Links and Buffer Management

Outline

• Overview
• Packet Departure
• Buffer Management
• Summary

Links and Simple Links

• Connect two nodes

\[ \text{n0} \quad \text{Link} \quad \text{n1} \]

• Consider “Simple Links” Only
• Simple Links models
  - Packet transmission time = \( \text{pkt\_size}/\text{BW} \)
  - Link propagation delay
  - Buffer management \( \Rightarrow \) Unlike real implementation (buffering is a functionality in routers)

Simple Links: Architecture

- **A stand-alone composite OTcl object**

![Diagram of Simple Links: Architecture]

- **head**: the entry
- **queue**: buffering
- **link**: model pkt transmission time and link propagation delay
- **ttl**: time to live
- **drophead**: dropping point

Simple Links: Architecture

- A stand-alone composite OTcl object

- *enqT*_ (+) = Trace pkt entering queue_
- *deqT*_ (-) = Trace pkt leaving queue_
- *drpT*_ (d) = Trace pkt dropped from queue_ (Buffer overflow)
- *rcvT*_ (r) = Trace pkt received by the terminating object

Simple Links: Architecture

• An example of trace file

<table>
<thead>
<tr>
<th>Type Identifier</th>
<th>Time</th>
<th>Source Node</th>
<th>Destination Node</th>
<th>Packet Name</th>
<th>Packet Size</th>
<th>Flags</th>
<th>Flow ID</th>
<th>Source Address</th>
<th>Destination Address</th>
<th>Sequence Number</th>
<th>Packet Unique ID</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.11308</td>
<td>0</td>
<td>2</td>
<td>cbr</td>
<td>1000</td>
<td>------</td>
<td>---------</td>
<td>1 0.0 3.0 2 8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>0.11308</td>
<td>2</td>
<td>3</td>
<td>cbr</td>
<td>1000</td>
<td>------</td>
<td>---------</td>
<td>1 0.0 3.0 2 8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>0.11308</td>
<td>2</td>
<td>3</td>
<td>cbr</td>
<td>1000</td>
<td>------</td>
<td>---------</td>
<td>1 0.0 3.0 2 8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r</td>
<td>0.11316</td>
<td>0</td>
<td>2</td>
<td>cbr</td>
<td>1000</td>
<td>------</td>
<td>---------</td>
<td>1 0.0 3.0 3 9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>0.11316</td>
<td>2</td>
<td>3</td>
<td>cbr</td>
<td>1000</td>
<td>------</td>
<td>---------</td>
<td>1 0.0 3.0 3 9</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- enqT_ = Trace enqueue, “+”
- deqT_ = Trace dequeue, “-”
- rcvT_ = Trace receive, “r”
- drpT_ = Trace packet drop, “d”, (Buffer overflow)

Simple Links: Creation

- From the OTcl (e.g., Tcl simulation script)

```tcl
$ns simplex-link $n1 $n2 <bandwidth> <delay> <queue_type>
$ns duplex-link $n1 $n2 <bandwidth> <delay> <queue_type>
```

- **simplex-link**: Create a one-directional simple link object
- **duplex-link**: Create two simplex-links in two directions

What are simple-link and duplex-link??

- Examples

```tcl
$ns duplex-link $n0 $n2 100Mb 5ms DropTail
$ns duplex-link $n1 $n2 100Mb 5ms DropTail
$ns duplex-link $n2 $n4 54Mb 10ms DropTail
```

Simplex Links

• Instproc simplex-link of class simulator
  - Create a uni-directional SimpleLink object

```tcl
//~ns/tcl/lib/ns-lib.tcl
Simulator instproc simplex-link { n1 n2 bw delay qtype args } {
    $self instvar link_ queueMap_ nullAgent_ useasim_
    switch -exact $qtype {
        /* See the detail in ~ns/tcl/lib/ns lib.tcl */
        default {
            set q [new Queue/$qtype $args]
        }
    }
    switch -exact $qtype0rig {
        /* See the detail in ~ns/tcl/lib/ns-lib.tcl */
        default {
            set link_($sid:$did) [new SimpleLink \
                $n1 $n2 $bw $delay $q]
        }
    }
}
$ns duplex-link $n0 $n2 100Mb 5ms DropTail
```

SimpleLink: OTcl Constructor

//~ns/tcl/lib/ns-link.tcl
SimpleLink instproc init { src dst bw delay q {  
   lltype "DelayLink"} } {  
   set ns [Simulator instance]  
   set drophead_ [new Connector]  
   $drophead_ target [$ns set nullAgent_]  
   set head_ [new Connector]  
   $head_ target $q  
   set queue_ $q  
   ...  
}

What is the class of drophead_? ( )
What is “target”? ( )

SimpleLink: OTcl Constructor

//~ns/tcl/lib/ns-link.tcl
SimpleLink instproc init { src dst bw delay q { 
lltype "DelayLink"} } {

set link_ [new $lltype]
$link_ set bandwidth_ $bw
$link_ set delay_ $delay
$queue_ target $link_
$link_ target [$dst entry]
$queue_ drop-target $drophead_ 
set ttl_ [new TTLChecker]
$ttl_ target [$link_ target]
$self ttl-drop-trace
$link_ target $ttl_
}

What is the class of $lltype_?
(DelayLink)

$ns duplex-link $n0 $n2 100Mb 5ms DropTail

Outline

• Overview
• Packet Departure
• Buffer Management
• Summary

Recap: Packet Forwarding

- Sending packet $\ast p$ to node $\ast target$
- Two types of packet forwarding
  1. Immediate packet forwarding
     - $\Rightarrow target->recv(p,h)$
  2. Delayed packet forwarding
     - $\Rightarrow schedule(target, p, delay)$

Packet Departure Process

- The amount of time needed to deliver a packet to the next node
  - Actual implementation
    • From: A packet enters the router A
    • To: The packet reaches the router B

- NS2 implementation
  • From: A packet enters the Link
  • To: A packet leaves the Link
  • Note: A packet is assumed to be at Node B as soon as it leaves the Link

Packet Departure Process

• The time consists of
  1. Packet transmission time
     - Time a packet spend inside router A
     - Pkt_size/link_speed

  2. Link propagation delay
     - Start after the entire packet leaves router A
     - Time the packet spend in the link

Packet Departure Process

- Which components implement the time? (link)
- C++ Class = LinkDelay
- OTcl Class = DelayLink

C++ Class LinkDelay

//~ns/link/delay.h
class LinkDelay : public Connector {
    public:
        LinkDelay(): dynamic_(0), latest_time_(0), itq_(0){
            bind_bw("bandwidth_", &bandwidth_);
            bind_time("delay_", &delay_);
        }
        void recv(Packet* p, Handler*);
    inline double txtime(Packet* p) { /* Packet TXT Time */
            return (8. * hdr_cmn::access(p)->size() / bandwidth_);
        }
    protected:
        double bandwidth_; 
        double delay_; 
        PacketQueue* itq_; 
        Event intr_; /* In transit */
};

C++ Class LinkDelay

<table>
<thead>
<tr>
<th>Variable</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>bandwidth_</td>
<td>Link speed (in bps)</td>
</tr>
<tr>
<td>delay_</td>
<td>Link propagation delay (in seconds)</td>
</tr>
<tr>
<td>itq_</td>
<td>Contains all packet in the links</td>
</tr>
<tr>
<td>intr_</td>
<td>Dummy event</td>
</tr>
</tbody>
</table>

C++ Class LinkDelay

- **Function** `tctime(p)`
  - **Compute** Packet transmission time
    ```cpp
    inline double tctime(Packet* p) { /* Packet TXT Time */
        return (8. * hdr_cmn::access(p)->size() / bandwidth_);
    }
    ```

- **Function** `recv(p, h)`
  - **Main** packet reception function
  - **Delay** the packet reception at the downstream NsObject
    - Packet transmission time
    - Link propagation delay
  - **How to delay it?** ➔ **Use** ( )

Function LinkDelay::recv(p,h)

```cpp
//~ns/link/delay.cc
void LinkDelay::recv(Packet* p, Handler* h)
{
    double txt = txtime(p);  // (1)
    Scheduler& s = Scheduler::instance();
    if (dynamic_) { /* See ~ns/link/delay.cc */ }
    else if (avoidReordering_) { /* See ~ns/link/delay.cc */ }
    else {
        s.schedule(target_, p, txt + delay_);  // (3)
    }
    s.schedule(h, &intr_, txt);  // (2)
}
```

1. Compute packet transmission time ($\text{txt}$)

**Function LinkDelay::recv(p, h)**

```cpp
//~ns/link/delay.cc
void LinkDelay::recv(Packet* p, Handler* h)
{
    double txt = txtime(p); (1)
    Scheduler& s = Scheduler::instance();
    if (dynamic_) { /* See ~ns/link/delay.cc */ }  
    else if (avoidReordering_) { /* See ~ns/link/delay.cc */ } 
    else {
        s.schedule(target_, p, txt + delay_); (3)
    }
    s.schedule(h, &intr_, txt); (2)
}
```

2. Schedule an event where a packet leave the router

⇒ invoke \(h->\text{handle}(\quad)\) after \(txt\) sec.

**Q: What is the input argument here?**

Function `LinkDelay::recv(p,h)`

```cpp
//~ns/link/delay.cc
void LinkDelay::recv(Packet* p, Handler* h)
{
    double txt = txtime(p);  //1
    Scheduler& s = Scheduler::instance();
    if (dynamic_) { /* See ~ns/link/delay.cc */ }
    else if (avoidReordering_) { /* See ~ns/link/delay.cc */ }
    else {
        s.schedule(target_, p, txt + delay_);  //3
    }
    s.schedule(h, &intr_, txt);  //2
}
```

3. Schedule an event where the packet arrive at the next node after `txt + propagation delay` seconds.

Outline

• Overview
• Packet Departure
• Buffer Management
• Summary

Buffer Management: Outline

- Overview
- Main C++ Classes
- Queue and Queue Handler
- Callback Mechanism

Buffer Management

- Buffer: Put incoming packets in the buffer before emitting them out.

- Why do we need a buffer?

Q: Suppose no buffer, what will happen if packet 2 arrives when the router is serving packet 1?
   (The packet “2” would be lost.)

- Handle burstiness of input traffic.

Buffer Management

- **NS2 implementation**
- **SimpleLink architecture**

- **Instvar** `queue_`
  - Receive packets,
  - Hold them until the link is clear, and
  - Transmit the next packet in queue.

C++ Main Classes

- **C++ Class** PacketQueue:
  - Model the buffer
  - Storage, enqueue, and dequeue packets

- **C++ Class** Queue:
  - Contain PacketQueue
  - Other functionalities, e.g., user interface

- **C++ Class** QueueHandler:
  - Derived from class Handler
  - Purpose: Define default action for a Queue object
Class Packet: Variables

// ~/ns/common/packet.h
class Packet : public Event {
    ...
public:
    Packet* next_;        
    static int hdrlen_;    
    ...
};

- The packet supports link list by itself.
- But it needs a manager:
  - Where the first packet is.
  - The number of packet in the link list.
  ➔ PacketQueue

C++ Class PacketQueue

//~ns/queue/queue.h
class PacketQueue : public TclObject {
    public:
        PacketQueue() : head_(0), tail_(0), len_(0), bytes_(0) {}
        virtual int length() const { return (len_); }
        virtual Packet* enqueue(Packet* p);
        virtual Packet* dequeue();
        virtual void remove(Packet*);
        Packet* head() { return head_; }
        Packet* tail() { return tail_; }
    protected:
        Packet* head_;
        Packet* tail_;
        int len_;
};
C++ Class Queue

```cpp
// ~ns/queue/queue.h
class Queue : public Connector {
    public:
        virtual void enqueue(Packet*) = 0;
        virtual Packet* dequeue() = 0;
        virtual void recv(Packet*, Handler*);
        void resume();
        int blocked() const { return (blocked_ == 1); }
        void unblock() { blocked_ = 0; }
        void block() { blocked_ = 1; }
        int limit() { return qlim_; }
        int length() { return pq_->length(); }
        virtual ~Queue();

    protected:
        Queue();
        void reset();
        int qlim_;
        int blocked_;
        int unblock_on_resume_;
        QueueHandler qh_;
        PacketQueue *pq_;
};

Receiving packets
Callback Mechanism

Buffer Management: Outline

- Overview
- Main C++ Classes
- Queue and Queue Handler
- Callback Mechanism

C++ Class QueueHandler

```cpp
class QueueHandler : public Handler {
public:
    inline QueueHandler(Queue& q) : queue_(q) {}  
    void handle(Event*);
private:
    Queue& queue_;  
};

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

Define default actions

Function `handle(e)`

=> function `resume()` of

the associated Queue object

Queue and QueueHandler

- Highly associated
- Queue accesses a QueueHandler object using a variable $qh_\_$
- QueueHandler accesses a Queue object using a reference to $queue_\_$

- Now Queue and QueueHandler can invoke functions of each other.

Queue and QueueHandler

• To associated a Queue object to a QueueHandler object
• The constructor of Queue: configuring qh_

```c++
//~ns/queue/queue.h
class Queue : public Connector {
    ...
    protected:
        QueueHandler qh_;  
    ...
};
```

Queue and QueueHandler

- The constructor of class `QueueHandler`: configuring `queue_`

```cpp
class QueueHandler : public Handler {
    public:
        inline QueueHandler(Queue& q) : queue_(q) {} 
        void handle(Event*);
    private:
        Queue& queue_;
};
```

- The connection between `Queue` and `QueueHandler` is established

QueueHandler

- Key functionality = ?
  
  - `QueueHandler::handle(Event*)`
    
    ```
    //~ns/queue/queue.cc
    void QueueHandler::handle(Event*)
    {
      queue_.resume();
    }
    ```

- `Queue::resume()`
  
  ➜ **Callback mechanism**
Buffer Management: Outline

• Overview
• Main C++ Classes
• Queue and Queue Handler
• Callback Mechanism

Callback Mechanism: Overview

• Recap:
  - NSObject
    • does not send
    • Tell the downstream object to receive
  - Packets are passed to downstream objects only.

• A signal which passed to an upstream object is called a callback signal
• The process is called “callback mechanism”
• Discussion context: Queue-Link connection

Queue-Link Connection: Implementation

• Main components
  - The Scheduler
  - Packet: base class = (Event)
  - QueueHandler: base class = (Handler)
  - LinkDelay: Put the packet in the simulation timeline

A Queue-Link Connection: Concept

1. queue_ sends a packet to link_
2. link_ receives the packet
3. queue_ waits until link_ indicate that it receives the packet
4. link_ sends a “callback” signal to queue_ after a certain delay time
5. queue_ realizes that the packet is received
6. Go to 1.

Queue-Link Connection:
Delay a callback message

//~ns/link/delay.cc
void LinkDelay::recv(Packet* p, Handler* h)
{
    double txt = txtime(p);
    Scheduler& s = Scheduler::instance();
    if (dynamic_) { /* See ~ns/link/delay.cc */ }  
    else if (avoidReordering_) { /* See ~ns/link/delay.cc */ } 
    else {
        s.schedule(target_, p, txt + delay_); (a)
    }
    s.schedule(h, &intr_, txt); (b)
}

Q: What do (a) and (b) do? When do these actions occur?

Queue-Link Connection: Queue blocking mechanism

• How do a queue_ “wait” for a callback message?
• I.E., How do we stop a Queue object from sending out packets after target->recv(p, &qh_) is executed.

• Queue blocking mechanism
  - Not caused by buffer being filled
  - Use the variable blocked_ of class Queue

```cpp
//~ns/queue/queue.h
class Queue : public Connector {
  ...
  protected:
    int unblock_on_resume_;  
};
```
Queue-Link Connection: Queue blocking mechanism

- **Blocked** \( (\text{blocked}_=1) \)
  - Not being able to send packets
  - After sending
  - Waiting for the callback signal from \text{link}_

- **Not blocked** \( (\text{blocked}_=0) \)
  - Able to send packets
  - After receiving the callback signal from \text{link}_

A Queue-Link Connection: Concept

1. queue_ sends a packet to link_
2. link_ receives the packet
3. queue_ waits until link_ indicate that it receives the packet
4. link_ sends a “callback” signal to queue_ after a certain delay time
5. queue_ realizes that the packet is received
6. Go to 1.

# Queue-Link Connection: Main Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>What it does</th>
<th>Invoked by</th>
</tr>
</thead>
<tbody>
<tr>
<td>resume()</td>
<td>Reception of a callback message</td>
<td>An associated QueueHandler object</td>
</tr>
<tr>
<td>recv(p,h)</td>
<td>Reception of a packet</td>
<td>An upstream object</td>
</tr>
</tbody>
</table>

```cpp
void Queue::resume()
{
    Packet* p = deque();
    if (p != 0)
        target_->recv(p, &qh_);
    else
        if (unblock_on_resume_)
            blocked_ = 0;
        else
            blocked_ = 1;
}
```

```cpp
void Queue::recv(Packet* p, Handler*)
{
    enqueue(p);
    if (!blocked_)
        p = deque();
    if (p != 0)
    {
        blocked_ = 1;
        target_->recv(p, &qh_);
    }
}
```

Queue-Link Connection:
Delay a callback message

//~ns/link/delay.cc
void LinkDelay::recv(Packet* p, Handler* h)
{
    double txt = txtime(p);
    Scheduler& s = Scheduler::instance();
    if (dynamic_) { /* See ~ns/link/delay.cc */ }
    else if (avoidReordering_) { /* See ~ns/link/delay.cc */ }
    else {
        s.schedule(target_, p, txt + delay_);
    }
    s.schedule(h, &intr_, txt);
}
Callback Mechanism in Queue-Link Interaction

Outline

• Overview
• Packet Departure
• Buffer Management
  – Queue and QueueHandler
  – Callback Mechanism
• Summary

Summary

• Link is a composite OTcl object used to connect two nodes

• Class SimpleLink
  - models buffer management and Link delay
  - is created by instrocs simplex-link or duplex-link of class Simulator
  - consists of: head_, queue_, link_, ttl_, and drophead_
Summary

- Packet delay = transmission delay + propagation delay

- NS2 Link
  - C++ Class = LinkDelay
  - OTclClass = DelayLink

- C++ Class LinkDelay
  - Two main variables: bandwidth_ and delay_.
  - Function recv(p, h) schedule two events:
    - Packet departure from the upstream object
    - Packet arrival at the downstream object

Summary

• Queue handle burstiness:
  ➔ Put a packet in queue and sent it out when the transmitter is ready.

• Class Queue and QueueHandler:
  - Queue: A variable qh_
  - QueueHandler: A reference variable queue_

• Callback Mechanism
  - Send a message upstream
  - Normally packet reception is performed in a downstream direction.
  - E.g., a LinkDelay object sends a message to a Queue to indicated that the packet has left the Queue.