An Implementation of the Reliable Data Protocol for Active Networking

Ted Faber

USC/ISI
4676 Admiralty Way
Marina del Rey, CA 90292
faber@isi.edu
,

1. Introduction

The Active Congestion Control/Active Reservation Protocol (ACC/ARP) project has chosen to implement the Reliable Datagram Protocol (RDP)[1,2] as a tool for exploring active congestion control implementations. This decision was motivated by the desire to have easy access to the entire transport protocol at the endpoints as well as the active code running in routers. The Internet's most used transport, the Transmission Control Protocol (TCP)[3], is usually implemented inside the endpoint's operating system. Although that code is accessible in many operating systems, the programming environment is considerably more hostile than user space. Rather than implement TCP in user space in an active endpoint, the project chose the simpler RDP.

ACC/ARP is primarily implementing RDP to experiment with using active networking techniques for congestion control, specifically using ACC[4]. Secondary reasons include having a native Java implementation of a reliable transport that ACC/ARP and ARP can use for experiments. Because the RDP implementation is a research vehicle, the implementation has focused on the projects' specific research agenda rather than on providing a production protocol.

Although it is simpler that TCP, RDP is an interesting and useful transport protocol. It is packet-based, rather than byte-stream-based, which simplifies much of the data handling. Under RDP connection termination has different semantics than under TCP; data sent but unacknowledged on an RDP connection is not resent after the connection is closed.

RDP has some features TCP does not. It allows for unsequenced delivery of packets, which TCP cannot − there is no way to reorder out-of-order bytes in a stream. Because RDP's data guarantees are looser than TCP's, particularly once a connection is closed, it makes more status available to the user. Internally, RDP uses selective acknowledgments as well as cumulative acknowledgments.

Although RDP does not specify a congestion control system, it is amenable to a TCP-like congestion control system[5]. Because the ACC/ARP project will be experimenting with congestion controls, the freedom to try new ideas without violating specifications or recoding an existing implementation is attractive. In modern TCP implementations, the congestion control code pervades the implementation. A conceptually small change to the TCP congestion control algorithms may require changing code in many modules and many files of the operating system source. A new RDP implementation will avoids that pitfall.

ACC/ARP is making use of the ARP project's Active Signalling Protocol Execution Environment (ASP EE)[6] and virtual networking system (VNET). ASP provides a consistent programming model, access to Active Networks Node OS functionality, dynamic application extensions, application isolation, and a virtual network, VNET. Because ASP is a Java environment, the ACC/ARP RDP implementation is written in Java. This paper describes the implementation in terms of the Java classes and interfaces that make it up.

As an Active Networking EE, ASP supports running multiple concurrent Active Applications (AAs). AAs implement network services, e.g, the RSVP reservation protocol[7,8]. ASP starts such AAs in response to packet arrivals or at router configuration time, demultiplexes packets to them, and ensures that

they do not interfere with each other. AAs can also be dynamically extended under ASP, allowing new protocol features to be added on the fly. RDP is generally used as a component by AAs.

The following sections describe the RDP protocol, explain the ASP/Java implementation of it, and discuss some implementation choices in the ACC/ARP implementation that are not fully covered in the specification.

2. RDP Overview

RDP is a packet-based, reliable transport protocol that allows for sequenced or unsequenced delivery of packets. It currently has no congestion control specified. The IETF considers it an experimental protocol, and as such is described in 2 RFCs[1,2].

RDP service is packet-based. Data is sent and received in contiguous chunks, which are the same size to sender and receiver. Applications must be able to deal with data in packet-sized quanta rather than breaking it up arbitrarily, but in many cases this is acceptable. RDP is packet-based because it makes the protocol simpler.

RDP provides reliability by resending lost packets, which are detected by acknowledgments from the receiver. The retransmission system is a standard sliding window with a go-back-n retransmission scheme. Both cumulative acknowledgments and selective acknowledgments are used. Selective acknowledgments are called extended ACKs in the RDP specification. Packets that are not acknowledged in a timeout period or that the protocol can deduce never arrived due to gaps in the selective acknowledgments are resent. Although RDP does not specify how to set the timeouts, the ACC/ARP implementation estimates the round trip time of the connection and uses that estimate to set the retransmission timer.

Although RDP delivery is reliable, the endpoints can specify whether the arriving packets should be delivered to the application in order or not. In applications for which packets order is not important, such as displaying an image, this can provide a speedup or at least a perceived speedup.

RDP does not currently provide congestion control, although we are in the process of adding not only congestion control, but congestion control that takes advantage of dynamically loaded code in the intermediate routers. This work is in progress.

RDP differs from TCP because RDP provides packet-based service and has simpler semantics on connection close. Packet-based service means that both sender and receiver must deal with data in arbitrary-sized pieces, unlike a byte-stream protocol. In practice many applications can use the packet model directly, and those that cannot can use an intermediate library. One such intermediate library is the VNET interface to RDP described in Section 3.3.

When an RDP connection is closed, the protocol makes no further efforts to deliver packets that have been passed to the protocol implementation, but have not yet been acknowledged. Unlike TCP, RDP does not guarantee that all packets passed to one endpoint's implementation are delivered to the other endpoint's implementation. This is another case where the RDP protocol chose simplicity of protocol implementation over a feature that not all applications need.

Practically, this means that if an endpoint requires that all its packets are received by the other endpoint, that endpoint must close its connection only when receipt of all packets has been confirmed. RDP specifies that this information must be made available to applications.

Although RDP close semantics differ from TCP in the above manner, RDP, like TCP, implements a `CLOSE_WAIT` state that prevents 2 connections from being established between the same host port pair too quickly. The purpose is to avoid starting a second connection which could accept data still in flight from the first connection[9].

3. RDP Under ASP

There are 2 interfaces to the RDP protocol under ASP. There is a stand-alone implementation, in which AAs use the RDP implementation classes directly, and a VNET interface that uses RDP to emulate Java's reliable byte-stream sockets. We will describe the stand-alone interface first, and then the glue used to connect the stand-alone implementation to VNET.

3.1. Stand-alone Implementation

The RDP implementation is visible to AAs as a collection of classes that embody packets, RDP connections, and connection status. These classes are `Packet`, `RDPConnection`, and `RDPStatus`. All classes described in this section are fully documented in the javadoc documents that come with the source.

The `Packet` class provides an interface to the data chunks that RDP sends. It provides methods to associate a `Packet` with a byte array, to add or trim bytes from the `Packet` efficiently, and to insert or remove various sized integers from the `Packet`. Integers are stored in `Packet`s in network byte order.

The `RDPConnection` connection is the AA's main interface to RDP. It provides methods to listen for connections from other hosts, to connect to other hosts, to send and receive packets, to examine a connection's status, and to close a connection. These methods are summarized below.

Method	Description
`open`	A static method to create connections. This call sets connection parameters and binds to addresses. Successful passive open calls return an `RDPConnection` that is listening for incoming connections. Active opens return an `RDPConnection` that has connected to another endpoint. Constructors are not directly used because open may block or return an error.
`accept`	When called on a listening `RDPConnection`, this returns a connected `RDPConnection`. If no other endpoint has connected, this call blocks until a connection is formed.
`recv`	Returns the next queued `Packet`, if any.
`send`	Sends the given `Packet` to the other endpoint.
`status`	Return an `RDPStatus` object that summarizes the connection state.
`close`	Terminate the connection immediately. As mentioned in Section 2, this has harsher semantics than a TCP close.

The `RDPStatus` object has no methods, and its members are:

Member	Description
`state`	The connection state, which can be used to tell if the other side has closed a connection, or if there has been a protocol error.
`unacked`	Number of packets sent but unacknowledged.
`queued`	Number of packets received but not delivered to the AA.
`recvmax`	Maximum size of an incoming packet.
`sendmax`	Maximum size of an outgoing packet.

An `RDPStatus` object is used to determine when it is safe to close a connection or to determine that the other side has closed it. It also allows the application to size its packets to meet the constraints imposed by the other endpoint during connection establishment.

Internally there are several other classes used by RDP. These classes are used to more easily manipulate packets, to provide a general timeout facility, and to send keepalive packets.

The `RDPPacket` class extends `Packet` to make constructing and parsing RDP header information easier. There are actually several further subclasses to make construction of various types of packets easier. For example there is a class to create an otherwise empty packet that acknowledges a given sequence number.

RDP needs to schedule three types of events: packet retransmissions, `CLOSE_WAIT` transitions, and keepalive packets. These events are scheduled by the `RDPTimer` class. `RDPTimer`'s `schedule` method arranges for the `timeout` method to be called on one of its arguments after a specified time. `schedule` can schedule one call or arrange for `timeout` to be called periodically. The object passed to `RDPTimer`

must implement the `Timeable` interface, which defines the `timeout` method that `RDPTimer` will call. `RDPTimer` also implements an `unschedule` that cancels any pending calls.

The following objects implement `timeout` in this implementation:

Class	Timeout Function
`RDPPacket`	Resend the packet
`RDPConnection`	Change state from `CLOSE_WAIT` to `CLOSED`
`RDPKeepalive`	Send a keepalive packet

Timeouts for each `RDPPacket` are scheduled when the packet is sent reliably and unscheduled when an acknowledgment arrives. Resend timeouts repeat themselves until the packet is acknowledged. Timeouts for the `RDPConnection` are scheduled when the other end of the connection closes, and unscheduled when the `RDPConnection` successfully changes state. An `RDPConnection` might fail to change state if there is unread but received data on the connection.

The `RDPKeepalive` object exists only to periodically resend the last acknowledgment unreliably to ensure that the connection remains open when both sides are idle. By resending a packet periodically, it avoids the possibility that a connection-closing packet is lost; receiving the acknowledgment triggers a new close packet. Each `RDPConnection` object allocates an `RDPKeepalive` object and schedules repeating events for it when the `RDPConnection` is created.

3.2. Stand-alone Example

The following example illustrates the implementation. These applications show the calling sequence for the RDP implementation. They have been pruned of some essential details of running in the ASP environment and error handling, and are not robust. They are intended to show the basics of the implementation. To write running code, consult the documentation available with the RDP source.

The example shows a sender and receiver that exchange `npkts` packets (`npkts` has been agreed on out-of-band) and then the receiver closes the connection. The receiver must run first and be waiting for the sender's active open. The receiver also illustrates the feature that a thread can `wait` on the `RDPConnection` object itself for a notification of packet arrival or state change.

3.3. VNET Interface

VNET provides the Java sockets interface between AAs and the ACC/ARP RDP implementation. It does the buffering and protocol conversion work to make a sequence of RDP packets look like a byte stream.

The VNET interface exports `ServerSocketV`, `ClientSocketV`, and `SocketV` objects that mimic the Java socket interface. A `ServerSocketV` passively opens an `RDPConnection` object and provides the familiar interface to it. A client connects to it by creating a `ClientSocketV` which does the active open. Calling `accept` on the `ServerSocketV` returns a `SocketV` which exports a byte stream interface. The `SocketV` provides the same `send/recv` interfaces as TCP sockets in the standard Java environment.

4. Implementation Notes

This implementation applies a loose interpretation to some of the details of the RDP specification[1,2] because of our operating environment. This implementation does not strictly enforce per-connection queueing limits, aggressively seeks half-open connections, and addresses lost SYN acknowledgments explicitly. Some of these interpretations deal with details of the specification,

Because the same memory resources are used by the AA and the RDP implementation, RCV.MAX, the RDP per-connection queue size, is initialized to the window size, and not strictly respected. A connection can queue as many packets as ASP allows RDP to allocate. In a system where kernel buffer space is allocated differently from process virtual memory, this limitation is more important.

```
class Sender {
  public void sender() {
    RDPConnection c;
    RDPStatus s;
    Packet p;
    int i,j;

    c = RDPConnection.open(
          RDPConnection.ACTIVE_OPEN,
          RDPConnection.ANY_PORT,
          (short)1500, to,
          RDPConnection.SEQUENCED);
    for ( i = 0; i < npkts; i++ ) {
     p = new Packet(20);
     p.putInt(i,0);
     c.send(p);
    }
    s = c.status();
    while (s.state!=RDPConnection.CLOSED) {
     synchronized(c) {
         c.wait();
     }
     s = c.status();
    }
  }
}


class Receiver
  public void receiver() {
     RDPConnection c, d;
     RDPStatus s;
     Packet p;
     int i,j;
     PrintWriter out;

     c = RDPConnection.open(
           RDPConnection.PASSIVE_OPEN,
           (short)1500,
           RDPConnection.ANY_PORT,
           null,
           RDPConnection.SEQUENCED );
     d = c.accept();
     for ( i = 0; i < npkts; i++ ) {
         s = d.status();
      while (s.queued == 0 ) {
        synchronized(d) {
            d.wait();
        }
        s = d.status();
       }
       p = d.recv();
       System.out.print(p.getInt(0)+"\n");
     }
     d.close();
     c.close();
   }
}
```

The RDP specification says that connections should avoid probing for half-open connections, but the ACC/ARP implementation is relatively aggressive. The most common cause of a half-open connection is when a one endpoint closes a connection and the RST packet that indicates to the other endpoint that the connection is closed is lost. The specification advises that application activity will generally detect half-

open connections, and that the protocol implementation should only detect and remove them *in extremis*, to avoid overloading the network. The ACC/ARP implementation is more aggressive because modern networks tend to have sufficient bandwidth that an occasional extra packet is not excessive overhead. Furthermore, keep alive packets are easy to implement while special-case code to identify resource starvation due to excessive numbers of half-open connections is more expensive.

RDP connection establishment can fail if the packet acknowledging the SYN packet that requests a new connection is lost. Because acknowledgments are sent unreliably, the lost packet is never resent, and because the connection hasn't been established, the SYN packet is not resent either. The specification is silent on how to address this issue.

The ACC/ARP implementation sends the SYN packet reliably. `SYN_RCVD` state actions have been altered to simply resend the original acknowledgment if the arriving SYN packet has the same parameters as the original. The implementation should set an upper bound on the number of times to resent the SYN packet, although the current code does not.

5. Conclusions

Our experience with RDP to date has been mostly positive. It has met the projects' needs for a simple transport protocol, and promises to be a good platform for congestion control experiments. RDP has been successfully used to load classes for the ASP protocol versioning system[6].

Conversely the ease of implementing RDP and integrating it with the VNET system has demonstrated that ASP is a reasonable development environment. ASP made a straightforward implementation of RDP easy because it basically stayed out of the implementor's way except where ASP services were required. The integration with VNET was a done seamlessly by a different implementor.

RDP shows promise as a research vehicle for active networking.

References

1. David Velten, Robert Hinden, and Jack Sax, "Reliable Data Protocol," *RFC-908* (July 1984).
2. Craig Partridge and Robert Hinden, "Version 2 of the Reliable Data Protocol (RDP)," *RFC-1151* (April 1990).
3. Jon Postel, ed., "Transmission Control Protocol," *RFC-793/STD-7* (September, 1981).
4. Theodore Faber, "ACC: Active Congestion Control," *IEEE Network*, vol. 12, no. 3 (July/August 1998).
5. Van Jacobson, "Congestion Avoidance and Control," *Proc. SIGCOMM Symposium on Communications Architectures and Protocols*, pp. 314-329, ACM SIGCOMM, Stamford, CA (Aug 16-19 1988).
6. Bob Braden, Alberto Cerpa, Ted Faber, Bob Lindell, Graham Phillips, and Jeff Kann, "ASP EE: An Active Execution Environment for Network Control Protocols," *Asp Documentation* (1999), available electronically from `http://www.isi.edu/active-signal/ARP/DOCUMENTS/ASP_EE.ps`.
7. Lixia Zhang, Stephen Deering, Deborah Estrin, Scott Shenker, and Daniel Zappala, "RSVP: A New Resource ReSerVation Protocol," *IEEE Network Magazine*, vol. 9, no. 4, pp. 8-18, IEEE (September 1993).
8. R. Braden, ed., L. Zhang, S. Berson, S. Herzog, and S. Jamin, "Resource ReSerVation Protocol (RSVP) − Version 1 Functional Specification," *RFC-2205* (September 1997).
9. Theodore Faber, Joe Touch, and Wei Yue, "The TIME-WAIT State in TCP and Its Effect on Busy Servers," *Proceedings of IEEE INFOCOM*, pp. 1573-1584, IEEE, New York, New York (March 21-25 1999).