Project 1 - Reliable Transport

UC Berkeley EE 122, Fall 2011
Version 0.1

Due: September 28, 2011

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. For extra credit, you may choose to implement one of several performance improvements which are described below.

Protocol Description

Our simple protocol has four message types: `start`, `end`, `data`, and `ack`. `start`, `end`, and `data` messages all follow the same general format.

```
start|<sequence number>|<data>|<checksum>
data|<sequence number>|<data>|<checksum>
end|<sequence number>|<data>|<checksum>
```

To initiate a connection, send a `start` message. The receiver will use the sequence number provided as the initial sequence number for all packets in that connection. After sending the `start` message, send additional 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 `end` message type to signal the receiver that the connection is complete. Your sender should accept acknowledgements from the receiver in the format:

```
ack|<sequence number>|<checksum>
```

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).

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 `start`, `data`, or `end`, 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 us 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+4; that is, the receiver operates with a window of five packets, and drops all packets that fall outside of that range. When the next unexpected packet is N+1 (due to N arriving), then the receiver will accept packet N+5.

You can assume that once a packet has been acknowledged by the sender, 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 (such as sliding window). The receiver's window size is five packets, and it will ignore more than this. 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:

[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.

- 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> -a <destination adddress> -p <port>`

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 (though this is a bells and whistles option).
- Your sender should support a window size of 5 packets (i.e., 5 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 sliding-window 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). If no input file is provided, your sender should read input from STDIN.
- Any packets received with an invalid checksum should be ignored.
- Your implementation must run on nova.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 should be written in Python and adhere to the standards given in PEP 8, Style Guidelines for Python Code.
- 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 sliding-window based sender. Note that a “Stop-And-Go” sender will not have adequate performance.

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-Go”, in which the sender transmits a single packet and waits for the receiver to acknowledge receiving it before transmitting more. You could start by

building a “Stop-And-Go” sender, and extending that for the full sliding window based sender. Understand that “Stop-And-Go” sender is only a stepping stone, and we require you to submit a full BEARS-TP sender for evaluation.

Bells and Whistles (Extra Credit)
Some of these may require modifying the provided receiver implementation; if you choose to do one, please make sure to provide your receiver implementation with your submission.

Variable size sliding window: For networks where packet loss, corruption, delay, and reordering are minimal, a large window size will obtain higher performance. A large window on a lossy network, however, will lead to a large amount of overhead due to retransmissions. Modify your sender to dynamically adjust its window size based on network conditions.

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). Modify the receiver to provide explicit acknowledgement of which packets have been received, and add support for selective retransmission to your sender.

Accounting for variable round-trip times: How long you should wait to retransmit an unacknowledged packet is tightly related to the time it takes a packet to travel between the sender and the receiver. If you know the round trip time for a packet to reach your receiver is 20ms, waiting 500ms to retransmit is inefficient: you can be quite sure that a packet has been lost if you haven’t heard back from the receiver after 20ms, give or take a few milliseconds. Modify your sender to determine the round trip time between the sender and the receiver, and adjust your retransmission timeout appropriately.

Bi-directional transfer: While our protocol as defined only support uni-directional data transfer, it could be modified to allow bi-directional transfer (i.e., both ends of the connection could send and receive simultaneously). Implement this functionality, modifying both the sender and receiver as necessary. Be sure to provide a description of your updated protocol in your README.txt file.

I’m a superhacker: Identify a problem with a widely deployed modern TCP implementation, then augment BEARS-TP to address that problem and outperform TCP. Describe and implement your improvement, and then benchmark your solution against the standard TCP.

README
You must also supply a README.txt file along with your solution. This should contain
(1) You (and your partner’s) names
(2) What challenges did you face in implementing your sender?
(3*) Name the extra credit options you implemented; describe what they do and how they work.

Downloads
The example code for this project, as well as the receiver and checksum implementations, are

available at https://github.com/shaddi/bears-tp.

What To Turn In

Turn in a .tar file with both you and your partner's last names in the file name (e.g. *project1-shenker-stoica.tar* or *project1-katz.tar*). The .tar file should contain a Python file, *Sender.py*, that implements the *basic protocol* outlined above (**without** the Bells and Whistles!). It should also include your *README.txt* file. You may optionally include additional files, *bears-tc-sender2.py* and *bears-tc-receiver2.py* with your extra credit versions of the protocols.

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://students.berkeley.edu/uga/conduct.pdf

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