Universidade de Coimbra
Faculdade de Ciências e Tecnologia

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA
Introdução às Redes de Comunicação

Exemplos de uso do TCP (Ficha não avaliada)

Ano Lectivo de 2015/2016

Nesta ficha vamos introduzir o estudo do TCP, que será a base do **Trabalho Prático 1**. Para isso, execute no ns-2 os dois scripts fornecidos em anexo (bwxd.tcl e ack_clock.tcl) usando os parâmetros propostos. Observe a transmissão e confirmação dos pacotes, a ocupação do meio de transmissão e o funcionamento da janela do TCP.

bwxd.tcl

By *F.Cela & A.Goller* on 02/16/01 (adapted)

Description: Issues regarding Stop-and-Wait protocols.

Stop-and-Wait protocols have some desirable properties:

- Since the sender cannot transmit a new packet until the acknowledgement for the last one sent is received, the protocol is ACK-clocked. These protocols automatically adjust the transmission speed to both the speed of the network and the rate the receiver sends new acknowledgements.
- They respect an underlying conservation principle of computer networks: "Given a producer sending packets to a consumer, if the system is in equilibrium, that equilibrium will be preserved if a new packet is sent only when a previously sent packet is consumed."

However, these protocols become rather inefficient if the path connecting the sender and the receiver has a large Bandwidth x Delay product (that is, the maximum number of bits that can be seen in transit on the link). A possible solution to this problem would be to give the source a credit of packets that can be sent without having to wait for acknowledgements. This would allow the source to fill the pipe, then, the arrival of acknowledgements would maintain and adjust a continuous flow of data. Therefore, a Stop-and-Wait protocol would be a particular case of such protocols, where a credit of one packet is given to the source. These issues are illustrated in the following 4 animations described below:

Animation 1:
Script used: bwxd.tcl
Parameters for this animation: `ns bwxd.tcl 10Mb 4ms 1 0.5`
Description: These two hosts are connected by means of a 10Mbit/s, 4ms delay link. Host 0 is using a Stop-and-Wait protocol. The line is very inefficiently used.

Animation 2:
Script used: bwxd.tcl
Parameters for this animation: `ns bwxd.tcl 10Mb 4ms 10 0.5`
Description: The two hosts are connected by means of a 10Mbit/s, 4ms delay link. Host 0 is using a window with capacity for 10 segments of 500bytes. The pipe is not being filled with packets and therefore the link is not being

efficiently used. The effective bandwidth is, in this case, approximately (10pkt*500bytes*8)/0.008 = 5Mbit/s, half of the available bandwidth.

Animation 3:
Script used: bwxd.tcl
Parameters for this animation: **`ns bwxd.tcl 10Mb 4ms 20 0.5`**
Description: Bandwidth x Delay = 40000bits = 10 packets of 500 bytes; we have adjusted the window to 20 packets but the pipe is still not full. Why? In our previous calculus we assumed the first ACK was sent exactly when the reception at n1 of the first packet begins. That is not true; the first ACK is sent when the reception of the first packet finishes and the packet is processed (you may need to slow down a bit the animation to see this fact).

Animation 4:
Script used: bwxd.tcl
Parameters for this animation: **`ns bwxd.tcl 10Mb 4ms 21 0.5`**
Description: We have added one packet more to the size of the window in the previous example; now we are keeping the pipe full.

------bwxd.tcl script ------ Ns version: ns2.1b7 ------

```
# TCP Sliding Window

if {$argc == 4} {
    set bandwidth [lindex $argv 0]
    set delay [lindex $argv 1]
    set window [lindex $argv 2]
    set time [lindex $argv 3]
} else {
    puts "            bandwidth"
    puts "     n0 --------------- n1"
    puts " TCP_window    delay"
    puts "Usage: $argv0 bandwidth delay window simulation_time"
    exit 1
}

# Create the 'Simulator' object
set ns [new Simulator]

# Open a file for writing the nam trace data
set nf [open out.nam w]

$ns namtrace-all $nf

# Add a 'finish' procedure that closes the trace and starts nam
proc finish {} {
    global ns nf
    $ns flush-trace
    close $nf
    exec nam out.nam &
    exit 0
}

$ns color 1 Red
$ns color 2 Blue
$ns color 1 Green

# Define the topology
set n0 [$ns node]
set n1 [$ns node]

#   object      from  to  bandwidth   delay    queue
$ns duplex-link $n0   $n1 $bandwidth $delay   DropTail
$ns duplex-link-op $n0   $n1 orient left-right

# Create a traffic source in node n0
set tcp [$ns create-connection TCP/RFC793edu $n0 TCPSink $n1 1]
$tcp set window_ $window
$tcp set packetSize_ 500

set ftp [new Application/FTP]
$ftp attach-agent $tcp

# When to start and to stop sending

$ns at 0.0 "$ftp start"
$ns at $time "finish"

####### END OF USER CODE ####################################################
############################################################################

# Start the simulation
$ns run
```

ack_clock.tcl

By *F.Cela & A.Goller* on 02/16/01 (adapted)

Description: Sliding Window

TCP implements an algorithm for flow control called Sliding Window; the reader will surely be familiar with this kind of algorithms which are used for flow control at the data link control layer of some protocols as well.

The "window" is the maximum amount of data we can send without having to wait for ACKs. In summary, the operation of the algorithm is as follows:

- Transmit all the new segments in the window.
- Wait for acknowledgement/s to come (several packets can be acknowledged in the same ACK).
- Slide the window to the indicated position and set the window size to the value advertised in the acknowledgement.

When we wait for an acknowledgement to a packet for some time and it has not arrived yet, the packet is retransmitted. When the acknowledgement arrives, it causes the window to be repositioned and the transmission continues from the packet following the one transmitted last.

TCP achieves following objectives by using the sliding window algorithm:

- The ACK policy makes the protocol self-clocking. Therefore, it dynamically adapts its transmission speed to both the speed of the network and the speed of the peer sending acknowledgements. If the conditions change in the network, so does the sender's transmission rate.
- The "credit" given by the window size makes possible an efficient use of the link.
- The receiver can regulate the information arriving rate by adjusting the sender's transmission window.

Animation 1:
Script used: ack_clock.tcl
Parameters for this animation: `ns ack_clock.tcl 0.2Mb 40ms 100 20 2`
Description: Links n0-n1 and n2-n3 have 1Mbit/s bandwidth and 10ms delay. Link n1-n2 has 0.2Mbit/s bandwidth, 40ms delay. Queue size at n1 is 100 segments and TCP uses a 20-segment window size. For the sake of simplicity, the TCP in n0 models the first TCP specification, which sends as much data as, the send window allows at the beginning of the transmission. Current TCP implementations use a less aggressive start (slow-start). After the connection establishment, we start sending as much data as the send window allows us. Note how the queue grows fast at the beginning of the bottleneck. Around t=0,56s we have sent the first 20 segments that the window allows us, then we run out of window and we have to wait for the first acknowledgement to open the window again. From now on, arrival of each new acknowledgement triggers the transmission of a new segment, the TCP source adapts to the effective capacity of the path, and we can see how the queue at the bottleneck remains stable.

Animation 2:
Script used: ack_clock.tcl
Parameters for this animation: `ns ack_clock.tcl 155Mb 2ms 100 20 1`
Description: Self-clocking is an interesting property of TCP that allows automatic adjustment of the transmission speed to the bandwidth and delay of the path. Therefore, it makes possible for TCP to operate over links with very different speeds. As an example, we can make the link 1-2 in the previous applet more than 700 times faster and TCP will still work without any problems. Links n0-n1 and n2-n3 are 1Mbit/s bandwidth, 10ms delay. Link n1-n2 is 155Mbit/s, 2ms delay. Queue size at n1 is 100 segments and TCP uses a 20-segment window size.

Animation 3:
Script used: ack_clock.tcl
Parameters for this animation: `ns ack_clock.tcl 64Kb 100ms 100 20 3`
Description: This animation shows how TCP works well even if we make the bottleneck slower. In this case we discover that as long as the window size is smaller than the actual size of the bottleneck path we send bursts of data rather than a continuous flow. Links n0-n1 and n2-n3 are 1Mbit/s bandwidth, 10ms delay. Link n1-n2 is 64Kbit/s, 100ms delay. Queue size at n1 is 100 segments and TCP uses a 20-segment window size. Around t=0.89s we run out of window and we have to wait for acknowledgements to open it again. We send bursts of packets rather than a continuous flow of packets.

------ack_clock.tcl script------ Ns version: ns2.1b7 ------

```
if {$argc == 5} {
    set bandwidth [lindex $argv 0]
    set delay [lindex $argv 1]
    set qsize [lindex $argv 2]
    set tcp_window [lindex $argv 3]
    set time [lindex $argv 4]
} else {
    puts "    1Mb      bandwidth      1Mb"
    puts "n0 ----- n1 ----------- n2 ----- n3"
    puts "    10ms      delay         10ms"
    puts "Usage: ns $argv0 bandwidth delay queue_size tcp_window time"
    exit 1
}

# Create the 'Simulator' object
set ns [new Simulator]

# Open a file for writing the nam trace data
set trace_nam [open out.nam w]

$ns namtrace-all $trace_nam

# Add a 'finish' procedure that closes the trace and starts nam
proc finish {} {
    global ns trace_nam
    $ns flush-trace
    exec nam out.nam &
    exit 0
}

# Color Codes
$ns color 1 Red
$ns color 2 Green
$ns color 3 Blue

# Define the topology
set n0 [$ns node]
set n1 [$ns node]
set n2 [$ns node]
set n3 [$ns node]

#   object      from  to  bandwith   delay    queue
$ns duplex-link $n0   $n1 1Mb        10ms    DropTail
$ns duplex-link $n1   $n2 $bandwidth $delay  DropTail
$ns duplex-link $n2   $n3 1Mb        10ms    DropTail

$ns duplex-link-op $n1 $n2 queuePos 0.5
$ns queue-limit $n1 $n2 $qsize
set qmon [$ns monitor-queue $n1 $n2 1 2]

# Create a traffic source in node n0
set ttcp0 [$ns create-connection TCP/RFC793edu $n0 TCPSink $n3 1]
$ttcp0 set window_ $tcp_window

set ftp0 [new Application/FTP]
$ftp0 attach-agent $ttcp0

# Create a traffic sink in node n1
set null0 [new Agent/Null]
$ns attach-agent $n2 $null0

# When to start and to stop sending
$ns at 0.1 "$ftp0 start"
$ns at $time "finish"

# Start the simulation
$ns run
```