RTP: A TRANSPORT LAYER IMPLEMENTATION PROJECT

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

ABSTRACT

This paper describes a project for use in computer networks courses, implementation of the Reliable Transport Protocol (RTP), that gives students hands-on experience with network protocol concepts and construction.

Lecture topics such as the protocol layering model, sliding window protocols, packet formats and headers, techniques for establishing and closing connections, and UDP sockets programming are all driven home via first-hand experience. Students gain general programming and debugging experience on a realistic, event-driven, asynchronous application as well, and necessarily exercise their knowledge of algorithms and data structures.

1. INTRODUCTION

With the phenomenal growth of the Internet, it is becoming increasingly important to expose undergraduate computer science majors to computer networking techniques and technologies. While many small colleges offer an upper-level networking course, a glance at sample syllabi shows that most fall firmly on the academic side of the spectrum, consisting primarily of textbook exercises supplemented with some simple sockets programming. This paper describes a student project, the implementation of the Reliable Transport Protocol (RTP), that has been used over the course of three semesters to increase the hands-on component of the course in an effort to both engage students more fully and communicate the key concepts more effectively. The scope of the implementation project also aligns our computer networks course more closely with other upper-level project-based courses such as compilers and graphics.

RTP implements reliable, connection-oriented datagram service atop UDP[1]. Students tackling the project gain detailed knowledge of reliable sliding-window transmission protocols, packet layouts and details, the protocol stack layering model, techniques for establishing and closing connections, and UDP sockets programming. They learn the value of protocol standards

[1]Software is used to make UDP appear significantly less reliable.

JCSC 16, 4 (May 2001)

as well. Students are given a detailed description of the RTP protocol, and those who implement it correctly are rewarded with interoperability — their RTP layer communicates with other correct implementations, dramatically demonstrating the value of precise specifications. The project also pulls together data structures and programming techniques from across the curriculum, making it an excellent capstone experience.

Currently, 48 students over three semesters have attempted RTP implementations, and the results have been extremely encouraging. Most have implemented at least the bulk of the required functionality, and a surprising number have surpassed the requirements and added additional features. Student feedback has shown that the project was perceived as challenging but rewarding, and several students have landed industry jobs primarily on the basis of their RTP experiences.

Section 2 gives an overview of our computer networks course and puts the RTP project into context. The RTP protocol is described in more detail in Sections 3 and 4. Our experiences are discussed in Section 5, and conclusions and guidelines are given in Section 6.

2. COURSE OVERVIEW

The networks course at Vassar College uses Tanenbaum's "Computer Networks" text [5], supplemented with material from other textbooks [4], references [2], and standards specifications[1]. Three small assignments in the first half of the semester lay the groundwork for the RTP project, which takes most of the second half of the semester. The first of these refreshes students? knowledge of command-line arguments, and introduces UNIX signals and signal handlers. (Students will later use a `SIGALRM` handler to periodically service RTP connections.) The second assignment familiarizes students with the details of reliable communication at the Data-Link layer by requiring that they debug a sliding-window protocol running in a simulator[3]. This protocol will later be incorporated into their RTP project. Students are introduced to basic socket programming with a simple client-server program as part of their third assignment. The RTP project itself is broken into three pieces: A design phase, a five-week implementation phase, and a short final assignment in which students implement an application atop RTP.

The course is an upper-level elective and has been taught three times in its current form, most recently in the Fall of 2000. Before taking the networks course, students must successfully complete CS1, CS2, and the third course in our core sequence (a more involved design and implementation course). Students taking networks have rated it highly overall, despite its well-deserved reputation for requiring an onerous amount of work.

3. RELIABLE TRANSPORT PROTOCOL

While the RTP protocol offers reliable, connection-oriented datagram service, the details of this service have been kept as simple as possible to reduce the scope of the project.[2] Connections are established via a traditional three-way handshake, with initial sequence numbers set from the system clock. The send and receive window sizes remain fixed for the duration of a connection. Connections are closed after an exchange of disconnect request messages, or due to inactivity. The RTP specification states that data arriving after a disconnect request has been sent can be ignored, and that undelivered data still in the receive window when a disconnect request arrives need not be delivered. This is an unrealistic approach, but simplifies the implementation significantly. The interface consists of just the seven functions described below.

3.1 Interface Routines

Each of these routines returns a value of type `Error`, an enumerated type containing entries for all anticipated errors. Error conditions are all defined to have negative values, so that some functions can cast and return positive values (e.g. connection IDs) and have these values distinguished from actual errors.

```
Error RTP_Init(int tickInterval, int windSize);
```

This routine must be called before any other RTP routines, and gives implementations a chance to initialize their data structures and perform other start-up tasks. The user supplies the desired network servicing interval, and the size of both the send and receive windows. The function returns an error code, (which could be 0 to indicate that all went well), and should only be called once.

```
Error RTP_Listen(int localPort);
```

Listens for incoming connection requests on the specified local port, and accepts a connection request from *any* destination. The function blocks until a connection is established or an error occurs. (This behavior is quite different from that of TCP.) The function returns an error code or unique connection ID.

```
Error RTP_Open(int port, char *destName,  int destPort);
```

Opens a connection between the specified local port and the named destination machine and port using a three-way handshake. If `port` is 0, RTP picks any available local port. The function is blocking, and returns an error code or unique connection ID. The remote machine

[2]There are many opportunities for independent projects and extensions as a result.

must have called `RTP_Listen` prior to this call, or the open request should eventually fail.

```
Error RTP_Send(int conNum, char *buf,  int dataSize);
```

Sends the data in buf along the specified connection. Note that the data might not actually cross the network until after the next `SIGALRM` handler runs. The function is therefore *not* blocking – it may return before the data is actually moved across the network – and returns an error code, or the number of bytes sent or queued for sending.

```
Error RTP_Receive(int conNum, char *buf, int bufSize);
```

Accepts a datagram from the RTP layer. If no data is available, the call returns immediately with a return value of zero. Otherwise, it returns an error code or the number of bytes received.

```
Error RTP_Close(int conNum);
```

Closes a connection using the technique outlined above, and only returns once the task is accomplished. The function returns an error code (which could be 0 to indicate that all went well).

```
Error RTP_PrintStats(int conNum);
```

Prints the number of explicit ACKs sent and received, the number of NAKs sent and received, the number of data packets sent and received, and the number of damaged packets that have arrived.

3.2 Implementation Overview

An RTP implementation must service the network relatively often to ensure that acknowledgements are sent in a timely manner, incoming data is pulled off of the network, and packets are periodically sent to keep the connection alive. These "housekeeping" operations could be performed each time the application calls an RTP routine (e.g. `RTP_Send`), but there are no guarantees that an application would make RTP calls with sufficient frequency to avoid inactivity timeouts and other difficulties. Instead, students register a `SIGALRM` handler that runs at known intervals to perform these chores.[3]

The handler-based approach guarantees frequent polling of the network (and sharing of the CPU between an application and the RTP layer), but introduces complexities for the students. First, code in the RTP interface routines must take pains to avoid potential race

[3]Although a separate thread could be used to service the network instead, the regular nature of handler invocations can also be used as a measure of time.

conditions when updating global data structures and variables, as the handler could run at any time. More importantly, the handler and the interface routines must often work in concert to carry out a request from the application. For example, a call to `RTP_Open` could immediately send a connection request to the remote machine, then block until the remaining stages of the handshake are completed. But if the packet was lost or damaged, it would be the `SIGALRM` handler's responsibility to detect the timeout and resend an identical connection request. Many students give the handler responsibility for completing the entire handshake and only "awaken" the `RTP_Open` call once the process is complete (or has failed). Students must clearly identify the responsibilities of both the handler and the interface routines to ensure they work together properly.

4. THE ASSIGNMENT

4.1 Preparation

Currently, the bulk of the networks course is organized around the project. The early assignments are tailored to introduce material and techniques required to implement RTP, and the chapters in the textbook are covered out of order so as to present transport layer concepts as quickly as possible. The first programming assignment is presented to students during the first week of the semester with the goal of introducing UNIX signals and handlers. Students write an application that registers a `SIGALRM` handler, specifies a delay between timer expirations, and disables the timer after the handler has been invoked a specified number of times. Meanwhile, the application spins on a global variable to wait out the required number of handler invocations.

In the second assignment, students become painfully familiar with the details of reliable Data Link layer protocols. In his book, Tanenbaum presents a series of protocols of increasing complexity culminating with protocol `p6`, which incorporates sliding windows, piggybacked acknowledgements, and negative acknowledgements for missing or damaged frames. Using a simulator, students run a slightly-modified version of `p6` containing a bug causing deadlock in the case of lost or damaged frames. Students must trace the behavior of the simulated protocol until the source of the deadlock is discovered, and are forced to determine the *correct* behavior of the protocol at each step so as to detect anomalous behavior. (For more information, see [3].) Later, `p6` can be used with very few modifications as the basis of the RTP Transport Layer protocol.

Students are introduced to socket programming in the third assignment, where they write a simple client and server. The assignment requires that students map host names to IP addresses, open and bind sockets, and send and receive UDP datagrams. This code, as well, can be used directly in the RTP project.

4.2 Design Phase

A project as large and complex as the RTP protocol requires a careful and thoughtful exploration of design issues, and students are required to complete a design before proceeding

with their implementation. Students are given a specifications document detailing the behavior of the RTP protocol and its interface, and an assignment writeup outlining possible implementation difficulties and requesting information about specific areas of their proposed design. In particular, they are asked to list the responsibilities of the `SIGALRM` handler, in order, and to describe the interactions between the handler and the code implementing the `RTP_Open`, `RTP_Send`, and `RTP_Receive` interface routines. Students are encouraged to complete a finite state diagram describing all possible states in which their protocol could find itself, and to use these states as a way of coordinating the behavior of the handler and the interface routines.

4.3 Supplied Materials

In addition to the RTP specifications document, students are given a variety of utility routines that they may modify or use directly in their implementations. These include functions to disable and enable interrupts (allowing students to temporarily disable the `SIGALRM` handler while modifying shared structures if necessary), a non-blocking routine that detects data waiting at a socket, and functions for setting and verifying checksums. (Students were asked to write checksum routines themselves the first semester the project was used, but small errors in the checksumming code resulted in long and frustrating debugging delays for many students. As the checksumming algorithm wasn't central to the protocol issues, a decision was made to provide working checksum code to all students in later semesters.) A timers class is provided that implements a collection of data packet timers multiplexed atop the single `SIGALRM` handler, and `.h` files specifying the packet and header layout are distributed.

Students are also provided with a simple menu-driven test application that runs atop an RTP implementation, allowing connections to be opened and closed and data to be exchanged. This can either be compiled with their RTP implementations, or with the executables for a working RTP layer that are also distributed. Students can therefore test their projects against a correct implementation to speed development and debugging.

4.4 Grading

A battery of functionality tests is used to evaluate finished RTP implementations, and the results determine 70% of a project?s final score. Many of these tests are performed automatically by a test application that checks for enforcement of maximum window sizes, packet payload size, and that proper error responses are returned in cases such as sending or receiving on a closed or nonexistent connection. Other tests, most of which involve sending and receiving data, are performed manually. Implementations are tested against themselves and against the reference RTP implementation, and are put through their paces on both noisy and noise-free lines. Program design, organization, and style make up the remaining 30% of the grade.

5. EXPERIENCES

Our experiences with the RTP project have been overwhelmingly positive. Students have come away with a deep understanding of design and implementation issues surrounding reliable transmission protocols, as well as enhanced programming and debugging skills. Implementing an entire layer of the protocol stack drives home the strengths of the layering approach: Students were able to use the services of the layer below[4], add functionality, and export these new services to an application above – the final assignment, in which they write an application atop RTP, completes the cycle. Actually implementing the three-way handshake and a (simplified) closing mechanism similarly reinforced lecture material, particularly when questions arose about the finer points of the technique in general, and required RTP behavior in particular.

As an additional benefit, students gained programming maturity during the implementation experience. For many of the students, RTP was the largest software project they had yet written, and completing the project was a substantial accomplishment. (Student implementations typically require 2-3,000 lines of C++ code.) Before the project, none of the students had worked with sockets, or signals and handlers, and few had worked with state machine-based designs.

Some of this maturity was obtained through the school of hard knocks, however. The project can be extremely tricky for students to debug, due to the asynchronous nature of the code. Race conditions and other subtle errors in the collaboration between the `SIGALRM` handler and the interface routines often proved difficult to track down. To make matters worse, the project has to be reasonably complete before any of it can be effectively tested and debugged. In three semesters, no student has ever failed to submit *something* for evaluation, but the quality of the submitted materials has varied a great deal. Almost a quarter of the projects have received a score of less than 50%[5]. The majority have completed respectable implementations, however, and a handful of students each semester have gone beyond the requirements and implemented additional functionality.

Encouragingly, course evaluations have shown a high degree of student satisfaction. Almost all have pointed to the project as the most useful and enriching component of the course, and recommend the course and the project highly. A common student lament has been that they did not start early enough[6]. This has been addressed in later semesters with closer supervision, more effective scare tactics, and more aggressive nagging from faculty during the early stages of the project. Students are also now given a list of suggestions to reduce complications. (For example, focus on getting `RTP_Open` to work on noise-free lines, and test just that much

---

[4]UDP is technically a transport-layer protocol, but we consider it a network-layer service for the purposes of the project.

[5]The actual figure is 11 out of 48 (23%). The average score among these 11 was 36%.

[6]Most of the students responsible for the below-50% scores fall into this category.

against the reference implementation before proceeding.) Some students have also been frustrated during the design stage of the project, as the scope and ramifications of their design decisions are often not yet appreciated. The design assignment has been clarified as a result, and significantly more lecture time is now spent discussing the project before students begin their designs.

6. CONCLUSIONS

This paper describes a computer networking project, the implementation of the Reliable Transport Protocol (RTP), that gives students hands-on experience with network protocol concepts and construction. Lecture topics such as the protocol layering model, sliding window protocols, packet formats and headers, techniques for establishing and closing connections, and UDP sockets programming are all driven home via first-hand experience. Students gain general programming and debugging experience on a realistic, event-driven, asynchronous application as well, and draw upon their knowledge of algorithms and data structures during the implementation. The RTP protocol specification, assignment details, and reference implementation have all been improved through three semesters of use, and are available at `www.cs.vassar.edu/~richards/RTP`. The author would welcome feedback from faculty considering the project, and is particularly interested in arrangements that would allow students at different institutions to communicate via RTP.

7. ACKNOWLEDGEMENTS

The author would like to thank Susan Hert for her valuable feedback on earlier drafts of this paper, and Professor Larry Landweber at UW-Madison for generously sharing the UDP garbling package.

REFERENCES

[1] IEEE. Carrier sense multiple access with collision detection. 802.3, IEEE, New York, 1985a.

[2] Postel, J. Transmission control protocol. RFC 793, DARPA, September 1981.

[3] Richards, B. Bugs as features: Teaching network protocols through debugging. In *Proc. of the Thirty-First ACM SIGCSE Technical Symposium on Computer Science Education* (March 2000).

[4] Stevens, W. R. *UNIX* Network Programming. Prentice Hall, 1990.

[5] Tanenbaum, A. S. *Computer Networks*. Prentice Hall, 1996.