

# Event Driven Simulation in NS2

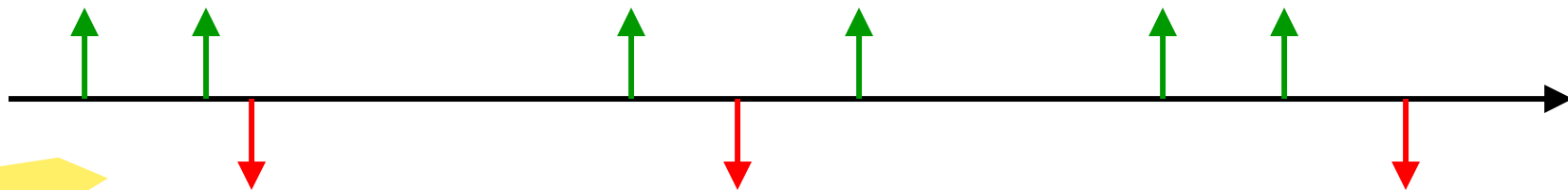
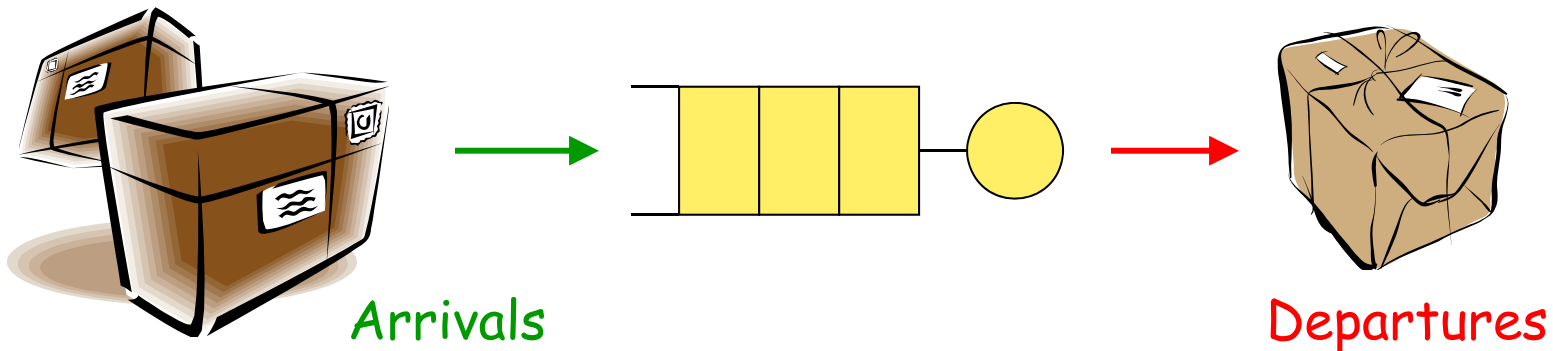


# Outline

- Recap: Discrete Event v.s. Time Driven
- Events and Handlers
- The Scheduler
- The Simulator
- Summary

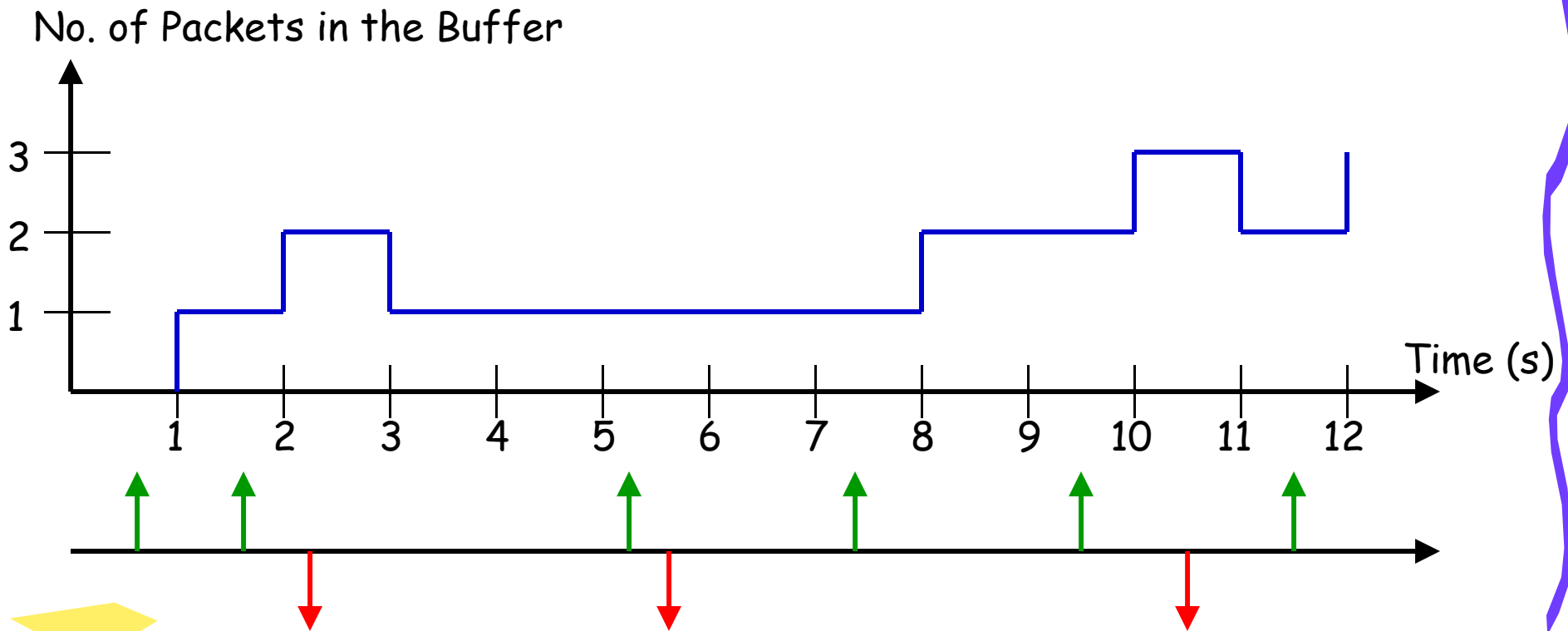
# Event-Driven v.s. Time-Driven

- Q: Time Driven = ( )
- Q: Event Driven = ( )
- Time Driven or Discrete Time Simulation
- Example: Packet arrivals and departures



# Time-Driven Simulation

- Observe the buffer for every FIXED period (e.g., 1 second)



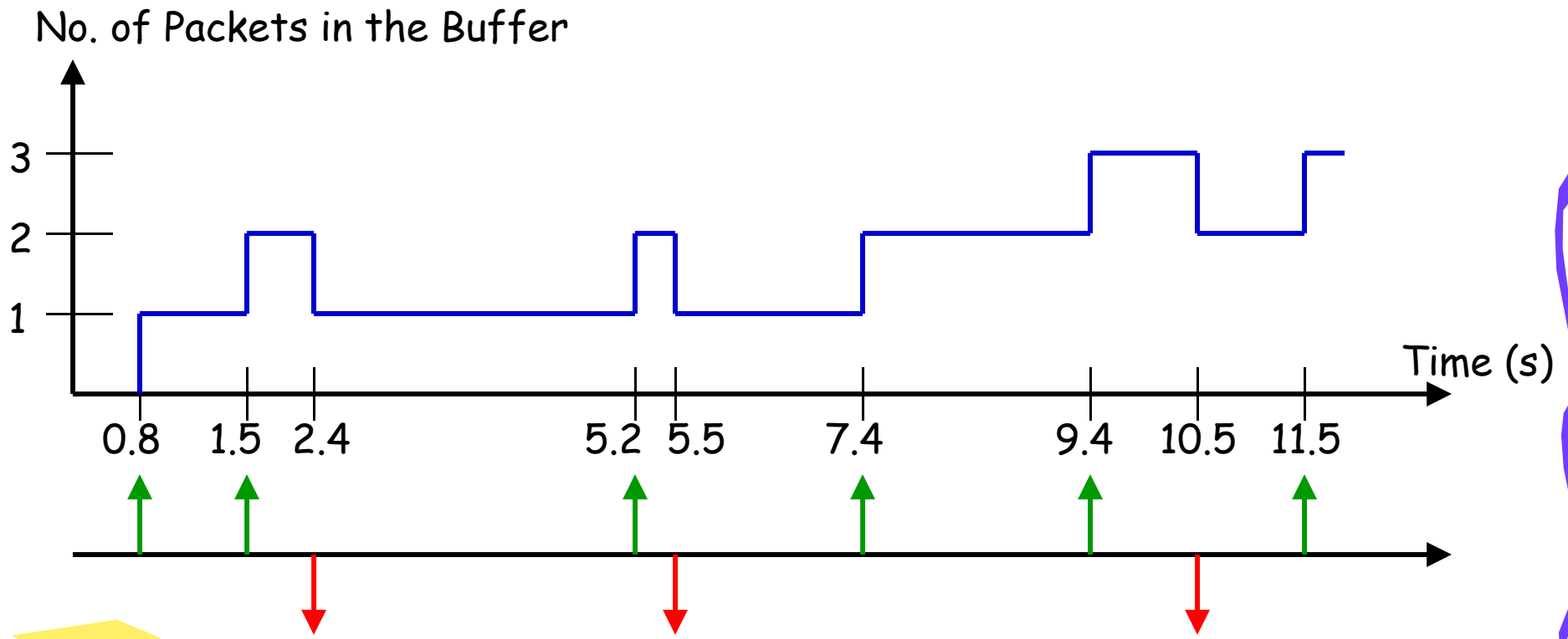
# Time-Driven Simulation

- Simulation event for every time slot (fixed interval)
- Example Pseudo Codes:

```
For t = 1 to sim_time {  
    if (arrival)  
        buffer = buffer + 1;  
    if (departure)  
        buffer = buffer -1;  
    print(buffer);  
}
```

# Event-Driven Simulation

- Go from one event to another
- Same Example



# Event-Driven Simulation

- Use a Scheduler
- Maintain a set of events
- Example

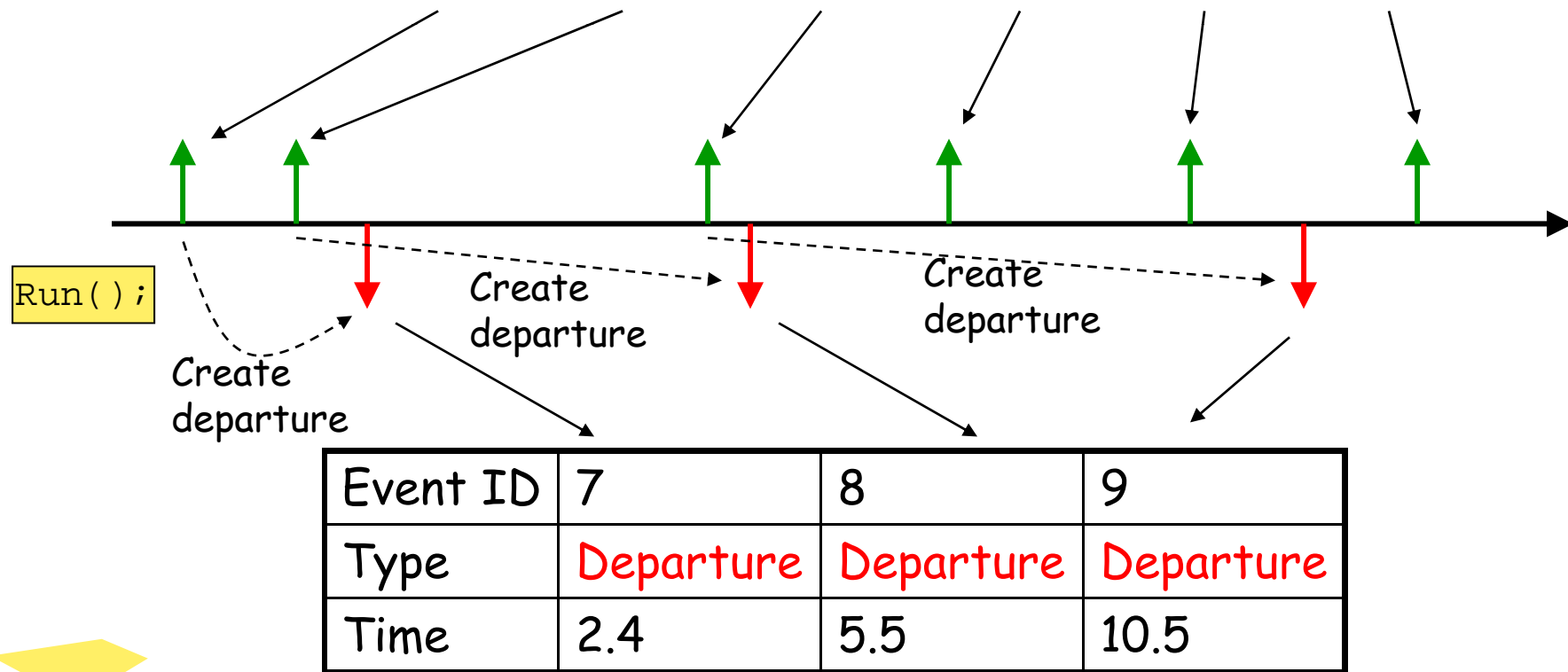
```
CreateEvent();  
Run ();
```

**Psudo Codes**

```
CreateEvent() {  
  Pkt1.arr(0.8)  
  Pkt2.arr(1.5)  
}
```

# Event-Driven Simulation

Event ID	1	2	3	4	5	6
Type	Arrival	Arrival	Arrival	Arrival	Arrival	Arrival
Time	0.8	1.5	5.2	7.4	9.4	11.5



# NS2 Simulation Concept

- Event-Driven Simulation
- Recap: Simulation Main Steps
  - Design
  - Simulation
    - Network Configuration Phase → `CreateEvent()`
    - Simulation Phase → `Run()`
  - Result Compilation



# Simulation

- Network Configuration Phase
  - Create topology
  - Schedule event (e.g., `CreateEvent()`)
- Simulation Phase
  - `Simulator::run()` (e.g., `Run()`)
  - Execute the scheduled events



# Outline

- Recap: Discrete Event v.s. Time Driven
- Events and Handlers
- The Scheduler
- The Simulator
- Summary

# Event and Handler: Outline

- Overview
- C++ Classes Event and Handler
- Two Main Types of Events
  - AtEvent
  - Packet



# Concepts of Events and Handlers

- Event-driven simulation
  - Put events on the simulation timeline
  - Move forward in time
  - When finding an event, take associated actions (i.e., execute the event)
- Main components
  - Events → C++ class `Event`
  - Actions → C++ class `Handler`



**Q: Give examples of events.**

# Event and Handler

- Examples of Events
  - Packet Arrivals/Departures
  - Start/Stop Application

```
$ns at 0.05 "$ftp start"  
$ns at 0.1  "$cbr start"  
$ns at 60.0 "$ftp stop"  
$ns at 60.5 "$cbr stop"  
$ns at 61  "finish"
```

# Event and Handler: C++ Classes

- `Class Event`: Define events (e.g., packet arrival)
- `Class Handler`: Define (default) actions associated with an event (tell the node to receive the packet)



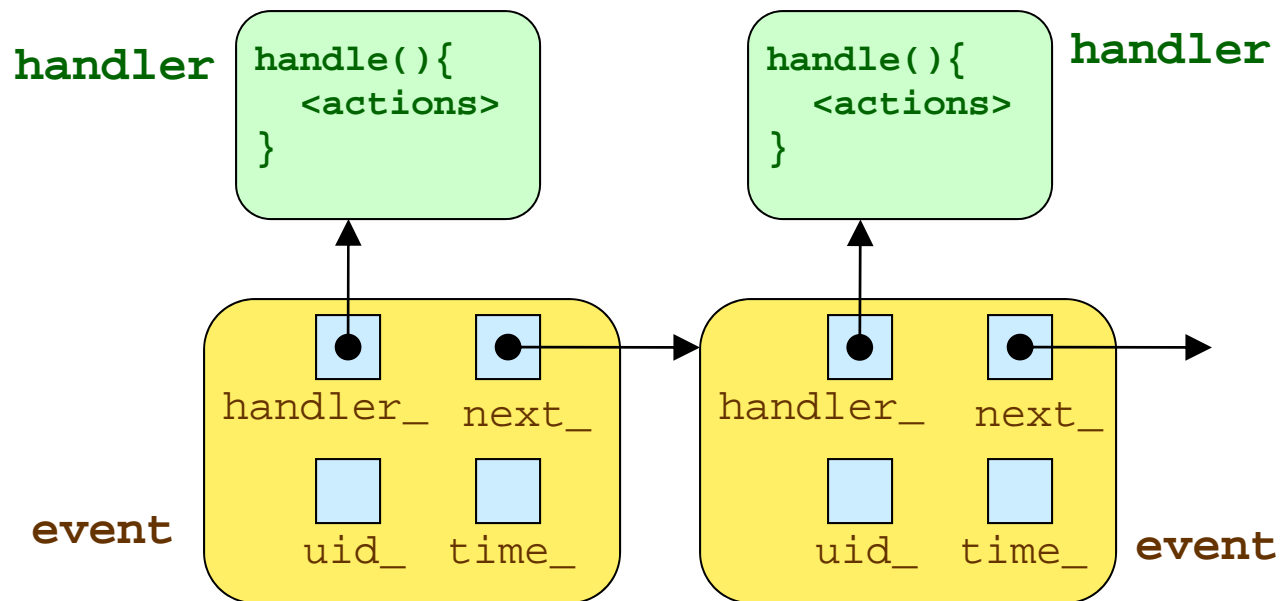
# C++ Class Event

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



# Class Event

- Main variables:
  - next\_: Next event
  - time\_: Time
  - uid\_: Unique ID
  - handler\_: Handler



# Class Handler

- Declaration

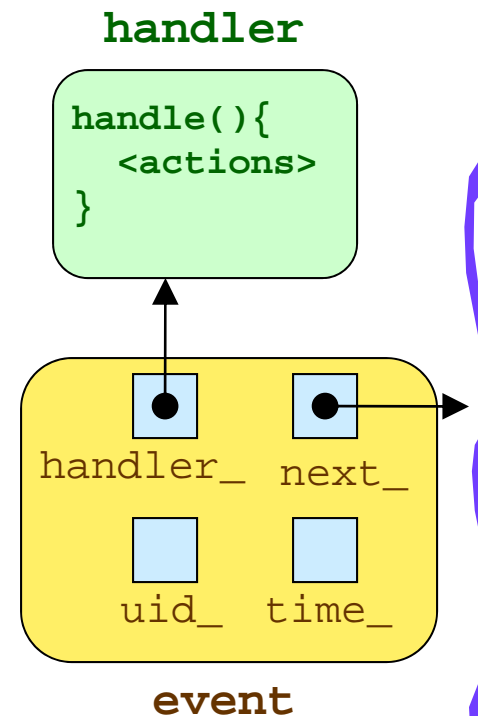
```
//~/ns/common/scheduler.h
class Handler {
public:
    virtual ~Handler () {}
    virtual void handle(Event* e) = 0;
};
```

What is this?  
What is the purpose?

- Define Default Actions

→ C++ function `handle(Event*)`

- Associated with an Event



# Handlers: Example

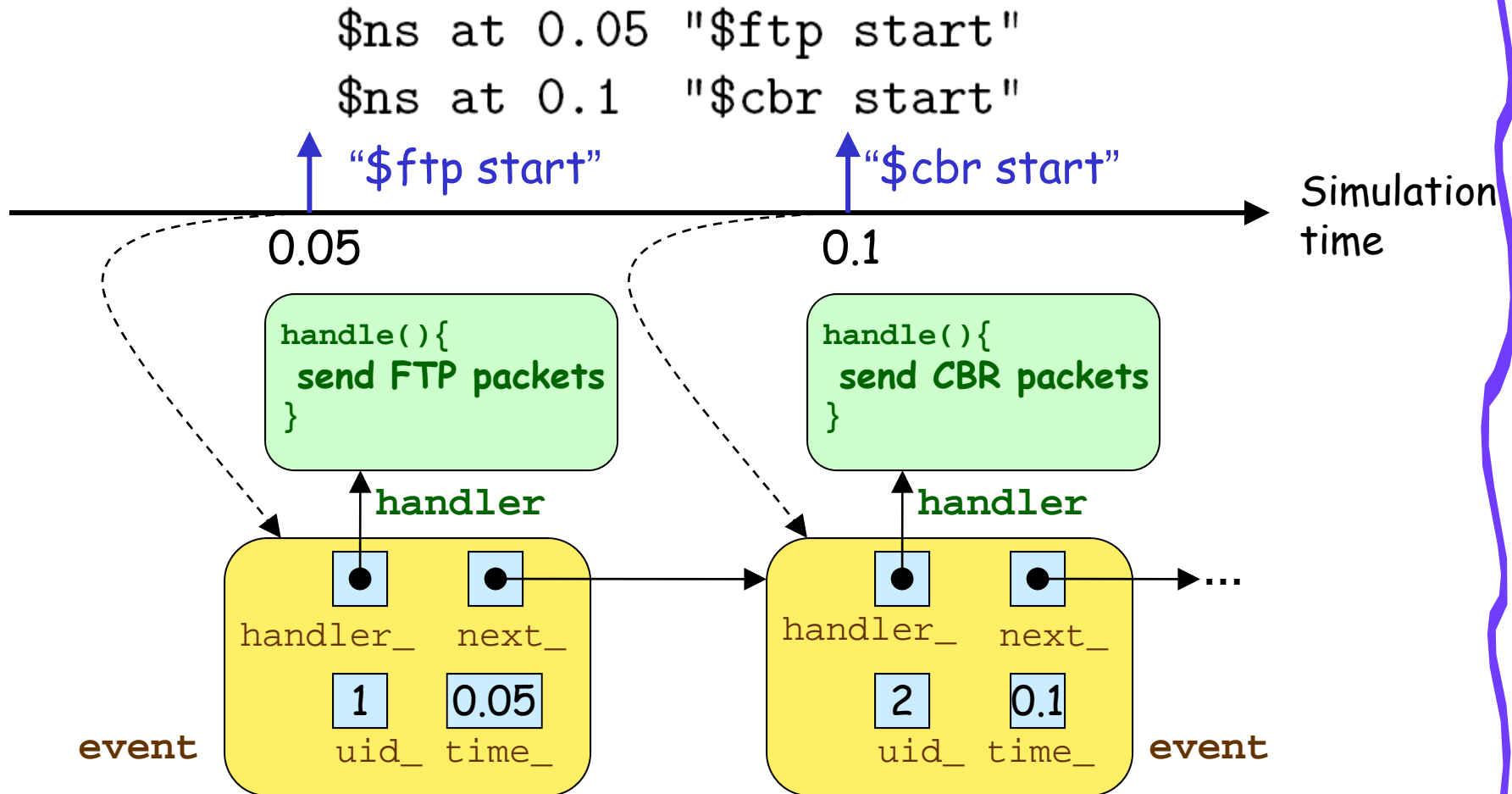
- Class `NsObject` (derived from class `Handler`)

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

- As we shall see, all network objects (e.g., `Connector`, `TcpAgent`) derived from class `NsObject`.
- Default action of all network objects is "to receive (using function `recv(...)`) a packet (cast from an event `e`)"

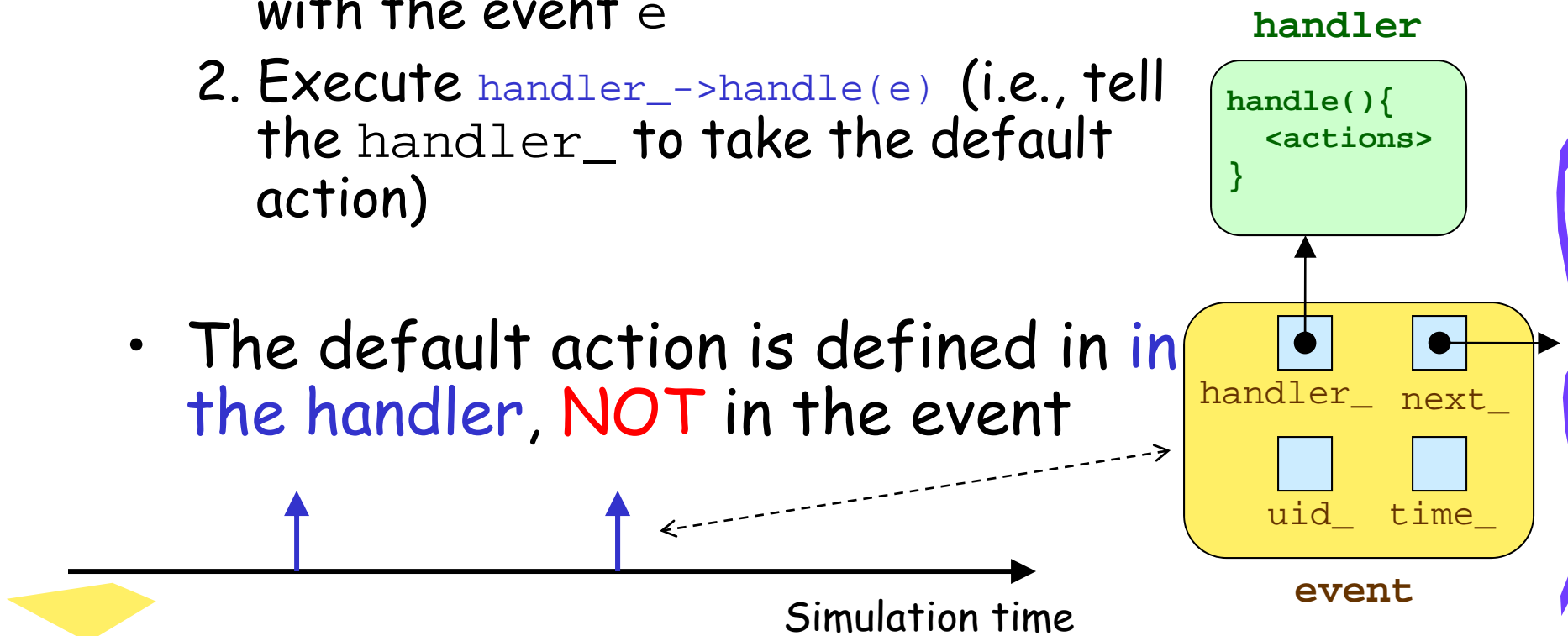


# Events and Handlers: Example



# Events and Handlers: Example

- When hitting an event  $e$ , a Scheduler
  1. Extract the `handler_` associated with the event  $e$
  2. Execute `handler_->handle(e)` (i.e., tell the `handler_` to take the default action)
- The default action is defined in **in the handler, NOT in the event**



# Question

- What is the main purpose of events ?
- What happen if NS2 does not define **classes** Event, Handler, and Scheduler?



# Event and Handler: Outline

- Overview
- C++ Classes Event and Handler
- Two Main Types of Events
  - AtEvent
  - Packet (Discussed Later)



# Two Types of Events

## 1. At Event: (Derives from Class `Event`)

- Action: Execute an OTcl command
- Examples:

```
$ns at 0.05 "$ftp start"
```

```
$ns at 0.1 "$cbr start"
```

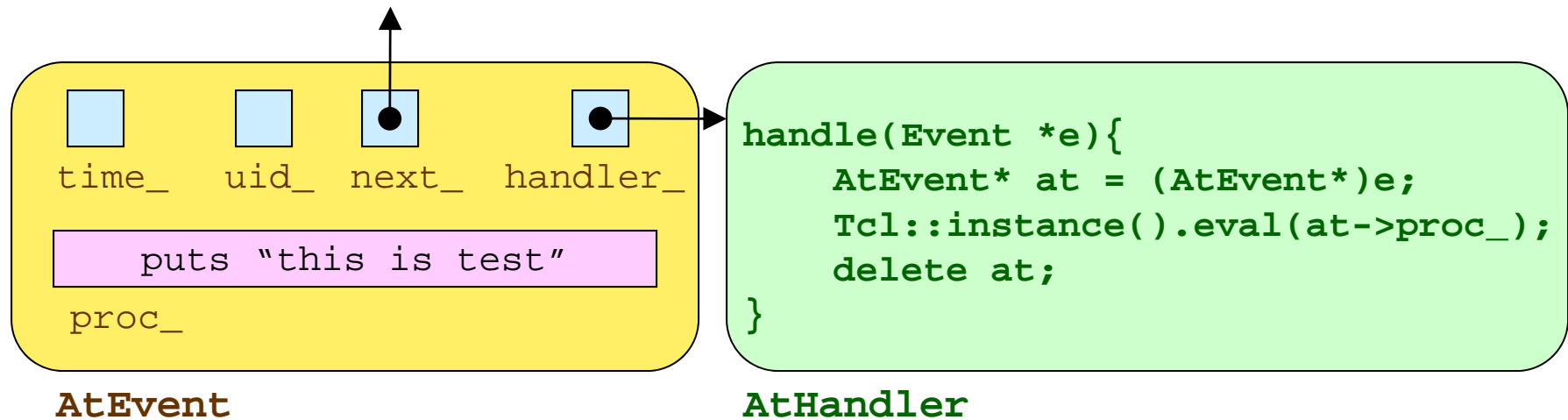
```
$ns at 60.0 "$ftp stop"
```

- C++ Class `AtEvent`
- Placed on the simulation timeline by `instproc` "at" with syntax

```
$ns at <time> <Tcl command>
```

# C++ Class AtEvent

```
class AtEvent : public Event {  
public:  
    AtEvent() : proc_(0) {}  
    char* proc_;  
};
```



# C++ Class AtEvent

- OTcl command: `$ns at <time> <Tcl command>`
- Implementation:

```
Scheduler::command(int argc, const char*const* argv)
{
    Tcl& tcl = Tcl::instance();
    if (argc == 4) {
        if (strcmp(argv[1], "at") == 0) {
            double delay, t = atof(argv[2]); const char* proc = argv[3];
            AtEvent* e = new AtEvent; int n = strlen(proc);
            e->proc_ = new char[n + 1];
            strcpy(e->proc_, proc);
            delay = t - clock();
            schedule(&at_handler, e, delay);
            return (TCL_OK);
        }
    }
    return (TclObject::command(argc, argv));
}
```

Q: `argv[0] = ? ( )`

# Two Types of Events

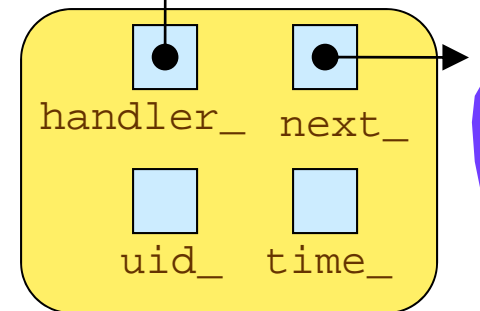
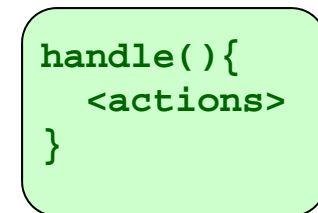
## 2. Packet: (Derives from Class Event)

- Action: Receive a packet

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

- C++ Class Packet (will be discussed later)

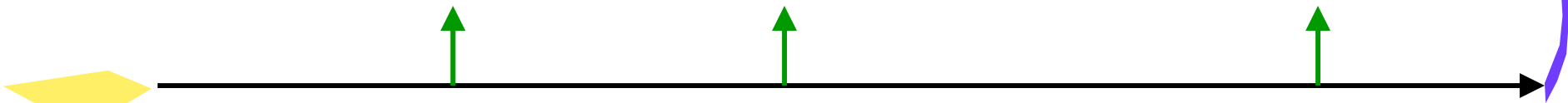
NsObject → ??



Event → Packet

# Questions

- Q: How do we put an `AtEvent` on the simulation timeline? ( )
- Q: Is it possible to put a `Packet` on the simulation timeline? Why or why not? ( )
- How do we put events on the simulation timeline? → Use **THE SCHEDULER**



# Outline

- Recap: Discrete Event v.s. Time Driven
- Events and Handlers
- The Scheduler
- The Simulator
- Summary

# The Scheduler: Outline

- Overview
- C++ Class Scheduler
- Unique ID and Its Mechanism
- Scheduling and Dispatching Mechanism
- Null Events and Dummy Events



# Event Handling: Recap

1. Put events on the simulation timeline
2. Take the default action assoc. with (i.e., handle) event → Handler
  - Also called "fire" or "dispatch"
  - function `handle()` of class `Handler`
3. Move to the next event → Scheduler
  - Through the pointer "`next_`" of an `Event` object

How do we "PUT", "TAKE", and "MOVE"?

# Recap

- Event  $e$  = An indication of future event
- Handler defines the default action (i.e., how to execute the event  $e$ ; `handler(e)`)
- NS2 moves forwards in time and tell the relevant handler to execute default actions.
- Execute = Fire = Dispatch
- What's more?
  - How to put an event on the simulation timeline?
  - Who should execute the actions assoc. with the event?

→ THE SCHEDULER

# The Scheduler

1. Put events on the simulation timeline

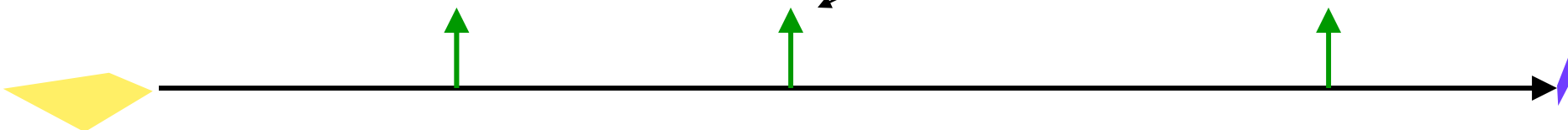
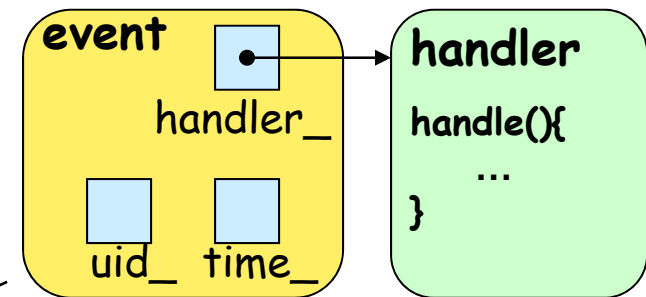
→ function `schedule(...)`

2. Take the default action

→ function `dispatch(...)`

3. Move forward in time

→ function `run(...)`



# C++ Class Scheduler



```
class Scheduler : public TclObject {
public:
    static Scheduler& instance() { return (*instance_); }
    void schedule(Handler*, Event*, double delay);
    virtual void run();
    virtual void cancel(Event*) = 0;
    virtual void insert(Event*) = 0;
    virtual Event* lookup(scheduler_uid_t uid) = 0;
    virtual Event* deque() = 0;
    virtual const Event* head() = 0;
    double clock() const { return (); }
    virtual void reset();

protected:
    void dispatch(Event*);
    void dispatch(Event*, double);
    Scheduler();
    virtual ~Scheduler();
    int command(int argc, const char*const* argv);
    double clock_;
    static Scheduler* instance;
    static scheduler_uid_t uid_;
    int halted_;
};
```

Current virtual time

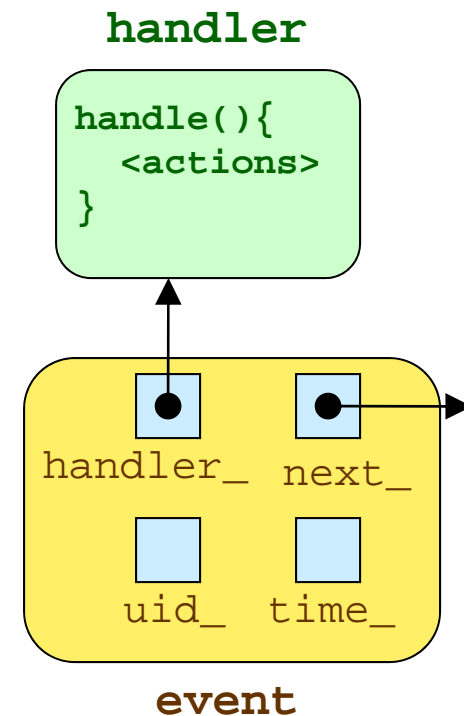
Unique ID: incremented for every new event

??



# Task 1: Put Event on the Simulation Timeline

- Use function `schedule(h, e, delay)`
  - Associate Event "e" with a handler "h"
  - Indicate the dispatching time
  - Assign unique ID
  - Put the Event "e" on the simulation time with delay "delay"



# Functions schedule(.)

```
void Scheduler::schedule(Handler* h, Event* e, double delay)
{
    < Checking for Error >
    e->uid_ = uid_++;
    e->handler_ = h;
    double t = clock_ + delay;
    e->time_ = t;
    insert(e);
}
```

New unique ID

Bind "e" and "h"

Update time

Put "e" on the time line

# Function schedule(.)

- 4 Possible errors

1. Null handler (i.e.,  $h = 0$ )

```
if (!h) { /* error: Do not feed in NULL handler */ };
```

We will talk about this error later

2. `uid_ of the event > 0` → Something wrong

```
if (e->uid_ > 0) {  
    printf("Scheduler: Event UID not valid!\n\n");  
    abort();  
}
```

This is a very common error message!!

# Function schedule(.)

- 4 Possible errors

3.  $\text{delay} < 0 \rightarrow$  Go back in time

```
if (delay < 0) { /* error: negative delay */ };
```

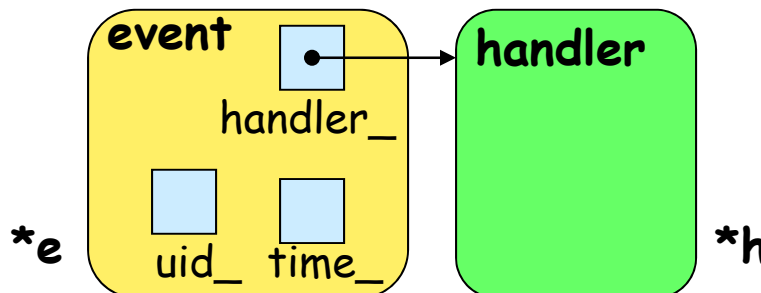
4.  $\text{uid}_ < 0 \rightarrow$  Use up the  $\text{uid}_$

```
if (uid_ < 0) {  
    fprintf(stderr, "Scheduler: UID space exhausted!\n")  
    abort();  
}
```

## Task 2: Take Default Actions

- NS2 “dispatches” a relevant handler to take default actions.

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



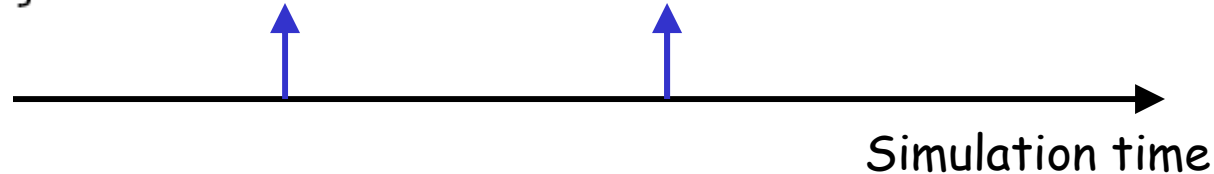
Why put negative?  
→ We will discuss about the sign of uid\_ later.

## Task 3: Move from One Event to the Next

- Function `run()` starts the simulation

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

Take the "next" event  
from the queue of  
events



# The Scheduler: Outline

- Overview
- C++ Class Scheduler
- Unique ID and Its Mechanism
- Scheduling and Dispatching Mechanism
- Null Events and Dummy Events



# Two types of Unique ID (UID)

- 1. Scheduler:
  - Global UID
  - Track the number of created UID

```
class Scheduler : public TclObject {  
public:  
    ...  
    static scheduler_uid_t uid_  
};
```

- 2. Event:
  - Individual UID
  - Event ID
  - Assigned by the Scheduler

```
class Event {  
public:  
    ...  
    scheduler_uid_t uid_;  
};
```

# Global UID

- A member variable of class Scheduler
- Always Positive
- Incremented for every new event (fn schedule(.))

```
void Scheduler::schedule(Handler* h, Event* e, double delay)
{
...
    if (uid_ < 0) {
        fprintf(stderr, "Scheduler: UID space exhausted!\n");
        abort();
    }
    e->uid_ = uid_++;
...
}
```



# Individual UID

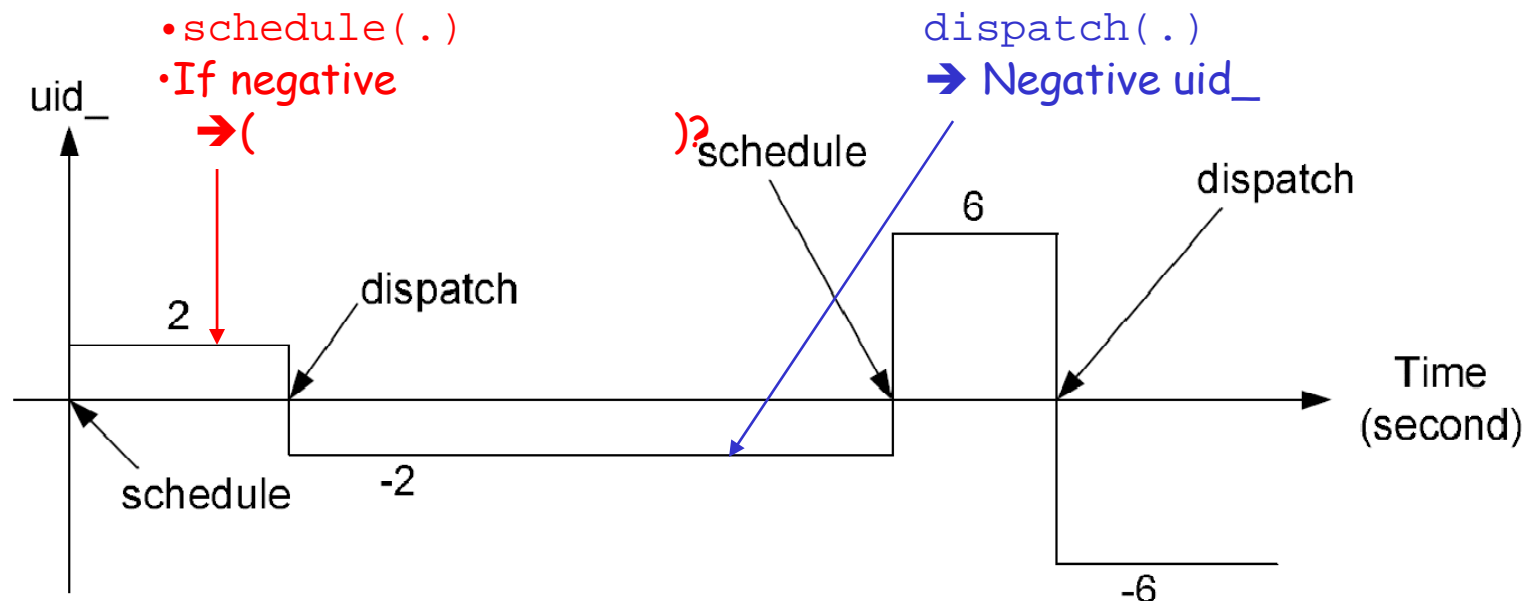
- Unique to each event
  - Set by the Scheduler
  - Assigned by the Scheduler within fn `schedule(.)`
  - Negated by the invocation of fn `dispatch(.)`

```
void Scheduler::schedule(Handler* h, Event* e, double delay)
{
    if (e->uid_ > 0) {
        printf("Scheduler: Event UID not valid!\n\n");
        abort();
    }
    e->uid_ = uid_++;
}
```

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

# Individual UID

- Unique to each event
  - Positive: assigned by fn `schedule(.)`
  - Negative: dispatched fn `dispatch(.)`
  - Dynamics: `uid_` is switching between +/- values



# Individual UID

- Positive UID
  - The event is on the simulation time line.
  - It is waiting to be executed.
  - Rescheduling the (undispatched) event here would **result in an error**

**uid\_ of the event > 0 → Something wrong:**

```
if (e->uid_ > 0) {  
    printf("Scheduler: Event UID not valid!\n\n");  
    abort();  
}
```

# Individual UID

- Positive UID
  - The event is on the simulation time line.
  - It is waiting to be executed.
  - Rescheduling the (undispatched) event here would result in an error
- Negative UID
  - The event has been executed.
  - It is ready to be rescheduled.



# The Scheduler: Outline

- Overview
- C++ Class Scheduler
- Unique ID and Its Mechanism
- Scheduling and Dispatching Mechanism
- Null Events and Dummy Events



# The Scheduler: Outline

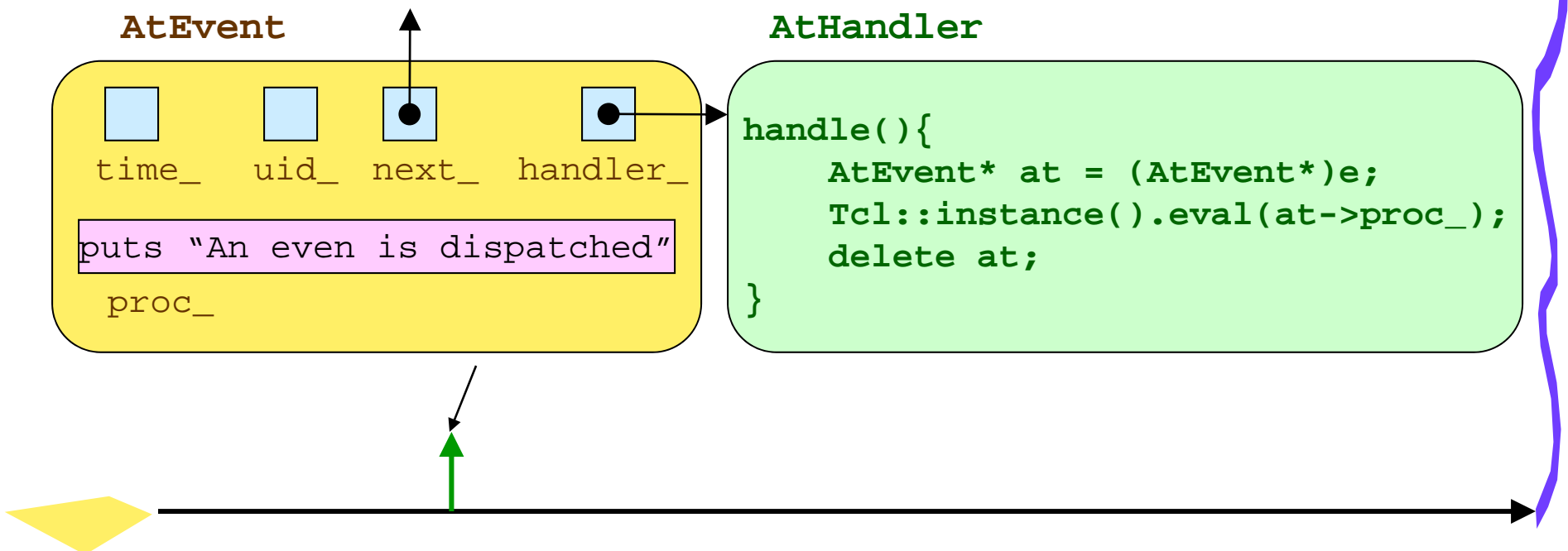
- Overview
- C++ Class Scheduler
- Unique ID and Its Mechanism
- Scheduling and Dispatching Mechanism
- Null Events and Dummy Events



# Scheduling-Dispatching Mechanism

- Example:

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



# Scheduling-Dispatching Mechanism

```
Scheduler::command(int argc, const char*const* argv)
{
    Tcl& tcl = Tcl::instance();
    if (argc == 4) {
        if (strcmp(argv[1], "at") == 0) {
            double delay, t = atof(argv[2]);
            const char* proc = argv[3];
            AtEvent* e = new AtEvent; int n = strlen(proc);
            e->proc_ = new char[n + 1];
            strcpy(e->proc_, proc);
            delay = t - clock();
            schedule(&at_handler, e, delay);
            return (TCL_OK);
        }
    }
    return (TclObject::command(argc, argv));
}
```



# The Scheduler: Outline

- Overview
- C++ Class Scheduler
- Unique ID and Its Mechanism
- Scheduling and Dispatching Mechanism
- Null Events and Dummy Events



# Null and Dummy Events Scheduling

- In general, we feed the event into the Scheduler.
- The event contains
  - Time where the event occurs, and
  - Ref. to an action taker (i.e., the handler)
- Example
  - Event = Packet
  - Time = Time where the packet is received
  - Default action = Receive a packet
  - Action taker = NSObject
- In some case, we the default action involves no event.
- E.g., Print a string after a certain delay
- What event would we feed to the function

`Scheduler::schedule(handler, event, delay) ?`

**Q: delay = ?; handler = ?; event = ?**

# Null and Dummy Events Scheduling

- Null Event: `set event = 0`

`Scheduler::schedule(handler, 0, delay)`

- Dummy Event:

- A member variable whose type is Event
- It does nothing but being placed in function

`schedule(handler, dummy_event, delay)`



# Null and Dummy Events Scheduling

- Dummy event example: class `LinkDelay`

```
//~ns/link/delay.h
class LinkDelay : public Connector {
    ...
    Event intr_;
};

//~ns/link/delay.cc
void LinkDelay::recv(Packet* p, Handler* h)
{
    ...
    s.schedule(h, &intr_, txt);
}
```

## Null and Dummy Events Scheduling

- Which one should we use? Null or Dummy?
- Null events
  - Simple, but no mechanism to preserve `uid_` conformance
  - You lose the scheduling-dispatching protection mechanism.
  - Suitable for simple cases
- Dummy events
  - Require a declaration in a class.
  - A bit more complicated, but will conform with NS2 scheduling-dispatching mechanism
  - Suitable for more complicated cases

# Outline

- Recap: Discrete Event v.s. Time Driven
- Events and Handlers
- The Scheduler
- The Simulator
- Summary

# The Simulator

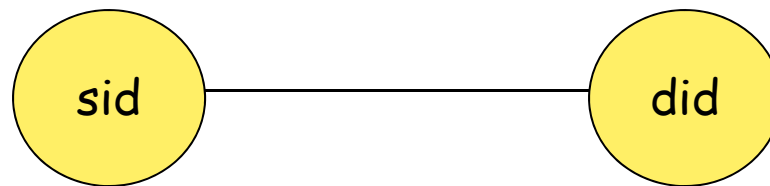
- Maintain assets which are **shared** among simulation objects
  - The schedulers → Event scheduling
  - The null agent → Packet destruction
  - Node reference → All nodes
  - Link reference → All links
  - Ref.to the routing component → Routing
- It does not do the above **functionalities**.
- It only provide the **ref.** to the obj which does the above functionalities

Q:What is an advantage of putting the ref. to the Simulator?



# The Simulator

- OTcl and C++ Classes `Simulator`
- OTcl Instvar
  - `scheduler_`: The scheduler
  - `nullAgent_`: The packet destruction object
  - `Node_(<nodeid>)`: stores node objects
  - `link_(sid:did)`: stores link objects connecting two nodes
  - `routingTable_`: stores the routing component



# C++ Class Simulator

```
//~ns/common/simulator.h
class Simulator : public TclObject {
public:
    static Simulator& instance() { return (*instance_); }
    Simulator() : nodelist_(NULL),
                rtobject_(NULL), nn_(0), size_(0) {}
    ...
private:
    ParentNode **nodelist_;
    RouteLogic *rtobject_;
    int nn_;
    int size_;
    static Simulator* instance_;
};
```

- **Function** `instance()`: Retrieve the static Simulator `instance_`.

# Retrieving the Simulator Instance

- **Instproc** instance{}

```
//~ns/tcl/lib/ns-lib.tcl
Simulator proc instance {} {
    set ns [Simulator info instances]
    if { $ns != "" } {
        return $ns
    }
    ...
}
```

- Q: What does info instances do?
- Q: Can it return more than one Simulator instance? Why? If so, which one do we choose?



# Running Simulation

- **Creating a Simulator object**

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

- **OTcl constructor:**

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

- **`$ns` is now a Simulator instance**



# Running Simulation

- Main instproc `run{}`: Start simulation

```
//~/ns/tcl/lib/ns-lib.tcl
Simulator instproc run {
    [$self get-routelogic] configure
    $self instvar scheduler_ Node_ link_ started_
    set started_ 1
    foreach nn [array names Node_] {
        $Node_($nn) reset
    }
    foreach qn [array names link_] {
        set q [$link_($qn) queue]
        $q reset
    }
    return [$scheduler_ run]
}
```



# Running Simulation

- Scheduler::run{ }

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

- Keep executing events until
  - no more event or
  - the simulation is halted

# Instprocs of Class Simulator

Instproc	Meaning
<code>now{}</code>	Retrieve the current simulation time.
<code>nullagent{}</code>	Retrieve the shared null agent.
<code>use-scheduler{type}</code>	Set the type of the Scheduler to be <code>&lt;type&gt;</code> .
<code>at{time stm}</code>	Execute the statement <code>&lt;stm&gt;</code> at <code>&lt;time&gt;</code> second.
<code>run{}</code>	Start the simulation.
<code>halt{}</code>	Terminate the simulation.
<code>cancel{e}</code>	Cancel the scheduled event <code>&lt;e&gt;</code> .

# Outline

- Recap: Discrete Event v.s. Time Driven
- Events and Handlers
- The Scheduler
- The Simulator
- Summary

# Summary

- NS2 Simulator is Event Driven
- Event
  - Unique ID + Time + Handler
  - Two derived classes: ( )
- Handlers
  - ( )
  - ( )



# Summary

- Scheduler
  - schedule(.): ( )
  - dispatch(.): ( )
  - run(): ( )
- Event UID Dynamics
  - schedule() → +,
  - dispatch() → -



# Summary

- Null event and Dummy Event
  - Purpose: ( )
  - Differences:
    - Null Event = ( )
    - Dummy Event = ( )
- Simulator
  - Maintain all common objects: Scheduler, null agent, nodes, links, and routing table
  - Start the simulation (e.g., "\$ns run")

