PESIT Bangalore South Campus

1Km before Electronic City, Hosur Road, Bangalore-560100.

DEPARTMENT OF COMPUTER SCIENCE

V SEMESTER

LAB MANUAL

SUBJECT: Computer Networks Laboratory

Sub Code: 15CSL57

SESSION: JULY 2017 – DECEMBER 2017

Prepared by Dr. Sarasvathi V and Sajeevan

Course objectives:

- Demonstrate operation of network and its management commands
- Simulate and demonstrate the performance of GSM and CDMA
- Implement data link layer and transport layer protocols.

Description (If any):

For the experiments below modify the topology and parameters set for the experiment and take multiple rounds of reading and analyze the results available in log files. Plot necessary graphs and conclude using any suitable tool.

PART A

1. Implement three nodes point – to – point network with duplex links between them. Set the queue size, vary the bandwidth and find the number of packets dropped.
2. Implement transmission of ping messages/trace route over a network topology consisting of 6 nodes and find the number of packets dropped due to congestion.
3. Implement an Ethernet LAN using n nodes and set multiple traffic nodes and plot congestion window for different source / destination.
4. Implement simple ESS and with transmitting nodes in wire-less LAN by simulation and determine the performance with respect to transmission of packets.
5. Implement and study the performance of GSM on NS2/NS3 (Using MAC layer) or equivalent Environment.

6. Implement and study the performance of CDMA on NS2/NS3 (Using stack called Call net) or equivalent environment.

PART B

Implement the following in Java:

7. Write a program for error detecting code using CRC-CCITT (16- bits).
8. Write a program to find the shortest path between vertices using bellman-ford algorithm.
9. Using TCP/IP sockets, write a client – server program to make the client send the file name and to make the server send back the contents of the requested file if present. Implement the above program using as message queues or FIFOs as IPC channels.
10. Write a program on datagram socket for client/server to display the messages on client side, typed at the server side.
11. Write a program for simple RSA algorithm to encrypt and decrypt the data.
12. Write a program for congestion control using leaky bucket algorithm.

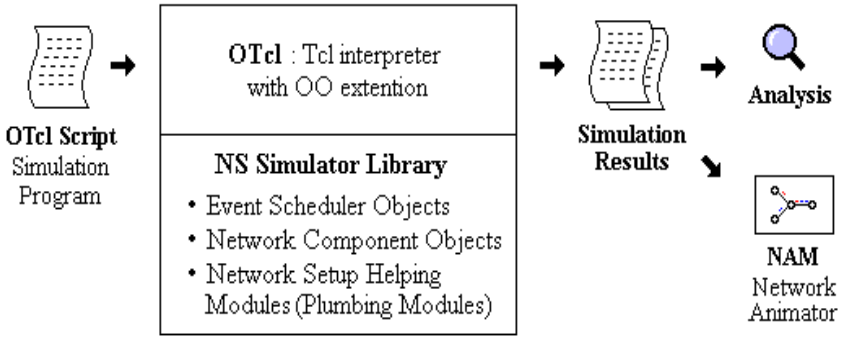
Part A - SIMULATION USING NS-2

Introduction to NS-2:

NS2 is an open-source simulation tool that runs on Linux. It is a discreet event simulator targeted at networking research and provides substantial support for simulation of routing, multicast protocols and IP protocols, such as UDP, TCP, RTP and SRM over wired and wireless (local and satellite) networks.

- Widely known as NS2, is simply an event driven simulation tool.
- Useful in studying the dynamic nature of communication networks.
- Simulation of wired as well as wireless network functions and protocols (e.g., routing algorithms, TCP, UDP) can be done using NS2.
- In general, NS2 provides users with a way of specifying such network protocols and simulating their corresponding behaviors.

Basic Architecture of NS2



TCL – Tool Command Language

Tcl is a very simple programming language. If you have programmed before, you can learn enough to write interesting Tcl programs within a few hours. This page provides a quick overview of the main features of Tcl. After reading this you'll probably be able to start writing simple Tcl scripts on your own; however, we recommend that you consult one of the many available Tcl books for more complete information.

Basic syntax

Tcl scripts are made up of *commands* separated by newlines or semicolons. Commands all have the same basic form illustrated by the following example:

```
expr 20 + 10
```

This command computes the sum of 20 and 10 and returns the result, 30. You can try out this example and all the others in this page by typing them to a Tcl application such as tclsh; after a command completes, tclsh prints its result.

Each Tcl command consists of one or more *words* separated by spaces. In this example there are four words: expr, 20, +, and 10. The first word is the name of a command and the other words are *arguments* to that command. All Tcl commands consist of words,

but different commands treat their arguments differently. The expr command treats all of its arguments together as an arithmetic expression, computes the result of that expression, and returns the result as a string. In the expr command the division into words isn't significant: you could just as easily have invoked the same command as

```
expr 20+10
```

However, for most commands the word structure is important, with each word used for a distinct purpose.

All Tcl commands return results. If a command has no meaningful result then it returns an empty string as its result.

Variables

Tcl allows you to store values in variables and use the values later in commands. The set command is used to write and read variables. For example, the following command modifies the variable x to hold the value 32:

```
set x 32
```

The command returns the new value of the variable. You can read the value of a variable by invoking set with only a single argument:

```
set x
```

You don't need to declare variables in Tcl: a variable is created automatically the first time it is set. Tcl variables don't have types: any variable can hold any value.

To use the value of a variable in a command, use *variable substitution* as in the following example:

```
expr $x*3
```

When a $ appears in a command, Tcl treats the letters and digits following it as a variable name, and substitutes the value of the variable in place of the name. In this example, the actual argument received by the expr command will be 32*3 (assuming that variable x was set as in the previous example). You can use variable substitution in any word of any command, or even multiple times within a word:

```
set cmd expr
set x 11
$cmd $x*$x
```

Command substitution

You can also use the result of one command in an argument to another command. This is called *command substitution*:

```
set a 44
set b [expr $a*4]
```

When a [appears in a command, Tcl treats everything between it and the matching] as a nested Tcl command. Tcl evaluates the nested command and substitutes its result into the enclosing command in place of the bracketed text. In the example above the second argument of the second set command will be 176.

Quotes and braces

Double-quotes allow you to specify words that contain spaces. For example, consider the following script:

```
set x 24
set y 18
set z "$x + $y is [expr $x + $y]"
```

After these three commands are evaluated variable z will have the value 24 + 18 is 42. Everything between the quotes is passed to the set command as a single word. Note that (a) command and variable substitutions are performed on the text between the quotes, and (b) the quotes themselves are not passed to the command. If the quotes were not present, the set command would have received 6 arguments, which would have caused an error.

Curly braces provide another way of grouping information into words. They are different from quotes in that no substitutions are performed on the text between the curly braces:

```
set z {$x + $y is [expr $x + $y]}
```

This command sets variable z to the value "$x + $y is [expr $x + $y]".

Control structures

Tcl provides a complete set of control structures including commands for conditional execution, looping, and procedures. Tcl control structures are just commands that take Tcl scripts as arguments. The example below creates a Tcl procedure called power, which raises a base to an integer power:

```
proc power {base p} {
    set result 1
    while {$p > 0} {
        set result [expr $result * $base]
        set p [expr $p - 1]
    }
    return $result
}
```

This script consists of a single command, proc. The proc command takes three arguments: the name of a procedure, a list of argument names, and the body of the procedure, which is a Tcl script. Note that everything between the curly brace at the end of the first line and the curly brace on the last line is passed verbatim to proc as a single argument. The proc command creates a new Tcl command named power that takes two arguments. You can then invoke power with commands like the following:

```
power 2 6
power 1.15 5
```

When power is invoked, the procedure body is evaluated. While the body is executing it can access its arguments as variables: base will hold the first argument and p will hold the second.

The body of the power procedure contains three Tcl commands: set, while, and return. The while command does most of the work of the procedure. It takes two arguments, an expression ($p > 0$) and a body, which is another Tcl script. The while command evaluates its expression argument using rules similar to those of the C programming language and if the result is true (nonzero) then it evaluates the body as a Tcl script. It repeats this process over and over until eventually the expression evaluates to false (zero). In this case the body of the while command multiplied the result value by base and then decrements p. When p reaches zero the result contains the desired power of base. The return command causes the procedure to exit with the value of variable result as the procedure's result.

Where do commands come from?

As you have seen, all of the interesting features in Tcl are represented by commands. Statements are commands, expressions are evaluated by executing commands, control structures are commands, and procedures are commands.

Tcl commands are created in three ways. One group of commands is provided by the Tcl interpreter itself. These commands are called *builtin commands*. They include all of the commands you have seen so far and many more (see below). The builtin commands are present in all Tcl applications.

The second group of commands is created using the Tcl extension mechanism. Tcl provides APIs that allow you to create a new command by writing a *command procedure* in C or C++ that implements the command. You then register the command procedure with the Tcl interpreter by telling Tcl the name of the command that the procedure implements. In the future, whenever that particular name is used for a Tcl command, Tcl will call your command procedure to execute the command. The builtin commands are also implemented using this same extension mechanism; their command procedures are simply part of the Tcl library.

When Tcl is used inside an application, the application incorporates its key features into Tcl using the extension mechanism. Thus the set of available Tcl commands varies from application to application. There are also numerous extension packages that can be incorporated into any Tcl application. One of the best known extensions is Tk, which provides powerful facilities for building graphical user interfaces. Other extensions provide object-oriented programming, database access, more graphical capabilities, and a variety of other features. One of Tcl's greatest advantages for building integration applications is the ease with which it can be extended to incorporate new features or communicate with other resources.

The third group of commands consists of procedures created with the proc command, such as the power command created above. Typically, extensions are used for lower-level functions where C programming is convenient, and procedures are used for higher-level functions where it is easier to write in Tcl.

Wired TCL Script Components

Create the event scheduler

Open new files & turn on the tracing

Create the nodes

Setup the links

Configure the traffic type (e.g., TCP, UDP, etc)

Set the time of traffic generation (e.g., CBR, FTP)

Terminate the simulation

NS Simulator Preliminaries.

Initialization and termination aspects of the ns simulator.

Definition of network nodes, links, queues and topology.

Definition of agents and of applications.

The nam visualization tool.

Tracing and random variables.

Features of NS2

NS2 can be employed in most unix systems and windows. Most of the NS2 code is in C++. It uses TCL as its scripting language, Otcl adds object orientation to TCL.NS(version 2) is an object oriented, discrete event driven network simulator that is freely distributed and open source.

- Traffic Models: CBR, VBR, Web etc
- Protocols: TCP, UDP, HTTP, Routing algorithms,MAC etc
- Error Models: Uniform, bursty etc
- Misc: Radio propagation, Mobility models , Energy Models
- Topology Generation tools
- Visualization tools (NAM), Tracing

Structure of NS

● NS is an object oriented discrete event simulator

– Simulator maintains list of events and executes one event after another

– Single thread of control: no locking or race conditions

● Back end is C++ event scheduler

– Protocols mostly

– Fast to run, more control

- Front end is OTCL

Creating scenarios, extensions to C++ protocols

fast to write and change

Platforms

It can be employed in most unix systems(FreeBSD, Linux, Solaris) and Windows.

Source code

Most of NS2 code is in C++

Scripting language

It uses TCL as its scripting language OTcl adds object orientation to TCL.

Protocols implemented in NS2

Transport layer(Traffic Agent) – TCP, UDP

Network layer(Routing agent)

Interface queue – FIFO queue, Drop Tail queue, Priority queue

Logic link contol layer – IEEE 802.2, AR

How to use NS2

Design Simulation – Determine simulation scenario

Build ns-2 script using tcl.

Run simulation

Simulation with NS2

Define objects of simulation.

Connect the objects to each other

Start the source applications. Packets are then created and are transmitted through network.

Exit the simulator after a certain fixed time.

NS programming Structure

- Create the event scheduler
- Turn on tracing
- Create network topology
- Create transport connections
- Generate traffic
- Insert errors

Sample Wired Simulation using NS-2

Creating Event Scheduler

- Create event scheduler: set ns [new simulator]
- Schedule an event: $ns at <time> <event>

– event is any legitimate ns/tcl function

$ns at 5.0 “finish”

proc finish {} {

global ns nf

close $nf

exec nam out.nam &

exit 0

}

● Start Scheduler

$ns run

-

Tracing

● All packet trace

$ns traceall[open out.tr w]

<event> <time> <from> <to> <pkt> <size>

…

<flowid> <src> <dst> <seqno> <aseqno>

+ 0.51 0 1 cbr 500 —— 0 0.0 1.0 0 2

_ 0.51 0 1 cbr 500 —— 0 0.0 1.0 0 2

R 0.514 0 1 cbr 500 ——0 0.0 1.0 0 0

● Variable trace

set par [open output/param.tr w]

$tcp attach $par

$tcp trace cwnd_

$tcp trace maxseq_

$tcp trace rtt_

Tracing and Animation

● Network Animator

set nf [open out.nam w]

$ns namtraceall

$nf

proc finish {} {

global ns nf

close $nf

exec nam out.nam &

exit 0

}

Creating topology

● Two nodes connected by a link

● Creating nodes

set n0 [$ns node]

set n1 [$ns node]

● Creating link between nodes

$ns <link_type> $n0 $n1 <bandwidth> <delay><queue-type>

$ns duplex-link$n0 $n1 1Mb 10ms DropTail

Data Sending

● Create UDP agent

set udp0 [new Agent/UDP]

$ns attach-agent $n0 $udp0

● Create CBR traffic source for feeding into UDP agent

set cbr0 [new Application/Traffic/CBR]

$cbr0 set packetSize_ 500

$cbr0 set interval_ 0.005

$cbr0 attach-agent$udp0

● Create traffic sink

set null0 [new Agent/Null]

$ns attach-agent$n1 $null0

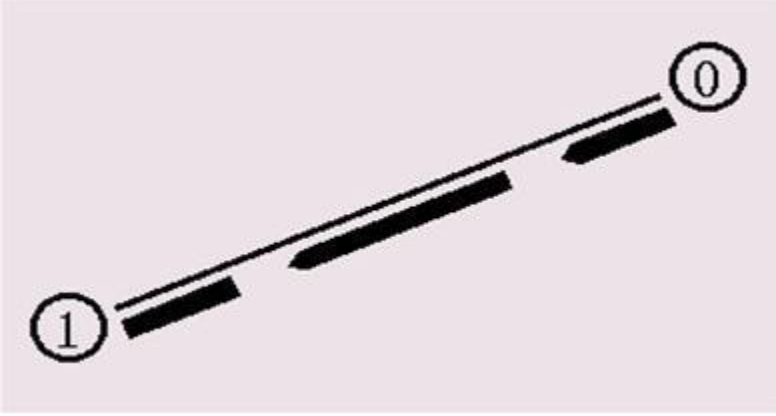
● Connect two agents

$ns connect $udp0 $null0

● Start and stop of data

$ns at 0.5 “$cbr0 start”

$ns at 4.5 “$cbr0 stop”



Traffic on top of TCP

● FTP

set ftp [new Application/FTP]

$ftp attach-agent$tcp0

● Telnet

set telnet [new Application/Telnet]

$telnet attach-agent$tcp0

PROCEDURE

STEP 1: Start

STEP 2: Create the simulator object ns for designing the given simulation

STEP 3: Open the trace file and nam file in the write mode

STEP 4: Create the nodes of the simulation using the 'set' command

STEP 5: Create links to the appropriate nodes using $ns duplex-link command

STEP 6: Set the orientation for the nodes in the simulation using 'orient' command

STEP 7: Create TCP agent for the nodes and attach these agents to the nodes

STEP 8: The traffic generator used is FTP for both node0 and node1

STEP 9: Configure node1 as the sink and attach it

STEP10: Connect node0 and node1 using 'connect' command

STEP 11: Setting color for the nodes

STEP 12: Schedule the events for FTP agent 10 sec

STEP 13: Schedule the simulation for 5 minutes

Structure of Trace Files

When tracing into an output ASCII file, the trace is organized in 12 fields as follows in fig shown below,

The meaning of the fields are:

Event	Time	From Node	To Node	PKT Type	PKT Size	Flags	Fid	Src Addr	Dest Addr	Seq Num	Pkt id

1. The first field is the event type. It is given by one of four possible symbols r, +, -, d which correspond respectively to receive (at the output of the link), enqueued, dequeued and dropped.
2. The second field gives the time at which the event occurs.
3. Gives the input node of the link at which the event occurs.
4. Gives the output node of the link at which the event occurs.
5. Gives the packet type (eg CBR or TCP)
6. Gives the packet size
7. Some flags
8. This is the flow id (fid) of IPv6 that a user can set for each flow at the input OTcl script one can further use this field for analysis purposes; it is also used when specifying stream color for the NAM display.
9. This is the source address given in the form of ―node.port‖.
10. This is the destination address, given in the same form.
11. This is the network layer protocol's packet sequence number. Even though UDP implementations in a real network do not use sequence number, ns keeps track of UDP packet sequence number for analysis purposes
12. The last field shows the Unique id of the packet.

XGRAPH

The xgraph program draws a graph on an x-display given data read from either data file or from standard input if no files are specified. It can display upto 64 independent data sets using different colors and line styles for each set. It annotates the graph with a title, axis labels, grid lines or tick marks, grid labels and a legend.

Syntax:

```
Xgraph [options] file-name
```

Options are listed here

/-bd <color> (Border)

This specifies the border color of the xgraph window.

/-bg <color> (Background)

This specifies the background color of the xgraph window.

/-fg<color> (Foreground)

This specifies the foreground color of the xgraph window.

/-lf <fontname> (LabelFont)

All axis labels and grid labels are drawn using this font.

/-t<string> (Title Text)

This string is centered at the top of the graph.

/-x <unit name> (XunitText)

This is the unit name for the x-axis. Its default is ―X‖.

/-y <unit name> (YunitText)

This is the unit name for the y-axis. Its default is ―Y‖.

Transmission of Ping messages: Experiment No. 2

Aim: Simulate the transmission of ping messages over a network topology consisting of 6 nodes and find the number of packets dropped due to congestion.

```
set ns [ new Simulator ]
set nf [ open lab4.nam w ]
$ns namtrace-all $nf
set tf [ open lab4.tr w ]
$ns trace-all $tf
set n0 [$ns node]
set n1 [$ns node]
set n2 [$ns node]
set n3 [$ns node]
set n4 [$ns node]
set n5 [$ns node]

$ns duplex-link $n0 $n4 1005Mb 1ms DropTail
$ns duplex-link $n1 $n4 50Mb 1ms DropTail
$ns duplex-link $n2 $n4 2000Mb 1ms DropTail
$ns duplex-link $n3 $n4 200Mb 1ms DropTail
$ns duplex-link $n4 $n5 1Mb 1ms DropTail

set p1 [new Agent/Ping] # letters A and P should be capital
$ns attach-agent $n0 $p1
$p1 set packetSize_ 50000
$p1 set interval_ 0.0001

set p2 [new Agent/Ping] #
letters A and P should be capital
$ns attach-agent $n1 $p2

set p3 [new Agent/Ping] # letters A and P should be capital
$ns attach-agent $n2 $p3
$p3 set packetSize_ 30000
$p3 set interval_ 0.00001
```

```
 set p4 [new Agent/Ping] #
 letters A and P should be capital
 $ns attach-agent $n3 $p4

 set p5 [new Agent/Ping] #
 letters A and P should be capital
 $ns attach-agent $n5 $p5

 $ns queue-limit $n0 $n4 5
 $ns queue-limit $n2 $n4 3
 $ns queue-limit $n4 $n5 2
Agent/Ping instproc recv {from rtt} {
$self instvar node_
puts "node [$node_ id]received answer from $from with round trip time $rtt msec"
}
#          please provide space between $node_ and id. No space between $ and
from. No space between and $ and rtt */

$ns connect $p1 $p5
$ns connect $p3 $p4

proc finish { } {
global ns nf tf
$ns flush-trace
close $nf
close $tf
exec nam lab4.nam &
exit 0
}
$ns at 0.1 "$p1 send"
$ns at 0.2 "$p1 send"
$ns at 0.3 "$p1 send"
$ns at 0.4 "$p1 send"
$ns at 0.5 "$p1 send"
$ns at 0.6 "$p1 send"
$ns at 0.7 "$p1 send"
$ns at 0.8 "$p1 send"
$ns at 0.9 "$p1 send"
$ns at 1.0 "$p1 send"
$ns at 1.1 "$p1 send"
$ns at 1.2 "$p1 send"
$ns at 1.3 "$p1 send"
$ns at 1.4 "$p1 send"
$ns at 1.5 "$p1 send"
$ns at 1.6 "$p1 send"
$ns at 1.7 "$p1 send"
$ns at 1.8 "$p1 send"
$ns at 1.9 "$p1 send"
$ns at 2.0 "finish"
$ns run
```

AWK file: *(Open a new editor using "vi command" and write awk file and save with ".awk" extension)*

```
BEGIN{
pingDrop=0;
}
{
 if($1= ="d" )
  {
   pingDrop++;
   }
}
END{
printf("Total number of ping packets dropped due to congestion is
=%d\n",pingDrop);
}
```

Simulation of Ethernet Lan Experiment No. 3

Experiment Specific Instructions

1. To analyze the given problem you have to write a Tcl script and simulate with ns2
2. Begin by specifying the trace files and the nam files to be created
3. Define a finish procedure
4. Determine and create the nodes that will be used to create the topology. Here in our experiment we are selecting 6 nodes namely 0, 1, 2, 3, 4, 5
5. Create the links to connect the nodes
6. Set up the LAN by specifying the nodes, and assign values for bandwidth, delay, queue type and channel to it
7. Optionally you can position and orient the nodes and links to view a nice video output with Nam
8. Set up the TCP and/or UDP connection(s) and the FTP/CBR (or any other application) that will run over it
9. Schedule the different events like simulation start and stop, data transmission start and stop
10. Call the finish procedure and mention the time at what time your simulation will end
11. Execute the script with ns

Simulation Script:

```
#Lan simulation
set ns [new Simulator]
#define color for data flows
$ns color 1 Blue
$ns color 2 Red
 #open tracefiles
set tracefile1 [open out.tr w]
set winfile [open winfile w]
$ns trace-all $tracefile1
#open nam file
set namfile [open out.nam w]
$ns namtrace-all $namfile
#define the finish procedure
proc finish {} {
global ns tracefile1 namfile
$ns flush-trace
close $tracefile1
close $namfile
exec nam out.nam &
exit 0
} #create six nodes
set n0 [$ns node]
set n1 [$ns node]
set n2 [$ns node]
set n3 [$ns node]
set n4 [$ns node]
set n5 [$ns node]
$n1 color Red
$n1 shape box
#create links between the nodes
$ns duplex-link $n0 $n2 2Mb 10ms DropTail
$ns duplex-link $n1 $n2 2Mb 10ms DropTail
$ns simplex-link $n2 $n3 0.3Mb 100ms DropTail
$ns simplex-link $n3 $n2 0.3Mb 100ms DropTail
set lan [$ns newLan "$n3 $n4 $n5" 0.5Mb 40ms LL Queue/DropTail
MAC/Csma/Cd Channel]
#Give node position
$ns duplex-link-op $n0 $n2 orient right-down
$ns duplex-link-op $n1 $n2 orient right-up
$ns simplex-link-op $n2 $n3 orient right
$ns simplex-link-op $n3 $n2 orient left
#set queue size of link(n2-n3) to 20
$ns queue-limit $n2 $n3 20
 #setup TCP connection
set tcp [new Agent/TCP/Newreno]
$ns attach-agent $n0 $tcp
set sink [new Agent/TCPSink/DelAck]
$ns attach-agent $n4 $sink
$ns connect $tcp $sink
```

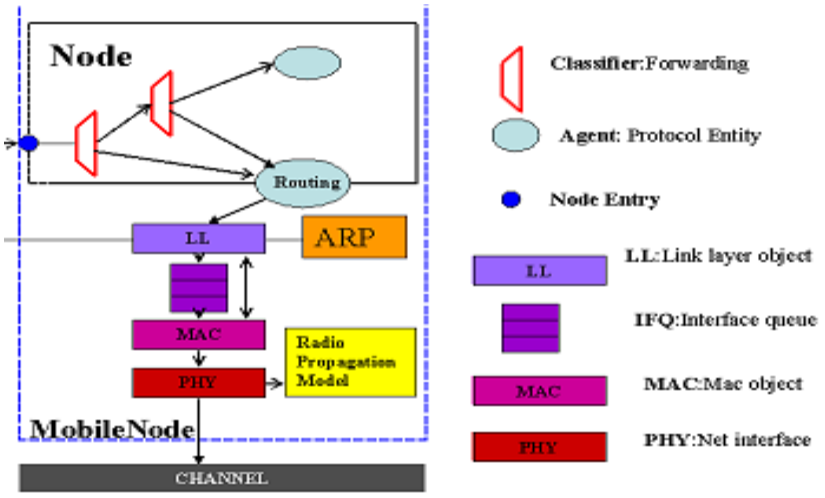
```
$tcp set fid_ 1
$tcp set packet_size_ 552
#set ftp over tcp connection
set ftp [new Application/FTP]
$ftp attach-agent $tcp
#setup a UDP connection
set udp [new Agent/UDP]
$ns attach-agent $n1 $udp
set null [new Agent/Null]
$ns attach-agent $n5 $null
$ns connect $udp $null
$udp set fid_ 2
#setup a CBR over UDP connection
set cbr [new Application/Traffic/CBR]
$cbr attach-agent $udp
$cbr set type_ CBR
$cbr set packet_size_ 1000
$cbr set rate_ 0.01Mb
$cbr set random_ false
#scheduling the events
$ns at 0.1 "$cbr start"
$ns at 1.0 "$ftp start"
$ns at 124.0 "$ftp stop"
$ns at 125.5 "$cbr stop"
proc plotWindow {tcpSource file} {
global ns
set time 0.1
set now [$ns now]
set cwnd [$tcpSource set cwnd_]
puts $file "$now $cwnd"
 $ns at [expr $now+$time] "plotWindow $tcpSource $file"
}
$ns at 0.1 "plotWindow $tcp $winfile"
 $ns at 125.0 "finish"
 $ns run
```

Wireless Simulation using NS-2

Simple Wireless Program in NS2 is the best way to learn about how to code in NS2. NS2 is one of the best simulation tool used by majority of scholars today due to its highlighted features like support for OOPs concept, C++ programming fundamentals, real time emulation support etc. NS2 is used to simulate both wired and wireless networks; here we have focused on wireless network simulation in NS2 due to its wide applicability. Regarding wired simulation in NS2, refer our other articles available in this site. Here, we have taken a simple wireless program in NS2 to explain the students about how to work with wireless networks in NS2.

Simulating a Wireless Network in NS2

- **Wireless Nodes**
  - A mobile node consists of network components:
    - Link Layer (LL)
    - Interface Queue (IfQ)
    - the MAC layer
    - the PHY layer: the wireless channel that the node transmit and receive signals from



  - At the beginning of a wireless simulation, we need to define the **type** for each of these network components.
  - Additionally, we need to define other parameters like:
    - the type of antenna
    - the radio-propagation model
    - the type of ad-hoc routing protocol used by mobilenodes etc.

  - **Configuring a Wireless Node**
    - Creating **wireless nodes** is also achieved using the **ns_ node** command:

```
set ns_  [new Simulator]    ;# Create a NS simulator object

set n1   [ns_  node]        ;# Create a WIRELESS node !!!
```

- **However:**

  BEFORE creating a **wireless node** you **MUST** first **select (configure)** the node configuration parameters to "become" a **wireless node**.

- The **NS2** command to **select (configure)** node configuration parameters is **node-config** and it is used as follows:

```
set ns_  [new Simulator]    ;# Create a NS simulator object

$ns_ node-config \
             -llType        LL
             -ifqType       "Queue/DropTail/PriQueue"
             -ifqLen        50
             -macType       Mac/802_11
             -phyType       "Phy/WirelessPhy"

                        -addressingType flat or hierarchical or expanded
             -adhocRouting   DSDV or DSR or TORA
             -propType      "Propagation/TwoRayGround"
             -antType       "Antenna/OmniAntenna"
             -channelType    "Channel/WirelessChannel"
             -topoInstance   $topo
             -energyModel    "EnergyModel"
             -initialEnergy  (in Joules)
             -rxPower       (in W)
             -txPower       (in W)

             -agentTrace     ON or OFF
             -routerTrace    ON or OFF
             -macTrace       ON or OFF
             -movementTrace  ON or OFF
```

- The value of most of the parameters are **simple values**
- **Except** for the value of the **-topoInstance** parameters.
- The **topology** is a **Topography** object that you must create.

  Example:

```
set topo  [new Topography]      ;# Create a Topography object

$topo load_flatgrid  500  500   ;# Make a 500x500 grid topology
```

- A **commonly used** wireless node configuration is:

```
set ns_  [new Simulator]    ;# Create a NS simulator object

$ns_ node-config -llType        LL
             -ifqType       "Queue/DropTail/PriQueue"
             -ifqLen        50
             -macType       Mac/802_11
             -phyType       "Phy/WirelessPhy"
```

```
                    -addressingType flat
            -adhocRouting   DSR or DSDV
            -propType       "Propagation/TwoRayGround"
            -antType        "Antenna/OmniAntenna"
            -channelType    "Channel/WirelessChannel"
            -topoInstance   $topo

            -agentTrace     ON
            -routerTrace    ON
            -macTrace       OFF
            -movementTrace  OFF
```

Sample Wireless Simulation Script

```
# Simulator Instance Creation
set ns [new Simulator]

#Fixing the co-ordinate of simutaion area
set val(x) 500
set val(y) 500
# Define options
set val(chan) Channel/WirelessChannel ;# channel type
set val(prop) Propagation/TwoRayGround ;# radio-propagation model

set val(netif) Phy/WirelessPhy ;# network interface type
set val(mac) Mac/802_11 ;# MAC type
set val(ifq) Queue/DropTail/PriQueue ;# interface queue type
set val(ll) LL ;# link layer type
set val(ant) Antenna/OmniAntenna ;# antenna model
set val(ifqlen) 50 ;# max packet in ifq
set val(nn) 2 ;# number of mobilenodes
set val(rp) AODV ;# routing protocol
set val(x) 500 ;# X dimension of topography
set val(y) 400 ;# Y dimension of topography
set val(stop) 10.0 ;# time of simulation end

# set up topography object
set topo [new Topography]
$topo load_flatgrid $val(x) $val(y)

#Nam File Creation nam – network animator
set namfile [open sample1.nam w]

#Tracing all the events and cofiguration
$ns namtrace-all-wireless $namfile $val(x) $val(y)

#Trace File creation
set tracefile [open sample1.tr w]

#Tracing all the events and cofiguration
$ns trace-all $tracefile
```

```
# general operational descriptor- storing the hop details in the network
create-god $val(nn)

# configure the nodes
$ns node-config -adhocRouting $val(rp) \
-llType $val(ll) \
-macType $val(mac) \
-ifqType $val(ifq) \
-ifqLen $val(ifqlen) \
-antType $val(ant) \
-propType $val(prop) \
-phyType $val(netif) \
-channelType $val(chan) \
-topoInstance $topo \
-agentTrace ON \
-routerTrace ON \
-macTrace OFF \
-movementTrace ON

# Node Creation
set node1 [$ns node]
# Initial color of the node
$node1 color black

#Location fixing for a single node
$node1 set X_ 200
$node1 set Y_ 100
$node1 set Z_ 0

set node2 [$ns node]
$node2 color black

$node2 set X_ 200
$node2 set Y_ 300
$node2 set Z_ 0
# Label and coloring
$ns at 0.1 "$node1 color blue"
$ns at 0.1 "$node1 label Node1"
$ns at 0.1 "$node2 label Node2"
#Size of the node
$ns initial_node_pos $node1 30
$ns initial_node_pos $node2 30
# ending nam and the simulation
$ns at $val(stop) "$ns nam-end-wireless $val(stop)"
$ns at $val(stop) "stop"

#Stopping the scheduler
$ns at 10.01 "puts \"end simulation\" ; $ns halt"
#$ns at 10.01 "$ns halt"
proc stop {} {
global namfile tracefile ns
$ns flush-trace
close $namfile
close $tracefile
```

```
#executing nam file
exec nam sample1.nam &
}

#Starting scheduler
$ns run
```

Experiment 5 & 6:

Implement and study the performance of GSM & CDMA on NS3 (Using MAC layer) or equivalent Environment.

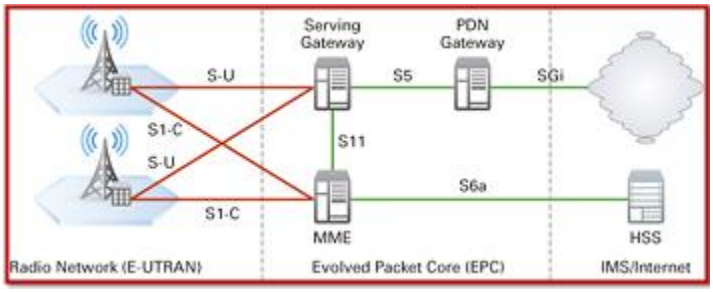
NS3 LTE Simulation

LTE is the latest high-speed cellular transmission network.LTE is a 4G technology with download speeds that run the gamut from 3 to 28 Mbps worldwide.4G LTE is one of several competing 4G standards along with Ultra Mobile Broadband (UMB) andWiMax (IEEE 802.16).Ns3 is the best choice among network simulator for simulating LTE framework.We provide ***customized NS3 LTE Simulation Projects*** based on customer Requirements.

Advantages of LTE:

- LTE will supports seamless connection to existing networks like GSM,CDMA and WCDMA.
- It has simple architecture because of low operating expenditure
- Time required for connecting network and is in range of a few hundred ms and power savings states can now be entered and exited very quickly
- High data rates can be achieved in both downlink as well as uplink.
- Both FDD and TDD can be used on same platform.
- Optimized signaling for connection establishment and other air interface and mobility management procedures have further improved the user experience.

Architecture of LTE:



LTE parameters:

- Transmission bandwidth.
- Mobility.
- Frequency range.

- Duplexing.
- Channel bandwidth.
- Channel coding.
- MIMO.
- Multi-antenna technology.

Sample code for LTE:

```
#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/mobility-module.h"
#include "ns3/lte-module.h"
#include "ns3/config-store-module.h"
using namespace ns3;
int main (int argc, char *argv[])
{
CommandLine cmd;
cmd.Parse (argc, argv);
ConfigStore inputConfig;
inputConfig.ConfigureDefaults ();
cmd.Parse (argc, argv);
Ptr<LteHelper> lteHelper = CreateObject<LteHelper> ();
lteHelper->SetAttribute ("PathlossModel", StringValue
("ns3::FriisSpectrumPropagationLossModel"));
NodeContainer enbNodes;
NodeContainer ueNodes;
enbNodes.Create (1);
ueNodes.Create (3);
MobilityHelper mobility;
mobility.SetMobilityModel ("ns3::ConstantPositionMobilityModel");
mobility.Install (enbNodes);
mobility.SetMobilityModel ("ns3::ConstantPositionMobilityModel");
mobility.Install (ueNodes);
NetDeviceContainer enbDevs;
NetDeviceContainer ueDevs;
enbDevs = lteHelper->InstallEnbDevice (enbNodes);
ueDevs = lteHelper->InstallUeDevice (ueNodes);
lteHelper->Attach (ueDevs, enbDevs.Get (0));
enum EpsBearer::Qci q = EpsBearer::GBR_CONV_VOICE;
EpsBearer bearer (q);
lteHelper->ActivateDataRadioBearer (ueDevs, bearer);
Simulator::Stop (Seconds (0.5));
lteHelper->EnablePhyTraces ();
lteHelper->EnableMacTraces ();
lteHelper->EnableRlcTraces ();
double distance_temp [] = { 1000,1000,1000};
std::vector<double> userDistance;
userDistance.assign (distance_temp, distance_temp + 3);
for (int i = 0; i < 3; i++)
{
```

```
Ptr<ConstantPositionMobilityModel> mm = ueNodes.Get (i)-
>GetObject<ConstantPositionMobilityModel> ();
mm->SetPosition (Vector (userDistance[i], 0.0, 0.0));
}
Simulator::Run ();
Simulator::Destroy ();
return 0;
}
```

PART B

Experiment No 7

7. Write a program for error detecting code using CRC-CCITT (16-bits).

Theory

The cyclic redundancy check, or CRC, is a technique for detecting errors in digital data, but not for making corrections when errors are detected. It is used primarily in data transmission.

In the CRC method, a certain number of check bits, often called a checksum, are appended to the message being transmitted. The receiver can determine whether or not the check bits agree with the data, to ascertain with a certain degree of probability whether or not an error occurred in transmission.

It does error checking via polynomial division. In general, a bit string

$b_{n-1}\ b_{n-2}\ b_{n-3}\ \ldots b_2\ b_1\ b_0$

As

$b_{n-1}X^{n-1} + b_{n-2}\ X^{n-2} + b_{n-3}\ X^{n-3} + \ldots b_2\ X^2 + b_1\ X^1 + b_0$

Ex: -

10001000000100001

As

$X^{16} + X^{12} + X^5 + 1$

All computations are done in modulo 2

Algorithm:-

1. Given a bit string, append 0^S to the end of it (the number of 0^s is the same as the degree of the generator polynomial) let B(x) be the polynomial corresponding to B.
2. Divide B(x) by some agreed on polynomial G(x) (generator polynomial) and determine the remainder R(x). This division is to be done using Modulo 2 Division.
3. Define T(x) = B(x) –R(x)

   (T(x)/G(x) => remainder 0)

4. Transmit T, the bit string corresponding to T(x).
5. Let T' represent the bit stream the receiver gets and T'(x) the associated polynomial. The receiver divides T1(x) by G(x). If there is a 0 remainder, the receiver concludes T = T' and no error occurred otherwise, the receiver concludes an error occurred and requires a retransmission.

***Figure 10.6:* Division in CRC encoder**

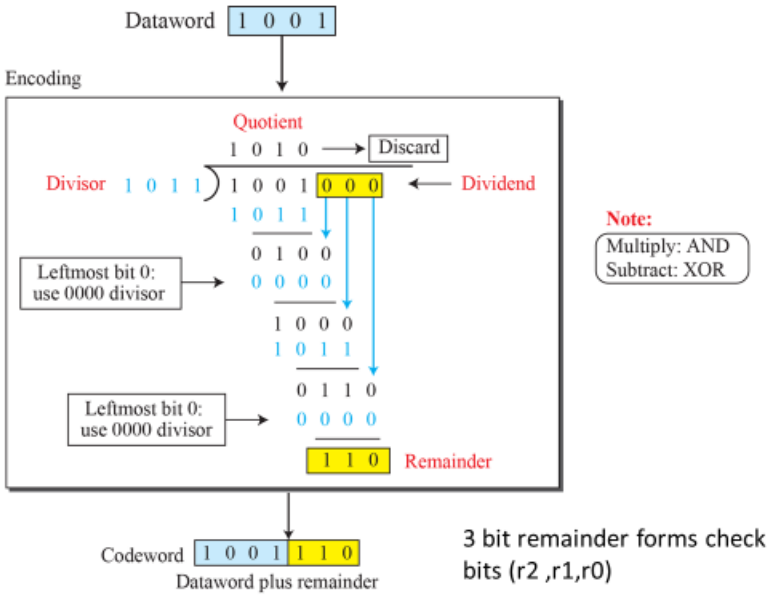
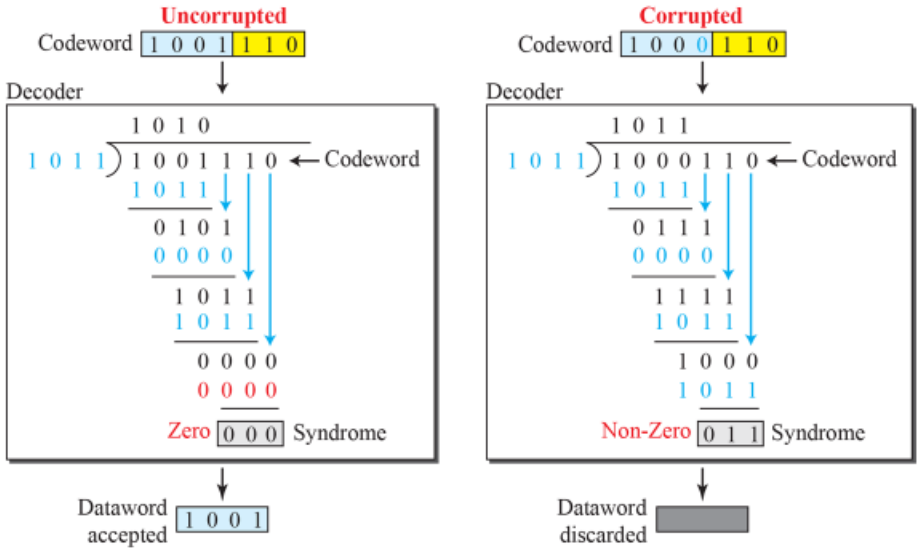


***Figure 10.7:* Division in the CRC decoder for two cases**



Experiment No 8

8. Write a program to find the shortest path between vertices using bellman-ford algorithm.

Theory

Routing algorithm is a part of network layer software which is responsible for deciding which output line an incoming packet should be transmitted on. If the subnet uses datagram internally, this decision must be made for every arriving data packet since the best route may have changed since last time. If the subnet uses virtual circuits internally, routing decisions are made only when a new established route is being set up. The latter case is sometimes called session routing, because a route remains in force for an entire user session (e.g., login session at a terminal or a file).

Routing algorithms can be grouped into two major classes: adaptive and nonadaptive. Nonadaptive algorithms do not base their routing decisions on measurement or estimates of current traffic and topology. Instead, the choice of route to use to get from I to J (for all I and J) is compute in advance, offline, and downloaded to the routers when the network ids booted. This procedure is sometime called static routing.

Adaptive algorithms, in contrast, change their routing decisions to reflect changes in the topology, and usually the traffic as well. Adaptive algorithms differ in where they get information (e.g., locally, from adjacent routers, or from all routers), when they change the routes (e.g., every ■*T* sec, when the load changes, or when the topology changes), and what metric is used for optimization (e.g., distance, number of hops, or estimated transit time).

Two algorithms in particular, distance vector routing and link state routing are the most popular. Distance vector routing algorithms operate by having each router maintain a table (i.e., vector) giving the best known distance to each destination and which line to get there. These tables are updated by exchanging information with the neighbors.

The distance vector routing algorithm is sometimes called by other names, including the distributed Bellman-Ford routing algorithm and the Ford-Fulkerson algorithm, after the researchers who developed it (Bellman, 1957; and Ford and Fulkerson, 1962). It was the original ARPANET routing algorithm and was also used in the Internet under the RIP and in early versions of DECnet and Novell's IPX. AppleTalk and Cisco routers use improved distance vector protocols.

In distance vector routing, each router maintains a routing table indexed by, and containing one entry for, each router in subnet. This entry contains two parts: the preferred out going line to use for that destination, and an estimate of the time or distance to that destination. The metric used might be number of hops, time delay in milliseconds, total number of packets queued along the path, or something similar.

The router is assumed to know the "distance" to each of its neighbor. If the metric is hops, the distance is just one hop. If the metric is queue length, the router simply examines each queue. If the metric is delay, the router can measure it directly with special ECHO packets hat the receiver just time stamps and sends back as fast as possible.

The Count to Infinity Problem.

Distance vector routing algorithm reacts rapidly to good news, but leisurely to bad news. Consider a router whose best route to destination X is large. If on the next exchange neighbor. A suddenly reports a

short delay to *X*, the router just switches over to using the line to *A* to send traffic to *X*. In one vector exchange, the good news is processed.

To see how fast good news propagates, consider the five node (linear) subnet of following figure, where the delay metric is the number of hops. Suppose A is down initially and all the other routers know this. In other words, they have all recorded the delay to A as infinity.

A	B	C	D	E	A	B	C	D	E	
∞		∞	∞	∞	Initially	1	2	3	4	Initially
1		∞	∞	∞	After 1 exchange	3	2	3	4	After 1 exchange
1		2	∞	∞	After 2 exchange	3	3	3	4	After 2 exchange
1		2	3	∞	After 3 exchange	5	3	5	4	After 3 exchange
1		2	3	4	After 4 exchange	5	6	5	6	After 4 exchange
						7	6	7	6	After 5 exchange
						7	8	7	8	After 6 exchange
							:			
						∞	∞	∞	∞	

Many ad hoc solutions to the count to infinity problem have been proposed in the literature, each one more complicated and less useful than the one before it. The split horizon algorithm works the same way as distance vector routing, except that the distance to X is not reported on line that packets for X are sent on (actually, it is reported as infinity). In the initial state of right figure, for example, C tells D the truth about distance to A but C tells B that its distance to A is infinite. Similarly, D tells the truth to E but lies to C.

Each node x begins with $D_x(y)$, an estimate of the cost of the least-cost path from itself to node y, for all nodes in N. Let $D_x = [D_x(y): y \text{ in } N]$ be node x's distance vector, which is the vector of cost estimates from x to all other nodes, y, in N. With the DV algorithm, each node x maintains the following routing information:

- For each neighbor v, the cost c(x,v) from x to directly attached neighbor, v
- Node x's distance vector, that is, $D_x = [D_x(y): y \text{ in } N]$, containing x's estimate of its cost to all destinations, y, in N
- The distance vectors of each of its neighbors, that is, $D_v = [D_v(y): y \text{ in } N]$ for each neighbor v of x

At each node, x:

```
1  Initialization:
2     for all destinations y in N:
3        D_x(y) = c(x,y)    /* if y is not a neighbor then c(x,y) = ∞ */
4     for each neighbor w
5        D_w(y) = ? for all destinations y in N
6     for each neighbor w
7        send distance vector D_x = [D_x(y): y in N] to w
8
9  loop
10    wait (until I see a link cost change to some neighbor w or
11          until I receive a distance vector from some neighbor w)
12
13    for each y in N:
14       D_x(y) = min_v{c(x,v) + D_v(y)}
15
16    if D_x(y) changed for any destination y
17       send distance vector D_x = [D_x(y): y in N] to all neighbors
18
19 forever
```

Experiment No 9
TCP Socket

9. Using TCP/IP sockets, write a client – server program to make the client send the file name to make the server send back the contents of the requested file if present. Implement the above program using as message queues or FIFOs as IPC channels.

TCP is a connection-oriented protocol. This means that before the client and server can start to send data to each other, they first need to handshake and establish a TCP connection. One end of the TCP connection is attached to the client socket and the other end is attached to a server socket. When creating the TCP connection, we associate with it the client socket address (IPaddress and port number) and the server socket address (IPaddress and port number). With the TCP connection established, when one side wants to send data to the other side, it just drops the data into the TCP connection via its socket.

With the server process running, the client process can initiate a TCP connection to the server. This is done in the client program by creating a TCP socket. When the client creates its TCP socket, it specifies the address of the welcoming socket in the server, namely, the IP address of the server host and the port number of the socket. After creating its socket, the client initiates a three-way handshake and establishes a TCP connection with the server.

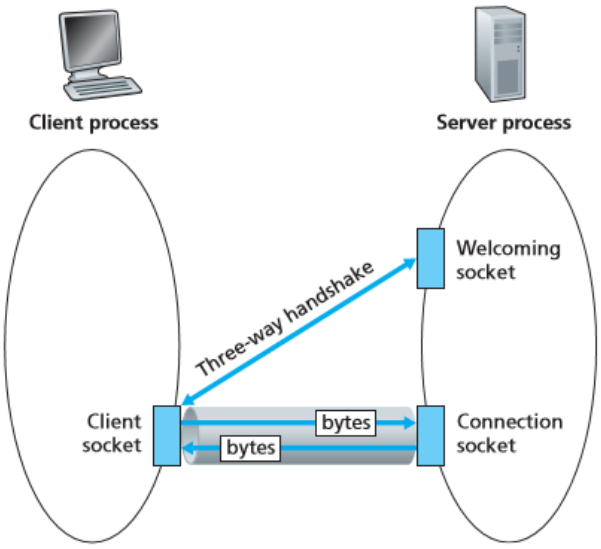


Figure 2.29 ♦ The `TCPServer` process has two sockets

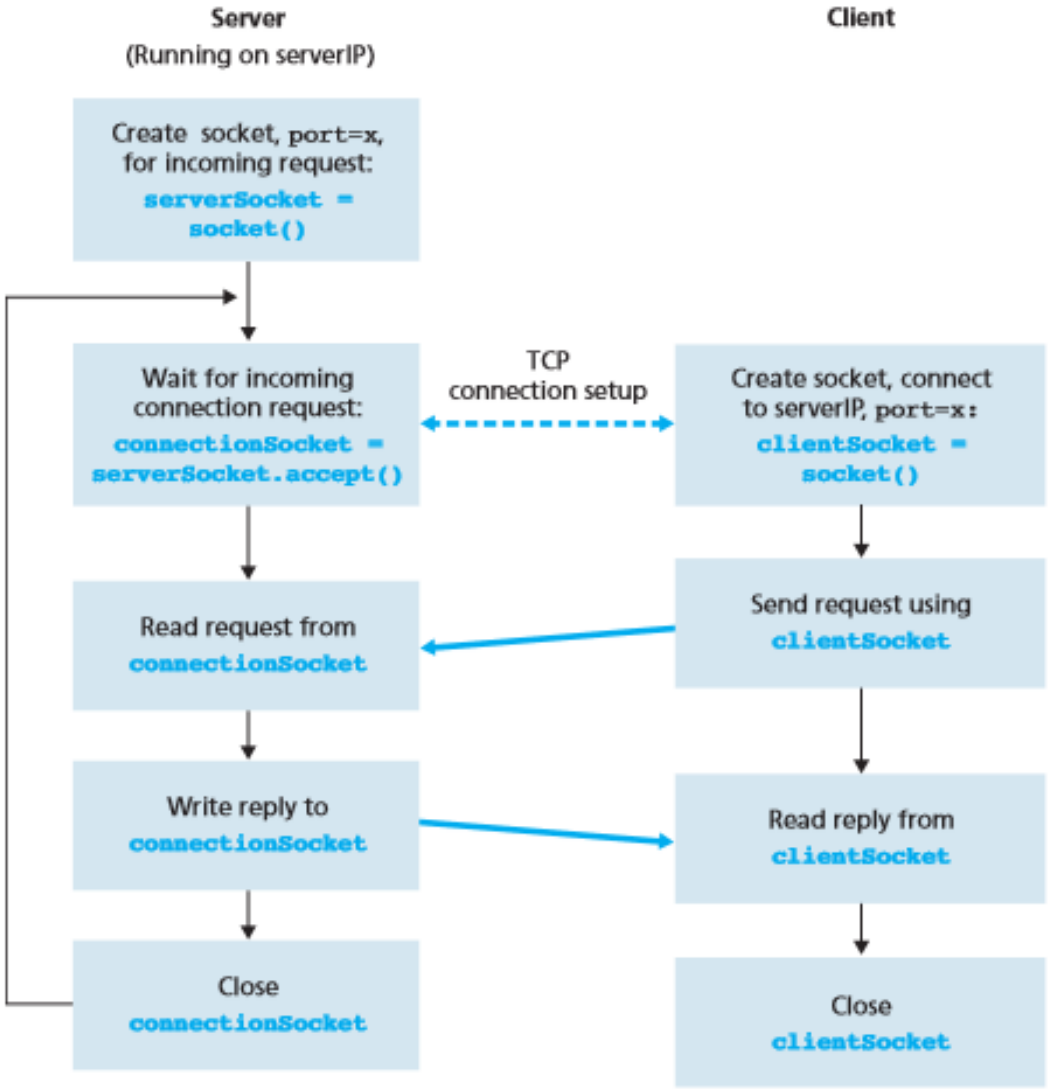


Figure 2.30 ♦ The client-server application using TCP

Example:

Here is the code for the client side of the application:

```
from socket import *
serverName = 'servername'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName,serverPort))
sentence = raw_input('Input lowercase sentence:')
clientSocket.send(sentence)
modifiedSentence = clientSocket.recv(1024)
print 'From Server:', modifiedSentence
clientSocket.close()
```

Server Program

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(('',serverPort))
serverSocket.listen(1)
print 'The server is ready to receive'
while 1:
   connectionSocket, addr = serverSocket.accept()
   sentence = connectionSocket.recv(1024)
   capitalizedSentence = sentence.upper()
   connectionSocket.send(capitalizedSentence)
   connectionSocket.close()
```

Experiment No 10

10. Write a program on datagram socket for client/server to display the messages on client side, typed at the server side.

Using User Datagram Protocol, Applications can send data/message to the other hosts without prior communications or channel or path. This means even if the destination host is not available, application can send data. There is no guarantee that the data is received in the other side. Hence it's not a reliable service.

UDP is appropriate in places where delivery of data doesn't matters during data transition.

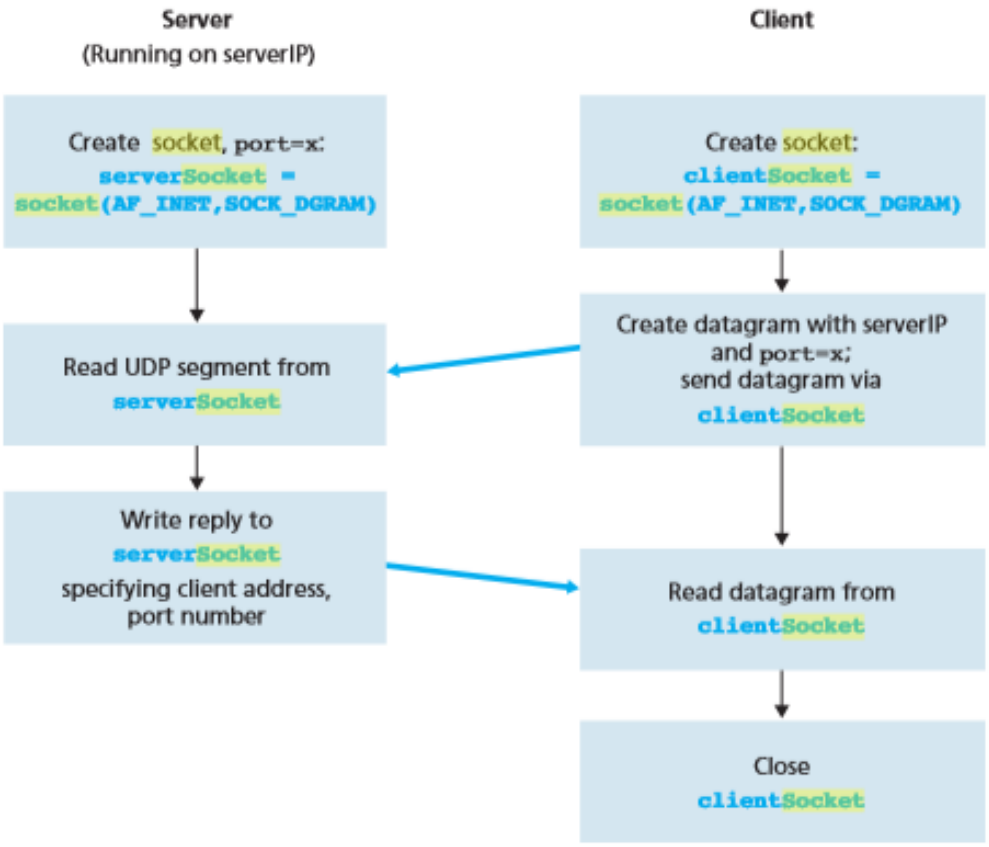


Figure 2.28 ♦ The client-server application using UDP

Experiment No 11
RSA Algorithm

11. Write a program for simple RSA algorithm to encrypt and decrypt the data.

Theory

Cryptography has a long and colorful history. The message to be encrypted, known as the plaintext, are transformed by a function that is parameterized by a key. The output of the encryption process, known as the ciphertext, is then transmitted, often by messenger or radio. The enemy, or intruder, hears and accurately copies down the complete ciphertext. However, unlike the intended recipient, he does not know

the decryption key and so cannot decrypt the ciphertext easily. The art of breaking ciphers is called cryptanalysis the art of devising ciphers (cryptography) and breaking them (cryptanalysis) is collectively known as cryptology.

There are several ways of classifying cryptographic algorithms. They are generally categorized based on the number of keys that are employed for encryption and decryption, and further defined by their application and use. The three types of algorithms are as follows:

1. Secret Key Cryptography (SKC): Uses a single key for both encryption and decryption. It is also known as symmetric cryptography.
2. Public Key Cryptography (PKC): Uses one key for encryption and another for decryption. It is also known as asymmetric cryptography.
3. Hash Functions: Uses a mathematical transformation to irreversibly "encrypt" information

Public-key cryptography has been said to be the most significant new development in cryptography. Modern PKC was first described publicly by Stanford University professor Martin Hellman and graduate student Whitfield Diffie in 1976. Their paper described a two-key crypto system in which two parties could engage in a secure communication over a non-secure communications channel without having to share a secret key.

Generic PKC employs two keys that are mathematically related although knowledge of one key does not allow someone to easily determine the other key. One key is used to encrypt the plaintext and the other key is used to decrypt the ciphertext. The important point here is that it does not matter which key is applied first, but that both keys are required for the process to work. Because pair of keys is required, this approach is also called asymmetric cryptography.

In PKC, one of the keys is designated the public key and may be advertised as widely as the owner wants. The other key is designated the private key and is never revealed to another party. It is straight forward to send messages under this scheme.

The RSA algorithm is named after Ron Rivest, Adi Shamir and Len Adleman, who invented it in 1977. The RSA algorithm can be used for both public key encryption and digital signatures. Its security is based on the difficulty of factoring large integers.

Algorithm

1. Generate two large random primes, P and Q, of approximately equal size.
2. Compute $N = P \times Q$
3. Compute $Z = (P-1) \times (Q-1)$.
4. Choose an integer E, $1 < E < Z$, such that GCD $(E, Z) = 1$
5. Compute the secret exponent D, $1 < D < Z$, such that $E \times D \equiv 1 \pmod{Z}$
6. The public key is (N, E) and the private key is (N, D).

Note: The values of P, Q, and Z should also be kept secret.
The message is encrypted using public key and decrypted using private key.

An example of RSA encryption

1. Select primes P=11, Q=3.
2. $N = P \times Q = 11 \times 3 = 33$
   $Z = (P-1) \times (Q-1) = 10 \times 2 = 20$
3. Lets choose E=3
   Check GCD(E, P-1) = GCD(3, 10) = 1 (i.e. 3 and 10 have no common factors except 1), and check GCD(E, Q-1) = GCD(3, 2) = 1
   therefore GCD(E, Z) = GCD(3, 20) = 1
4. Compute D such that $E \times D \equiv 1 \pmod{Z}$ compute $D = E^{\wedge -1} \bmod Z = 3^{\wedge -1} \bmod 20$
   find a value for D such that Z divides (($E \times D$)-1) find D such that 20 divides 3D-1.

Simple testing (D = 1, 2, ...) gives D = 7
Check: (E x D)-1 = 3.7 - 1 = 20, which is divisible by Z.

5. Public key = (N, E) = (33, 3) Private key = (N, D) = (33, 7).

Now say we want to encrypt the message m = 7,

Cipher code = $M^{\wedge E}$ mod N

= $7^{\wedge 3}$ mod 33

= 343 mod 33

=13.

Hence the ciphertext c = 13.

To check decryption we compute Message' = $C^{\wedge D}$ mod N =

$13^{\wedge 7}$ mod 33

= 7.

Note that we don't have to calculate the full value of 13 to the power 7 here. We can make use of the fact that a = bc mod n = (b mod n).(c mod n) mod n so we can break down a potentially large number into its components and combine the results of easier, smaller **calculations to calculate the final value.**

Experiment No 12
Leaky Bucket

12. Write a program for congestion control using Leaky bucket algorithm.

Theory

The congesting control algorithms are basically divided into two groups: open loop and closed loop. Open loop solutions attempt to solve the problem by good design, in essence, to make sure it does not occur in the first place. Once the system is up and running, midcourse corrections are not made. Open loop algorithms are further divided into ones that act at source versus ones that act at the destination.

In contrast, closed loop solutions are based on the concept of a feedback loop if there is any congestion. Closed loop algorithms are also divided into two sub categories: explicit feedback and implicit feedback. In explicit feedback algorithms, packets are sent back from the point of congestion to warn the source. In implicit algorithm, the source deduces the existence of congestion by making local observation, such as the time needed for acknowledgment to come back.

The presence of congestion means that the load is (temporarily) greater than the resources (in part of the system) can handle. For subnets that use virtual circuits internally, these methods can be used at the network layer.

Another open loop method to help manage congestion is forcing the packet to be transmitted at a more predictable rate. This approach to congestion management is widely used in ATM networks and is called traffic shaping.

The other method is the leaky bucket algorithm. Each host is connected to the network by an interface containing a leaky bucket, that is, a finite internal queue. If a packet arrives at the queue when it is full, the packet is discarded. In other words, if one or more process are already queued, the new packet is unceremoniously discarded. This arrangement can be built into the hardware interface or simulate d by the host operating system. In fact it is nothing other than a single server queuing system with constant service time.

The host is allowed to put one packet per clock tick onto the network. This mechanism turns an uneven flow of packet from the user process inside the host into an even flow of packet onto the network, smoothing out bursts and greatly reducing the chances of congestion.