CSC/ECE 573 Internet Protocols Project

Fall 2008

SRTP: Simple Reliable Transport Protocol

1. SRTP Overview

You are to implement a Simple Reliable Transport Protocol (**SRTP**) on top of **UDP** on any Unix or Windows workstation. **UDP** is technically a transport protocol in the Unix kernel, but we will consider the **UDP/IP** protocols to be the "network layer" (i.e., the "transmission medium") for this project. This will allow you to implement **SRTP** outside of the Unix kernel rather than inside the kernel. **UDP** is better suited for this strategy, than, for example, **TCP**, because **UDP** implements a datagram service at the transport level. **SRTP** is reasonably complex in the sense that it can support both ends of a full-duplex connection. The protocol should provide reliable transmission of data over the unreliable network layer (**UDP**). To allow you the flexibility in the design of your protocol, you will not be given a "standard" for this protocol. The disadvantage of this approach is that your implementation of **SRTP** will not be able to talk to another team's implementation of **SRTP**.

The following is a non-exhaustive list of general issues that you will need to address in your project.

- Naming
- Data structures
- Buffer management strategy
- Timer management strategy
- Reliable transport
  - lost, corrupted, delayed, or duplicated messages
    - retransmission strategy
    - sequence numbers
    - checksum
  - flow control
    - sliding window
    - slow start and congestion avoidance
  - connection management
    - three-way handshake for establishing a connection
    - maintain a connection (what if one end goes away?)
    - close a connection
    - multiple connections
  - packet
    - size and format
    - headers
- Run-time statistics gathering
- Network management

2. SRTP Interface

You are to provide at least the following six interfaces for the user of **SRTP**. All interfaces except **SRTP_Listen()** and **SRTP_Receive()** should be non-blocking.

For connection establishment:

```
connection_number=SRTP_Open(udp_host_addr,port_number);
int connection_number, port_number;
struct sockaddr *udp_host_add;
connection_number=SRTP_Listen(port_number);
int connection_number, port_number;
```

For stream-oriented data transmission:

```
status=SRTP_Send(connection_number,buffer_ptr,buffer_size);
int connection_number, buffer_size, status;
char *buffer_ptr;
```

Status returns the number of bytes accepted by **SRTP** or -1 for Error.

```
status=SRTP_Receive(connection_number,buffer_ptr,buffer_size);
int connection_number, buffer_size, status;
char *buffer_ptr;
```

Status returns the number of bytes returned by **SRTP** or -1 for Error.

For graceful disconnection (by active site only):

```
status=SRTP_Close(connection_number);
int connection_number, status;
```

Status returns a -1 for Error and a 0 otherwise.

For checking the status of a connection:

```
status=SRTP_Status(connection_number);
int connection_number, status;
```

The following return codes are to be implemented:

1 - Attempting to Open
2 - Listening
3 - Active Open
4 - Passive Open
5 - Close requested - active
6 - Close requested - passive
-1 - Connection does not exist

3. SRTP Design Requirements

Stream-Oriented Data Transmission and Congestion Control. The **SRTP** protocol provides a stream-oriented data transmission service. That is, data is delivered in the order it was sent, but message boundaries from the user are not preserved. Reliable transmission will require the use of checksums and sequence numbers. The protocol should use a fixed number of buffer space to be shared among all connections. Efficiency of the protocol is an important consideration. Features such as piggybacking of acknowledgements must be incorporated. Flow control must be implemented, and it is suggested that you implement a sliding window protocol. The timeout interval should be constant, and its value will be varied to collect performance statistics. You also need to implement TCP's slow start and congestion avoidance/control mechanisms as discussed in class: (1) Slow-start, (2) window increase/decrease, and (3) fast retransmit and fast recovery.

Connection Establishment. You must implement TCP's three-way handshake for establishing a connection. Your implementation must be robust, in the sense that the connection must be established correctly even in the presence of packet loss or packet duplicates. During connection setup you do not need to negotiate the use of checksums: always use checksums. However, you need to negotiate the maximum segment to be used by **SRTP** for data transfer. Doing so will allow you to measure the performance of **SRTP** as you vary the maximum segment size (see Section 4).

Graceful Disconnection. The close request should only be allowed at the active end of a connection. Otherwise, you should return an error. The active end of **SRTP** should send a **Close-Connection-Request** message after all of the data in the transport buffers has been sent. When a **Close-Connection-Request** message is received at the passive end, **SRTP** waits until all of the data with lower sequence numbers have been delivered to the user, and then sends a **Close-Accept** message back to the active site. As in connection establishment, disconnection must work despite the occurrence of network errors (lost packets, duplicate packets, etc.).

Buffer Management. To be able to retransmit, **SRTP** must maintain memory buffers, say, 50-200 KB (you can make this a parameter and test the performance of your implementation as you vary the amount of buffer space). Data passed to **SRTP** by the application through the **SRTP_Send()** call are copied to **SRTP**'s buffer space. The data are kept in **SRTP**'s buffers until the other end of the connection acknowledges receipt of the data. Buffer management can be tricky and complex. I suggest that you implement a simple circular buffer scheme.

Similarly, data received by **SRTP** from the other end are copied to the application's buffer space as indicated by the **SRTP_Receive()** call. To keep things simple, you may want to adopt a fixed buffer allocation scheme in which the buffer space of **SRTP** is partitioned statically into two sections, a "transmitting" section that buffers data from the application for transmission, and a "receiving" section that buffers incoming data to be passed to the application.

Addressing. A Transport layer address consists of a host address and a port number. The host address corresponds to the network layer address (to be used by **UDP**). The port number is used to identify the clients which are using **SRTP**. A particular **SRTP** server (passive end of a connection) should listen on a port number of your choosing in the range [1024,65535].

Network Layer. You will be using the **UDP** transport protocol (which in turn uses the **IP** network protocol) as the "network layer" for **SRTP**. Data can be sent and received over **UDP** by making appropriate system calls. These calls require the host addresses to be bound to "sockets."

The system calls for sending and receiving data are as follows:

```
status=sendto (socket_descriptor,packet_ptr,packet_size,flags,to,tolen);
status=recvfrom (socket_descriptor,packet_ptr,max_packet_size,flags,from,fromlen);
```

For detailed information about these calls consult the Unix **man** pages on **recv(2)** and **send(2)** (note that the number in parantheses is the section of the **man** pages where these calls are described), or Appendix 1 of Vol. III of the Comer text.

Handling Multiple Connections. You will need to multiplex multiple connections over a single **UDP** socket so that a single client can open several connections or several clients can be active at the same time. You need not attempt any complicated buffer management strategies in the transport layer. Due to time constraints you might not be able to implement any complicated scheme and debug it. So implement some simple (maybe even static) allocation scheme.

The main aim is to implement the transport layer with multiple connections "correctly." Performance is something that is left to the last week (if you still have time). In other words, buffer allocation would not be at the top of your priority list. Ideally, you should have designed a clean interface to the buffer allocation routines. So changing the policy for buffer allocation should not affect the rest of your design. It is better to keep it that way because you might want to try different buffer allocation strategies if you have the time.

Don't bother about handling an asymptotically infinite number of connections. There is a limit to which the transport layer can handle connections, and it is acceptable not to accept new connections when you do not have buffer space, etc. Also, it is acceptable to break connections if you run into deadlock situations. Design your implementation to work for, say, a maximum of 10 connections. It should be possible that by changing some constants it will work for more connections. But in the workload you will be testing with, you will probably have only 2 or 3 file transfers going on at a time or 3-4 talks. Sometimes it is acceptable to drop packets that have arrived without errors due to lack of buffer space.

Network Management. It is very useful, both for debugging and when demonstrating your project, to be able to query the protocol and to display, at both the transmitter and the receiver, run-time statistics at the time of the query (rather than at the end of a connection. In order to provide this, a separate window would be useful, or perhaps a special signal handler which accepts queries about the state of the system (e.g., number of packets transmitted or retransmitted, average packet delay, number of connections, checksum errors, etc.)

One way of doing this would be to implement a network management protocol, which sends status information from the transmitter and receiver to a manager module both periodically and in response to queries. Statistics would then be computed by the manager module and displayed. In addition, the management protocol might allow **SRTP** clients to set run-time parameters for the **SRTP** protocol (e.g., timeout values, maximum datagram size, etc.). Extra credit will be given for projects, which include a quality management protocol.

4. SRTP Implementation and Testing

You must implement (1) a talk facility and (2) a file transfer facility as clients for the **SRTP**. The **SRTP** clients must use the **SRTP** interface described in Section 2 to open or close a connection and to transfer data with **SRTP**.

You must include code in the **SRTP** protocol to measure the performance of your implementation. Performance measures of interest include **average throughput** for file transfer and **average response time** for talk. Both sending and receiving ends of a connection must also maintain information about the state of the connection. Relevant state information includes the transmission or receiving times, respectively, of all packets, number of packets transmitted, number of packets received in error (checksum errors), number of duplicate packets received, number of packets retransmitted, etc. At the end of a connection, this information will allow you to obtain the average delay or average throughput, as well as to create plots of sequence numbers against time. They will also make it easy to compare different (constant) values for the timeout for different network paths (e.g., two hosts on the same subnetwork vs. two hosts over a routed network).

5. Important Notes:

1. **You will not have access to the class lab for your projects.** So, you have to make sure that you have the system requirements that are needed for your project implementation. A project presentation is due during the last week of classes. You should make sure that you can make your final presentation on-campus even if the actual implementation is done off-campus.

2. Projects are done in teams of **at most two** students. Selecting partners, as well as dividing up the work among team members, is your responsibility. You can use the course message board to find team members. The grade will be assigned to the project as a whole. However, all team members will be asked to specify their contributions.

6. Milestones and Deliverables:

It is extremely important to adhere to the due dates for the project milestones. Due dates are posted on the course web site. Only **one** submission per team is required, and can be done by any team member. All submissions are to be done through Wolfware.

Milestone 1. Identification of team members
Milestone 2. Preliminary Design: A 4-page single-spaced document (Word format or PDF format) describing in detail the design of the your programs. That is, make sure to describe all programming interfaces, message types, packet formats, mechanisms for establishing and releasing connections, flow control and error recovery strategies, buffer management, and timer management, etc. You should include a brief description of the important data structures and of the organization of the code you plan to implement. You should also state how the implementation will be divided between team members.
Milestone 3. Final Project Submission: The final submission should include the following:

1. An 8-page single-spaced document (Word format or PDF format) fully documenting all aspects of your project. This document should include performance measurement figures of every aspect of your project testing thbe impact of important parameters like the retransmission timers, etc. Highlight changes to your initial design and the problems you encountered during the implementation.
2. Source code, Makefile and a Readme file that contains the following information: Documentation of your source code tree, build and invocation instructions. Also, include in the Readme file the state of your programs, whether there are any bugs and include platform specific information.

Project Demonstration: You will be expected to give a demo of your project during the last week of classes to the instructor or to the TA. Advance sign up for the demonstration is required.

7. Grading

30%: **Documentation and Deliverables:** Were all milestones completed as expected? Is the final report complete and clear?

70%: **Quality:** Is the final implementation addressing every required feature? Are there any bugs in your implementation?