# CS3251 Project Proposal

Susanna Dong

04 November 2015
Version 1.0

# Table of Contents

# Defenestrating Transport Protocol (DTP)

## Description

The Defenestrating Transport Protocol (DTP) is a transport-layer protocol that provides reliable, connection-oriented, and byte-stream services to endpoint systems! DTP will act similarly to the Go-Back-N (GBN) protocol, with some modifications to make the protocol more efficient. DTP will also provide a 4-way handshake for connection establishment to provide better network security.

## How does it handle…?

- ***Lost packets?*** Lost packets are assumed to have never arrived to the receiver. The sender is not notified (i.e. sender does not receive a specific packet to indicate a lost packet) and will wait until a timeout to retransmit the packet.
- ***Corrupted packets?*** Packets (both regular or acknowledgement) are checked using the checksum algorithm. If a received packet is corrupted via the checksum algorithm, the packet is dropped. The sender is not notified and will wait until a timeout to retransmit. See the *Algorithms* section for a detailed description of the checksum algorithm.
- ***Duplicate or out-of-order packets?*** The receiver will check the sequence number of the packet vs. the sequence number it anticipates getting. If the packet is found to be a duplicate or an out-of-order packet (the sequence number does not match the expected number), the receiver will drop the packet and retransmit a cumulative acknowledgement of the last ACKed packet. The sender should respond by retransmitting the requested packet. In this current implementation, the protocol utilizes a GBN protocol, so out-of-order packets are not stored by the receiver. Therefore, the sender must retransmit packets that may have already been sent.
- ***Pipelining?*** Pipelining is handled by utilizing a sliding window protocol. This allows multiple packets within a “window” up to a maximum size to be sent at once, while only sending a limited number of packets at one time until a packet is ACKed.
- ***Window-based flow control?*** The protocol will utilize a modified GBN protocol for flow control. A maximum of *w* packets can be sent at one time before getting an acknowledgement. An ACK for one packet will allow the window of packets to shift. Packets that arrive out-of-order or are duplicates will result in sending a cumulative ACK.
- ***Bi-directional data transfers?*** Sender and receiver packets can simultaneously be data packets and acknowledgement packets. The packet will contain the segment number, as well as the acknowledgement number of the next packet the sender anticipates to receive. This can allow both sides to simultaneously send data and acknowledge received data.

- **Optimizations?** Because GBN may cause some unnecessary retransmitted packets, there may be some optimizations for the protocol. One example is fast retransmit - if 3 of the same ACKs are detected, the sender will transmit the packet before the timeout.

# DTP Packet Structure

**Maximum packet size (header + data)**: 1500 bytes
**Packet header size**: variable, minimum 20 bytes

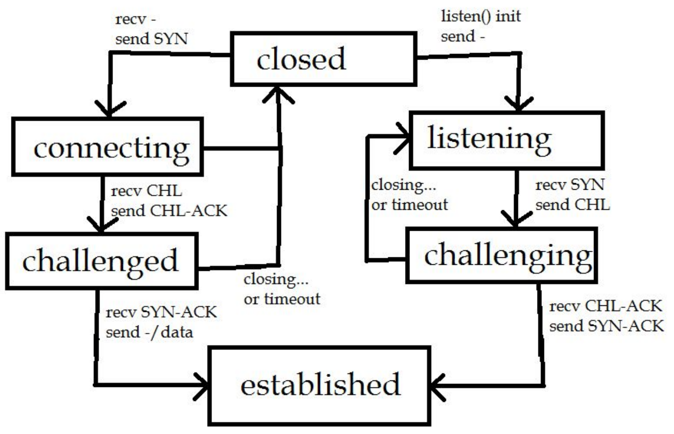
| | | |
|---|---|---|
| Source Port (16 bits) | | Destination Port (16 bits) |
| Sequence Number (32 bits) | | |
| Acknowledgement Number (32 bits) | | |
| Checksum (16 bits) | | Window Size (16 bits) |
| Flags (8 bits) | Header Length (8 bits) | Options (variable amount) |
| *Data (variable amount)* | | |

## Header Fields

- ***Source port and destination port***: the ports at which the packet is sent from and sent to respectively. server_port = client_port + 1 and therefore only one field is necessary to interface with NetEmu. However, for the sake of completeness, both fields will be kept.
- ***Sequence number***: the sequence number of the packet, valued in bytes.
- ***Acknowledgement number***: the acknowledgement number of the packet, valued in bytes. This indicates the next sequence number the receiver anticipates in receiving.
- ***Checksum***: a number calculated to see whether the packet is corrupted or not.
- ***Header length***: the length of the header. This allows flexibility for expanding the header.
- ***Window size***: the size of the sender's receiving window. Window size can range from 1 to 2^15. This allows the receiver to gauge the sender's receiving capabilities.
- ***Flags***: flags to raise. The current implementation will have the following flags:
    - (empty, reserved 4 bits for future implementations)
    - **SYN**: indicates that the packet is used for connection establishment.
    - **CHL**: indicates that the data is used for connection authentication.
    - **ACK**: indicates that the packet is acknowledging data.
    - **FIN**: indicates that the packet is used for connection shutdown.
- ***Options***: an options field, depending on the header length. For this implementation, the options field will not be used.
- ***Data***: the payload of the packet.

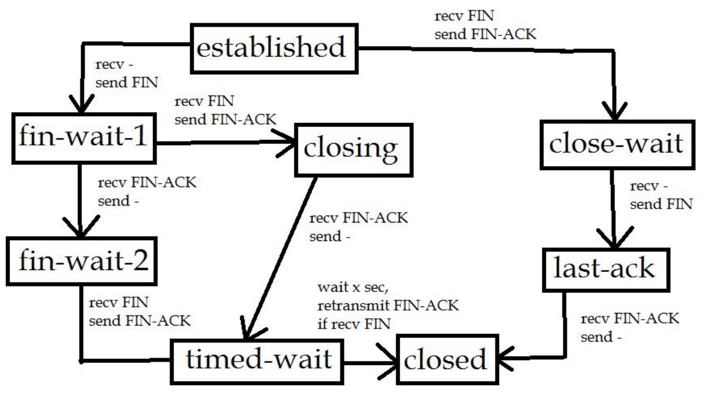
# Finite State Machine Diagrams

## DTP Connection Establishment FSM



Client wants to connect with server and sends a SYN. Assuming server is listening for incoming clients, server will create a challenge message, tag the message as CHL, and send it to the client. The client will answer the challenge (see *Algorithms* section for more details) and send the answer back as CHL-ACK. The server will compare the client's answer with the server's answer. If they match, the server sends an acknowledgement SYN-ACK and will deem the client worthy of transmitting data. Both server and client will have the connection established. If at any time, the client is unable to authenticate, the server will forget the client's state and send a packet acknowledging that the connection was not established.

## DTP Connection Shutdown FSM



The FSM for shutdown is the same procedure as TCP.

## DTP Send/Receive FSM

- Sender
    - If sending window size <= 0, do not send new packets and alert caller
    - Otherwise, for all packets in window that has not been sent yet:
        - Calculate sequence and acknowledgement numbers
        - Calculate checksum for packet and store at header
        - Send packet
        - If packet is first packet of the window, reset the timer
    - If timeout (fixed at 0.1 seconds for current implementation - do not want the user to start noticing a considerable delay due to timeout length if the network is poor, and the fast retransmit will also speed up the process)
        - Reset timeout and resend unACKed packets in window
    - If get an ACK:
        - Check for checksum; if not valid, drop packet
        - Keep track of ACK number; if ACK number >= 3, perform fast retransmit
        - Check for the cumulative ACK; ACK all packets in window up to the last cumulative ACK
        - Shift window by n = number of packets acknowledged
- Receiver
    - If receiving window size <= 0, do not receive new packets
    - Otherwise, if a packet has been received:
        - Check for valid sender; if not valid, drop packet and send cumulative ACK
        - Check for checksum; if not valid, drop packet and send cumulative ACK
        - Check for duplicate/out-of-order; if packet sequence number != expected sequence number, drop packet and send cumulative ACK
        - ackNumber += numBytesInPayload
        - If also sending data, perform sender calculations
        - Send cumulative ACK

# Algorithms

## Checksum Algorithm

The checksum will be calculated by the following:

1. Create a pseudo-header (consisting of the source IP address, destination IP address, and DTP packet length). Break down into 16-bit words and sum them together. Words that are less than 16 bits long will have 0's padded on the left. Call this value *pseudo_header_sum*.
2. Break down the DTP header (excluding checksum) and data into 16-bit words and sum the words together. Words that are less than 16 bits long will have 0's padded on the left. Call this value *dtp_packet_sum*.
3. Let *actual_checksum* = *pseudo_header_sum* + *dtp_packet_sum*.
4. Let *expected_checksum* = checksum from the DTP packet header.
5. If *actual_checksum* & (~*expected_checksum* + 1) = 0, then the packet is good. Otherwise, the packet is corrupted.

*Reference*: http://www.tcpipguide.com/free/t_TCPChecksumCalculationandtheTCPPseudoHeader-2.htm

## 4-Way Handshake

During connection establishment, the client must prove to be a trusted client wanting to connect.

The 4-way handshake will be as followed:

1. Client requests connection with the server by sending a SYN packet.
2. Server sends a challenge via a 32-character string of random character in response and sends the information in a CHL packet. The server will keep the challenge string and packet sequence number.
3. Client must send an answer back to the server in order to authenticate. The answer must consist of the MD5 hash of the concatenation of the client's IP address, the challenge packet's sequence number, and the challenge string. answer = md5(clientIP+sequenceNum+challenge)
4. Server must check if the challenge string is valid. The server will calculate its own answer by following Step 3 by its stored information for the client. If the answers match, the server will send a SYN-ACK.

# DTP API

## DTP Server Socket API

`DTPServerSocket(), DTPServerSocket(int portNumber)`
Creates a server socket. Can also provide a port number to bind to. Throws exception if an error occurs during creation.

`boolean window(int windowSize)`
Manually sets the window size for the server. Window size can range from 1 to 2^15. Default window size is 5. Invalid numbers will result in the method returning false, true otherwise. Current implementation will not let the maximum window size automatically resize for congestion control.

`void bind(int portNumber)`
Server socket binds to a given port number. If a socket already is binded to a port number by the constructor, an exception is thrown. Throws exception if an error occurs during binding.

`void listen()`
Server will begin to listen for incoming clients. If the socket’s listening buffer is full, an exception is thrown. Throws exception if an error occurs while attempting to listen.

`DTPSocket accept()`
Accepts a client. Will establish connection and return a socket that is already connected to the client socket. Will block if there are no clients currently pending to connect with the server. Throws exception if an error occurs while attempting to accept.

`void close()`
Closes the server socket. Throws exception if an error occurs attempting to close.

## DTP Client Socket API

`DTPSocket(int portNumber)`
Creates a client socket binded to the port number provided. Throws exception if an error occurs during creation.

`boolean window(int windowSize)`
Manually sets the window size for the client. Window size can range from 1 to 2^15. Default window size is 5. Returns false if the windowSize is invalid, true otherwise. Current implementation will not let the maximum window size automatically resize for congestion control.

`boolean connect(String destAddress, int destPort)`
Attempts to connect the client socket to the server socket, based on the IP address and port number. Blocks until accepted by the server it is attempting to connect to. Returns true if a connection is established with the server, false otherwise.

`int send(byte[] buffer)`
Sends a string of data from the buffer to the server. The data is guaranteed to be sent to the given IP address and port number exactly as presented. Returns the number of bytes copied from the buffer. Throws exception if an error occurs during sending.

`int recv(byte[] buffer)`
Receives a string of data to the buffer from the server. The data is guaranteed to be received from the given IP address and port number exactly as presented in the server. The method will block until data has been received. Returns the number of bytes copied to the buffer. Throws exception if an error occurs during receiving.

`close()`
Closes the socket. Throws exception if an error occurs during closing.