TNK092 NETWORK SIMULATION
LAB ASSIGNMENT #2
SEPT. 13th 2011

NS2 – Adding modules and results manipulation.

The students will focus on how to add their own modules in NS2. They will learn how to add a new agent into NS2 and how to adapt source codes accordingly. For the purpose of reporting, students will also learn how to use awk to analyze system performance using the trace file.

Requirements:

- Linux commands
- Familiarity with tcl
- Basic NS2 understanding

Objectives

- Following the step-by-step instructions below you will implement a new UDP and a new CBR agent in NS2, and
recompile the software.
- You will use the awk code provided in the lectures to calculate system parameters (throughput, jitter, delay…)
- You will visualize the result by using gnuplot (preferably), or other graph-plotting software you are familiar with. [*refer to assignment #1, on how results should be presented in the report.*]

Detailed instructions see below (also available from ns by example (slightly changed))

Reporting:

The expected result of this assignment is the successfully implementation of the new module and use the awk codes. Include the simulation scenario snapshot from nam, performance graphs (minimum requirement is: to present: (1) throughput, (2) jitter, and (3) delay) into your report.

Goal

We want to build a multimedia application that runs over a UDP connection, which simulates the behavior of an imaginary multimedia application that implements a "*five rate media scaling*" scheme which can, to some extent, respond to network congestion, by changing encoding and transmission policy pairs associated with the scale parameter values.

In this implementation, it is assumed that when a connection is established, the sender and the receiver agree on 5 different sets of encoding and transmission policy pairs, and associate them with 5 integers {0..4}. For simplicity reasons, it is also assumed that a **constant transmission rate** is determined for each of the encoding and transmission policy pairs, and every pair uses **packets of the same size** regardless of the encoding scheme.

Basically, this "five rate media scaling" works as follows. The sender starts with the transmission rate associated with scale 0, and changes rates according to the scale value that the receiver sends back. The receiver is responsible for monitoring network congestion and determining the scale factor.

For congestion monitoring, a simple periodical (for every RTT second) packet loss monitoring is used, and even **a single packet loss at each period is regarded as network congestion**. If congestion is detected, the receiver reduces the scale parameter value to half and notifies the sender of this value. If no packet loss is detected, the receiver increases the value by one and notifies the sender.

Analysis

Before implementing this application, examining the UDP agent implementation reveals one major problem: Since a UDP agent allocates and sends network packets, all the information needed for

application level communication should be handed to the UDP agent as a data stream. However, the UDP implementation allocates packets that only have a header stack. Therefore, **we need to modify the UDP implementation** to add a mechanism to send the data received from the application. It is also noted that we might want to use this application for further research on IP router queue management mechanisms. Therefore, we need a way to distinguish this type of multimedia stream from other types of streams. That is, **we also need to modify UDP agent to record data type in one of IP header fields that is currently not used**.

A. Implementation

For this application, we take the CBR implementation and modify it to have the "five level media scaling" feature. We examined the C++ class hierarchy. We name the class of this application as "**MmApp**" and implement as a child class of "Application". The matching OTcl hierarchy name is therefore "**Application/MmApp**".

The sender and receiver behavior of the application are implemented together in **MmApp**. We name the modified UDP agent who will support **MmApp**, as "**UdpMmAgent**" and implement it as a child class of **UdpAgent**. The matching OTcl hierarchy name is "**Agent/UDP/UDPmm**".

1. MmApp Header:

For the application level communication, we define a packet header whose structure in C++ is **hdr_mm**. Whenever the application has information to transmit, it will hand it to the **UdpMmAgent** using the **hdr_mm** structure format. Then, **UdpMmAgent** allocates one or more packets (depending on the data packet size) and writes the data to the multimedia header of each packet.

Table 1: MM Packet Header Structure, from *udp-mm.h* –see attached files

```
// Multimedia Header Structure
struct hdr_mm {
   int ack;      // is it ack packet?
   int seq;      // mm sequence number
   int nbytes;   // bytes for mm pkt
   double time;  // current time
   int scale;    // scale (0-4) associated with data rates

   // Packet header access functions
         static int offset_;
         inline static int& offset() { return offset_; }
         inline static hdr_mm* access(const Packet* p) {
                  return (hdr_mm*) p->access(offset_);
         }
};
```

Table 2: MM Class, from *udp-mm.cc*

```
// Mulitmedia Header Class
static class MultimediaHeaderClass : public PacketHeaderClass
{
public:
   MultimediaHeaderClass() :
PacketHeaderClass("PacketHeader/Multimedia",
                                          sizeof(hdr_mm)) {
          bind_offset(&hdr_mm::offset_);
   }
} class_mmhdr;
```

Table 1 above shows the header definition, in which a structure for the header, and a header class object, **MultimediaHeaderClass** (derived from **PacketHeaderClass**)are defined. In defining this class, the OTcl name for the header ("**PacketHeader/Multimedia**") and the size of the header structure defined are presented. Do notice that **bind_offset()** ***must*** be called in the constructor of this class.

We also add the following **red bold** lines to *packet.h* and *ns-packet.tcl* as shown in Table 3 (see also Slides from seminar #2) to add our "Multimedia" header to the header stack. At this point, the header creation is complete, and **UdpMmAgent** can access the new header using the **hdr_mm::access()** member function.
-Refer to the NS Manual for detailed information on header creation and access methods. For the rest of the application and the complete modified UDP agent description, refer directly to the *mm-app.h*, *mm-app.cc*, *udp-mm.h* and *udp-mm.cc* files, as you feel is needed.

Table 3: the packet header and tcl codes

1. In *packet.h*
A. Look for:
```
// insert new packet types here
Static packet_t PT_NTYPE = 62; // This MUST be the LAST one
```
And insert new packet types **between those 2 lines:**
```
    static const packet_t PT_Multimedia = 70;
```
B. Look for:
```
  class p_info {
```
In → `public:`
In → `static void initName()` in the long list of name_ add:
```
    name_[PT_Multimedia] = "Multimedia";
```

2. In *ns-packet.tcl*
- Look for:
```
    foreach prot {
```
And in the list of protocols add:
```
Multimedia
```

2. MmApp Sender:

The sender uses a timer for scheduling the next transmission of an application data packet. We defined the "**SendTimer**" class derived from the **TimerHandler** class, and wrote its "**expire()**" member function to call the **send_mm_pkt()** member function of the **MmApp** object. Then, we included an instance of this object as a *private member* of the **MmApp** object referred to as "**snd_timer_**". Table 4 shows the **SendTimer** implementation.

Table 4: the **SendTimer** implementation.

```
class SendTimer : public TimerHandler {
  public:
      SendTimer(MmApp* t) : TimerHandler(), t_(t) {}
      Inline virtual void expire(Event*);
  protected:
      MmApp* t_;
};
void SednTimer::expire(Event*)
{
  t_->send_mm_pkt();
};


class MmAPP : public Application{
  public:
  MmApp();
  …
  private
  …
  SendTimer snd_timer_;
  …
};


MmApp:: MmApp() : running_(0), snd_timer_(this), ack_timer_(this)
{
  bind_bw("rate0_", &rate[0]);
  …
  bind_bw("rate4_", &rate[4]);
  bind( "pktsize_", &pktsize_);
  bind_bool( "random_", &random);
}
```

```
void MmApp::send_mm_pkt()
{
  hdr_mm mh_buf;
  if (running_) {
  agent_->sendmsg(pktsize_, (char*) &mh_buf); //send to UDP

  // Reschedule the send_pkt time
  double next_time_ = next_snd_time();
   if (next_time_ > 0) snd_timer_.resched(next_time_);
  }
}
```

Before setting this timer, **MmApp** re-calculates the next transmission time using the transmission rate associated with the current scale value and the size of application data packet that is given in the input simulation script (or using the default size). The **MmApp** sender updates the scale parameter, when an application-ACK packet arrives from the receiver side.

3. MmApp Receiver:

The receiver uses a timer, named "**ack_timer_**", to schedule the next application-ACK packet transmission of which the interval is equal to the mean RTT. When receiving an application-data packet from the sender, the receiver counts the number of received packets and also counts the number of lost packets using the packet sequence number. When the **ack_timer_** expires, it invokes the **send_ack_pkt** member function of **MmApp**, which adjusts the scale value looking at the received and lost packet accounting information, resets the received and the lost counters to 0, and sends an ACK packet with the adjusted scale value. Note that the *receiver doesn't have any connection establishment or closing methods*. Therefore, starting from the first packet arrival, the receiver periodically sends ACK packets and never stops (this is a “quick and dirty solution”).

4. UdpMmAgent:

The **UdpMmAgent** is modified from the **UdpAgent** to have the following additional features: (1) writing to the sending data packet MM header the information received from a **MmApp** (or reading information from the received data packet MM header and handing it to the **MmApp**), (2) segmentation and re-assembly (the **UdpAgent** only implements segmentation), and (3) setting the priority bit to 15 (max priority) for the **MmApp** packets.

5. Modify "agent.h":

To make the new application and agent running with your NS distribution, you should add two methods to **Agent** class as **public**. In the **command** member function of the **MmApp** class, there is defined an **attach-agent** OTcl command. When this command is received from OTcl, the **MmApp** tries to attach itself to the underlying agent. Before the attachment, it invokes the **supportMM()** member function of the underlying agent to check if the underlying agent support for multimedia transmission (i.e. can it carry data from application to application), and invokes **enableMM()** if it does. Even though these two member functions are defined in the **UdpMmAgnet** class, it is not defined in its base ancestor **Agent** class, and the two member functions of the general **Agent** class are called. Therefore, when trying to compile the code, it will give an error message. Inserting the two methods as public member functions of the Agen" class (in **agent.h** as follows will solve this problem.

Table 5.Adding two member functions to "Agent" class.

```
class Agent : public Connector {
  public:
       Agent(int pktType);
     …..
     virtual int supportMM() {return 0 ;}
     virtual void enableMM() {}
     virtual void sendmsg(int nbytes, const char *flags = 0);
}
```

6. Modify *app.h* : You also need to add an additional member function "**recv_msg(int nbytes, const char *msg)**" to the **Application** class as shown in Table 6. This member function, which was included in the Application class in the old versions of NS (ns-2.1b4a for sure), is removed from the class in the latest versions (ns-2.1b8a for sure). Our multimedia application was initially written for the ns-2.1.b4a, and therefore requires the **Application::recv_msg()** member function for versions after ns-2.1b8a

Table 6. Adding a member function to "Application" class.

```
class Application : public Process {
  public:
       Application();
     virtual void send (int nbytes);
     virtual void recv (int nbytes);
     virtual void recv_msg(int nbytes, const char *msg = 0){}
     virtual void resume();
     …
}
```

7. Set default values for new parameters in the *ns-default.tcl* :

After implementing all the parts of the application and agent, the last thing to do is to set default values for the newly introduced configurable parameters in the *ns-default.tcl* file. Table 7 shows an example of setting the default values for configurable parameters introduced by **MmApp**.

Table 7. Settings Default parameter values.

```
…
        Application/MmApp set rate0_ 0.3mb
        Application/MmApp set rate1_ 0.6mb
        Application/MmApp set rate2_ 0.9mb
        Application/MmApp set rate3_ 1.2mb
        Application/MmApp set rate4_ 1.5mb

        Application/MmApp set pktsize_ 1000
        Application/MmApp set random_ false
```

B. Download and Compile

Here is a checklist that should be done before recompiling your NS.

0. Download *mm-app.h*, *mm-app.cc*, *udp-mm.h* and *udp-mm.cc* to your ns directory.

1. Make sure you registered the new application header by modifying *packet.h* and *ns-packet.tcl* as shown in **table 3**.

2. Add the **supportMM()** and **enableMM()** methods to the **Agent** class in *agent.h* as shown in **table 5.**

3. Add the **recv_msg()** method to the **Application** class in *app.h* as shown in **table 6**

4. Set default values for the newly introduced configurable parameters in ns-*default.tcl* as described in **table 7**.

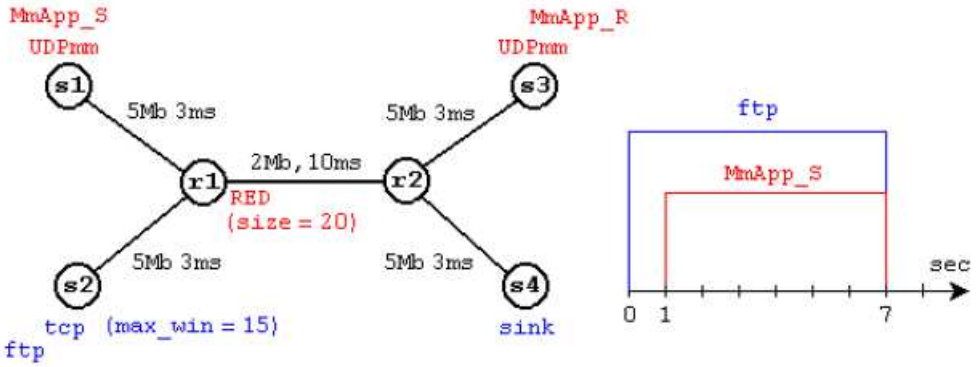
Be SURE to complete this last step. Otherwise, all five-scale rates are initialized to zero unless specified in the input simulation script (i.e., the test simulation script given below will **not** transmit any frames).

After you've done all things in the checklist, modify your *Makefile* as needed (see lecture #2) and re-compile your NS.

Be SURE to run "**make clean**" and "**make depend**" before you re-compile your modified NS, otherwise the new application may not transmit any packets.

C. Test Simulation & reporting

Figure 1 below shows a simulation topology and scenario that is used to test "MmApp", while in table 8 the test simulation script is included. Download this script from the assignment web page of the course and test your newly added components. Perform any additional changes you see fit to include in your report, implementing the required awk codes.



Good Luck!

Lab Assignments Contact:
Qing He, SP8202
tnk092@gmail.com

Table 8: the test script

```
set ns [new Simulator]

#Define different colors for data flows
$ns color 1 Red
$ns color 2 Blue

#Open the nam trace file
set nf [open out.nam w]
set tf [open out.tr w]
$ns namtrace-all $nf
$ns trace-all $tf

#Define a 'finish' procedure
proc finish {} {
        global ns nf tf
        $ns flush-trace
        #Close the trace file
        close $nf
        close $tf
        #Execute nam on the trace file
        exec nam out.nam &
        exit 0
}

set node_(s1) [$ns node]
set node_(s2) [$ns node]
set node_(r1) [$ns node]
set node_(r2) [$ns node]
set node_(s3) [$ns node]
set node_(s4) [$ns node]

$ns duplex-link $node_(s1) $node_(r1) 5Mb 3ms DropTail
$ns duplex-link $node_(s2) $node_(r1) 5Mb 3ms DropTail
$ns duplex-link $node_(r1) $node_(r2) 2Mb 10ms RED
$ns duplex-link $node_(s3) $node_(r2) 5Mb 3ms DropTail
$ns duplex-link $node_(s4) $node_(r2) 5Mb 3ms DropTail

#Setup RED queue parameter
$ns queue-limit $node_(r1) $node_(r2) 20
Queue/RED set thresh_ 5
Queue/RED set maxthresh_ 10
Queue/RED set q_weight_ 0.002
Queue/RED set ave_ 0

$ns duplex-link-op $node_(r1) $node_(r2) queuePos 0.5

$ns duplex-link-op $node_(s1) $node_(r1) orient right-down
$ns duplex-link-op $node_(s2) $node_(r1) orient right-up
$ns duplex-link-op $node_(r1) $node_(r2) orient right
$ns duplex-link-op $node_(s3) $node_(r2) orient left-down
$ns duplex-link-op $node_(s4) $node_(r2) orient left-up

#Setup a MM UDP connection
set udp_s [new Agent/UDP/UDPmm]
set udp_r [new Agent/UDP/UDPmm]
$ns attach-agent $node_(s1) $udp_s
$ns attach-agent $node_(s3) $udp_r
$ns connect $udp_s $udp_r
$udp_s set packetSize_ 1000
$udp_r set packetSize_ 1000
$udp_s set fid_ 1
$udp_r set fid_ 1

#Setup a MM Application
set mmapp_s [new Application/MmApp]
set mmapp_r [new Application/MmApp]
$mmapp_s attach-agent $udp_s
$mmapp_r attach-agent $udp_r
$mmapp_s set pktsize_ 1000
$mmapp_s set random_ false

#Setup a TCP connection
set tcp [$ns create-connection TCP/Reno $node_(s2) TCPSink $node_(s4) 0]
$tcp set window_ 15
$tcp set fid_ 2

#Setup a FTP Application
set ftp [$tcp attach-source FTP]

#Simulation Scenario
$ns at 0.0 "$ftp start"
$ns at 1.0 "$mmapp_s start"
$ns at 7.0 "finish"

$ns run
```