

Bugs as Features: Teaching Network Protocols Through Debugging

Brad Richards

Computer Science Department
Vassar College
Poughkeepsie, NY 12604
richards@cs.vassar.edu

Abstract

Being exposed to well-written code is a valuable experience for students — especially when the code is larger or more complex than they are currently capable of writing. In addition to the mechanics of a particular computation, students learn organization and documentation skills, and general concepts illustrated by the specific program. However, to obtain these benefits, students must thoroughly familiarize themselves with the code. This paper describes recent successes using software bugs as a means to force familiarization with network protocol code. The bugs become tools by which the students learn the inner workings of network protocols. As a side benefit, the approach provides a concrete basis for the discussion of debugging approaches and techniques. The technique is appropriate for any course involving programming, and is especially good for upper-level courses like networks, operating systems, and parallel and distributed programming, where difficult concepts can be illustrated via sample programs.

1 Introduction

It is common practice to expose computer science students to code written by others: Teachers present solutions to programming assignments, textbooks typically contain sidebar material showing small programs or program fragments, and there is growing interest in introducing students to larger and more complex bodies of code [1, 3, 5]. By inspecting good code, students learn software organization and documentation skills as well

as the actual mechanics of a particular computation. More importantly, they can learn the *concept* illustrated by the program. But the exercise only has value if students can be motivated to examine code in sufficient detail. It seems unreasonable to ask detailed questions about a large piece of code on a quiz or exam, and it can be difficult to do a sufficiently detailed “walk-through” in class.

This paper describes recent successes in coercing students to learn network protocol code via intentionally-introduced software bugs. Students worked with Data Link layer protocol code running in a simulator. The protocol code contained a subtle bug, causing it to deadlock consistently when the simulator parameters allowed data loss. To find and fix the bug, students had to become familiar both with the operation of the buggy protocol code and with the protocol’s intended behavior.

Currently, 36 students over two semesters have been subjected to this assignment, and the results have been extremely encouraging. Not a single student has failed to find the bug (though some take far longer to find it than others). In the process, they became *extremely* familiar with the protocol code — 300 lines of tricky C++, and gained a deep understanding of reliable data exchange via retransmission. The assignment has also produced valuable dialogs on debugging approaches and methodologies.

Section 2 gives an overview of our computer networks course and puts the protocol assignment into context. The protocol simulator and the bug itself are described in more detail in Sections 3 and 4. The assignment and our experiences are discussed in Sections 5 and 6, and conclusions and guidelines are given in Section 7.

2 Course Overview

The networks course at Vassar College is based on Tanenbaum’s “Computer Networks” text [7], supplemented with material from other textbooks [6], references [4], and standards specifications [2]. Three small assignments in the first half of the semester lay the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGCSE 2000 3/00 Austin, TX, USA
© 2000 ACM 1-58113-213-1/00/0003...\$5.00

```

Time:   Machine 0:                                     Machine 1:
====   =====                                     =====
1       <0: data (ack 7)> ----->
2       <1: data (ack 7)> ----->
3
4
5
6
7       <2: data (ack 7)> ----->
8
9       <0: data (ack 1)> <----- (corrupted)
9               <nak 0> ----->
10      <3: data (ack 7)> ----->
      :

```

Figure 1: Sample output from modified simulator

groundwork for a large programming project — building reliable datagram services atop UDP — that takes most of the second half of the semester.

The project requires students to have a working knowledge of signals, interrupts, sockets, and reliable network protocols — all of which are introduced through the small assignments preceding the project. The “bug” assignment, one of these initial exercises, is given during coverage of the Data Link layer (Chapter 3 in Tanenbaum), and both drives home the workings of a reliable Data Link layer protocol and prepares students to write their own reliable Transport layer protocol when the project arrives, as the basic notion of reliability via retransmission remains the same.

The course is an upper-level elective and has been taught twice in its current form, most recently in the Spring of 1999. Students taking the course have rated it highly, despite its well-deserved reputation for requiring an onerous amount of work.

3 Protocol Simulator

Tanenbaum supplies a simulator for the Data Link layer protocols in his textbook.¹ The simulator takes arguments specifying delay parameters, the percentage of frames to be lost, and the percentage damaged. Hosts are simulated by forking a pair of communicating processes, and the simulator uses one of six different protocols from the book to manage host-to-host communication. The simulator processes print information to the screen describing their actions as they run.

On our Sun workstations, output from the simulator processes was interleaved on the screen such that it was essentially unintelligible. We therefore modified the simulator so the processes write to log files as they run

¹See www.cs.vu.nl/~ast/books/cn3-simulator.zip.

instead of printing to the screen. Our postprocessor merges the logs and formats the output in a more intuitive fashion. The text in Figure 1 is a fragment of an actual output file from the modified simulator, and shows messages being exchanged between the two simulated hosts. One frame is being shown as lost during injection into the network. Another arrives corrupted at the other side. Displayed frames include sequence numbers, frame types, and acknowledgement information as appropriate. Work is currently underway on a Java-based implementation of the simulator that will display similar output as the simulation runs.

4 Bug Description

Errors were introduced into Tanenbaum’s protocol p6, the most sophisticated and realistic of the Data Link layer protocols presented. P6 is bidirectional, uses a sliding-window mechanism to allow out-of-order receipt of frames, and sends negative acknowledgements (NAKs) when damaged or missing frames are detected.

The bug was introduced serendipitously during an experimental effort to extend the NAKing mechanisms of p6. NAKs in p6 do not carry sequence numbers — the host sending a NAK makes no effort to describe *which* frame was lost or damaged. The NAK’s receiver assumes the earliest unacknowledged frame was involved and retransmits. We changed p6 such that NAKs carried the number of the frame being negatively acknowledged, but unwittingly introduced an error: The NAKs were also treated as positive acknowledgements (ACKs).

Figure 2 shows a fragment of protocol p6 as pseudocode. The protocol is event driven; only two of the possible events are shown. In the original p6, line 11 retransmits the oldest unacknowledged frame. In our “enhanced” version, it inspects the sequence number sent with the NAK to determine which frame to resend. Unfortunately,

```

1  case: frame_arrival
2      Retrieve frame from network
3      IF (frame contains data) THEN
4          IF (not next in sequence) THEN
5              Send NAK
6          IF (falls within window) THEN
7              Store frame in window
8              Try to pass up frames
9
10     IF (frame is a NAK) THEN
11         Retransmit appropriate frame
12
13     Examine ACK information
14     Mark ACK'd frame as received
15     break;
16
17 case: checksum_error
18     Send NAK
19     break;
    :

```

Figure 2: Pseudocode fragment of p6

to avoid adding additional fields to the frame, we chose to send this information in the same field used for acknowledgement sequence numbers. Thus, lines 13–14 treat NAK information as a positive acknowledgement as well. If the retransmitted frame is lost or damaged, the protocol deadlocks: The original sender is convinced the frame has been successfully transmitted and refuses to try again. The receiver is equally convinced that the frame never arrived, and refuses to make forward progress until it does.

The modified p6 works properly when simulated conditions allow error-free transmissions. Only when frames can be lost or damaged do NAKs get sent and potentially lead to deadlock.

5 The Assignment

Before giving students the defective protocol code, we cover Data Link layer protocols in lecture. Protocol p6 is discussed in detail, and the class does a walk-through of the code. The walk-through familiarizes students with the basic layout and construction of the protocol, but is typically not sufficient for a thorough understanding of how the protocol ensures reliable transmission.

The assignment requires that students, who must work individually, run the supplied version of p6 in the simulator with various transmission parameters, watching for deadlocks. It is recommended that, once a deadlock is found, they work their way through the simulator output, deducing and recording the values of key protocol variables (*e.g.*, send and receive window contents)

at each step. Students know initial values for the protocol variables, and determine updated values by consulting the protocol code at each simulated event. This is the key to the method's success: During the manual execution of code, students learn its details first hand. Eventually students reach a point where the predicted behavior of the ideal protocol no longer matches the simulated protocol's behavior. Some detective work is then required to discover the source of the bug.

Since our modified p6 is different from the code shown in the book, a sharp-eyed student could well discover the bug by comparing the two protocols instead of focusing on the simulator output and expected behavior. We mislead students somewhat to avoid this shortcut. They are told (truthfully) that p6 has been modified during conversion to C++, and that the on-line version of p6 should be considered authoritative. When we then say "there is a bug in p6", students assume it was flawed from the start, instead of looking for bugs introduced during the conversion.

Students were required to describe the bug in person before proceeding with a solution, ensuring they had found and understood the intended bug and not something spurious. This introduces grading difficulties, however, since all students eventually find and fix the proper bug. Our remedy was a second assignment phase: Students modify the protocol to collect timing information and report an estimated average round-trip time including only transmission delays. This provides additional work on which to base a grade, and forces students to think in detail about which round-trip scenarios to measure and how best to add the timing code. The details of the second phase can also be modified from semester to semester to keep the project fresh.

6 Experiences

The assignment has produced dread and fear in the typical student when handed out. Students know from previous debugging experience how long it can take to find bugs in their *own* code; now they must find flaws in a mysterious and complex program written by someone else. For most students, the initial fears fade once they begin debugging and become more comfortable with the protocol. They quickly discover which transmission parameters lead to deadlock, and learn how to compare the simulated behavior with the expected. After finding the bug, the students are almost uniformly appreciative of the assignment.

Students have found and fixed the error in as little as two or three hours. More typically, students spent several times that long on the assignment. One or two students a semester had great difficulty with the assignment and required significant guidance. Even these

students eventually found and fixed the error, though after fairly large investments of time.

Most importantly, students became extremely familiar with the workings of protocol p6 (and reliable protocols in general) during the course of the assignment. The bug descriptions delivered by students have been detailed and knowledgeable, and often include communications diagrams or other high-level summaries of how the protocol should operate. Even students that have yet to find the bug show a satisfying familiarity with the protocol's workings when asking questions or requesting guidance. The overall comprehension is far greater than after the code walk-through and discussion alone.

7 Conclusions

Our experiment with teaching network protocols via debugging has been a success. Students learned how reliable Data Link layer protocols work, and did so quickly and thoroughly. Making students debug intentionally flawed code has a number of advantages over more traditional programming assignments:

- **Familiarity:** Students must understand how a program works and what it is supposed to do before they can fix it. While this familiarity could be obtained by writing programs instead of debugging them, debugging allows access to larger programs than students could reasonably write.
- **Efficiency:** Bug-driven assignments are an effective way for students to learn complex concepts quickly, as they spend time focusing on code illustrating the concept rather than getting bogged down in the details of their own design.
- **Exposure:** If the flawed code is otherwise well written, students can learn valuable lessons about organization, style, and commenting.
- **Debugging experience:** Students learn about debugging approaches and methodologies in a controlled environment.

However, the approach is not without weaknesses:

- **Hard to start:** Some students have difficulty deciding how to start the assignment, since the code is completely foreign. Giving an overview of the code in lecture helps most students.
- **Ease of cheating:** Much effort goes into learning the code and finding the bug. Unfortunately, a description of the bug could then be passed to another student fairly quickly. This has not been a problem for us in practice, in part because students must give detailed bug descriptions in person.

Finally, we expect our experiences to generalize to domains other than network protocols. Our approach would be especially good for upper-level courses like operating systems, parallel programming, or artificial intelligence, where difficult concepts can be illustrated via sample programs. The following guidelines are offered for those interested in duplicating this technique in other areas:

- Ensure the *expected* behavior of the code is well defined. Network protocols are ideal in this regard, as one can easily determine what they *should* be doing at each step.
- Design a bug and context that forces students to execute the code manually — determining the expected behavior by stepping through code drives home its operation quickly.
- Allow bug-free execution paths through the code if possible. Students will be more comfortable if they can compile and run *something* before starting to look for bugs. Also, students must then discover which execution paths contain bugs — a valuable experience in its own right.

Acknowledgements

The author would like to thank Susan Hert for her valuable feedback on earlier drafts of this paper.

References

- [1] Astrachan, O., and Reed, D. AAA and CS 1: the applied apprenticeship approach to CS 1. In *Proceedings of the SIGCSE technical symposium on Computer Science Education* (1995), ACM Press, pp. 1–5.
- [2] IEEE. Carrier sense multiple access with collision detection. 802.3, IEEE, New York, 1985a.
- [3] Linn, M. C., and Clancy, M. J. The case for case studies of programming problems. *Communications of the ACM* 35, 3 (March 1992), 121–132.
- [4] Postel, J. Transmission control protocol. RFC 793, DARPA, September 1981.
- [5] Sharp, H., and Hall, P. A multi-media approach to providing software project experience for postgraduate students. In *Proceedings of the conference on Integrating Technology into Computer Science Education (ITiCSE)* (1996), ACM Press, pp. 109–115.
- [6] Stevens, W. R. *UNIX Network Programming*. Prentice Hall, 1990.
- [7] Tanenbaum, A. S. *Computer Networks*. Prentice Hall, 1996.