06_saw

Jump to bottom

jemusser edited this page on Oct 27, 2022 · 4 revisions

Reliable UDP (Part 1)

The purpose of this assignment is to extend the `netster` file-transfer behavior so that it supports reliable network communication over UDP. We call this Reliable UDP or **RUDP**. You already implemented an application protocol over TCP and UDP and you will now have an opportunity to test your existing code over an unreliable network. Once the limitations of UDP over an unreliable channel become clear you should feel motivated to solve the issue at hand with your own RUDP implementation.

This assignment requires both careful planning and likely several iterations of your design before it meets the specifications put forth in the tasks below. You will have more time to complete this assignment, but be sure to start early and begin experimenting with a lossy network to understand how things work. Good luck!

Emulating your local coffee shop's free public Wi-Fi

The `solar` and `lunar` machines have some "bad" ports[fn:note] - they have more delay and sometimes drop random packets! Up to this point you've been using the "good" ports that generally work reliably. But for this assignment your messages must survive the harrowing journey through this sketchy part of the network.

These are the port ranges with the bad characteristics:

dst port range	latency	loss
2048-2304	25ms	0%
4096-4352	25ms	1%

8192-8448	25ms	30%

[fn:note]: The “badness” is something we have simulated on purpose with a “traffic control” program called tc (we know which ports are bad because we chose them). In the real world, bad networks occur spontaneously for real-world reasons like congestion or bad physical connectors.

How to develop and test with packet loss and delay

- Other ports (outside the ranges described in the table above) are normal, with no special latency or loss behavior.
- The test ports add 25ms latency in both directions, or 50ms round-trip-time (RTT).

You can observe the impact of 30% loss across the network using our UDP file transfer from the previous assignment:

Server (on `solar.open.sice.indiana.edu`) is missing some messages from the client:

```
$ ./netster-ref -p 8199 -u -f newfile.txt
Hello, I am a server
$
```

Let’s assume that we have a file called `example.txt` containing a large amount of text. Client (on `lunar.open.sice.indiana.edu`) is missing some responses from the server:

```
$ cat example.txt
Hello this is an example text file.
It has lots of words blah blah are you still reading?
Anyway pretend this is many hundreds of bytes long for the purpose of this example.
Sincerely,
The end.

$ ./netster-ref -p 8199 solar.open.sice.indiana.edu -u -f example.txt
Hello, I am a client
$
```

After the above interaction, we can examine the contents of `newfile.txt` on the server:

```
$ cat newfile.txt
Hello is an exaptext file.
```

```
It has lots
Anyway prend this many dreds of bytes lng for the pur of this exa Sincerely,
The .

$
```

Some parts of the file are missing, because the transport was unreliable! Next, if we replace the `-u` option with `-r 1` and try again, it may take a little longer (more messages), but we'll end up with a perfect copy of the original file:

```
# server
$ ./netster-ref -p 8199 -f newfile.txt -r 1   # NOTE - we added the -r 1
Hello, I am a server
$



# client
$ ./netster -p 8199 solar.open.sice.indiana.edu -f example.txt -r 1  # NOTE - added -r 1 opti
```

Now we can check the file we sent on the server side:

```
$ cat newfile.txt
Hello this is an example text file.
It has lots of words blah blah are you still reading?
Anyway pretend this is many hundreds of bytes long for the purpose of this example.
Sincerely,
The end.
```

It's the same as `example.txt`, which means we have prevailed over the unreliable network!

Task 1 - Implement alternating bit, stop-and-wait protocol

You will implement a stop-and-wait protocol (i.e. alternating bit protocol), called **rdt3.0** in the book and slides.

Since you implemented the UDP client/server in the previous assignment, you already have experience with a similar unreliable channel interface (i.e. `udt_send()` and `udt_recv()` from **rdt3.0**). You must now implement the `rdt_send()` and `rdt_rcv()` interface that your client and server will use when running in RUDP mode. In other words, your `rdt_` methods should provide the reliable service for the calling application and use the underlying UDP `sendto()` and `recvfrom()` methods to access the unreliable channel.

There are a number of ways to implement this task correctly, but the following tips indicate some necessary and optional features:

- You **must** introduce a new RUDP header that encapsulates the application data sent in via `rdt_send()`. This should include fields to support sequence numbers, message type (ACK, NACK, etc.), and potentially other fields like length.
- You **must** use a timer.
- You **must** treat any potentially corrupt packets the same as if they were simply lost.
- You **may** assume unidirectional data transfer (e.g. client → server).
- You **may** make adjust the how often data is sent into your RUDP interface for debugging purposes.
- You **may** use both ACK and NACK control messages instead of duplicate ACKs.
- You **may** assume only a single client RUDP session at a time.
- You **may** assume that UDP handles the checksum error detection for you.

Your strategy for state management and handling control messages is up to you. The **rdt3.0** state machines from the book and slides may be guides but the expectation is that many different implementations should arise from this assignment.

Setup

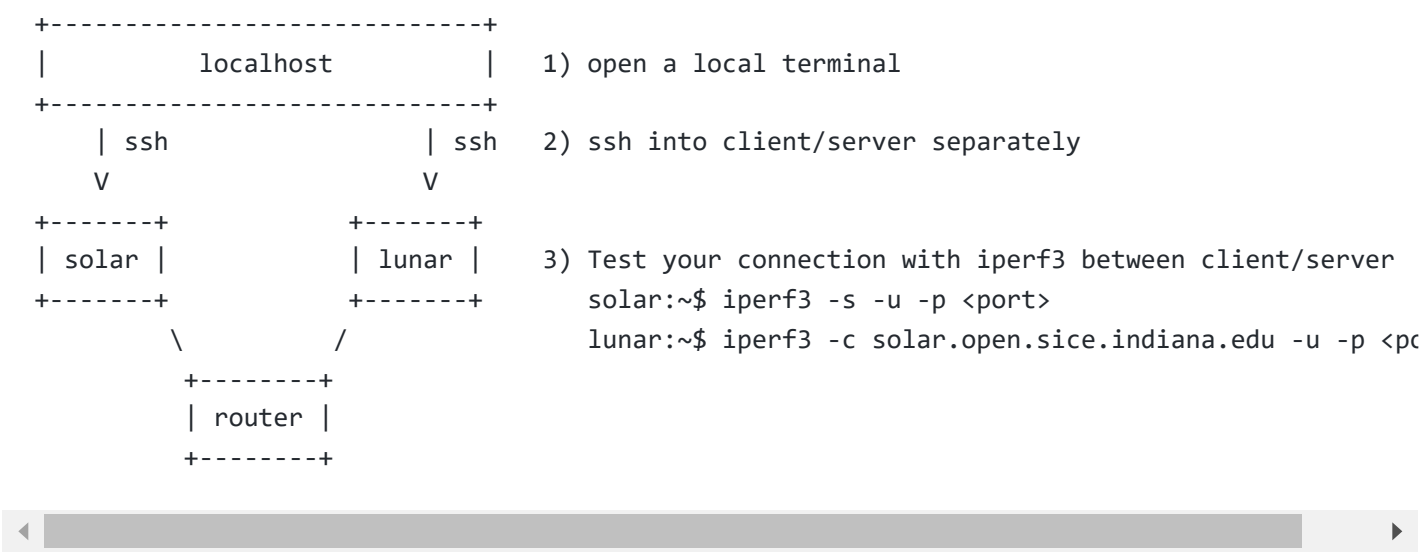
Ensure you have up-to-date code from https://github.iu.edu/SICE-Networks/Net-Fall22 either by downloading the repository or using git commands.

Run `make part3` (one time only) to copy the relevant files from the .templates folder into your working directory.

Edit the file:

- edit `stopandwait.c` if using C language, and then run `make` without arguments to build it, *or*
- edit `stopandwait.py` if using Python language

Testbed



Instructions

For this assignment, there is only one task: extend your `stopandwait_client` and `stopandwait_server` functions to use stop-and-wait to transfer files.

- This should function correctly for all 3 port ranges
- To use RUDP, include the `-r` and `-f` command line arguments for both server and client:
  - server: `./netster -p PORT -f FILENAME -r 1`
  - client: `./netster server.ip.address.example.com -p PORT -f FILENAME -r 1`
- You must still support the `-i IFACE` argument but for testing it's fine to use the defaults.

TO SUBMIT

- Submit by uploading your files to the autograder here (C version) or here (Python version). Make sure that you are submitting to the correct assignment.
- Upload your `stopandwait.c` file.
- If your code completes and you are happy with your grade, you are done (you may submit up to 5 times).

Clone this wiki locally

```
https://github.iu.edu/SICE-Networks/Net-Fall22.wiki.git
```