Note For NS2: How NS2 Connects Two Nodes

Teerawat Issariyakul

July 19, 2006
Chapter 1

Introduction

1.1 Overview

This note explains how NS2 really connects two nodes.

- Basically, it just uses a Tcl object called "link" to connect two nodes: \( n_1 \rightarrow \text{Link} \rightarrow n_2 \).
- Links consists of three main parts (inside the square bracket): \( n_1 \rightarrow [\text{head} \rightarrow \text{queue} \rightarrow \text{link} \rightarrow \text{ttl}] \rightarrow n_2 \).
- To connect two objects, NS2 uses a based class \text{Connector} (implemented in C++). Here all three main components—\text{queue}, \text{link}, \text{and ttl}—derive from class \text{Connector}.

1.2 Basic Tcl Script

- Use command 
  \[
  \$\text{ns simplex-link } n_1 \ n_2 <\text{bw}> <\text{delay}> <\text{qtype}> <\text{args}>.
  \]
- \( [f=\text{ns-lib.tcl};c=\text{Simulator};f=\text{simplex-link};\text{asm= droptail and no } <\text{args}>]: \)
  
  (i) Create a queue object: 
  \[
  \text{set q [new Queue/}\$\text{qtype]}
  \]
  
  (ii) Create a link object: 
  \[
  \text{set sid [}\$\text{n1 id]}; \text{set did [}\$\text{n2 id] and set link(}\$\text{sid:}\$\text{did) [new SimpleLink } n_1 \ n_2 <\text{bw}> <\text{delay}> <\text{q}]
  \]
- C++: \text{Connector} \rightarrow \text{Queue} \rightarrow \text{DropTail}.
Chapter 2

Connector

2.1 Overview

- **Location:** ns2/common/connector.*.

- **Functionality:** connect two NsObjects; provide a dropping point (e.g., to dump a packet in case of a packet drop).

- When a connector receives a packet, it will call function `recv(.)`. Different child classes overwrite the function `recv(.)`. However, most likely, almost every child class will again call `target_->recv(.)` or `drop_->recv(.)`.

2.2 Main Member Variables

- `NsObject* target_, drop_; int dynamic_; Event intr_; PacketQueue* itq_`

- `itq` is a pointer to a linklist of packets. It is used for local manipulation of packets received.
2.3 Main Public Function

2.3.1 Function inline NsObject* target() { return target_; }

2.3.2 Function void target (NsObject *target) { target_ = target; }

2.3.3 Function virtual void drop(Packet* p);

2.3.4 Function void setDropTarget(NsObject *dt) {drop_ = dt; }

2.4 Main Protected Function

The child classes will inherit these functions as public functions.

2.4.1 Function void recv(Packet* p, Handler* h) {send(p,h)}

2.4.2 Function inline void send(Packet* p, Handler* h)

        target_->recv(p, h);

2.4.3 Function virtual void drop(Packet* p, const char *s);

        if (drop_ != 0)
        drop_->recv(p, s);
        else
        Packet::free(p);
Chapter 3

Simple Link

- Tcl: Link → SimpleLink; Class Link has a variable n1_ which store the id for the next link.
- [fl=ns-link.tcl;cl=SimpleLink;fn=init;asm= droptail and no <args>]:

3.1 Class Link (fl:ns-link.tcl)

- Member Variables: id_, fromNode_, toNode_
- Fn: init {src dst}

3.2 Tcl Class SimpleLink ← Link (fl:ns-link.tcl)

- Additional Member Variables: link_, queue_, head_, ttl_, drophead_
- Fn=init { src dst bw delay q {lltype "DelayLink"}}
  
  (i) Call base constructor:
  $self next $src $dst
  
  (ii) Create a connector drophead_ and point its target_ to NULL; drophead_ connects a
  SimpleLink object to a drop target:
  set ns [Simulator instance]
  set drophead_ [new Connector]
  $drophead_ target [$ns set nullAgent_]

\[the\ default\ value\ of\ lltype\ is\ DelayLink.\]
(iii) Create a connector head. Include a variable link as a member of head. Set head\_.link as the current SimpleLink object; head connects an outside object to a SimpleLink object:

```
set head_ [new Connector]
$head_ set link_ $self
```

(iv) Connect head\_\rightarrow queue\_\rightarrow link\_\rightarrow [$dst entry]::

```
$head_ target $q
set queue_ $q
set link_ [new $lltype]
$link_ set bandwidth_ $bw
$link_ set delay_ $delay
$queue_ target $link_
$link_ target [$dst entry]
$queue_ drop-target $drophead.
```

From the above the default $lltype is DelayLink.

(v) Insert ttl between link and [$dst entry]; ttl is used to check the time to live of each packet:

```
set ttl_ [new TTLChecker]
$ttl_ target [$link_ target]
$self ttl-drop-trace
$link_ target $ttl_
```
Chapter 4

C++ Class LinkDelay ← Connector (fl:link/delay.*)

This class is implemented in C++.

4.1 Main Member Variables

- double bandwidth_, delay_; int dynamic_; Event intr_; PacketQueue* itq_.
  - itq is a pointer to a linklist of packets. It is used for local manipulation of packets received.

4.2 Binded Variables (C++ ⇔ (<type>) Tcl)

- bandwidth_ ⇔ (bw) bandwidth_
- delay_ ⇔ (time) delay_
- avoidReordering_ ⇔ (bool) avoidReordering_;

4.3 Main Public Function

4.3.1 Function void recv(Packet* p, Handler* h);
  - Assumptions: Link is no dynamic; We do not want to avoid packet reordering.
• h is the pointer to the upstream queue. It is actually of class QueueHandle.
• s.schedule(target_, p, txt + delay_): Let the target handle the packet at txt + delay. after.
• s.schedule(h, &intr_, txt);: This line puts an event on the scheduler. At the right time, the scheduler will call h->handle(&intr_). This line will call QueueHandle::handle(), which simply runs only one command queue_.resume();. queue_.resume(); will caused the queue object to become unblocked (see detail in the class Queue).

4.3.2 Function void send(Packet* p, Handler*);
Simply call target_->recv(p, (Handler*) NULL).

4.3.3 Function void handle(Event* e);
• Called by the scheduler at the proper time.
• E.g., if the link is dynamic, the LinkDelay delays the handling of the received packet for for a period of transmission time (txt) plus propagation delay (delay_). In the recv(.), the delay is implemented by scheduling the packet handling at time after:

Scheduler& s = Scheduler::instance();
if (dynamic_) {
  Event* e = (Event*)p;
  e->time_= txt + delay_;
  itq_->enque(p); // for convinience, use a queue to store packets in transit
  s.schedule(this, p, txt + delay_);
}

4.3.4 Function inline double txtime(Packet* p)
Return the transmission time of the packet <*p>. 
Chapter 5

Queue

5.1 Overview

- This class is implemented in C++.
- Consist of three main classes
  
  (i) **Queue**: Based class with functionalities `enqueue()`, `dequeue()`, and so on,
  
  (ii) **QueueHandler** provides the queue reference to another (the next downstream) object, and , and **PacketQueue** maintains the link-list of packets stored in the queue. In fact, the “real” packets are not stored in the **Queue** but in **PacketQueue**. Class **Queue** indicates when to enqueue/dequeue a packet, while **PacketQueue** tells which packet should be enqueued.

5.2 Queue Operation

The fundamental point of the queue is the service time. The service time starts from when the queue starts transmitting a packet and ends when the packet is completely out of the queue. The service time depends on bandwidth and packet length (and in some case on propagation time).

When the queue is transmitting a packet, no other packet is allowed to be transmitted (obviously). NS2 uses a variable `block_` to indicate whether a queue is transmitting a packet. `block_` is set to one if a packet is in the head of the queue (i.e., queue is transmitting a packet). When the transmission is complete, the downstream object, which in most cases would be **LinkDelay** who has a point referenced to the queue, will call the function `resume()` of the queue to reset `block_` to zero. This will allow queue to transmit another head of line packet.
5.2.1 Function `recv(Packet* p, Handler* h)`

```c
double now = Scheduler::instance().clock();
enqueue(p);
if (!blocked_) {
    /*
     * We're not blocked. Get a packet and send it on.
     * We perform an extra check because the queue
     * might drop the packet even if it was
     * previously empty! (e.g., RED can do this.)
     */
    p = dequeue();
    if (p != 0) {
        utilUpdate(last_change_, now, blocked_);
        last_change_ = now;
        blocked_ = 1;
        target_->recv(p, &qh_);
    }
}
```

Basically, it just enques (enqueue(p)) a packet. If the queue is not blocked, the deque (p = dequeue();), transmit (target_->recv(p, &qh_)), and block (blocked_ = 1;) the queue.

Note that when transmitting, the queue sends the pointer to its handler qh_ so that the downstream object will tell the queue to resume after the packet transmission process is complete.

qh_ is a pointer to an object of class QueueHandler. We shall see the detail in the next section.

5.2.2 Function `resume()`

```c
double now = Scheduler::instance().clock();
Packet* p = dequeue();
if (p != 0) {
    target_->recv(p, &qh_);
} else {
    if (unblock_on_resume_) {
        utilUpdate(last_change_, now, blocked_);
        last_change_ = now;
        blocked_ = 0;
    }
}
else {
    utilUpdate(last_change_, now, blocked_);
    last_change_ = now;
    blocked_ = 1;
}
}

What we need to know now is that, upon `resume()`, the queue `deque(Packet* p = deque();)` a packet and transmit (`target_->recv(p, &qh_);`) it to the downstream object.

### 5.3 QueueHandler

#### 5.3.1 QueueHandler Functionality

- Derived from `Handler` class, `QueueHandler` overwrites the method `handle(Event*)`.

- Basically, NS2 schedules (i.e., put an event on the time line) a call for `QueueHandler` using the method `schedule()`. At the proper time (e.g., when transmission process is complete), the scheduler fires the event `QueueHandler::handle()`, which is

  ```
  void QueueHandler::handle(Event*)
  {
      queue_.resume();
  }
  ```

- Intuitively, we need to provide a connection between queue and its handler.

#### 5.3.2 Connection to Queue

- Use private variable `Queue& queue_;`

- In the constructor, `inline QueueHandler(Queue& q) : queue_(q)`

- This constructor is called by class `Queue` which provide itself as an input argument.

- Class `Queue` has a protected member variable `QueueHandler qh_;`. In the constructor, it is initialized by `qh_(*this)`
5.4 PacketQueue

- Again, this class maintains a link list of packets in queue.

- Protected member variable:

  Packet* head_;  
  Packet* tail_;  
  int len_;        // packet count  
  int bytes_;      // queue size in bytes  

- Each packet has a pointer Packet* next_ defined in class Packet which point to the next packet.

- This way, PacketQueue can maintain the link-list of packets by keeping track of only two variables: head_ and tail_.

- So the main function of PacketQueue include for example enqueue, dequeue, lookup, enqueuehead, etc.