Project 2 - Reliable Transport

UC Berkeley CS 168, Fall 2015

Version 1.0

Due: 11:59pm, October 26th, 2015

In this project, you will build a simple reliable transport protocol, "BEARS-TP" (BTP). Your protocol must provide in-order, reliable delivery of UDP datagrams, and must do so in the presence of packet loss, delay, corruption, duplication, and re-ordering.

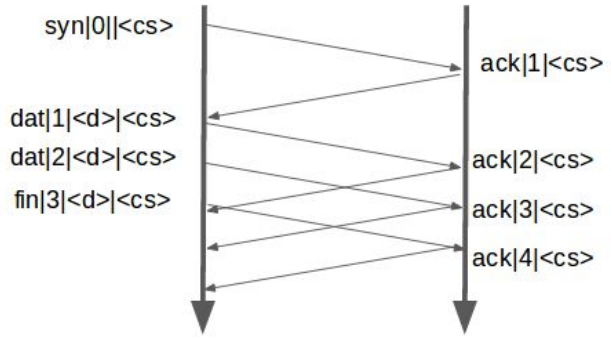
There are a variety of ways to ensure a message is reliably delivered from a sender to a receiver. We will provide you with a reference implementation of a receiver (which you must use) that returns a cumulative ACK whenever it receives a data packet. This is further explained with an example later. Your job is to implement a sender that, when sending packets to this receiver, achieves reliable delivery.

Protocol Description

Our simple protocol has four message types: `syn`, `fin`, `dat`, and `ack`. `syn`, `fin`, and `dat` messages all follow the same general format.

```
   syn|<sequence number>||<checksum>
dat|<sequence number>|<data>|<checksum>
fin|<sequence number>|<data>|<checksum>
   ack|<sequence number>|<checksum>
```

- To initiate a connection, send a `syn` message with a random initial sequence number. The receiver will use the sequence number provided as the initial sequence number for all packets in that connection. After sending the `start` message, the sender waits for an `ack` packet to finish a handshake
- After the handshake, send actual data packets in the same connection using the data message type, adjusting the sequence number appropriately.
- Unsurprisingly, the last data in a connection should be transmitted with the fin message type to signal the receiver that the connection is complete.
- The following diagram gives an example for the bears-tp protocol



An important limitation is the maximum size of your packets. The UDP protocol has an 8 byte header, and the IP protocol underneath it has a header of ~20 bytes. Because we will be using Ethernet networks, which have a maximum frame size of 1500 bytes, this leaves 1472 bytes for your entire packet (message type, sequence number, data, and checksum). However, small packet size hurts performance. Therefore, *in order to meet the performance requirement, packet size should be larger than 1000 bytes (unless it is the last packet in the stream), but less than 1472 bytes*.

The angle brackets ("<" and ">") are not part of the protocol. However, you should ensure that there are no extra spaces between your delimiters ("|" character) and the fields of your packet. For specific formatting details, see the sample code provided.

Receiver specification

We will provide a simple receiver for you; the reference implementation we provide will also be used for grading, so make sure that your sender is compatible with it.[1] The BEARS-TP receiver responds to data packets with cumulative acknowledgements. Upon receiving a message of type `syn`, `dat`, or `fin`, the receiver generates an `ack` message with the sequence number it expects to receive next, which is the lowest sequence number not yet received. In other words, if it expects a packet of sequence number N, the following two scenarios may occur

1. If it receives a packet with sequence number not equal to N, it will send "`ack`|N".
2. If it receives a packet with sequence number N, it will check for the highest sequence number (say M) of the in-order packets it has already received and send "`ack`|M+1". For example, if it has already received packets N+1 and N+2 (i.e. M = N+2), but no others past N+2, then it will send "`ack`|N+3".

Let's illustrate this with an example. Suppose packets 0, 1, and 2 are sent, but packet 1 is lost before reaching the receiver. The receiver will send "`ack`|1" upon receiving packet 0, and then "`ack`|1" again upon receiving packet 2. As soon as the receiver receives packet 1 (due to retransmission from the sender), it will send "`ack`|3" (as it already has received, and upon receiving this acknowledgement the sender can assume all three packets were successfully received.

If the next expected packet is N, the receiver will drop all packets with sequence number greater than N+6 that is, the receiver operates with a window of seven packets, and drops all packets that fall outside of that range. When the next expected packet is N+1 (due to N arriving), then the receiver will accept packet N+7.

---

[1] We will test your senders in a variety of scenarios: e.g., packet loss or data corruption. Just because your sender is compatible with the receiver in one test environment does not ensure a perfect score! Your sender must be capable of handling all scenarios outlined in this document.

You can assume that once a packet has been acknowledged by the receiver, it has been properly received. The receiver has a default timeout of 10 seconds; it will automatically close any connections for which it does not receive packets for that duration.

Sender specification

The sender should read an input file and transmit it to a specified receiver using UDP sockets. It should split the input file into appropriately sized chunks of data, specify an initial sequence number for the connection, and append a checksum to each packet. The sequence number should increment by one for each additional packet in a connection. Functions for generating and validating packet checksums will be provided for you (see Checksum.py).

Your sender must implement a reliable transport algorithm (Go Back N). Remember that the receiver's window size is seven packets, and any packet it receives outside this window will be ignored. Your sender must be able to accept `ack` packets from the receiver. Any `ack` packets with an invalid checksum should be ignored.

Your sender should provide reliable service under the following network conditions:

- Loss: arbitrary levels; you should be able to handle periods of 100% packet loss.
- Corruption: arbitrary types and frequency.
- Re-ordering: may arrive in any order, and
- Duplication: you could see a packet any number of times.
- Delay: packets may be delayed indefinitely (but in practice, generally not more than 10s).

Your sender should be invoked with the following command:

```
python Sender.py -f <input file>
```

Some final notes about the sender:

- **The sender should implement a 500ms retransmission timer to automatically retransmit packets that were never acknowledged (potentially due to `ack` packets being lost).** We do not expect you to use an adaptive timeout.
- Your sender should support a **window size of 7 packets** (i.e., 7 unacknowledged packets).
- Your sender should roughly meet or exceed the performance (in both time and number of packets required to complete a transfer) of a properly implemented Go Back N based BEARS-TP sender.
- Your sender should be able to handle arbitrary message data (i.e., it should be able to send an image file just as easily as a text file).
- Any packets received with an invalid checksum should be ignored.
- Your implementation must run on hive[##].cs.berkeley.edu. However, it should work on any system that runs the appropriate version of Python (we tested on version 2.7.1 under Ubuntu 11.04 and 2.6.5 under SunOS 5.10).
- Your sender MUST NOT produce console output during normal execution; Python

exception messages are ok, but try to avoid having your program crash in the first place.

We will evaluate your sender on correctness, time of completion for a transfer, and number of packets sent (and re-sent). Transfer completion time and number of packets used in a transfer will be measured against our own reference implementation of a Go Back N based sender. Note that a “Stop-And-Wait” sender [described below] will not have adequate performance.

Performance Improvements

Besides Go Back N, you are also required to implement Fast Retransmit and Selective Acknowledgements.

Fast Retransmit: Go Back N sender retransmits the unacknowledged packet only on a timeout event, which can wait up to 500ms. Instead, we can detect the packet loss event by duplicated ack event. If we receive three duplicate acknowledgements (that is, receiving the same acknowledgement number for 4 times), we are almost sure the packet has been lost and we should immediately retransmit the packet.

For example, if the sender sends five packets with sequence numbers 0,1,2,3,4, but packet 1 is lost. On receiving packet 0,2,3,4, the receiver will send cumulative acknowledgement `ack|1` four times back to the sender. Upon receiving the fourth `ack|1`, the sender should retransmit packet 1.

Selective Acknowledgements*:* If the sender knows exactly what packets the receiver has received, it can retransmit only the missing packets (rather than naively retransmitting the last N packets). We have modified the receiver to provide explicit acknowledgement of which packets have been received. You should implement SACK, or **S**elective **Ack**nowledgement, functionalities at the sender.

The Receiver.py runs in SACK mode if you enable the -k or --sack option. In SACK mode, in addition to sending a cumulative acknowledgement, the receiver also sends selective acknowledgements in the following format.

`sack|<cum_ack;sack1,sack2,sack3,...>|<checksum>`

cum_ack is the cumulative acknowledge, which is the same as the one in an ack packet. Following the *cum_ack* is a *semicolon* and a list of *sack*’s separated by *comma*.

For example, if the sender sends five packets with sequence numbers 0,1,2,3,4, but packet 1 and 2 are lost. The following table shows the sack packets sent by the receiver when a packet is received.

Packet Received	sack packet sent by the receiver
0	`sack|1;|<checksum>`

3	`sack|1;3|<checksum>`
4	`sack|1;3,4|<checksum>`

On getting a timeout (or three duplicate acknowledgements, if you have implemented fast retransmit), the sender should therefore retransmit only packets with sequence numbers 1 and 2.

The sender should be running in SACK mode when the -k or --sack option is on. We test your sender by running:
Receiver side: `python Receiver.py -k`
Sender side: `python Sender.py -f <filename> -k`
In SACK mode, your sender should only retransmit packets that are missing. For instance, in the previous example that packet 1 is missing, the sender should only retransmit packet 1 after the timeout.

Testing

You are responsible for testing your own project. In order to make this process easier for you, we've included a test harness that you can use to develop tests for your project. See the README in the code for the project for more details, as well as TestHarness.py and the example test cases in the tests directory. Note that **passing the example test cases is a necessary but not sufficient condition for a good score** on this project! These are only meant as tests of basic functionality and examples for your own test cases; you must think about the types of edge cases your sender could encounter, as well as the various conditions required for correctness.

Hints and Tips

To begin with, just focus on the simple case where nothing bad ever happens to your packets. After you have that case working, you can consider how to handle packet loss.

It may help to build your way up to a full-fledged reliable sender. The simplest reliable transport mechanism is "Stop-And-Wait", in which the sender transmits a single packet and waits for the receiver to acknowledge its receipt before transmitting more. You could start by building a "Stop-And-Wait" sender, and extending that for the full Go Back N based sender. Understand that "Stop-And-Wait" sender is only a stepping stone, and we require you to submit a full BEARS-TP sender for evaluation.

README

You must also supply a README file along with your solution. This should contain:

- You (and your partner's) names and login
- How challenging was this project on a scale of 1 (easiest) to 10 (hardest)?
- How long did it take you?

Code

Check the lecture schedule page for the latest copy of the project code.

Submission Instructions

You may submit your code for testing once per day, we will run a series of tests on it, and you can see how you're doing. The tests we run daily are similar to at least some of the tests we will run for the final grading, though we may not test all aspects of the project in the daily tests.

Your final grade will be the greater of 70% * highest_daily_score and score_on_final_tests. In other words, if you get 100% on your daily tests, you will not do worse than a 70% on the project. The one exception is that we do not run the full set of anti-cheating tests in the daily runs, so if you manage to get 100% while cheating on the daily tests, you may still get 0% on the final grading. You will use the provided OK tool to submit your work. See http://cs61a.org/articles/using-ok.html for some basic info on OK.

Please submit using ok as early as possible to see if there is any problem with your ok account. Also, please indicate your partner's email in okpy.org. Or we won't know your partnership.

Collaboration Policy

The project is designed to be solved independently, but you may work in partners if you wish. Grading will remain the same whether you choose to work alone or in partners; both partners will receive the same grade *regardless of the distribution of work between the two partners* (so choose a partner wisely!).

You may **not** share code with any classmates other than your partner. You **may** discuss the assignment requirements or your solutions (e.g., what data structures were used to store routing tables) -- *away from a computer and without sharing code* -- but you should not discuss the detailed nature of your solution (e.g., what algorithm was used to compute the routing table). Assignments suspected of cheating or forgery will be handled according to the Student Code of Conduct.[2] Apparently 23% of academic misconduct cases at a certain junior university are in Computer Science[3], but we expect *you all* to uphold high academic integrity and pride in doing *your own work.*

[2]http://sa.berkeley.edu/code-of-conduct

[3]http://www.pcworld.com/article/194486/why_computer_science_students_cheat.html