

Project 2

Edit 2022-10-07: Added example for running the traffic generation tool.

Overview

Goal: Construct a real-time transport tool that provides "almost reliable" message delivery under a time constraint and evaluate its performance in the presence of packet loss.

You should work on this assignment in groups of exactly 2 students. Contact us if you cannot find a partner and we will arrange one for you.

Your programs should be written using C or C++.

The **initial design** submission date is **Monday, October 3, 2022, at the beginning of class.**

The **final submission** date is **Friday, October 21, 2022, 11:59pm.**

Transport Tool Details

Using the unreliable UDP/IP protocol, you should write a tool that transmits a data stream from one computer to another, such that messages are delivered within a specific latency constraint. Within that latency constraint, delivery should be as reliable as possible.

The tool consists of two programs:

1. `rt_srv` - the sender process.
2. `rt_rcv` - the receiver process.

In order to test your programs, we will provide a traffic generation tool that will send data locally over UDP to your sender (`rt_srv`) process, and a receiver "application" that your receiver (`rt_rcv`) process should deliver data to, again using UDP.

Note that in practice you would typically write a transport protocol like this as a library with an API that other programs could link with, rather than a standalone program that communicates with the application over UDP. However, this design choice will simplify your implementation, and will allow you to experiment with real applications in addition to our toy traffic generation application if you choose (see video demo bonus opportunity below).

Your programs should work with any application that sends UDP traffic (not only our traffic generation tool). Therefore, you should NOT rely on the header structure of the generated traffic. You should treat received traffic as an opaque payload and encapsulate it in your own header that includes all the information your program needs.

Sender Process

A sender process (`rt_srv`) should be run on the source machine with the following interface:

```
./rt_srv <loss_rate_percent> <app_port> <client_port>
```

`loss_rate_percent` is the loss percentage to emulate for sent packets

`app_port` is the port on which the `rt_srv` process should receive incoming UDP messages from the application

`client_port` is the port that the `rt_srv` process will use for communicating with clients (`rt_rcv` processes). The `rt_srv` process will use this port to receive requests from clients (`rt_rcv` processes) that would like to receive the data stream and to receive feedback messages (ACKs/NACKs) once a client has requested a transfer.

You can assume the `rt_srv` process runs forever. It only needs to handle one connected `rt_rcv` process. If a new `rt_rcv` process tries to connect to the server while it already has a connected client, the server should reject the request.

Receiver Process

A receiver process (`rt_rcv`) should be run on the destination machine with the following interface:

```
./rt_rcv <loss_rate_percent> <server_ip>:<server_port> <app_port> <latency_window>
```

`loss_rate_percent` is the loss percentage to emulate for sent packets

`server_ip` is the IP address of the host where the `rt_srv` process runs

`server_port` is the port the `rt_srv` process receives messages on (i.e. this should match the `client_port` you give the `rt_srv` process)

`app_port` is the port where the application is waiting to receive messages. The `rt_rcv` process should "deliver" messages when their time comes by sending them over UDP to this destination port. You can assume the application program runs on the same machine as the `rt_rcv` process (i.e. you are sending to localhost 127.0.0.1:app_port)

`latency_window` is the amount of time to buffer each received packet before delivering, in milliseconds (see description of `deliveryTime` calculation below)

On startup, the `rt_rcv` process should try to initiate a connection the `rt_srv` server. If the server rejects its request, the `rt_rcv` client should simply report this and exit immediately (it should not retry the connection). If the `rt_rcv` process's request is accepted, you can assume it runs forever (i.e. it does not need to exit until you manually kill it, e.g. with `ctrl-c`).

The `rt_rcv` process should attempt to maintain a smooth delivery pattern (i.e. reproduce the input pattern), similarly to what we discussed for SRT. Based on its initial connection to the server, it should calculate the `baseDelta` as we discussed (`baseDelta` = `recvTime` - `sendTS`), where `recvTime`

is the time (on the receiver's clock) when the packet was received, `sendTS` is the timestamp (on the sender's clock) that indicates when the packet was sent, and `baseDelta` is equal to the oneway network delay plus the difference between sender and receiver's clocks. (Hint: be careful to ensure your calculation is still accurate even if there is loss during the initial request/accept phase.) Similarly to what we discussed in lecture, your `rt_rcv` process should compute the target time to deliver each packet as $\text{deliveryTime} = \text{sendTS} + \text{baseDelta} + \text{latencyWindow}$. Note that `sendTS` in this case should represent the original time the packet was sent (it should not be updated after retransmission).

For this project, you can assume that clock skew and roundtrip time DO NOT CHANGE once the initial request is accepted.

Note that the size of the buffer you need to maintain at the sender and receiver depends on the bitrate of the input stream (which determines how many packets are sent per second) and the latency window (which determines the length of the history that needs to be kept). You may assume a maximum bitrate of 20Mbps, 1400-byte packets, and a maximum latency window of 1 second, and may use a fixed size buffer large enough to accommodate these maximums.

Loss Emulation

As in project 1, to check the software, each process (both `rt_srv` and `rt_rcv`) should be calling a wrapper routine named `sendto_dbg` instead of the `sendto` system call. The wrapper routine will allow control of an emulated network loss percentage for each sent packet. **This should be done for every send operation between your `rt_srv` and `rt_rcv` processes (both for the sender and the receiver).** You do NOT need to inject loss when delivering packets from `rt_rcv` to the client application -- for these calls **only**, you should use `sendto` instead of `sendto_dbg`.

Reporting

Every 5 seconds, both the sender (`rt_srv`) and receiver (`rt_rcv`) should report:

1. The total amount of "clean" data (i.e. not counting retransmissions) successfully transferred so far:
 - A. in megabytes
 - B. in packets
2. The average transfer rate (for clean data) for the whole transfer so far:
 - A. in megabits/sec
 - B. in packets/sec
3. The sequence number of the highest packet sent/delivered so far

For the receiver only, you should additionally report:

1. The total number of packets lost/dropped so far (hint calculate based on difference between highest received sequence and total packets received)

2. The average, minimum, and maximum oneway delay for **receiving** packets (note that this is different than the delay for **delivering** them, since it doesn't include the extra waiting time to smooth out the delivery)

For the sender only, you should additionally report:

1. The total number of retransmissions sent so far

Evaluating and Analyzing Performance

Similar to Project 1, you will perform your evaluation using an emulated wide-area environment using GENI.

Emulated Wide Area Setup

We have given you a GENI request rspec that you can use to set up your GENI evaluation environment. The rspec is located at:

https://sites.pitt.edu/~babay/courses/cs2520/ex2_request_rspec.xml
(https://sites.pitt.edu/~babay/courses/cs2520/ex2_request_rspec.xml)

The rspec includes 2 nodes in one site and a 20 Mbps link between them. You should use node-0 in the topology as your sender (rt_srv) process and node-1 as your receiver (rt_rcv) process. Node-0 has IP address 10.0.1.100 and Node-1 has IP address 10.0.1.101.

You should emulate a roundtrip latency of 40ms between your nodes. You should use the Linux netem tool to do this.

Specifically, to add latency, you should run the following on EACH of your two VMs (node-0 and node-1):

```
sudo tc qdisc add dev eth1 root netem delay 20ms
```

This adds 20ms latency in each direction, for a round-trip delay of 40ms.

To test that it was successful, you should test pinging between the two nodes.

On node-0, run:


```
ping 10.0.1.101
```

You should see delay measurements of about 40ms.

To remove the added delay you can run on EACH of your two VMs (node-0 and node-1):

```
sudo tc qdisc del dev eth1 root
```

You can test pinging again to check that it was successful. You should now see delay less than 1ms.

For more information on netem, see: <https://wiki.linuxfoundation.org/networking/netem> 
(<https://wiki.linuxfoundation.org/networking/netem>)

Running the Traffic Generation Tool

We provide the `udp_stream` tool to generate a stream of UDP data, and the `udp_stream_rcv` tool to receive a stream of UDP data. For your performance benchmarks, these will serve as your application.

You should run the `udp_stream` tool on the same GENI node as your `rt_srv` process and the `udp_stream_rcv` tool on the same GENI node as your `rt_rcv` process. The `udp_stream` tool will send traffic to your `rt_srv` process on a local UDP port (the `app_port`), and your `rt_rcv` process should similarly "deliver" its data to the `udp_stream_rcv` tool over a local UDP port.

As an example, you can run:

On GENI node 1, in 2 separate terminal windows:

```
./udp_stream_rcv 7777
```

```
./rt_rcv 0 10.0.1.100:6666 7777 180
```

On GENI node 0, in 2 separate terminal windows:

```
./rt_srv 0 5555 6666
```

```
./udp_stream 127.0.0.1:5555 25000
```

In this example, `udp_stream` sends to `rt_srv` on port 5555, and `rt_rcv` connects to `rt_srv` on port 6666, and `rt_rcv` sends to `udp_stream_rcv` on port 7777 (the specific port numbers are arbitrary; you could choose any ports).

This example shows a 0% loss rate and 180ms latency window. For your benchmarks, you will experiment with a range of loss rates and latency windows.

Performance measurements

For your analysis, you should analyze the relationship between the emulated loss rate, the latency window, and the loss rate experienced by your protocol.

You should test the following loss rates: 0%, 1%, 5%, 10%, 20%, 30%.
and the following latency windows: 10ms, 50ms, 100ms, 180ms

You should test all combinations of loss rates and latency windows, for a total of 24 experiments. You should run each experiment at least 3 times and average the results. For each experiment, you

should use the provided traffic generation program to send 25,000 messages (each experiment should take about 30 seconds to run).

Please create a **table** with all your results and produce a **graph**, with the x-axis being the emulated loss rate (0-30%), the y-axis being the loss rate experienced by your protocol (dropped packets/total packets), and a separate line for each latency window.

Discuss the results. Do they match your expectation, based on how the protocol works? Can you write down a formula for the "experienced" loss rate based on the emulated loss rate, latency window, and round trip time?

Submission

Two separate submissions are required.

1) An electronic submission of your initial design. Due **Monday October 3 at the beginning of class**. You should also bring your initial design document to class on October 3 (either print it out or bring a laptop, as we will discuss your designs in class).

An initial design document should be a well-written, complete design of your `rt_srv` and `rt_rcv` programs. This includes the main ideas behind your protocol, the internal data structure in each program, the type and structure of the messages used, and the protocol (algorithm) description. It should be 1-2 pages long with diagrams as needed.

A final design document is required with your final submission. The final design document should be updated to reflect any changes made to your initial design.

2) An electronic submission of all the documents/code described above. Due Friday October 21 at 11:59pm. You should make a zip file named with both teammates' Pitt IDs with all your documents and code and upload the zip file to Canvas.


The complete submission should include your final design, performance results and discussion, `rt_srv.c` code, `rt_rcv.c` code, and a Makefile to build your programs.

The programs should be complete, sound, and well documented. We should be able to easily run all test cases without modifying your code.

Bonus Opportunities

For an extra challenge, you can go beyond the basic description given above. Each opportunity is worth 8 points, for a total of 24 possible bonus points.

Opportunity 1: Video Demonstration

Rather than demonstrate your program only with the toy traffic generation program we gave, you can show it working with actual video traffic. We will provide a [sample video](https://canvas.pitt.edu/courses/165007/files/10477052?wrap=1) (<https://canvas.pitt.edu/courses/165007/files/10477052?wrap=1>)  (https://canvas.pitt.edu/courses/165007/files/10477052/download?download_frd=1) and [instructions](https://canvas.pitt.edu/courses/165007/pages/project-2-bonus-video-demo-instructions) (<https://canvas.pitt.edu/courses/165007/pages/project-2-bonus-video-demo-instructions>) for playing it out over UDP to your `rt_srv` program. Then, you can use VLC media player (<https://www.videolan.org/vlc/>) to play the UDP stream delivered by your `rt_rcv` program. Note that this will require you to run the `rt_rcv` program on the same machine where you are running VLC (e.g. your laptop). You may want to use a Linux VM on your laptop for this, particularly if you use Windows.

The GENI request `rspec` we provide will provision a public IP address on your node-0. You should use this public IP address as the `server_ip` for your `rt_rcv` instance when you run your demo. The machine where your `rt_rcv` instance runs does not need a public IP address, since the server will use the IP address and port it receives the initial request on to determine where to send the data.

You should produce a short screen capture (video recording) that demonstrates your program working with different loss rates.

Opportunity 2: Extended Video Demonstration

Augment your `rt_srv` program and `rt_rcv` programs with an option to just directly use UDP (with `sendto_dgb`) instead of your realtime recovery protocol. Produce a short screen capture that compares using plain UDP to using your protocol.

Opportunity 3: Dynamic Adaptation

Modify your programs to ensure smooth delivery even if your sender and receiver clocks drift over time, or the RTT changes. Your design document should describe the changes you made AND how you tested that they work. If you do this, please submit BOTH versions of your `rt_srv.c` and `rt_rcv.c` programs (i.e. the original static version and updated dynamic version).