The Network Workbench: Network Simulation Software for Academic Investigation of Internet Concepts

J. Mark Pullen

Department of Computer Science, George Mason University, Fairfax, VA 22030, USA
email: mpullen@gmu.edu

Abstract

Simulation offers significant advantages as a basis for academic projects in computer networking. Because many unimportant details can be abstracted away, and also because simulations can be completely repeatable, it is possible to address the same concepts more quickly than is possible with actual networks. At the same time, students who program a protocol for a network simulator come to understand the protocol much better than if they learn only from reading and lectures. This paper reports on a new network simulator, the Network Workbench, which is intended for use in the academic environment. It is based on discrete event simulation and structured around a five-layer stack abstracted from the Internet protocols (TCP/IP family). While the Workbench is less powerful than some tools used for investigation of larger networks or more complex protocols, it has compensating advantages. Its use can be learned quickly and it is sufficiently powerful, comprehensive, and extensible to allow investigation of a considerable range of problems. The Workbench, which is available to the academic community under no-cost license, includes a set of protocol programming exercises for introductory networking courses and it also has proved usable for more advanced student research projects. This paper describes the philosophy behind the workbench, gives a brief outline of its history, explains its internal structure, and describes its use in computer network teaching and research.

1. Introduction

Teaching introductory networking poses a challenge. We know that students learn best those concepts that are reinforced by activities that require them to use the concepts taught. However, networking is a complex, interdisciplinary subject. The simplest meaningful project could easily bog the student down for long periods of time in tangential details of socket-level programming, debugging programs on active networks, and understanding the relationships of the numerous protocols with few supporting tools. Furthermore, creating a running protocol at the lower layers of the stack typically requires modification of the computer's operating system, which is not a suitable activity for an introductory course. A project is needed that can support the broad range of topics needed to convey a basic understanding of computer networking today.

Faced with these difficulties, the author has concluded that a network simulation environment represents the best compromise between the problems associated with student network programming and the need for students to reinforce classroom learning by doing a real project. Simulation is widely used to abstract away non-essential physical details (which are also likely to be non-repeatable in testing) while exercising the critical aspects of a protocol. Simulations run as user programs in the computer, avoiding problems associated with modifying the operating system.

This paper reports on the results of a five-year effort which has produced a new network simulator, called the Network Workbench, aimed at academic investigation of Internet protocol concepts. The Workbench is available to the academic community under no-cost license. It contains a complete protocol stack, abstracted from the Internet stack, and a set of exercises that focus on critical protocol algorithms in the Internet stack. It also has proved sufficiently powerful, comprehensive, and extensible to serve as a basis for advanced student research projects.

Over the past five years the Workbench has matured. Starting as a programming assignment given to Master's-level classes, it has grown into a capable network simulator. The guiding principles in its development have been:

- abstract away unnecessary details, while ensuring that the student must grapple with the central algorithms of the protocols by programming them in C++;
- keep the whole simulation system simple, understandable, and independent of physical presence in the lab where the simulation software is maintained, so that it can run on the student's personal computer under Windows, as well as in the Unix laboratory;
- provide as much infrastructure as possible to support the learning process; in particular, provide working interfaces between layers in the protocol stack;
- provide code of good quality, with all details visible except the actual solutions to graded exercises (as a side-effect of this project, students gain valuable experience in working with code written by others, a task many can expect to take on after graduation); and
- provide complete visibility into all aspects of the network being simulated, including the ability to see protocol data units as they move between the layers in the stack, and statistics showing performance metrics for each trial run.

The remainder of this paper begins by explaining the motivation for developing the Network Workbench. Following this are descriptions of the structure of the Workbench at the system level, and then from perspectives of detailed mechanisms, software structure, network architecture, and student projects. The paper ends by describing the author's experiences in using the Workbench for teaching and graduate academic research.

2. Motivation

The author teaches networking courses in the George Mason University (GMU) Computer Science curriculum to graduate students and to undergraduate Seniors. The ideal project assignment for introductory courses builds on the Computer Science students' ability to program, because in creating a working program they necessarily come to understand the protocol algorithms. Therefore the author set out to create a course project where students program important protocol algorithms at all levels of the stack associated with the Internet protocols (TCP/IP family).

Simulation offers abstraction and insulation from irrelevant problems of the physical world. These are very attractive capabilities for use in an educational environment. For this reason, starting in 1993 the author began assigning network simulations as part of introductory courses for Computer Science graduate students and later in courses for undergraduate Seniors. At first these assignments required the student to develop the entire simulation. However, it soon became apparent that students were spending too much time on simulation concepts, and too little time on protocol concepts. It was therefore natural to consider using an existing computer network simulation system to support their learning process.

Several very powerful network simulators are available today. However, available network simulators which work at the protocol level, such as OPNET [5] and ns (see http://www-mash.cs.berkeley.edu/ns/ns.html) are highly complex tools that take a long time to master, and require an advanced Unix workstation environment. Analytic simulations such as COMNET (http://www.caciasl.com) are simpler to use but do not provide exposure to the internal working of the protocols. Other available simulation systems that work at the protocol level, *e.g.* BoNES (see http://www.cadence.com/alta/

products/bonesdat.html), provide powerful capabilities but also require an extensive learning period.

For example, the Optimized Network Engineering Tools system (OPNET) from MIL3, Inc. provides detailed discrete event simulations of several different networking technologies, including common Internet protocols. OPNET has a powerful graphic interface and very useful tools for creating test drivers and capturing/graphing network response. At GMU we have built an entire family of models for proposed Internet resource-reserved multicast protocols, which have been downloaded for use by several other groups following its announcement through the Internet Engineering Task Force [9]. In developing these models we found that students using OPNET require several months to develop proficiency.

Another powerful network simulator is ns, which has been developed under the VINT project by the University of Southern California-Information Sciences Institute, the University of California-Berkeley, Lawrence Berkeley Laboratory, and Xerox Palo Alto Research Center. ns combines simulation and emulation to represent a WAN with high fidelity, using a system of workstations interconnected by a LAN. Protocols to be investigated in ns are programmed to run on a real network. The simulator adds models of point-to-point wide-area links so that the overall ns system emulates a network where the protocols function in an environment that is very much like a real WAN. The ns environment has the advantage that it is not subject to physical problems of an actual WAN, such as unexpected link degradation and difficulties in observing behavior of a distant system. However, like OPNET, ns is quite a sophisticated system with a steep "learning curve." It requires the user to program actual, working protocols in real, working workstations, with all of the attendant complexities of a functional environment. While this is an excellent way to develop new protocols, the time investment required simply is too large for an introductory course intended to introduce students to networking fundamentals.

Clearly a greater level of abstraction is appropriate in projects for introductory courses which introduce the entire protocol stack in one semester. One possibility is to use a protocol development environment such as the *x*-kernel rather than a simulation. The *x*-kernel has an educational role similar to that sought from simulation in a teaching environment, in fact its use is featured in a networking textbook [7]. It provides a mature protocol implementation environment that is easier to learn than OPNET or ns. However, because the *x*-kernel requires an actual protocol implementation, it gives up some advantages that are available from simulation. It requires a multi-workstation Unix system and a major student time commitment to learn the tool. Also the project must be tightly focused on the network and transport layers, rather than the broader range of project topics enabled by a simulation-based project.

After considering all options, simulation is still the most promising basis for projects in an introductory course, but no existing system was found appropriate to this purpose. Each of the major network simulation systems described has undeniable merits, but requires too much learning time, and demands a hardware and software environment that is too complex. Therefore the Network Workbench has been developed to provide an infrastructure that allows students to focus on protocol algorithms without worrying about details of the simulation. The Workbench has progressed through four distinct phases:

Version 0: In 1994 - 1995 an experimental discrete event simulation infrastructure was developed by students in the GMU HyperLearning Center (formerly Center for the New Engineer or CNE) [3] under guidance from the author. This was part of a larger collection of "workbench" software designed to expose students to important concepts in networking and other areas (see http://cne.gmu.edu/workbenches). The Network Workbench subsequently grew in different directions from the other CNE workbenches.

Version 1: By Fall, 1994 the author and students had expanded the Workbench into a usable collection of software, based around the experimental Workbench discrete event simulation infrastructure, and complete enough to be used for class exercises. Version 1 lacked a coherent system structure. In many cases the modules were hastily programmed and were released to students only as binary executables because of their low quality. Nevertheless it still provided a highly valuable learning experience.

Version 2: Starting in Fall, 1997 the author undertook a complete redesign of the earlier Workbench, providing a coherent system structure, fully object-oriented programming, and visibility of all code modules (except project solutions) in the form of C++ source code. In many academic programs C++ is now the first programming language taught. Therefore the C++ abstraction *class* (which is not available in C) becomes a natural medium of understanding for students that can be used to good advantage in the project. Version 2 also served as the basis for projects in a more advanced, project-based graduate course for Master's and doctoral students.

Version 3: The latest version of the Workbench builds on Version 2 to provide both LAN and WAN models. It contains an internetted architecture that supports teaching many key aspects of today's networking environment. These include network topology, DLC error control, CSMA/CD collision backoff, optimal route computation, reliable transport, multicast, and LAN/WAN integration. In its current form, the Workbench consists of about 9000 lines of heavily-commented C++ code, plus associated documentation and installation scripts for Unix and Windows systems. Version 3.2 is available for download from http://netlab.gmu.edu/ networkbench, under no-cost license.

3. Structure of the Network Workbench

When the Workbench software runs it creates a simulation of a network, using student-programmed protocols. The simulated network passes messages representing an "email" server application, which the simulated hosts read from a set of data files. These data files are distributed with the software. The Workbench and its projects have been structured in such a way that students must grapple repeatedly with the protocol stack layers and their functions, the protocol algorithms, and their data structures. The data structures are nested, just as in any working protocol stack. Frames contain packets, which in turn contain transport segments.

3.1 System-level structure

Figure 1 shows the general structure of the Workbench. The top-level system blocks are:

Header files: A small, compiler-specific header file establishes any code definitions that differ among the supported compilers (currently Unix compilers Sun C++ under Solaris and Gnu G++ under Linux, and also Windows compilers Borland C++ Builder and Microsoft Visual C++). This is followed by the main header file, wkb.h, which defines the base classes as well as data types, data structures, and state enumerations for the entire Workbench.

Main program: The main execution sequence demonstrates the basic simplicity of the Workbench:

```
int stack::simulation()
{
  set_runparms();// set printout conditions
  print_authors();// print out authors' names
  if(read_net())   // initialize network
   {
    if(WAN)create_topology();
    print_topology();
    if(WAN)compute_routes();
    startup_simulation();
    }
 // run simulation
 while(next_event() && keep_running);
 // print performance summary
  statistics();
  return SUCCESS;
}
```

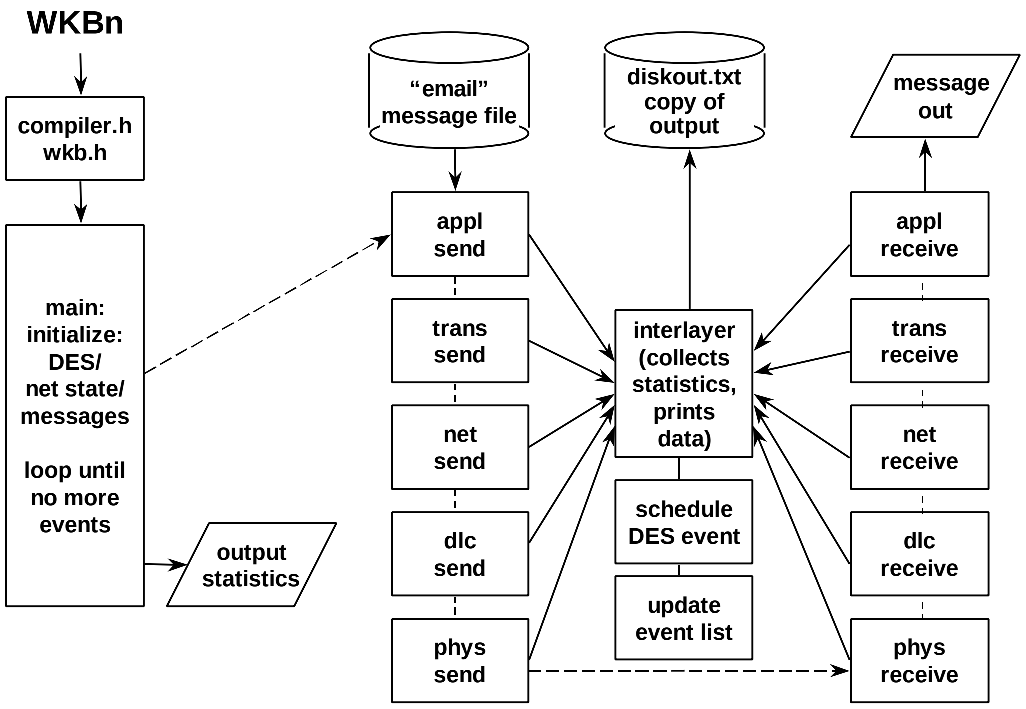

Fig. 1. Network Workbench System Architecture

The core of the simulation is the single line *while(next_event( ) && keep_running);* that repeatedly invokes the DES function *next_event(),* which in turn invokes the C++ function corresponding to the next event on the DES event list. The *keep_running* test allows an "escape" that can be used if any Workbench module needs to create a "panic stop."

Interlayer module: This is the single component that is responsible for much that is unique in the Workbench. Whereas in a normal protocol stack each layer invokes the next lower layer to obtain service, in the Workbench this invocation passes through an intermediate function contained in the *interlayer* module. This module collects statistics about the operation of the simulated network, prints an optional trace between any two layers, and invokes the DES subsystem to create a future event that will cause the next lower layer to be invoked. The architecture facilitates exposing or hiding actions of various layers as required by the project at hand, using a profile array which can be reset by the user during the simulation. It also provides a mechanism whereby execution times of the individual protocol layers can be represented explicitly in the simulation.

Input files: The "email" application consists of a collection of files generated by the Unix *fortune* program containing tidbits of wit and wisdom. Files are *email1.txt* through *email3.txt*, each holding a number of messages appropriate to the assignment. The application layer adapts itself to the simulation to be performed, representing WAN, LAN, or internetted hosts.

The network topology input comes from a set of four files (one mandatory, three provided as needed). The mandatory file is *nnets.txt*, which contains a single two-digit number that becomes *nnets* in the simulation. Because the Workbench simulates an internetwork, *nnets* is both the number of routers (if *WAN=TRUE* in the simulation) and the number of LAN subnets (if *LAN=TRUE*). The other three files provide the links matrix for the WAN, the number of hosts for each LAN, and the number of multicast hosts for each LAN (when *MULTICAST=TRUE*).

Protocol stack: the Workbench stack is abstracted from the Internet (TCP/IP) stack. Like the Internet stack it contains five layers. Much of the detailed C++ code in the Workbench is found in modules for the application layer *al*, transport layer *tl*, network layer *nl*, datalink control layer *dl*, and physical layer *pl*.

Discrete event simulation subsystem: this small and highly robust part of the Workbench is responsible for maintaining a doubly-linked event list, removing the latest event from the list when *next_event( )* is invoked, and invoking the function corresponding to that event.

Output files: the text outputs of the Workbench are the initial network description, the results of interlayer traces, and the summary statistics (selectable according to the layer from which they were collected). All text outputs are also written to file *diskout.txt* so the results of the simulation run can be reviewed. A common evaluation strategy for student Workbench projects is to have the student-written C++ code file and file *diskout.txt* submitted as attachments to electronic mail, for paperless grading. (The assignment includes an admonition that the grader may run the code in order to confirm that it yields the solution submitted.)

Project code stubs: each code module to be written by students is provided in the Workbench library in the form of a stub module with correct C++ interfaces, but containing no protocol code. There is a module *partn.cpp*, where *n* is one of the project parts, that establishes default conditions appropriate to the project part. It also presets the layers at which protocol data transfer will be traced and selects the statistics to be output, however these presets can be overridden dynamically in student-written code for selective tracing. A *setup* process is available to move all required data files and code files to a working directory, in order to make initial use of the Workbench simple for the student.

Executable module library: The basic functions of the Workbench are collected in a single compiled executable module, *wkcore.obj*. Executables for all other code modules are available in a library file. The library includes all project solution modules (for these, C++ code is not made available to the student). It also includes other modules that the student might want to replace, for example the bit error pseudo-random number generator for network links.

3.2 Basic Mechanisms Embodied in the Workbench

Internetting: The Workbench models an abstracted version of the Internet architecture. Each node in the Workbench network has two integer attributes: *netnum* (equivalent to the Class A, B or C network number in an Internet address) and *nodenum* (equivalent to the host number in an Internet address). Each interface of each node has three integer attributes: *netnum* and *nodenum* (associated with the host) and *ifacenum* (which uniquely identifies the interface within the host). *ifacenum*=0 is assigned to the node's interface to its LAN subnet. All nodes on the same LAN have the same value of *netnum*. Other values of *ifacenum* connect to serial links among nodes. There are conventions for the *nodenum* assigned to the LAN's router, its one multicast group, and its broadcast function, and to the *netnum* for WAN's one multicast group.

Each interface in the Workbench has an eight-bit layer-two address or "port number" which is used to identify that interface in DLC and MAC frames, and is globally unique like an Ethernet address. The first 2**nlinks* addresses identify the ends of WAN links; subsequent addresses are assigned to LAN interfaces. These conventions, and also the fact that addresses are associated with nodes rather than interfaces, are considerable simplifications on the Internet architecture. Nevertheless experience has shown that, while they simplify the Workbench considerably, they do not cause unacceptable deviations from the basic concepts of the Internet protocols.

Discrete Event Simulation: In a computer network there are many asynchronous (unsynchronized) events occurring constantly, yet the whole network must function as a coordinated, distributed system. Discrete Event Simulation (DES) is widely used to study such complex systems by abstracting the key distributed system functions [6]. In the Workbench, every simulated event is discretized to a time that is represented by an integer value *sim_timer* which is a count of some basic time unit such as microseconds. Every event happens on a specific value of *sim_timer*. The DES routines are responsible for accounting, in an efficient way, for all events that "have not yet happened." Typically the result of an event happening is for one or more new events to be scheduled. When events "happen" they invoke C++ functions. This arrangement provides a simple mechanism for representing the multiple, asynchronous threads of control needed to simulate a network.

Workbench DES Function *next_event( )* causes the function for the next event "waiting to happen" to be invoked. The top level of the Workbench program consists of invoking this function indefinitely, until the event list is empty or a time limit is reached. The DES routines maintain a two-dimensional linked list of events, with the first dimension ordered by "happen time" and second dimension FIFO by order of scheduling within a particular "happen time." This supports efficient event list search and update. After an event is scheduled, the current Workbench DES has no mechanism for "unscheduling" it. When the LAN simulation for the Workbench was developed, it became evident that such a mechanism would be useful, however in light of the complexity this would have added to the DES functions, an efficient alternative was discovered. The approach, described in [10], schedules frames to be received but deletes the frame at receive time if a collision has occurred.

Stochastic simulation: The Workbench uses pseudo-random number generators to create two types of events critical to behavior of networks: message arrivals and datalink errors. Each message stream and each datalink has its own independent stream of pseudo-random events, which are generated by functions that can be replaced by the user. The built-in functions are deterministic (fixed time interval) for best-effort and reliable messages, intended to reflect the behavior of humans generating email. Poisson (exponential inter-arrival) is used for multicast traffic representing multimedia streams, and also for bit errors. Because the random number generators associated with the various compilers supporting the Workbench do not produce consistent results, the Workbench contains its own random number generators based on the mixed congruential method [4]. The pseudo-random number sequences are seeded individually with values that are the same from run to run, so that the events, while "random," always occur at the same point (a considerable help in debugging). To change a random number generator distribution, the student needs only to provide a generator functions of the same name.

Input/Output: The "email" inputs, WAN topology, number of hosts on each LAN, and multicast topology are contained in files with simple text format that can be edited by students. The interfacing "interlayer" module collects statistics on invocation of the various layers in the Workbench. It also contains a trace function that can be switched on and off by student-provided modules during execution, in order to observe protocol operation. There is a set of user-defined output functions that can be supplemented to provide any desired output when *interlayer( )* is invoked. An optional interactive mode limits the output to chunks small enough to fit on a screen. At the end of simulation, summaries of the inter-layer statistics are automatically displayed to the screen. Text output to the screen also is recorded in file *diskout.txt*. A graphic run-time output display is now under development.

3.3 Software structure: class hierarchy

Beginning with Version 2 the Workbench became object-oriented. Figure 2 shows the class hierarchy, which consists of the following classes:

Simulation_control: this class contains control parameters that affect the entire simulation, for

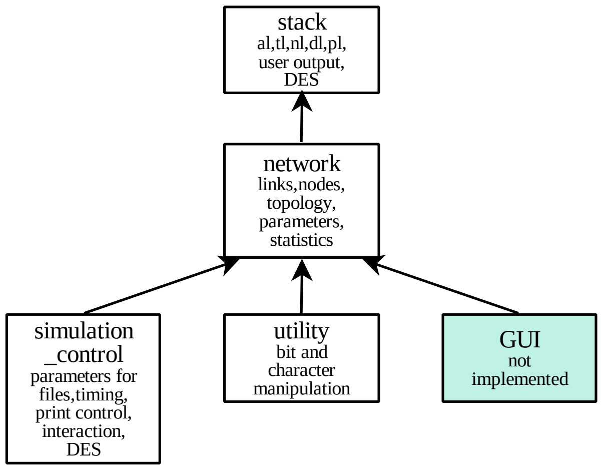

Fig. 2. Network Workbench Class Hierarchy

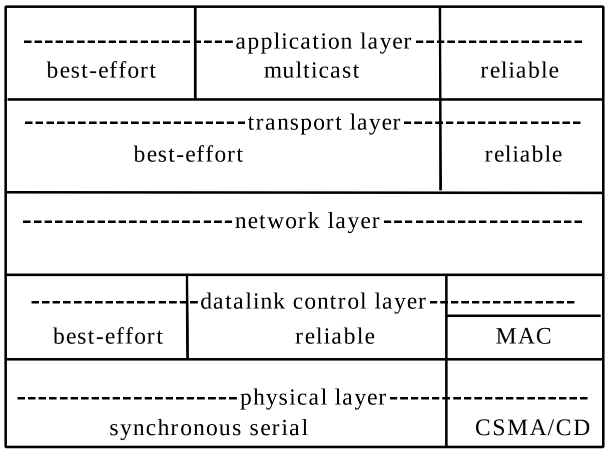

Fig. 3. Network Workbench Protocol Stack

example *LAN*, *WAN*, *MULTICAST*, *INTERNET*, and *time_step*.

Utility: data types and functions that support various low-level operations used by the Workbench, such as bit string manipulation.

GUI: reserved for the planned graphical user interface.

Network: data types and functions associated with the structure of the network, for example the *links[ ][ ]* matrix.

Stack: data types and functions for the main Workbench protocol stack, for example *al*, *tl*, *nl*, *dl* and *pl*, as well as *interlyr*, *userout*, and *des*. These form the "main works" of the Workbench.

3.4 Network structure: protocol stack

Figure 3 shows the Workbench protocol stack in more detail. The supported protocols are:

application layer (al): For best-effort and reliable transport, an "email" protocol which reads a "message" from a file and sends it to the *netnum*, *nodenum* indicated in the message; best-effort sends messages at a fixed interval until all are sent, while reliable waits until one message is sent and acknowledged to send the next. A different application represents the stream data that is typically sent over multicast with a long sequence consisting entirely of a filler character.

transport layer (tl): For best-effort and multicast applications there is a simple encapsulation protocol *utl* which is motivated by the Internet User Datagram Protocol (UDP). For the reliable application there is a reliable transport protocol *rtl* which is motivated by the Internet Transmission Control Protocol (TCP). The logic for sending segments reliably is broken out into a separate function, as are formatting and deformatting of transport segments. The Workbench transport layer is simpler than its Internet counterpart in two important ways: it does not support multiplexing of connections via ports, and the reliable transport does not piggyback acknowledgements on data segments of a reverse flow.

network layer (nl): This layer has the simplest code among the five layers in the Workbench stack, because the functions of a datagram protocol are limited and straightforward. The primary function is to select between a unicast packet forwarding function and a multicast forwarding function. Actual formatting and unformatting of packets is done in separate functions.

datalink control layer or DLC (dl): This layer also has three distinct functions, but they are not necessarily associated with any particular transport protocol. The reliable and best-effort DLCs are distinguished by the form of their Automatic Request for Retransmission (ARQ) logic, which is contained in collection of separate functions. The third DLC function is actually a sublayer, media access control (MAC), which implements an abstracted version of the Ethernet protocol. MAC shares common frame formatting processes with the DLC.

physical layer (pl): The physical layer comes in two forms: a synchronous, serial point-to-point link for the WAN, and a carrier-sense multiple access with collision detection (CSMA/CD) channel used with Ethernet as described in [10]. Each form processes a physical frame only once, imposing any and all bit errors when it is invoked, and scheduling the frame to be received after the sum of propagation delay and transmission time.

3.5 Project structure

The current Workbench available for download (version 3.2) offers eight project options. The specific nature of these assignments is illustrated in the Appendix, which shows the assignment, algorithm and code stub for Part 3. The project options are:

1. a study of WAN topology, requiring that students program a function which generates the matrices used by the Workbench to describe the WAN

2. a study of DLC frame formatting, requiring that students program functions for bit stuffing/unstuffing cyclic redundancy check calculations

3. a study of DLC flow and error control, requiring that students program the decision logic for ARQ

4. a study of CSMA/CD local area networks, requiring that students program a function for binary exponential backoff

5. a study of network layer routing, requiring that students program functions for route optimization and packet forwarding

6. a study of reliable transport, requiring that students program a function that implements the core logic of the sending end for an abstracted form of TCP

7. a study of multicast networking, requiring that students program a function which generates the set of WAN interfaces that participate in the multicast tree

8. a study of internetworking, where all previous project parts are combined with the result that reliable transport of email messages is interleaved with multicast stream transmission

In addition to these eight, there is a generic exercise intended to be expanded by the student as part of a different project that is either proposed by the student, or assigned by the instructor.

4. Experience with the Network Workbench

4.1 Introductory Courses

The Workbench was developed to increase learning efficiency for introductory courses in Computer Networking intended for graduate students and undergraduate Seniors in Computer science at GMU. It replaced previous assignments that had the same purpose: to teach about network operation by programming the protocols. The previous assignments were effective in teaching about one or two protocols, but at the cost of absorbing large amounts of the students' time to create the DES framework and protocol interfaces. From its inception, the Workbench was intended to minimize the time students spend in mastering such infrastructural issues while maximizing the amount of time available for investigation of protocol operation by programming. Version 1 of the Workbench was first used with students in Fall, 1994. From the beginning it was apparent that the approach adopted would be successful. Although students expressed frustration with the somewhat ragged quality of the software, they were quick to acknowledge that they were learning much more about the operation of network protocols than they could have learned in a purely lecture-based course.

Feedback from course-end critiques indicated the need for additional Workbench features such as the summary of statistics produced at the end of a simulation. Under effects of the twin stimuli of student enthusiasm and helpful feedback, the base of project assignments in the Workbench grew from two project parts in Fall of 1994 to six parts in Fall of 1997. The larger the Workbench grew, the more good suggestions were received from students. However, the Workbench had acquired an unhappy characteristic of software that grows without a planned structure: the internal relationships among its parts (which mostly had been contributed by students) were inconsistent and ill-defined. As a result it became increasingly difficult to add new project parts without finding unexpected side-effects and triggering latent bugs in the software. Also there were increasing student criticisms aimed at the fact that the Workbench software, while an interesting exercise in working with legacy code, was not an example of good quality programming.

At this point the author concluded that, in order for the Workbench to grow into its true potential as an educational tool, it would be necessary to take the sage Software Engineering advice of Brooks to "plan to throw one away" [1]. Therefore the Workbench was completely redesigned, retaining its overall architecture within the context of an object-oriented design. This in turn set the stage for the Workbench to grow, from WAN simulations involving only routers with point-to-point links, to a more complex internetting model. The latest version, Network Workbench 3.2, is currently used as courseware by four faculty members at GMU, in three different courses. The projects have been used successfully to supplement several texts used in our courses [2,11,12,13].

4.2 Advanced, project based use

Stimulus for an internetting Workbench came from expanded usage of Version 2 beyond the introductory networking projects for which it had been developed. In Fall 1997 and again in Spring 1998 graduate students (in two cases, doctoral students) who were familiar with the capabilities of the Workbench chose to use it as a network simulator for advanced, research-oriented projects. These were projects of somewhat limited scope: a model of logger discovery for reliable multicast, and a model of resource reservation for streaming multicast applications, both intended for use in Distributed Virtual Simulation [8]. The students involved were free to select among available simulation systems. They elected to use the Network Workbench in the expectation that its relative simplicity would prove beneficial when compared to other available tools such as OPNET. This was demonstrated to be true when the students using the Workbench were able to complete more complex projects within the context of a graduate one-semester advanced project course than students using other simulation systems.

4.3 Extensibility

The projects in Spring 1998, undertaken by doctoral students, used the object-oriented beta version of Workbench Version 2. These students' reports indicated that the object-oriented software in the rebuilt Workbench was highly extensible, and indicated several extensions that should be included. Several of these were in fact incorporated, in rewritten code, into Version 3 of the Workbench. They have expanded its power as a platform for advanced student use. Functionality added includes a best-effort stream application for multicast, ability to use reliable and best-effort transport protocols in the same simulation run, and user output functions. Given this demonstrated ease of extensibility it is reasonable to expect that the Workbench will continue to grow, within the framework of the structures described above.

5. Conclusions and Future Work

All experience to date indicates that the Network Workbench is an exceptionally useful platform for academic investigations of Internet concepts. It is useful both in protocol programming exercises for introductory courses, and as an easy-to-use network simulator for advanced student projects. The simplicity and modularity of the Workbench result in much shorter time to learn how to implement protocol models than would be the case with more powerful simulators such as OPNET and ns. The latest version of the Workbench is available to the academic networking community at http://netlab.gmu.edu/networkbench under no-cost license, with a request that any new Workbench-related code be provided back to the author for potential inclusion in future versions.

Development of the Workbench continues. At present the author plans to incorporate a routing protocol abstracted from Open Shortest-Path-First (OSPF) and a LAN protocol abstracted from Fiber Distributed Data Interface (FDDI). Version 4 of the Workbench will revise the DES structure to allow events to be "unscheduled," and use this function to improve the WAN simulation. It also will expand and refine the Workbench classes to hide from student-written modules any global details that would not be available to the actual protocol being simulated. Support for mobile networking also is contemplated.

However, based on past experience, it is likely that many of the best ideas for additions to the Workbench will come from students. By paying careful attention to student projects and reimplementing them in high quality, well documented software, the author expects to continue increasing the scope and utility of the Network Workbench for an expanded range of academic investigations while retaining its simple, easy to learn structure.

References

[1] F. Brooks, *The Mythical Man-Month*, Chapter 11, Addison-Wesley, 1995

[2] D. Bertsekas, and R. Gallager, Data Networks, 2nd Ed., Prentice-Hall, 1992

[3] P. Denning, "Business Designs for the New University," *Educom Review* Vol. 31 No 6, November/December 1996, 20-30

[4] F. Hillier and G. Lieberman, *Operations Research, 2nd Edition*, Holden-Day, 1974, 626-628

[5] I. Katzela, *Modeling and Simulating Communications Networks*, Prentice-Hall, 1999

[6] M. MacDougall, *Simulating Computer Systems: Techniques and Tools*, Chapter 1, MIT Press, 1987

[7] L. Peterson and B. Davie, *Computer Networks: A Systems Approach*, Morgan-Kaufmann, 1996

[8] J. Pullen, "Networking for Distributed Virtual Simulation," *Computer Networks and ISDN Systems*, Vol. 27, No. 3, December 1994, Elsevier, 387-394

[9] J. Pullen, L. Lavu, R. Malghan, G. Duan, J. Ma and H. Nah "A Simulation Model for IP Multicast with RSVP," RFC 2490, Internet Society, Reston, VA, 1999

[10] J. Pullen, "Discrete Event Simulation of CSMA/CD Local Area Networks in the Network Workbench," *Proceedings of the Computer Networks and Distributed Systems Modeling and Simulation Conference*, Society for Simulation, San Diego, CA, January 1999

[11] W. Shay, *Understanding Data Communications and Networks*, PWS Publishing Company, 1995

[12] W. Stallings, *Data and Computer Communications*, 5th Ed., Prentice-Hall, 1997

[13] A. Tanenbaum, *Computer Networks*, Prentice-Hall, 1996

ABOUT THE AUTHOR

J. Mark Pullen is an Associate Professor of Computer Science and a member of the C^3I Center at George Mason University, where he heads the Networking and Simulation Laboratory. He holds BSEE and MSEE degrees from West Virginia University and the Doctor of Science in Computer Science from the George Washington University. He is a licensed Professional Engineer and a Fellow of the IEEE. Prior to joining the GMU faculty he was an officer in the U.S. Army, in which capacity he served four years on the faculty of the U.S. Military Academy at West Point, New York and seven years at the Defense Advanced Research Projects Agency (DARPA). At DARPA he managed programs in high performance computing, networking, and simulation. Dr. Pullen teaches courses in computer networking, and has active research projects in networking for distributed virtual simulation and networked multimedia tools for distributed education. He recently received the IEEE's Harry Diamond Memorial Award for his work in networking for distributed simulation.

Appendix

Sample Project Protocol Assignment, Algorithm and Code Stub

Project Part 3: Datalink Control Automatic Repeat Request

Preparatory Homework for Part 3 from *part3hw.txt:*

1. Module dl.c contains the algorithms for a reliable DLC using go-back-n. Function dl_send operates in three distinct states: waiting, sending, sending_supv. Link state variable dl_send_state indicates which state it will enter when called. Create a state-transition diagram showing what conditions cause transitions among these states. HINT: look in wkb.h for definitions and init.C for initial state.

2. A very useful function will be found immediately following the initial comments in code/dllogic.cpp:

```
bit LTwindow(byte Nmin, byte Nmax)
// tests whether the range between two numbers is less
// than the window size DLC_WINDOW_FRAMES,
// given counter range DLC_WINDOW_MAX;
// useful in dlc_send
```

You are to write a companion function, which I believe you will find useful in project part3:

```
bit INwindow(byte Nmin, byte Nmax)
// tests whether range between two numbers is within
// the window size DLC_WINDOW_FRAMES, given
// DLC counter range DLC_WINDOW_MAX;
// useful in dlc_receive
```

Assignment for Part 3, DLC ARQ from *assign3.txt*:

You are to use the Workbench to create a discrete event simulation (DES) of a full duplex data link using a link control protocol similar to HDLC to pass one sequence of frames from A to B, and another sequence from B to A.

The Workbench software provides everything but your DLC, in source or object form. There is a DLC routine in the workbench code library, dl.cpp, that links with the other layers provided and works. The ARQ logic for this DLC must be incorporated into dllogic.cpp, for which a stub version is available in the wrkbch32/code directory. The logic for ARQ is included in the comments in dl.cpp; you will need to analyze the dl.cpp comments and code in order to complete dllogic.cpp.

The physical layer code provides an error generator so you can demonstrate your ARQ. The simulation application is email; two files email1.txt and email2.txt are passed between nodes 1 and 2. The messages were made up from "fortune" calls. You will see the application layer in the Workbench forming the email and sending it down to the data link layer.

For debugging you will probably want to start with one email file, zero bit error rate, and possibly interactive operation. You can cause this to happen by adding statements in WKB3.cpp such as:

```
number_of_nodes_sending_email = 1;
link_bit_error_rate = 0;
interactive = TRUE;
```

For your submission, you are to remove any such statements and run with the WKB3 defaults. Submit dllogic.cpp and diskout.txt as email attachments, and include any special conditions necessary for your code to work in the body of the email. Be sure the code files will run, as they may be run during grading.

Hints:

(1) If your code is working correctly it will get all the packets exchanged eventually, even in the presence of bit errors. However it has been my experience that debugging with two files or even just with errors is difficult, so the Workbench has been set up to let you tackle the problem by phases. I recommend doing it like this:
1. one file and no errors
2. one file and errors
3. two files no errors
4. two files and errors

(2) You will probably find functions LTwindow() and INwindow() are useful in completing the assignment. Many student errors with this part of the project arise from not making sliding window tests correctly.

(3) The ARQ algorithm provided is conservative in that if there is no new data to send and frames in the buffer have not been ACKed, it simply retransmits them during idle line time. This can result in better

performance in the presence of errors, in that a frame that suffered an error may be retransmitted before a NACK is received and thus arrive sooner than it otherwise would have. A consequence of this behavior is that when there are no frames to send, the DLC will be sending the oldest frame previously sent, or a supervisory frame.

Algorithm for Part 3, DLC ARQ from *dl.cpp*:

```
//
//  This is how the DL send works after the link is
//  initialized:
//
//  Each end can send up to DL_WINDOW_FRAMES
//  frames while waiting for an ACK.  The algorithm
//  for the sending side is below
//
//  (initial dl_send_state=dl_sending).
//
//  NOTE: the physical layer will automatically
//  schedule a call to dlc_send each time a frame
//  transmission is completed. It is important to have
//  dl_send_state correct so dlc_send picks up in the
//  right place when this happens.
//
//  case waiting:
//      (in this state the DLC has been sending but has
//      been blocked by the fact the window is full; it
//      wakes up here after a timeout and takes some
//      action):
//      if the range between SNmin and SNmax is not
//      smaller than the window,
//      the window is full so continue with the proper
//      actions for waiting:
//         send the frame in buffer position SNmin,
//         after transmission continue with case waiting.
//      otherwise (when the range between SNmin and
//         SNmax is smaller than the window)
//         an ACK has been received so continue with
//         case dl_sending
//
//   case dl_sending:
//      (in this state the DLC expects to dequeue and
//      send the next frame):
//      (the first step is to dequeue and buffer the frame):
//      if range between SNmin and SNmax is smaller
//      than the window,
//         attempt to take a frame from the input queue,
//           if there is a frame available
//           set SN=SNmax,
//           buffer the frame in position SNmax,
//           increment SNmax|mod DL_WINDOW_MAX.
//           else if frames remain unacked we need to
//           continue re-sending the  SNmin frame so
//             set SN=SNmin
//           otherwise nothing remains to be sent, so
//             set link_active to FALSE
//             set send-state to waiting,
//      and escape
//
//      this leaves the next frame to be sent in buffer
//      position SN; finish that frame by inserting RN,
//      CRC and stuffing
//
//      if the window becomes full with this frame
//         set dl_send_state to waiting so no more frames
//         are sent until ACKs are received.
//
//      finish by sending the frame that was just made to
//      the physical layer
//
//  NOTE: This algorithm is conservative in that, if it
//    has no new data to send and frames in the buffer
//    have not been ACKed, it simply retransmits them
//    during idle line time.  This can result in better
//    performance in the presence of errors, in that a
//    frame that suffered an error may be retransmitted
//    before a NACK is received and thus arrive sooner
//    than it otherwise would have (this is a  reasonable
//    practice for a point-to-point link, but not for a
//    shared link layer such as Ethernet where other
//    senders might use the capacity instead).  A
//    consequence of  this behavior is that, when there
//    are no frames to send, a node will be sending the
//    last frame sent, or a supervisory frame.
//
//   case sending_supv:
//      in this state we are sending a receiver_ready
//      supervisory frame because a send_supv packet
//      has been received indicating dl_receive found
//      a need for an ACK and queued  that packet
//
//*******************************************
//
//  This is how the DL receive works:
//
//  When a frame passing the CRC check is received,
//    if RN of that frame > SNmin but within the current
//    window,
//       set SNmin = RN of that frame.
//    if SN of that frame == RN sent to other end,
//       release the contents to the network layer,
//      and increment RN using modular arithmetic.
```

C++ code stubs for DLC ARQ logic function from *dllogic.cpp* (boxes highlight student requirements):

```
//   DLC ARQ logic for Networking Workbench.
//
//   This version defines stub functions providing logic
//   for an HDLC-like DLC that uses go-back-n
//   (n=DL_WINDOW_FRAMES).
//
//   These functions are all very short, one to four lines
//   of code. They provide the logic for various steps in
//   the DLC ARQ.

//  function to test whether range between two numbers
//  is smaller than window (DL_WINDOW_FRAMES),
//  given DLC counter range (DL_WINDOW_MAX)
//
//  Nmin is logically lower end of range, the lowest
//  number not ACKed
//  Nmax is logically higher end of range, the next
//  number to be used
//
bit stack::LTwindow(byte Nmin,byte Nmax)
{
   int testmax=Nmax;
   if(Nmin > Nmax)
       testmax = testmax+DL_WINDOW_MAX;
   return(testmax-Nmin < DL_WINDOW_FRAMES);
}

//  function to test whether range between two numbers
//  is within window (DLC_WINDOW_FRAMES),
//  given DLC counter range (DLC_WINDOW_MAX)
//
//  Nmin is logically lower end of range, the lowest
//  number not ACKed
//  Nmax is logically higher end of range, the next
//  number to be used
//
bit stack::INwindow(byte Nmin, byte Nmax)
{
// student must replace FALSE with appropriate code
return FALSE;
}

//  function to determine whether the window has
//  become full
//
bit stack::window_full( line_interface*  link_iface)
{
// student must replace TRUE with appropriate code
   return TRUE;
}

//  function to confirm that frames have been sent for
//  which ACKs have not been received
//
bit stack::frames_remain_unacked( line_interface*
   link_iface)
{
// student must replace TRUE with appropriate code
return TRUE;
}

//  function to increment SNmax for an interface within
//  the modular range of size DL_WINDOW_MAX
//
void stack::increment_SNmax( line_interface*
   link_iface)
{
// student must provide code
   return;
}

// function to update interface SNmin using received RN
//
void stack::update_SNmin( line_interface*
   link_iface,byte received_RN)
{
// student must provide code
   return;
}

//  function to update interface RN using received SN
//
void stack::update_RN(line_interface*  link_iface,byte
   received_SN)
{
// student must provide code
   return;
}

// function to decide whether to accept frame, based on
// its SN and the RN associated with receiving interface
//
bit stack::accept_frame(byte received_SN,byte
   interface_RN)
{
// student must replace FALSE with appropriate code
   return FALSE;
}

// function to determine whether to send an ACK frame
//
bit stack::send_ack( line_interface*  link_iface)
{
// student must replace FALSE with appropriate code
   return FALSE;
}
```