CS 3251 - Reliable Data Transfer 04/03/2017

Project Group

- Chad Miller (cmiller86@gatech.edu)
- Noam Lerner (nlerner3@gatech.edu)
- Robert Bradshaw (underwaterbasketweaving@gatech.edu)

Compiling and Running

All commands are run from the project root directory (i.e., the directory that contains the folders "client" and "server").

Server

Because the server is written in Python, no compilation is needed. To run, run the command: `python server/reldat-server.py [port number] [max window size]`.

Client

Because the client is written in Java, compilation is required. To compile, run the command: `javac client/src/ReldatClient.java`. To run, run the command: `java client/src/ReldatClient [IP address]:[port number] [max window size]`.

For convenience, the pre-compiled .class files are included. Should you choose to use those files instead, to run, run the command: `java client/bin/ReldatClient [IP address]:[port number] [max window size]`.

Files submitted

Server Files (./server/*.py)

reldat.py

This file contains the RELDAT server which implements the protocol. Three of the functions are meant to be called in an external loop:

- `listen()` - listens for incoming packets, and sends any packets that need to be sent.
- `resend_packets()` - resent unacknowledged packets that have timed out.
- `check_connection()` - ensures that the server is still connected to the client.

Details on each function are provided in the Design Documentationn section of this file.

packet.py

Contains helper functions for constructing and parsing packets. Addititionally contains a packet class to encapsulate RELDAT packet information, a packet iterator class that converts a string of data into RELDAT packet objects, several packet factory functions, and exceptions thrown when a parsed packet has a corrupted header or payload.

reldat-server.py

The entry point for the server. Boots the server and starts a loop which will continually call the three required functions described in the section explaining `reldat.py`

Client Files (./client/src/*.java)

CommandReader.java

The `CommandReader` class is a wrapper for a `Scanner` object. An instance of this class is meant to be run in a separate thread and provides a non-blocking way to read input from stdin.

ReldatClient.java

The `ReldatClient` class is the entry point for the client. Boots the client, opens a connection with the target server, and starts a loop which will parse stdin input and execute a valid `transform` or `disconnect` command.

reldat/ReldatConnection.java

The `ReldatConnection` class is a complex class that represents a connection between the client and server. An instance of this class serves as a state machine that handles all connection activities, including setup and teardown; data transmission, and packet acknowledgement. Additionally, this class manages the send and receive windows that send a file to the server and receive the transformed file from the server, respectively. Finally, this class performs extensive bookkeeping operations for file transmissions, including management of packet timeout counters, window indexes for pipelined transmisions, and counters for retransmissions to determine whether or not connection to the server has been lost.

reldat/ReldatHeader.java

The `ReldatHeader` class encapsultates the header of a RELDAT packet, including converting between the high-level header object itself and byte arrays suitable for transmission over a UDP connection (and vice-versa). For more details, refer to the "Packet Header Structure" subsection of the "Packet Design" section below.

reldat/ReldatPacket.java

The `ReldatPacket` class encapsulates an entire RELDAT packet, including the packet header and payload data. This class also contains methods for converting between the high-level packet object itself and byte arrays suitable for transmission over a UDP connection (and vice-versa). For more details, refer to the "Packet Design" section below.

reldat/exception/HeaderCorruptedException.java

The `HeaderCorruptedException` class is an Exception subclass that signifies that a RELDAT packet header has been corrupted.

reldat/exception/PayloadCorruptedException.java

The `PayloadCorruptedException` class is an Exception subclass that signifies that a RELDAT packet payload has been corrupted.

Design Documentation

All methods are documented in the files.

Packet Design

Each packet has a maximum size of 1000 bytes and contains a header and a payload. Each header a one-byte flag field, a four-byte sequence number field, a four-byte acknowledgement number field, a four-byte payload size field, a 16-byte payload checksum field, and a 16-byte header checksum field. In total, a header contains 45 bytes of information; therefore, a packet can contain up to 1000 - 45 = 955 bytes of payload data.

Packet Header Structure

```
0[N][E][D][R][A][C][O]             1 byte (flags)
[Sequence Number]                  4 bytes
[ACK Number]                       4 bytes
[Payload Size]                     4 bytes
[Payload Checksum]                16 bytes
[Header Checksum]                 16 bytes
----------------------------
[ P   A   Y   L   O   A   D ] <= 955 bytes
```

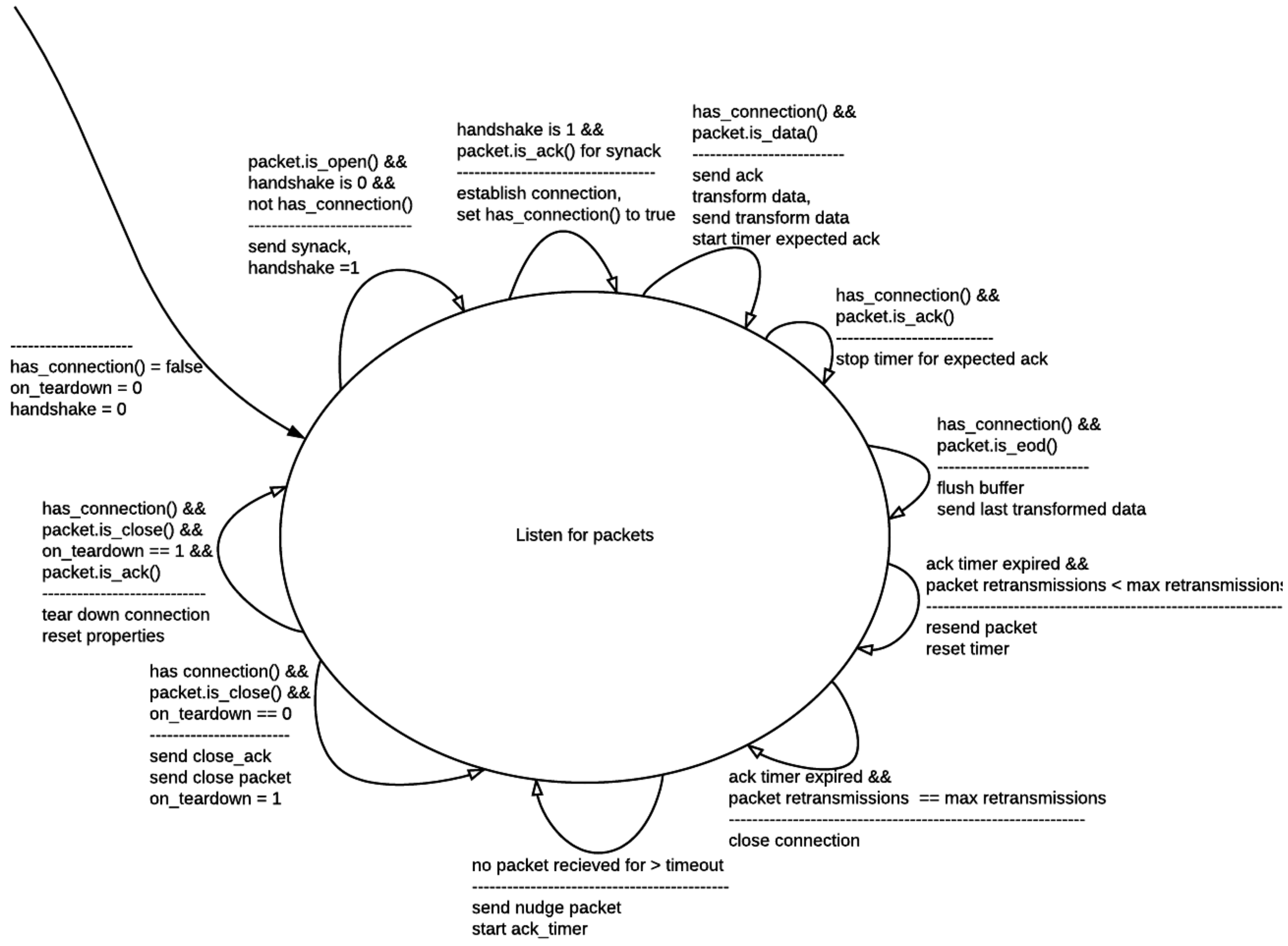
Packet Header Flags

Seven flags are defined, as shown in the Packet Header Structure section. Each flag is identified with a single letter:

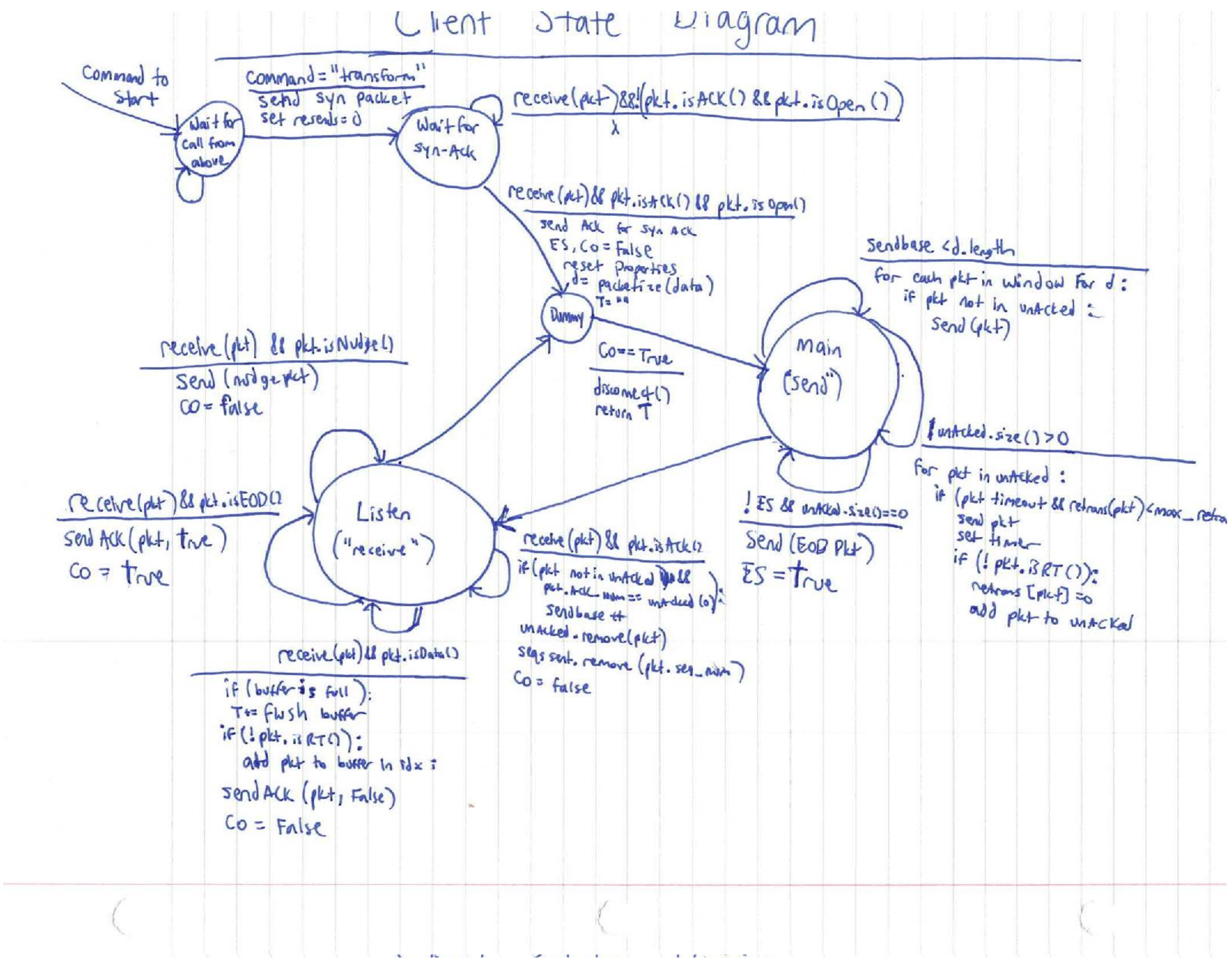
- N = Nudge (NUDGE) - a packet containing this flag is used to test the connection between the client and the server.
- E = End-Of-Data (EOD) - a packet containing this flag is used to signify that no more data should be expected until after a file transformation is complete.
- D = Data (DATA) - Packet contains data. In this case, the ACK number is 0 and the sequence number is nonzero.
- R = Retransmission (RETRANSMIT) - a packet containing this flag has already been transmitted, but for some reason went unacknowledged.
- A = Acknowledgement (ACK) - a packet containing this flag is acknowledging a data packet. In this case, the ACK number is the same as the sequence number of the packet being acknowledged, and the sequence number is 0.

- C = Close Request (CLOSE) - a packet containing this flag wishes to initiate the connection termination process.
- O = Open Request (OPEN) - a packet containing this flag wishes to initiate the connection open process.

Server State Machine Diagram


Listen for packets
--------------------- has_connection() = false on_teardown = 0 handshake = 0
packet.is_open() && handshake is 0 && not has_connection() --------------------------- send synack, handshake =1
handshake is 1 && packet.is_ack() for synack ---------------------------------- establish connection, set has_connection() to true
has_connection() && packet.is_data() ------------------------- send ack transform data, send transform data start timer expected ack
has_connection() && packet.is_ack() -------------------------- stop timer for expected ack
has_connection() && packet.is_eod() ------------------------- flush buffer send last transformed data
ack timer expired && packet retransmissions < max retransmissions -- resend packet reset timer
ack timer expired && packet retransmissions == max retransmissions -- close connection
no packet recieved for > timeout --- send nudge packet start ack_timer
has connection() && packet.is_close() && on_teardown == 0 ----------------------- send close_ack send close packet on_teardown = 1
has_connection() && packet.is_close() && on_teardown == 1 && packet.is_ack() --------------------------- tear down connection reset properties

Client State Machine Diagram



How the RELDAT Protocol Works

The RELDAT protocol implements reliable data transfer using a combination of packet timeout counters, lifeline checks, and a sliding window-based pipeline for sending and receiving data. Several safeguards to recover from the effects of an unreliable network are included; these are documented further below in this section.

Opening a Connection

The client initiations a connection by sending a packet with the OPEN flag set. A three-way handshake is then done, with the following packet exchanges occurring:

1. Client -> Server; Flags: OPEN; Payload: Client's max window size
2. Server -> Client; Flags: OPEN | ACK; Payload: Server's max window size
3. Client -> Server; Flags: ACK; Payload: Nothing

These initial packets are not treated any differently from other packets. In other words, they are subject to the same timeout and corruption checks that any other type of packet is subject to.

Transmitting Data

Our RELDAT protocol implements a reliable connection using two UDP sockets: an in-socket and an out-socket. The in-socket is bound to the port specified in the program arguments; the out-socket uses a random available port. With these two ports, bi-directional data transfer is possible. An end of the connection can and will send data while it is still receiving data. In this case, it will both send data from its data buffer while acknowledging data it receives from the other end of the connection.

Additionally, our RELDAT protocol implements a sliding window-based pipelined transmission algorithm based on Go-Back-N.

A window of packets is sent by the sender. Each packet is separately acknowledged by the recipient. The sender slides its window one packet at a time: if the acknowledged packet is an ACK number one greater than the sequence number of the packet at index 0 of the window, the window increments one packet. If there are no more packets left to slide the window over, the lower bound of the window increments, thus shrinking the window until the window has no packets left.

The recipient has a buffer containing packets it received. Packets are not appended to the buffer, but indexed. A "standardizer" for the window begins at the sequence number 3, the next sequence number expected after the three-way connection open handshake. Each subsequent packet is indexed by subtracting this standardizing integer from its sequence number. When the buffer is full (i.e., every index in the buffer contains a packet), the buffer is flushed and the standardizing integer is set to the sequence number of the next packet received.

Note that this provides a built-in way of protecting against out-of-order packets: packets are guaranteed to be placed in the buffer at their correct index instead of appended onto the buffer haphazardly. Note that this also provides a built-in way of protecting against duplicate data: the index is calculated using the sequence number, meaning if a packet has been sent twice, it will simply overwrite the exact same packet in the buffer (although this is not the case - see the "Duplicate Packets" subsection of the "Handling Common Problems" section below).

What it means to "flush the buffer" differs in the context of the client and the server. When the client flushes its buffer, that means its buffer is full of transformed data it received from the server. The client appends the contents of its buffer, sequentially, to a string containing all of the data it has received so far. All packets are then removed from the buffer and the buffer is ready to accept more data.

In the context of the server, when the server flushes its buffer, it takes the contents of each index in the buffer, transforms the data, and then sends the data back to the client in a pipelined fashion. The packets in the buffer are then transferred to the server's send window and the buffer is ready to accept more data.

Once a sender has no more data to send, it waits for the final packets in the last window it sent to be acknowledged. Once all of these packets have been acknowledged, a packet with the EOD flag is set, letting the recipient know that it should not expect any more data until another transformation operation is initiated. The recipient, upon receiving this EOD, sends a packet back with its EOD and ACK flags sent.

Lifeline Checks

To ensure connectedness between the client and server when the client is idle, the server sends a "nudge packet" every five seconds when no data is being transmitted. This packet is a packet with the NUDGE bit set. When the client receives a nudge packet, it sends an acknowledgement back with both the NUDGE and ACK bits set. If the server does not receive a nudge acknowledgement, it attempts to re-send it three more times before determining the client has died and returning to an unconnected state.

During data transmission, if the same packet goes unacknowledged more than three times, the sender assumes the recipient has died. If the sender is the client, the client displays an error message and exits. If the sender is the server, the server displays an error message and returns to an unconnected state.

Closing a Connection

The client initiates a connection close by sending a packet with the CLOSE flag set. A four-way handshake is then done, with the following packet exchanges occurring:

1. Client -> Server; Flags: CLOSE; Payload: Nothing
2. Server -> Client; Flags: CLOSE | ACK; Payload: Nothing
3. Server -> Client; Flags: CLOSE; Payload: Nothing
4. Client -> Server; Flags: CLOSE | ACK; Payload: Nothing

Like the connection open process, these packets are subject to the same timeout and corruption checks that any other type of packet is subject to.

Handling Common Problems

Our RELDAT protocol's implementation can correctly account for the following issues caused by an unreliable connection:

Duplicate Packets

When a packet is sent multiple times, it is guaranteed to have its RETRANSMIT bit set. If a packet with the RETRANSMIT bit is encountered, the recipient checks its buffer to make sure it has not already received the packet by comparing sequence numbers. If the packet has already been received, it is acknowledged, but the packet's data is not stored in the buffer again.

Corrupted Packets

Each packet contains two MD5 checksums: a checksum for the header and a checksum for the payload. First, the checksum for the header is calculated, and then compared to the header checksum in the packet. If they are not identical, then the packet is ignored. Otherwise, the checksum for the payload is calculated, and then compared to the payload checksum in the packet. If they are not identical, the packet is ignored. Because the packets go unacknowledged, the sender believes they have been lost, and the sender re-sends them.

Lost Packets

A packet is determined to be lost in two situations:

1. If the sender sends a DATA packet and an ACK is not received for it within 1000 milliseconds.
2. If the recipient sends an ACK packet but receives its corresponding DATA packet (i.e., another packet with the same sequence number) again sometime in the future.

In the event of (1), the sender detects that the ACK was not received in a timely manner and assumes the packet was lost. It then re-sends the packet. In the event of (2), the recipient simply treats the re-received packet as a duplicate packet.

Re-Ordered Packets

Packets are not appended to the buffer, but indexed. A "standardizer" for the window begins at the sequence number 3, the next sequence number expected after the three-way connection open handshake. Each subsequent packet is indexed by subtracting this standardizing integer from its sequence number. Thus re-ordered packets are still indexed at the correct position in the buffer.

Limitations

- A packet does not necessarily have to have a payload. In this case, the minimum packet size is 45 bytes - the size of the header.
- The maximum amount of data that can be transmitted at any one time is 2,050,846,882,885 bytes =~ 2.051 terabytes. This value was calculated by taking the maximum payload size (955 bytes) times the maximum sequence number (2^31 - 1), for a total of (955 * ((2^31) - 1)) bytes. This limit will typically not be reached by the average computer user.