Computer Networks Lab Practicing with the NS simulator

B. Stojcevska[1], A. Misev[2], M. Gusev[3]

Abstract – **In this paper we overview lab practicing exercises for generic computer networks course curricula. We will also present statistical data that shows how the new way of teaching affects the students' skills and knowledge obtained, and also their average grades and score.**

I. Introduction

Upgrading of the teaching process does not only mean introducing new curricula and new courses, but also developing and applying new teaching and grading methods. Advancing through the ECTS, we introduced the system to assign the students various homework assignments and projects, supported by tools that will provide them with the necessary environment to realize the tasks.

The Computer Networks course consists of several major topics: data communication standards, various protocol models, data transmission, transmission media, coding, communication interfaces, flow control, multiplexing, circuit and packet switching, LAN and IP protocols. It represents a theoretical introduction to the main concepts of the computer networks. The realization of the curriculum was done through lectures, exercises and laboratory exercises. It is hosted at: http://twins.pmf.ukim.edu.mk/courses/mrezi/. This paper addresses lab practicing part of the course, where students could score 8 - 10% of the total credit.

II. NS simulator projects

The *ns* (Network Simulator) is an object-oriented platform aimed for research in the domain of computer networks. *ns* is a tool for discreet simulation of events and supports implementation and utilization of a set of network protocols and topologies.

The simulator is a part of the VINT (Virtual InterNet Testbed simulator) as a joint effort by people from UCB (University of California Berkeley), LBNL (Lawrence Berkeley National Laboratory), USC/ISI (University of Southern California's Information Sciences Institute) and Xerox PARC (Palo Alto Research Center).

ns supports simulation of variety of IP networks and protocols, such as TCP and UDP, traffic sources with FTP, Telnet, Web, CBR and VBR behavior, queuing management mechanisms, routing protocols and MAC protocols for local area networks. It has an open architecture and supports growth and adding of new protocols. That is why it is widely used in the Internet and computer networks research community. We used the version *ns-2* as a tool for examining TCP behavior in different network scenarios.

A. Using ns

The simulator is an OTcl interpreter and is used by writing OTcl scripts with simulation scenario specifications. These scripts contain: basic network elements – nodes, links and queues; the transport connections and the traffic that goes over the transport connections at the application layer; event handling procedures, and events.

The new network objects can be defined by using the objects from the ns library. The data path between the objects is defined in the OTcl scripts by object plumbing and specifying the behavior of the data sources. The simplicity of object combining makes ns a powerful tool for developing and testing new network protocols, especially during their development.

There are a variety of possibilities to display data, which are the result of a running of a script. They can be sent to standard output, but a more often way is to put the output data in trace files. The contents of a trace file do not include overall simulation information, but only records about packet events. Calculation of the performance parameters, such as throughput or packet loss, based on the trace files usually requires writing specific programs or importing the data into spreadsheet application.

The ns platform includes the Network Animator (NAM) tool, which can do a graphical simulation display. NAM has a GUI interface with buttons that control the animation flow. It can produce graphical information about the simulation parameters, and give information about the objects in the simulation, but it cannot be used for accurate simulation analysis.

B. The simulator structure

The basic ns components are: event scheduler; library of network components and plumbing modules; events and event handlers.

[1]B. Stojcevska is with the University "Sts. Cyril and Methodius", Faculty of Natural Sciences and Mathematics, Arhimedova 5, 1000 Skopje, Macedonia, E-mail: biljanas@ii.edu.mk

[2]A. Misev is with the University "Sts. Cyril and Methodius", Faculty of Natural Sciences and Mathematics, Arhimedova 5, 1000 Skopje, Macedonia, E-mail: anastas@ii.edu.mk

[3]M. Gusev is with the University "Sts. Cyril and Methodius", Faculty of Natural Sciences and Mathematics, Arhimedova 5, 1000 Skopje, Macedonia, E-mail: marjan@ii.edu.mk

An event in ns is a unique packet identifier, time of execution and event handler that will perform an action. The event scheduler schedules the events in an event queue. When an event needs to take place, the scheduler activates the network component that should handle the event. The network component communicates by exchanging packets.

ns is implemented in C++ and it uses OTcl (Object Tool Command Language) as a command and configuration interface. There is a C++ class hierarchy and an OTcl class hierarchy. Both hierarchies are closely related *Figure 1*. The C++ classes contain the implementation of the objects that handle the packets as a compiled library. Each C++ object corresponds to an OTcl interpreted object. The OTcl classes are used for object handling and simulation configuration.

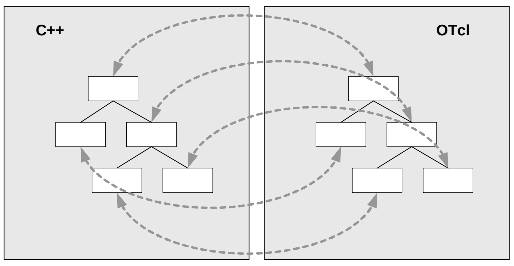

Figure 1: C++ and OTcl duality

The choice of the programming language depends on the application. The structure of the platform and the characteristic of the languages imply the following recommendations:

- ≠ C++ for data – when a packet handler is implemented or the required function is not implemented
- ≠ OTcl for management – for object manipulation, defining simulation parameters and time scheduling.

C. Student projects

We used the ns educational scripts database available at http://www.isi.edu/nsnam/repository/topics.html. The database contains the following simulation scripts divided in several topics:

Application

≠ multipleappPa.tcl: multiple pareto sources on a link. Defines three pareto traffic sources on a link.

TCP

≠ bwxd.tcl: Demonstrates and explains TCP's stop-and-wait behavior. Stop-and-Wait protocols have some desirable properties. They are ACK-clocked and automatically adjust the transmission speed to both the speed of the network and the rate the receiver sends new acknowledgements. They respect the conservation principle of computer networks.

≠ A1-stop-n-wait.tcl: Another script showing TCP's stop-and-wait behavior. Stop-n-wait" is the fundamental technique to provide reliable transfer under unreliable packet delivery system. After transmitting one packet, the sender waits for an acknowledgment (ACK) from the receiver before transmitting the next one. In this way, the sender can recognize that the previous packet is transmitted successfully and we could say "stop-n-wait" guarantees reliable transfer between nodes.

≠ ack_clock.tcl: Demonstrates and explains TCP's sliding window algorithm. Similar to A1-stop-n-wait.tcl.

≠ SlowStart.tcl: Shows TCP's slow-start algorithm. This scenario shows TCP using slow-start and congestion avoidance.

≠ NoSlowStart.tcl: Shows TCP without slow-start. This simulation shows the behavior of TCP before Van Jacobson added features like slow start, a revised RTT estimator, and congestion avoidance.

≠ A4-slow-start.tcl: Another script demonstrating slow-start. "slow start" is kind of "sliding window", the difference between "sliding window" and "slow start" is that "slow start" allows various window size while "sliding window" has fixed window size. Using "slow start", the sender can transmit as much packets as the network can deliver. The basic idea behind "slow start" is to send packets as much as the network can accept. It starts to transmit 1 packet and if the that packet is transmitted successfully and receives an ACK, it increases its window size to 2, and after receiving 2 ACKs it increases its window size to 4, and then 8, and so on. "Slow start" increases its window size exponentially.

≠ NoSlowStart[1].tcl: Another script demonstrating TCP with no slow-start.

≠ D1-m-decrease.tcl: Shows multiplicative decrease of TCP's window size. When a network is getting congested and packets are lost or when a network delay increases and packets timed out, TCP adjusts its window size to fit the congested environment. "Multiplicative Decrease" in TCP exists for that. Whenever the sender recognizes that there is a collision - by receiving duplicated ack or timeout -, it decrease its window size by half of current window size (while the value is greater than 2) and transmits data in "slow-start" fashion, so that TCP can deliver its data reliably and efficiently even in a congested environment. This feature is necessary in the Internet.

≠ fast-retransmit.tcl: Shows fast-retransmit behavior of TCP's congestion avoidance algorithm. Fast retransmit is a modification to the congestion avoidance algorithm. As in Jacobson's fast retransmit algorithm, when the sender receives 3rd duplicate ACK, it assumes that the packet is lost and retransmits it without waiting for a retransmission timer to expire. After retransmission, the sender continues normal data transmission. That means TCP does not wait for the other end to acknowledge the retransmission.

≠ Fast-retransmit.tcl: Another script for fast-retransmit

≠ Fast-recovery.tcl: Shows fast-recovery behavior of TCP's congestion avoidance algorithm. Fast Recovery is now the last improvement of TCP. With using only Fast Retransmit, the congestion window is dropped down to 1 each time network congestion is detected. Thus, it takes an amount of time to reach high link utilization as before. Fast Recovery, however, alleviates this problem by removing the slow-start phase. Fast Recovery algorithm has been implemented in TCP since Reno release. It collaborates with Fast

Retransmit algorithm in an algorithm called "Fast Retransmit/Fast Recovery Algorithm".

≠ fast-rxt-recovery.tcl: Another script showing fast retransmit/fast recovery algorithm

≠ multiple-drops.tcl: Shows TCP's failure to do fast retransmit/fast recovery incase of multiple drops in the same window. This script demonstrates TCP failure to trigger fast retransmit and fast recovery algorithms when there are multiple drops in the same window. The cwnd drops to 1 due to a timeout and TCP enters slow start.

≠ delayed_acks.tcl: Shows delayed ack method used by TCP for cumulative acknowledgements. TCP uses cumulative acknowledgements, that is, it can acknowledge up to a certain sequence number in the same ACK. Until the moment we have seen the receiver acknowledge each segment received individually, but it is possible to wait a certain amount of time before sending the acknowledgement in order to acknowledge several segments in the same ACK

Other transport protocols

≠ chain.tar: Demonstrates SRM's (a reliable multicast protocol) loss recovery for a chain topology. SRM is a reliable multicast protocol, building the reliability on an end-to-end basis. SRM members, who detect loss wait a random time and then multicast their repair request. Any node, which has the data, waits for a random time and sends the repair data. A Chain network topology, where all nodes in the chain are members of the multicast session. The loss recovery is deterministic in this topology. The deterministic suppression will ensure that there will be only one request and one repair. The animation illustrates scenarios where SRM handles losses in a chain topology and interaction of SRM session messages and loss recovery.

≠ star.tar: Demonstrates SRM's (a reliable multicast protocol) loss recovery for a star topology. SRM is a reliable multicast protocol, building the reliability on an end-to-end basis. SRM members, who detect loss wait a random time and then multicast their repair request. Any node, which has the data, waits for a random time and sends the repair data. A Star network topology, where all nodes in the chain are members of the multicast session.

≠ star-cmplx.tar: The animation illustrates scenarios where SRM handles losses in a star topology. SRM is a reliable multicast protocol, building the reliability on an end-to-end basis. SRM members, who detect loss wait a random time and then multicast their repair request. Any node, which has the data, waits for a random time and sends the repair data. A Star network topology, where all nodes in the chain are members of the multicast session. The loss recovery is purely probabilistic in this topology. The number of request and repair messages in this topology is dependent on the randomness of the backoff timer values picked by the nodes. The nodes estimate the group size and timer using the session messages. The animation illustrates scenarios where SRM handles losses in a star topology and there are duplicate requests for a single loss.

≠ star-repair.tar: The animation illustrates scenarios where SRM handles losses in a star topology pgm.tar: Script showing PGM, another reliable multicast protocol. SRM members, who detect loss wait a random time and then multicast their repair request. Any node, which has the data, waits for a random time and sends the repair data. A Star network topology, where all nodes in the chain are members of the multicast session. The loss recovery is purely probabilistic in this topology.

≠ pgm-disable.tar: Shows aspects of PGM behavior. PGM is Pragmatic General Multicast, another reliable multicast protocol that guarantees reliability by making the intermediate nodes aware of the losses. Protocol is based on sequenced packets. Node, which detects loss, sends a NAK upstream back to the source hop-by-hop and each node records the state and replies with a NCF or the NAK confirmation. This provides the reliability of the NAK for each hop.

≠ pgm-rptrtx.tar: Another script that shows aspects of PGM behavior.

Routing

≠ test-mcast-PimDm.tcl: Shows PIM (Protocol independent multicast) behavior for dense mode. Protocol Independent Multicast (PIM) is a multicast protocol, which is not dependent on any unicast routing protocols. This independency makes PIM contrast to the traditional multicast routing protocols such as DVMRP and MOSPF. DVMRP is based on distance-vector unicast routing protocol. MOSPF is based on link-state unicast routing protocol. There are 2 implemented modes of PIM: PIM dense mode (PIM-DM) and PIM sparse mode (PIM-SM or PIM). PIM-DM is suitable for an area which members of a multicast group are distributed densely. PIM-SM is developed for a situation that members of a multicast group are distributed sparsely in a network. This simulation is about PIM-DM.

≠ dm.tar: Another script for PIM-DM

≠ st.tar: Shows PIM for sparse mode. PIM-SM is protocol independent multicast for sparse multicast membership. The PIM is independent of the underlying unicast routing protocol. The animation here illustrates the PIM-SM for a random network topology.

≠ dvinfty.tcl: Demonstrates the classic count-to-infinity problem faced by distance-vector routing algorithm

Queue management

≠ red-queue.tcl: Shows RED (Random Early Detection) mechanism used for congestion avoidance. RED (Random Early Detection) is one of congestion avoidance mechanisms. It is placed in gateways (routers). Each gateway is programmed to monitor its own queue length. When it detects that congestion is imminent, it notifies the source, which is believed to be the cause of congestion, by dropping one of its packets.

≠ droptail-queue.tcl: Shows simple drop-tail (first-in, first-out) policy

Each student was given one of the scripts in the database. The student's task was to perform the simulations with the script and then analyze the output. Based on the results, they had to write a seminal paper including the following: analysis of the simulation script; description of the protocols and mechanisms that are performed in the script; graphical representation of the relevant simulation parameters based on the trace files; conclusions about the protocol performance.

Our main goal was to train the students to use ns for understanding various computer network architectures and network protocol behaviors. This is not a trivial task since ns is research oriented platform and doesn't include user-friendly training tools adequate for beginners usage.

D. Implementation of the projects

During the course, we have spent 4 hours for training the students to use ns for performing simulations and interpretation of the trace files. The students required 6 hours to implement their tasks.

In addition, we briefly present one of the projects that were graded with highest points. The student performed the script fast-rxt-recovery.tcl that shows the fast retransmit/fast recovery algorithm in TCP. The network topology is a simple topology with two nodes, source and destination nodes, connected with a 10 Mbps link. There are two TCP agents at each node. The simulation duration is 1 sec. At time t=0.1216 a packet is dropped on the line After running the script, the student gave a detailed description about the packets behavior, the values of the congestion window and the values of the congestion threshold in a scenario where. The main accent was set on explaining what happens when the packet is dropped and how TCP fast retransmit and fast recovery are triggered. Instead of going to slow start, TCP keeps sending with congestion window with constant size.

Figure 2 gives a graphical representation of the packet sending process on a time axis.

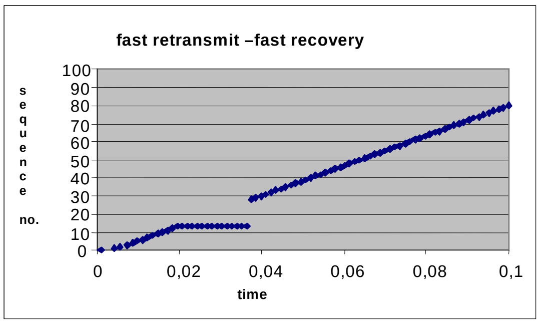

Figure 2 Fast retransmit/fast recovery

Our overall opinion is that the students managed to become skilled in using ns at beginner level successfully. This knowledge helped them to understand the inner working of the TCP/IP architecture, be able to create computer networks with various topologies and analyze the results of ns simulations. The students' response to the projects was extremely high. Out of 51 students, 45 students completed the ns project. The average score of the project was 17 out of 20. The point distribution is given in *Table* I.

Table *I*
Point distribution

Points	*Students*
19-20	25
17-18	4
15-16	4
13-14	0
11-12	4
1-10	9

III. Achieved results

Out of the 51 students that took the course "Computer Networks", 20 students finished the course by the end of the semester, approaching the 40% with average grade of 71%. Additional 10 students passed the course by exams instead of colloquia. Their average grade is 86%. The overall average grade was 76%. Our estimate is that the low average grade is mostly due to the nature of the courses. An interesting fact is that the average grade is higher for the students that took the exam. This is due to the fact that most of these 10 students took the first colloquia, but were not satisfied with the results and decided to take the exams. Over 90% of the students finished their *ns* homework with respect to the deadline. This was also the case with the second, programming project.

The grade distribution of the students passing the course is given in [5]. We evaluate the ns usage in the computer networks course as very useful. Each student had the chance to experiment on its own since each student was given individual and different homework. Unfortunately, we cannot compare the results with previous years, since this is the first course in computer networks. We propose to continue the usage of the simulator in the teaching process.

Literature

1. Fall, K., Varadhan, K., "The ns Manual", LBNL, http://www.isi.edu/nsnam/ns/ns-documentation.html, Nov 2001
2. The Network Simulator - ns-2, the VINT project, http://www.isi.edu/nsnam/ns/
3. NS Educational Module Repository, http://www.isi.edu/nsnam/repository/index.html
4. Chung, J., Claypool, M., NS by Example - http://nile.wpi.edu/NS/
5. M. Gusev, "New methodology and evaluation system", Proc. TEMPUS CD JEP 16160-2001 project workshop