15744 Fall 2004: Computer Networking

Homework 2 - Solutions

Question 1: What is the default maximum buffer size of the queue in a link? Which command can you use to limit the maximum buffer size of the queue in the link between nodes n1 and n2 to 5 packets?

The variable corresponding to the default maximum buffer size is `limit_` and its default value is 50. There are two ways to find out what the default value is. You can either look into the file ns-default.tcl or you can create a queue and then print the value of `limit_` as follows.

```
set l [$ns duplex-link $n0 $n1 10ms DropTail]
set q [$l set queue_]
set s [$q set limit_]
puts $s
```

The command to change the maximum buffer size between nodes n1 and n2 to 5 is the following:

```
$ns queue-limit $n1 $n2 5
```

Question 2: Write a Tcl procedure that takes as an argument an integer n and creates a dumb-bell topology with n nodes on either side. Make the link between R1 and R2 a 5Mbps/10ms duplex links with a DropTail queue. Make all the other links 100Mbps/10ms duplex links with DropTail queues.

Here is one way to do it:

```
proc dumbbell {n} {
     global ns
     set r1 [$ns node]
     set r2 [$ns node]
     $ns duplex-link $r1 $r2 5Mb 10ms DropTail

     for {set i 0} {$i < $n} {incr i} {

         set s($i) [$ns node]
         $ns duplex-link $s($i) $r1 100 Mb 10ms DropTail

         set d($i) [$ns node]
         $ns duplex-link $r2 d($i) 100 Mb 10ms DropTail
     }
}
```

Question 3: Read the ns manual to find out how to write similar code to set up a *TCP* connection. Then use your procedure from Question 3 to create a dumb-bell topology with 5 S_i-D_i pairs. Set up a TCP connection for each of the S_i-D_i pairs. For all connections use an object of type `Agent/TCP`.

One way to do this is to extend your dumbbell procedure from the previous question as follows.

```
proc dumbbell {n} {
     global ns
     set r1 [$ns node]
```

```
    set r2 [$ns node]
    $ns duplex-link $r1 $r2 5Mb 10ms DropTail

    for {set i 0} {$i < $n} {incr i} {

        set s($i) [$ns node]
        set tc($i) [new Agent/TCP]
        $ns duplex-link $s($i) $r1 100 Mb 10ms DropTail
        $ns attach-agent $s($i) $tc($i)

        set d($i) [$ns node]
        set ts($i) [new Agent/TCPSink]
        $ns duplex-link $r2 d($i) 100 Mb 10ms DropTail
        $ns attach-agent $d($i) $ts($i)

        $ns connect $tc($i) $ts($i)
    }
}
```

Question 4: The code you wrote in the previous question creates a TCP agent that implements TCP Tahoe. Find out what other flavors of TCP are supported in ns.

The following flavors of TCP are supported

- Agent/TCP - a Tahoe TCP sender
- Agent/TCP/Reno - a Reno TCP sender
- Agent/TCP/Newreno - Reno TCP with a modification
- Agent/TCP/Vegas - Vegas TCP
- Agent/TCP/Sack1 - Reno TCP with selective acknowledgements
- Agent/TCP/FACK - Reno with forward acknowledgement congestion control

Question 5: Create and attach an FTP application to each of the S_i nodes from Question 3 and connect each to the corresponding D_i nodes. Start the FTP sources at 4 sec and stop them at 8 sec. Now change the code so that instead of explicitly stopping the FTP sources after 8 sec they stop after sending 10 packets. Save the resulting scripts in files called 5a.tcl and 5b.tcl, respectively.

For the first part you can simply extend your dumbbell procedure by adding the following code after the `connect` command.

```
set agent($i) [new Application/FTP]
$agent($i) attach-agent $tc($i)
$ns at 4.0 "$agent($i) start"
$ns at 8.0 "$agent($i) stop"
```

For the second part (having the source send 10 packets) you can add either the following lines to your dumbbell procedure

```
set agent($i) [new Application/FTP]
$agent($i) attach-agent $tc($i)
$ns at 4.0 "$agent($i) produce 10"
```

or these lines

```
set agent($i) [new Application/FTP]
$agent($i) attach-agent $tc($i)
$ns at 4.0 "$agent($i) send 10000"
```

The latter solution works because the default packet size is 1000 bytes.

> **Question 6:** Instead of deterministically starting a source it is sometimes useful to have a source start firing randomly. Change the code from the previous question so that the FTP sources start firing at random instants of time. Stop the sources after 100 sec. Save the script in a file called 6.tcl.

Just as a reminder, the reason we often want to start sources randomly is to avoid that TCP sources get synchronized. Randum numbers can be generated through the use of a random number generator provided by ns:

```
set rng_ [new RNG]
$rng_ seed 0
set tmp [$rng_ uniform 0 100]
```

The random numbers generated in this example will be uniformly distributed between 0 and 100. Another equivalent method is to use random variables provided by ns:

```
set u [new RandomVariable/Uniform]
$u set min_ 1
$u set max_ 100
set tmp [$u value]
```

Or you can use the `rand()` function that tcl provides.

> **Question 7:** Create a trace file for the simulation in the previous question. Start the tracing 20 sec after simulation begin and report the following
>
> - The loss rate on each of the flows.
> - The number of packets sent, received and dropped on each flow.

Here you can either use `trace-all` and then filter out the relevant information using awk or a perl script or you can use `trace-queue` on the bottleneck link (since this is the only link that will have drops).

A flow consists of all the packets between a given source and destination pair that belong to the same connection. Hence in our case there are 5 flows.

The numbers should look similar to the following:

Flow#	sent	recvd	drop	loss rate (in %)
1	15427	15427	0	0
2	15434	15434	0	0
3	15420	15420	0	0
4	15426	15426	0	0
5	885	787	98	11.07

Depending on when your TCP sources start firing the output might be slightly different.

Question 8: Trace the `cwnd_` variable of an arbitrary flow from the previous question and obtain a plot of the evolution of the congestion window with time.

Depending on whether you trace the cwnd_ variable of one of the flows that had no loss or one with loss your plot should look similar to Figure 1 or Figure 2 (see last page).

Question 9: Create 3 nodes n1, n2, and n3. Connect n1 with n2, n1 with n3, and n3 with n2. Then create a CBR agent and attach it to node 1. Connect this CBR agent to a sink on node 2. Have the CBR agent start sending at 0.5s and have it stop at 4.5s. Do not specify a routing protocol. At time 2.5 let the link between node 1 and 2 go down for a second. What do you observe?

The packets are dropped although there exists an alternative path.

Question 10: What happens after the link fails and why? (Hint: what is the default routing strategy?). How can you modify the code to solve this 'problem'.

The default routing strategy that ns uses is `Static`. In this case the routes are computed once before the start of the simulation and not recomputed during simulation. Therefore when the link fails the packets are dropped although there exists an alternative route. This problem can be solved by specifying for example `DV`:

```
$ns rtproto DV
```

Question 11: Having two languages raises the question of which language should be used when implementing a new component. What is the advantage of doing it purely in Otcl space? What is the advantage of doing it in C++ space?

The advantage of writing code in Otcl is that it is easier to modify (no recompiling). It is there convenient to use if the component needs to be modified frequently or for quick prototyping of a new idea. On the other hand it is less efficient than a compiled language like C++. Therefore, if a component is complex and/or requires per packet action C++ is a better solution.

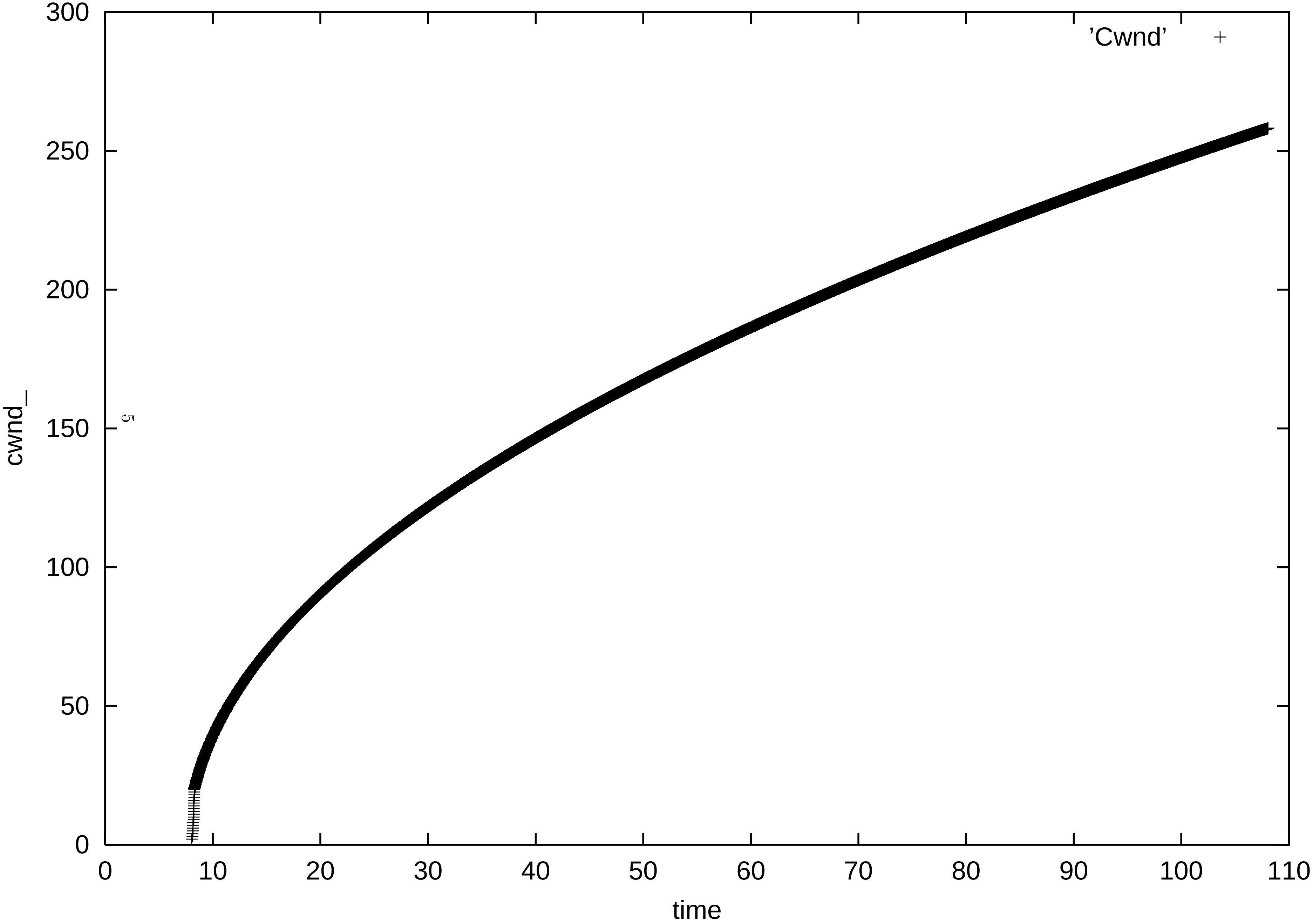
Question 12: Implement the following three methods so that the new Tcl classes `Sender` and `Receiver` can be used as specified above.

```
Sender instproc send-next args { ....}
Sender instproc recv msg {....}
Receiver instproc recv msg {....}
```

Do the entire implementation in a file called ping.tcl. Create a Sender and a Receiver and attach them to two nodes that are directly connected by a 1Mbps/10ms duplex link with a DropTail queue. Set the seq_ variable to 100. Start the sender 1 sec after simulation begin and stop it 5 sec after simulation begin. What is the output you get?

Here is a sample implementation:

```
Class Sender -superclass Agent/Message
```


'Cwnd'
cwnd_
time
0
50
100
150
200
250
300
0
10
20
30
40
50
60
70
80
90
100
110

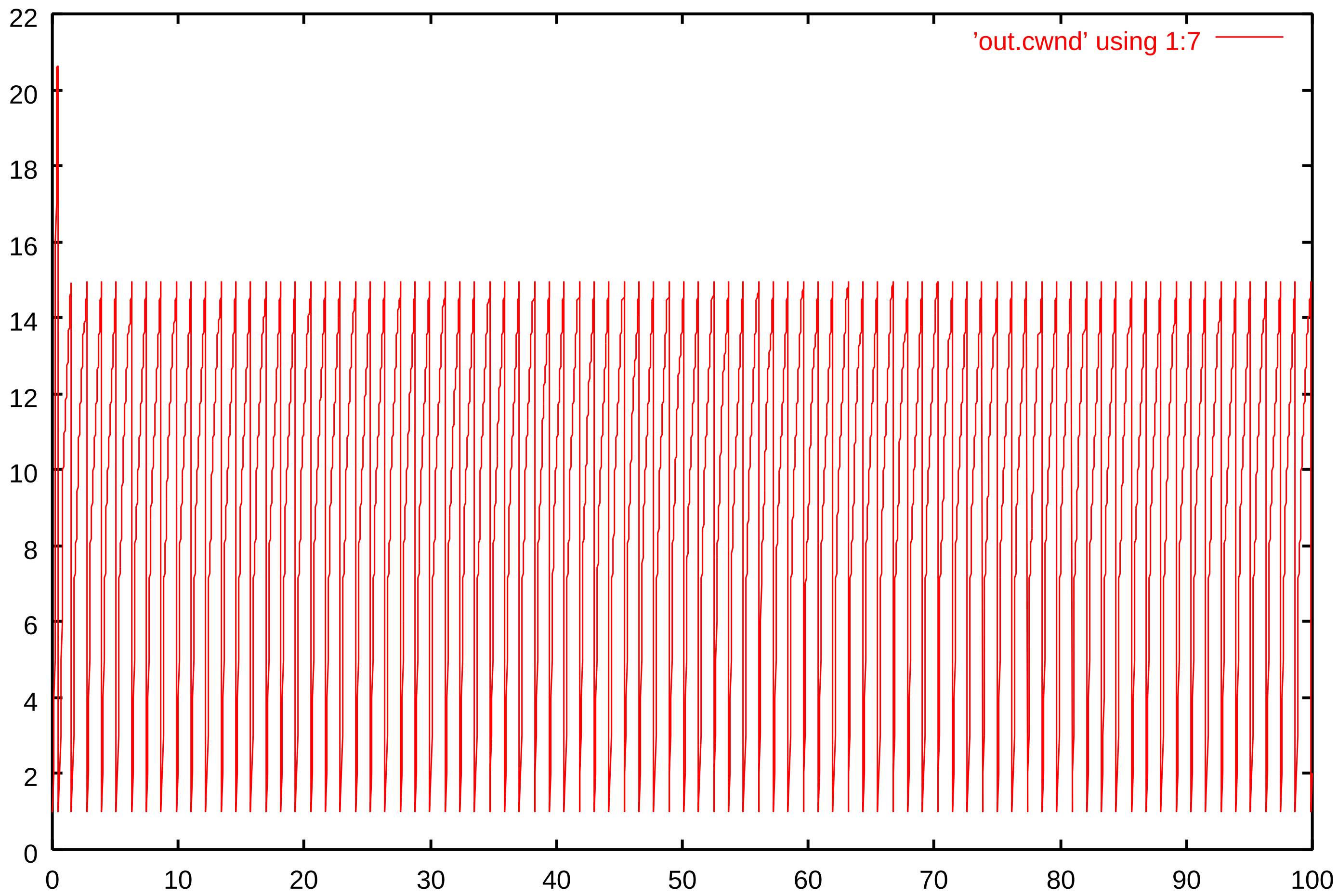


Figure 2: cwnd_ over time for a flow with loss

```
Sender instproc send-next args {
  $self instvar seq_ agent_addr_
  $self send "$agent_addr_ send $seq_"
  incr seq_
  global ns
  $ns at [expr [$ns now]+1] "$self send-next"
}

Sender instproc recv msg {
   $self instvar agent_addr
   global ns
   set sdr [lindex $msg 0]
   set seq [lindex $msg 2]
   set time_now [$ns now];
   puts "Sender gets ack $seq from $sdr at time $time_now"
}

Class Receiver -superclass Agent/Message
Receiver instproc recv msg {
   $self instvar agent_addr_
   global ns
   set sdr [lindex $msg 0]
   set seq [lindex $msg 2]
   set time_now [$ns now];
   puts "Receiver gets seq $seq from $sdr at time $time_now"
   $self send "$agent_addr_ ack $seq"
}
```

The output should look similar to the following:

```
Receiver gets seq 100 from 3 at time 1.1837500000000001
Sender gets ack 100 from 5 at time 1.3675000000000002
Receiver gets seq 101 from 3 at time 2.1837499999999994
Sender gets ack 101 from 5 at time 2.3674999999999988
Receiver gets seq 102 from 3 at time 3.184605553196151
Sender gets ack 102 from 5 at time 3.3683555531961504
...
```