events which do not take part in default actions. A dummy event is usually declared as a member variable of a C++ class, and is used repeatedly in a scheduling–dispatching process.

Consider a packet departure event which is modeled by class `LinkDelay` (see Sect. 7.2) for example. During simulation, an NsObject informs a `LinkDelay`

object to schedule packet departure events. At the firing time, the packet completely departs the NsObject, and the NsObject is allowed to fetch another packet for transmission. The packet departure event takes no part in the default action, since a new packet is fetched or created by another object.

As we shall see, a packet departure event is represented by a dummy event variable "`intr_`" of class `LinkDelay`, and the packet departure is scheduled through the variable "`intr_`" only. Since the variable "`intr_`" is a dummy `Event`, its unique ID follows the semantic in Fig. 4.2. An attempt to schedule an undispatched event would immediately cause runtime error. Note that "`intr_`" is a member variable of class `LinkDelay`. It is used over and over again to indicate packet departure from a `LinkDelay` object.

As a final note, under a simple configuration, it is recommended to use the null event scheduling approach. For a complicated configuration, on the other hand, the dummy event scheduling is preferable, since it provides a protection against scheduling of undispatched events.

4.4 The Simulator

OTcl and C++ classes `Simulator` are the main classes which supervise the entire simulation. Like the `Scheduler` object, there can be only one `Simulator` object throughout a simulation. This object will be referred to as *the Simulator* hereafter. The Simulator contains two types of key components: simulation objects and information-storing objects. While simulation objects (e.g., the Scheduler) are the key components which drive the simulation. On the other hand, Information-storing objects (e.g., the reference to created nodes) contain information which is shared among several objects. These information-storing objects are created via various instprocs (e.g., `Simulator::node{}`) during the Network Configuration Phase. Most objects access these information-storing objects via its instvar "`ns_`" (set by executing "`set ns_ [Simulator instance]`"), which is a reference to a Simulator.

4.4.1 Main Components of a Simulation

4.4.1.1 Interperted Hierarchy

Created by various instprocs, the main OTcl simulation components are as follows:

- *The Scheduler* (`scheduler_` created by the instproc `Simulator::init`) maintains the chain of events and executes the events chronologically.

- *The null agent* (`nullAgent_` created by the instproc `Simulator::init`) provides the common packet dropping point.[5]
- *Node reference* (`Node_` created by the instproc `Simulator::node`) is an associative array whose elements are the created nodes and indices are node IDs.
- *Link reference* (`link_` created by the instprocs `simplex-link{...}` or `duplex- link{...}`) is an associative array. Associated with an index with format "`sid:did`," each element of "`link_`" is the created unidirectional link which carries packet from node "`sid`" to node "`did`."
- *Reference to routing table* (`routingTable_` created by the instproc `Simulator::get-routelogic{}`,) contains the global routing table.

4.4.1.2 Compiled Hierarchy

In the compiled hierarchy, class `Simulator` also contains variables and functions as shown in Program 4.9. Variable "`instance_`" (Line 18) is a pointer to the Simulator. It is a static variable, which means that there is only one variable "`instance_`" of class `Simulator` for the entire simulation. Variable "`node list_`" (Line 14) is the link list containing all created nodes. The link list can contain up to "`size_`" elements (Line 17), while the total number of nodes is "`nn_`" (Line 16). Variable `rtobject_` (Line 15) is a pointer to a `RouteLogic` object, which is responsible for the routing mechanism (see Chap. 6).

Function `populate_flat_classifiers(...)` (Line 7) pulls out the routing information stored in the variable `*rtobject_` and installs the routing table in the created nodes and links (see Sect. 6.5). Function `add_node(...)` (Line 8) puts the input argument `node` into the link list of nodes (`nodelist_`). Function `get_link_head(...)` returns the link head object (see Chap. 7) of the link with ID "`nh`" which connects to a `ParentNode` object `*node`. Function `node_id_by_addr(addr)` (Line 10) converts node address "`addr`" to node ID. Function `alloc(n)` (Line 11) allocates spaces in `nodelist_` which can accommodate up to "`n`" nodes, and clears all components of `nodelist_` to `NULL`. Function `check(n)` immediately returns if "`n`" is less than `size_`. Otherwise, it will create more space in `nodelist_`, which can accommodate up to "`n`" nodes. Static function `instance()` in Line 3 returns the variable "`instance_`" which is the pointer to the Simulator.

4.4.2 Retrieving the Instance of the Simulator

From the interpreted hierarchy, we can also retrieve the simulator instance by invoking the instproc `instance{}` of class `Simulator` (see program 4.10).

[5] By "dropping a packet," we mean "removing a packet" from the simulation. We will discuss the dropping mechanism in Chap. 5. For the moment, it is sufficient to know that "`nullAgent_`" drops or removes all received packets from the simulation.

Program 4.9 Declaration of class `Simulator`

```
    //~ns/common/simulator.h
 1  class Simulator : public TclObject {
 2  public:
 3      static Simulator& instance() { return (*instance_); }
 4      Simulator() : nodelist_(NULL),
                rtobject_(NULL), nn_(0), size_(0) {}
 5      ~Simulator() { delete []nodelist_;}
 6      int command(int argc, const char*const* argv);
 7      void populate_flat_classifiers();
 8      void add_node(ParentNode *node, int id);
 9      NsObject* get_link_head(ParentNode *node, int nh);
10      int node_id_by_addr(int address);
11      void alloc(int n);
12      void check(int n);
13  private:
14      ParentNode **nodelist_;
15      RouteLogic *rtobject_;
16      int nn_;
17      int size_;
18      static Simulator* instance_;
19  };
```

Program 4.10 Retrieving the instance of the Simulator using instproc `instance` of class `Simulator`

```
    //~ns/tcl/lib/ns-lib.tcl
1 Simulator proc instance {} {
2     set ns [Simulator info instances]
3     if { $ns != "" } {
4         return $ns
5     }
6     ...
7 }
```

This instproc executes the OTcl built-in command "`info`" with an option "`instances`." This execution returns all the instances of a certain class. Since there is only one `Simulator` instance, the statement "`Simulator info instances`" returns the `Simulator` object as required.

4.4.3 Simulator Initialization

Simulator initialization refers to the process in the Network Configuration Phase, which creates the Simulator as well as its components. The Simulator is created by executing "`new Simulator`". This statement invokes the constructor (i.e., the instproc `init{...}` of class `Simulator`) shown in Program 4.11.

Program 4.11 Instprocs `init` and `use-scheduler` of class `Simulator`

```
    //~ns/tcl/lib/ns-lib.tcl
1   Simulator instproc init args {
2       $self create_packetformat
3       $self use-scheduler Calendar
4       $self set nullAgent_ [new Agent/Null]
5       $self set-address-format def
6       eval $self next $args
7   }

8   Simulator instproc use-scheduler type {
9       $self instvar scheduler_
10      if [info exists scheduler_] {
11          if { [$scheduler_ info class] == "Scheduler/$
                type" } {
12              return
13          } else {
14              delete $scheduler_
15          }
16      }
17      set scheduler_ [new Scheduler/$type]
18  }
```

The constructor first initializes the packet format in Line 2, and executes the OTcl statement "`use-scheduler{type}`" in Line 3 to specify type of the Scheduler. By default, type of the Scheduler is `Calendar`. Line 4 creates a null agent (`nullAgent`). Line 5 sets the address format to the default format. The instproc `use-scheduler{type}` (Lines 8–18) deletes the existing Scheduler if it is different from that specified in the input argument "`type`." Then it will create a scheduler with type = `type`, and store the created `Scheduler` object in the instvar "`scheduler_`."

4.4.4 Running Simulation

The Simulation Phase starts at the invocation of the instproc `run{}` of class `Simulator`. As shown in Program 4.12, this instproc first invokes the instproc `configure{}` of class `RouteLogic` (Line 2), which in turn computes the optimal routes and creates the routing table (see Chap. 6). Lines 5–10 reset nodes and queues. Finally, Line 11 starts the Scheduler by invoking the OTcl command `run{}` of class `Scheduler`, which in turn invokes the C++ function `run()` of class `Scheduler` shown in Program 4.6. Again, this function executes events in the chain of events one after another until the Simulator is halted (i.e., variable "`halted_`" of class `Scheduler` is 1), or until all the events are executed.

Program 4.12 Instproc : : run of class simulator

```
    //~/ns/tcl/lib/ns-lib.tcl
1   Simulator instproc run  {
2       [$self get-routelogic] configure
3       $self instvar scheduler_ Node_ link_ started_
4       set started_ 1
5       foreach nn [array names Node_] {
6           $Node_($nn) reset
7       foreach qn [array names link_] {
8           set q [$link_($qn) queue]
9           $q reset
10      }
11      return [$scheduler_ run]
12  }
```

4.4.5 Instprocs of OTcl Class Simulator

The list of useful instprocs of class `Simulator` is shown below.

`now{}`	Retrieve the current simulation time.
`nullagent{}`	Retrieve the shared null agent.
`use-scheduler{type}`	Set the scheduler to be `<type>`.
`at{time stm}`	Execute the statement `<stm>` at `<time>` second.
`run{}`	Start the simulation.
`halt{}`	Terminate the simulation.
`cancel{e}`	Cancel the scheduled event `<e>`.

4.5 Chapter Summary

This chapter explains the details of event-driven simulation in NS2. The simulation is carried out by running a *Tcl simulation script*, which consists of two parts. First, the *Network Configuration Phase* establishes a network and configures all simulation components. This phase also creates a chain of events by connecting the created events chronologically. Second, the *Simulation Phase* chronologically executes (or dispatches) the created events until the Simulator is halted, or until all the events are executed.

There are four main classes involved in an NS2 simulation:

- Class `Simulator` supervises the simulation. It contains simulation components such as the Scheduler, the null agent. It also contains information storing objects which are shared by other (simulation) components.
- Class `Scheduler` maintains the chain of events and chronologically dispatches the events.

- Class `Event` consists of the firing time and the associated handler. Events are put together to form a chain of events, which are dispatched one by one by the Scheduler. Classes `Packet` and `AtEvent` are among the classes derived from class `Event`, which can be placed on the simulation timeline (i.e., in the chain of event). They are associated with different handlers and take different actions at the firing time.
- Class `Handler`: Associated with an event, a handler specifies default actions to be taken when the associated event is dispatched. Classes `NsObject` and `AtHandler` are among classes derived from class `Handler`. They are always associated with `Packet` and `AtEvent` events, respectively. Their actions are to receive a `Packet` object and to execute an OTcl statement specified in the `AtEvent` object, respectively.

4.6 Exercises

1. What are the definitions, similarities/differences, and relationship among the following NS2 components:

   a. Simulation timeline
   b. Scheduler
   c. Event
   d. Event handler
   e. Simulator
   f. Firing time or dispatching time

   Show an example to support your answer.
2. What are the similarities/differences/relationship between a TclObject and an NSObject?
3. What is a chain of events? Explain how NS2 creates a chain of events, and how NS2 locates a particular event on the chain.
4. What is event unique ID? Where does NS2 store this value? What is its data type? What is the implication when its value is positive, negative, or zero?
5. What are the two simulation phases? Explain the key objectives of each of the phases.
6. NS2 has two types of built-in events: Packet reception events and AT events. Design another type of events. Explain their purposes, show how these events can be integrated into NS2, and write an NS2 program to support your answer.
7. Explain the sequence of actions that occurs at the firing time. Use an OTcl statement execution event as an example.
8. How does NS2 start a simulation in the OTcl domain? What happens in the C++ domain after the simulation has started? When and under what condition will the simulation terminate?

9. What are four common errors associated with event scheduling in NS2? Explain the reasons and suggest general solutions.
10. What are Null events and dummy events? What are their purposes? Explain their similarities and differences. Show example usage of both types of events.
11. Write statements for the following purposes. Run NS2 to test your answer.

    a. Show the current virtual time on the screen in both C++ and OTcl domain.
    b. At 10 s, print out "Hello NS2 Users!!" on the screen. In the C++ domain, use `printf(...)` or `cout`. In the OTcl domain, use `puts{...}`).
    c. Store a `Simulator` object in a local variable `csim` in the C++ domain, and `osim` in the OTcl domain.
    d. Send a packet `*p` to an NsObject `*obj` at 15 s in future (C++ only).