EE122: Introduction to Computer Networks — Fall 2007

Project 2: Design a Reliable Transfer Protocol*

Phase 1 — Due Tue Nov 13, 2007 11PM
Phase 2 — Due Sat Dec 8, 2007 11PM

Prof. Vern Paxson / Lisa Fowler / Daniel Killebrew / Jorge Ortiz

Introduction

In Project 1, you built a functional Web applications that could interact with other Web servers and clients in the world at large. Building these in an interoperable fashion required adherence to application-layer specifications such as RFCs, but leveraged the Internet's underlying, standard TCP transport service.

In this project, you will apply the networking principles you learned in this course to design your own protocol for reliable data transport. You will deepen your understanding of design issues concerning reliability and congestion control by building this protocol on top of the unreliable, datagram-oriented service provided by UDP. To evaluate the effectiveness of your custom protocol, you will implement programs to use your protocol to transfer a file from a sender to a receiver (and, optionally, "port" the Project 1 Web server and client to use your custom protocol instead of TCP).

Project Structure

This project consists of the following two phases:

Phase	Due Date	Overview of Tasks	Deliverable	% Grade
1	Nov 13 11pm	Design a reliable transport protocol on top of UDP.	Design Report	20%
2	Dec 8 11pm	Implement your reliable transport protocol as designed in Project 2 Phase 1. Develop programs to use your protocol to transfer files. Optionally revise your preferred Project 1 Web client and server to use your new reliable transport protocol.	Working implementation	80%

*Version 3: Tue Nov 20 – Porting Project #1 client/server is now optional for extra credit. This changes the specifics of what programs you turn in for the main credit (they are now simple file transfer programs.) Version 2: Fri Nov 2 – Clarified abstracted system calls.

Team Work

You can work in teams of 2 for this project. You are free to choose your own partner. You may also work on your own (no extra credit or additional time granted for working alone.) Since this project involves a significant amount of design and implementation, you are encouraged to work with a partner. Be prudent in your choice of partner, as you will each rely significantly on the other in order to effectively divide up the work. If you choose to have one, you should select your teammate by the Phase 1 deadline, and turn in a single writeup with both of your names on it.

1 Phase 1: Designing a Reliable Transport Protocol

TCP is the workhorse of today's Internet, carrying the majority of today's Internet traffic. TCP provides a reliable, in-order, byte-stream abstraction. It employs flow control to prevent fast senders from overwhelming slow receivers, and congestion control mechanisms to react to network congestion by lowering the sending rate. In Project 1, you used TCP to reliably transfer data between your Web server and your Web client.

In this project, you will design and implement your own reliable transport protocol on top of an unreliable transport protocol, namely UDP. In Phase 1, you will design this protocol.

Your protocol should have the following features:

1. **Reliable Data Transfer:** The protocol should reliably transfer data from a sender to a receiver. The underlying packets containing your data (*e.g.*, UDP packets) may be lost, re-ordered, replicated, or delayed inside the network. Your protocol must handle these conditions.
2. **High Performance Data Transfer:** The protocol should transfer data quickly by using an approach such as Sliding Window.
3. **Congestion Control:** Your protocol should "play nice" with the network world at large by modifying its behavior in response to network congestion as indicated by packet drops. This, along with the need for high-performance data transfer, means you need to adopt an appropriate response to the state of network congestion or its absence, such as AIMD.
4. **Flow Control:** The receiver should be able to limit the rate at which the sender sends it data, *i.e.*, a fast sender should not overwhelm a slow receiver.

1.1 Additional Considerations

- You do NOT have to implement all the features of TCP. You may borrow mechanisms from TCP to implement the features described above.
- TCP is a symmetric protocol; a single TCP connection can simultaneously transfer data in both directions. Your protocol should transfer data in a single direction *only*—from sender (connection originator) to receiver (connection responder). You should optimize the protocol (and the headers associated with it) for unidirectional data transfer.

  This means that your design should differ somewhat from TCP.

  Optional: If you wish to support bidirectional transfers, note this in your design writeup. In addition, when doing so, you still must use a design other than copying TCP.

- TCP was *standardized* more than 25 years ago. Do not feel you must restrict yourself to the styles used in TCP! If you have a method that can improve on TCP's approaches, use it!

1.2 Phase 1 Deliverables

Due Nov 13 2007 11PM

Phase 1 is worth 40 points.

You need to submit a design report that contains the following information:

- A description of your proposed protocol, including any extra messages or headers that you might use. Sketch the headers, including the size of different fields and their format/representation.
- A finite-state machine description or diagram illustrating the operation of your proposed protocol from both the sender's and the receiver's perspectives.
- Describe how your protocol achieves reliability. Briefly discuss how your protocol deals with:
  1. Lost data.
  2. Corrupted data.
  3. Reordered data.
  4. Replicated/duplicated data.
  5. Delayed data.
- Discuss any limitations imposed by your design, including (but not limited to):
  1. How does your protocol deal with extreme loss?
  2. How well can your protocol take advantage of a very high bandwidth path?
- To what degree is your congestion control scheme fair?

Keep the report short and concise. If you think you need more than 5 pages (12 point font, single or double column), contact Lisa *in advance*. You do not need to fill 5 pages if you can express your design adequately in less.

The report must be in one of the following formats: `doc, pdf, ps, html` or `txt`. You can structure the report in any way you deem fit, but clarity and presentation will be valued during grading.

We will aim to return feedback and grades for your design within 3 days after the Phase 1 submission deadline.

1.3 Phase 1 Submission Guidelines

1. Log in to your instructional account.
2. Create a directory called `proj2phase1: mkdir proj2phase1`
3. Change to that directory: `cd proj2phase1`
4. Copy all the files/subdirs that you wish to submit to this directory
5. Run the submit program: `submit proj2phase1`
6. Make sure that no error messages are displayed. If an error occurs, retry. If it still occurs, contact the TAs.

2 Phase 2: Implementing Your Reliable Transport Protocol

In this phase, you will implement your design (modified based on the feedback you received for Phase 1) to produce a working transport protocol implementation. You will then test your implementation by developing programs to use your protocol to transfer a file from one host to another. You can attain extra credit by in addition porting your Project 1 Web client and server to interoperate in a basic way using your transport protocol.

As discussed above, for full credit the transport protocol you implement needs to use a transmission scheme that achieves good performance and robust reliability, and provides congestion control. You can however receive partial credit for implementing a simpler Stop-and-Wait scheme; it may be prudent to first get this fully working (*i.e.*, including implementing your programs for file transfer, and then extend your implementation to support your higher-performance scheme.

2.1 Additional Considerations

- The computers available for testing your implementation are all located in the same building, possibly on the same LAN. In such a scenario, the RTT is often very low, and packet drops are rare. In order to simulate packet drops, packet re-ordering and high RTTs, you will send your UDP packets through a network emulator and not directly with the regular UDP `sendto()` function. You do this by using the **MNL Emulator** (MNL = *My Network Layer*), which replaces `sendto()` with its own custom call, `MNL_sendto()`. Thus, for datagram transmission your transport protocol does NOT operate over regular UDP, but instead uses the MNL emulator.

  See the Appendix for more information on how to use the MNL emulator.

- Because you are using your own reliable transport protocol, your file transfer programs will **not** be able to directly use many of the socket-oriented system calls, such as `connect()`, `listen()` or `accept()`.[1]

  To this end, we recommend when implementing your transport protocol you provide your own equivalent system calls (with different names), such as: `my_connect()`, `my_listen()`, `my_read()`, `my_write()`, `my_bind()`, *etc.*. You can then implement your file transfer programs in terms of invoking these calls rather than the corresponding originals that you would use in TCP. Similarly, if you pursue the extra credit of porting your Web client and server to use your custom protocol, you can do so by just changing their invocations of the original system calls to instead be to these.

  Remember, within each of your system-call equivalents you **will** use another set of socket calls in order to layer your protocol on top of MNL/UDP. Make sure that you keep a clear notion of the call or layer at which you are interacting or interfacing: your reliable transport protocol runs over the MNL Emulator, which in turn runs over UDP, which is provided by the kernel via `sys/socket.h` system calls. Each layer has its own notion of the above system calls. Take care to know which you are using!

- For the extra credit, you do not have to worry about ensuring that your Project 1 implementation fully complies with all of the Project 1 requirements. As long as your Web client can make a valid GET request and the server can reply to it with the corresponding file, that suffices for this project. For example, you do not need to worry about HTTP errors, chunking, or the server dealing with multiple clients at the same time.

[1] Note that you don't need to use `select()`, since to keep the project focused on the transport protocol's workings, your server needs to only support **one** connection at a time, pending or admitted.

If you are concerned about your chosen implementation, feel free to use the reference solutions from Project 1 instead.

- Note that since your custom reliable data transport protocol is *unidirectional*, for the extra credit you will need to change the Project 1 client and server to use a pair of connections (each using your protocol) between them, one for each direction, rather than a single TCP connection.

2.2 Simple File Transfer

To evaluate your transport protocol, you will implement two programs that use it to transfer a file from one host to another. The first program is the *sender* (named `file_send`), and it initiates the connection used for the transfer. (A correct term for this program would be "client," but some students find that confusing because they're used to clients being the processes that receive data. This by no means needs to be the case—for example, an HTTP client can transfer data to an HTTP server using `POST`.) The second is the *receiver* (named `file_recv`), and it is the responder to the connection initiated by the sender.

To run these programs, you first start up the receiver, which does the equivalent of `accept()` to await an incoming connection. You then run the sender, directing it to connect to the receiver and specifying to it which local file it should transfer to the receiver. When the receiver gets the entire file, it stores it in the current directory.

Your receiver should operate with the following command-line invocation:

> `file_recv` *listening-port filename*

which means that it should listen for a new connection on the given port and store whatever that connection transfers to the given local file.

For your sender, the invocation looks like:

> `file_send` *receiver-host receiver-port filename*

which means that it should connect to the receiver on the given host and port, and send to it the contents of the given local file.

For example:

```
file_recv 8899 newstuff
```

where `8899` in this example is the port on which your receiver is listening, and it will record the data it receives to the file `newstuff` in its current directory.

```
file_send 127.0.0.1 8899 ~/awesome.txt
```

where `127.0.0.1` is the IP address of the receiver (you should support hostnames too), `8899` is the port on which the receiver is listening, and `~/awesome.txt` is file that you wish your sender to transfer to the receiver.

Sender Algorithm

1. Retrieve the specified file
2. Connect to receiver
3. Fill outgoing buffer with file contents
4. While there's still data to send or have acknowledged:
   (a) Send what's allowed
   (b) Receive incoming ACKs and/or process timers that have expired
5. Close connection (by your definition of connection termination)

Receiver Algorithm

1. Wait to accept an incoming connection
2. While not closed (by your definition of connection termination):
   (a) Receive data and generate ACKs
3. Process close of connection (by your definition of connection termination)
4. Save file to disk

Note: These steps are given in high-level terms. You will have to augment them with considerations with regard to the features you have designed into your protocol, such as sliding window, flow control, congestion control, loss detection, and so on.

Feel free to take code from your previous HTTP client-server implementation or elsewhere, provided that you note the original source.

2.3 Grading

Due Dec 8 2007 11PM

Phase 2 is worth 160 points:

- **Reliable Transport (70 points)**
  - *Sliding Window-based (or other efficient transfer) scheme (70 points)*
    If you implement an efficient transfer scheme, such as one based on Sliding Window, you can get up to the maximum 70 points for the reliable transport portion of the project, as well as possibly full credit for flow control and congestion control.
  - *Stop-and-Wait (45 points)*
    If you instead implement a simple Stop-and-Wait scheme, you can get up to 45 points for the reliable transport portion of the project. *However, note that you cannot then receive any credit for implementing flow control, and at most 10 points for congestion control (exponential timer backoff), for a total possible score of 90 out of 160 points.*

  You should implement **ONE** of the two possible reliable transport schemes. You will be graded for only one of the schemes. You must specify which scheme you wish to be graded for in your `README` file.

- **Flow Control (15 points)**
  You will receive 15 points for implementing correct flow control.

- **Congestion Control (40 points)**
  You may use a subset of TCP-like congestion control mechanisms. Be sure to describe the congestion control scheme(s) you use in `README`. Scoring reflects the degree to which your implementation is comprehensive:

  - *A form of retransmission timeout (RTO) backoff (10 points)*
  - *A form of Slow Start (10 points)*
  - *A form of congestion control that governs how to increase your sending rate in the absence of congestion, and decrease it in the presence of congestion, such as AIMD (10 points)*
  - *A recovery scheme that does not require timeouts, but includes a congestion response, such as Fast Retransmit (10 points)*

- **Simple File Exchange Application (20 points)**
  You need to demonstrate your protocol's ability to correctly provide reliable data transfer. To do so, you must build a simple single-use sender (connection originator) and receiver (connection responder), as described in Section 2.2.

- **Packet Parser/Printer (15 points)**
  You need to build a parser that can interpret your packets appropriately according to your custom header format and print a representation of each packet. You can use `tshark` or `tcpdump -s 0` to capture the packets. We will provide a skeleton program written in C that reads tcpdump/tshark save files, for which you then supply a "printer" function that parses the raw captured packets and prints them in a human-readable form, according to your own custom protocol specification.

  If you choose to not build a protocol parser, you will not receive credit for this part, however you must provide descriptive logging in your programs to demonstrate the operation of your congestion control mechanisms (see below).

- **EXTRA-CREDIT: Modified Web Server and Client (20 points)**
  You can receive up to 20 additional points for modifying your preferred Project 1 Web server and client (either your own, or the reference solutions) to use your custom reliable transport protocol.

2.4 Phase 2 Deliverables

Due Dec 8 2007 11pm

Your project must build two programs, `file_send` and `file_recv` in the root directory of `proj2phase2`. These are invoked as discussed in Section 2.2.

You must describe in your `README` file which schemes you chose to implement for your transport protocol.

Additionally, your packet parser must also build to the root directory and must be executable by the name `proto_parser`. The parser is invoked:

```
proto_parser -f binary_trace_file
```

where *binary_trace_file* is a packet capture file generated by specifying the option `-w` *binary_trace_file* to `tshark` or `tcpdump -s 0`.

To demonstrate your congestion control mechanism(s), you need to include a descriptive sample output that shows the timestamps and sequence numbers of each packet as it was received by the receiver, and of each packet as sent by the sender. You will generate the data for this by first simulating particular network conditions using the provided MNL configuration files, transferring files of varying sizes (also provided to you) between your sender and receiver, capturing the packets using `tshark` or `tcpdump`, and then parsing the packet capture file using your own packet parser program.

You must then process the output into table form, so that the data appears as comma-separated values with a single header line. Generate two files, `cc_send_x.csv` and `cc_recv_x.csv`, for traces from the sender's perspective and the receiver's perspective, respectively, where `x` indicates the number of the corresponding config file and transfer file.[2] You will find the particular MNL configuration files and the files that you must transfer between the sender and receiver at `http://inst.eecs.berkeley.edu/~ee122/fa07/projects/p2files/`, along with examples of what the `.csv` files look like.

If you pursue the extra credit, you need to note this fact in your `README`, and your project must build to also include your modified `http_client` and `http_server` in the root directory of `proj2phase2`. In the `README`, briefly explain what functionality these programs provide and any considerations in running them.

2.5 Submission Guidelines

1. Log in to your instructional account.
2. Create a directory called `proj2phase2`: `mkdir proj2phase2`
3. Change to that directory: `cd proj2phase2`
4. Copy all the files/subdirs that you wish to submit to this directory
5. Run the submit program: `submit proj2phase2`
6. Make sure that no error messages are displayed. If some error occurs, retry. If it still occurs, contact the TAs.

[2] *E.g.,* you will pair `mnl_conf_3.txt` with `file3.txt` by running `MNLDaemon mnl_conf_3.txt` and then have your sender transfer `file3.txt` to your receiver. You will capture the packets exchanged using either `tshark` or `tcpdump`. You will then parse the results, and populate and name the sample files `cc_send_3.csv` and `cc_recv_3.csv` .

Appendix: My Network Layer (MNL) Emulator

You must use the My Network Layer (MNL) simulator while implementing the reliable transport portion of Project 2. The MNL simulator can artificially delay, drop and re-order UDP packets. This will help you easily test if your reliable transport protocol works correctly under different network conditions.

2.6 Downloading and Unpacking MNL

Download the latest version of the MNL simulator (MNL version 0.1a) from the course webpage at
`http://inst.eecs.berkeley.edu/~ee122/fa07/projects/MNL_0_1a.tar.gz`
Untar/Unzip it into the directory containing your source code.

2.7 Example Programs

The MNL distribution comes with example programs that send and receive packets: `TestSend.c` and `TestRecv.c`. These programs can be compiled using `Makefile.MNL` as follows:

`make -f Makefile.MNL` Before you can run a program X that uses MNL, you must start the MNL-Daemon in a separate terminal window. The MNLDaemon should be run from the same directory as the program X. MNLDaemon is started as follows: `MNLDaemon samplecfg/dnone0.txt` After starting MNLDaemon, start `TestRecv PORT_NUM` in a different terminal window. Then start `TestSend PORT_NUM` in a third terminal window. TestRecv will print out the packets sent by TestSend. The same `PORT_NUM` should be specified for TestRecv and TestSend.If you have trouble starting the MNLDaemon, please refer to `README.txt` for possible solutions.

2.8 Using MNL in your program

The best way to learn how to use MNL in your program is to study the example program `TestSend.c`.

Steps

- Include the header file `MNLClient.h` in your program.
- Before calling any other MNL function, you must call the function `MNLClientInit()`. This function will return 1 if initialization succeeded. If initialization failed, you must print an appropriate error message and exit.
- To send a UDP packet, use the `MNL_sendto()` function instead of the regular `sendto` function.
- Link the `MNLClient.o` object file to your executable — just add `MNLClient.c` to your list of source files (See `Makefile.MNL`).

`MNL_sendto()` has the following signature:

```
ssize_t MNL_sendto(
              int s,
              const void* buf,
              size_t len,
```

```
int flags,
const struct sockaddr *to,
socklen_t tolen,
int iSeqNumStart,
int iDataLen,
int iWindowSize,
int *ipError)
```

The first 6 arguments and the return value have the same semantics as the standard UDP `sendto` function (`man sendto` for more information). The additional arguments have the following meaning:

- `ipError`: If some internal error occurs within `MNL_sendto`, the integer pointed to by `ipError` is set to 1. Otherwise it is 0.
  Note that dropping a packet purposely is NOT an error. If you ever find `ipError` set to 1, please notify the TAs.
- `iSeqNumStart`: The sequence number of the first data byte contained inside the packet you are sending; this value is only used for logging purposes. If you are not implementing a sliding scheme, pass 0 for this argument.
- `iDataLen`: The number of actual data bytes (i.e., *not* including header bytes used by your transport protocol) in `buf`, also used for logging purposes. Note that `iDataLen` is not the same as `len`. `len` includes any transport protocol headers you may use in your protocol; `iDataLen` counts only the actual data bytes you are transferring via your transport protocol.
- `iWindowSize`: The current window size (in bytes). Use 0 if not using a sliding-window based protocol. Also just used for logging purposes.

When a UDP packet reaches its destination after being sent using `MNL_sendto`, the source port number in the packet will be the source port of the MNLDaemon, and not the source port of your sending program. This means that your receiving program must be able to obtain the port number to which it must reply from within the payload of the UDP packet. `TestRecv.c` does this — the first two bytes of the UDP payload contains the source port number put in by `TestSend.c`. There is no `MNL_recv()` function; use the standard UDP `recvfrom()` function to receive packets sent by `MNL_sendto()`.

2.9 Running an MNL-enabled program

You must first start MNLDaemon in a separate terminal window before running any program that uses MNL. Start MNLDaemon from the same directory from which you start the program (*e.g.*, `TestSend`) that will use MNL. After your program has finished running, you must terminate the MNLDaemon using `Ctrl-C`.

2.10 Simulating Different Network Conditions

The configuration file passed to MNLDaemon describes the network conditions the MNLDaemon will simulate.

The first line of the configuration file represents the mean RTT in microseconds. Packets sent via `MNL_sendto` are delayed by half of this value before they reach the destination.

The second line of the configuration file represents the variance in the RTT. If the variance is 0, every packet is delayed by exactly the same amount (there will still be some variation due to system clock granularity effects). If the variance is high, packets may even be reordered.

The third line describes the packet dropping policy. MNLDaemon supports four types of dropping policies:

1. `drop_none` – No packets are intentionally dropped. If you send packets at a very fast rate, the OS kernel might drop packets even if MNL did not intend to.
2. `drop_periodic:X` – Drop every Xth packet.
3. `drop_probabilistic:P` – Drop packets with probability P. P is between 0 and 1.
4. `drop_specific:N1:N2:N3` – Drop the packet sent via the `N1`th call to `MNL_sendto`, the `N2`th call and so on.

Some sample configuration files are given in the `MyNetworkLayer/samplecfg` subdirectory.

2.11 Non-Solaris Platforms

The MNLDaemon executable in the distribution is compiled for Solaris. This means that it will not work on Linux or Windows. You will need to compile the MNLDaemon from source for Linux or Windows. Source code for the MNLDaemon is included in the distribution. Please see the `README.txt` file for more details.

2.12 More Information

For more information on MNL, please have a look at the `README.txt` file in the distribution.