Yeonjoon Choi
Oleg Filatov

ATCP

Advanced TCP

November 4th, 2016

Overview

The protocol we are implementing is ran on top of UDP and resembles TCP in a lot of ways, but it also differs noticeably. It is Stop-and-Wait (i.e., **not** pipelined), utilizes sequence and ACK numbers, MD5 checksums, cumulative ACKs (which should help with reducing the number of ACKs needed to be sent overall and thus time), and supports bidirectional data transfer once the connection between parties is established.

Header Structure and Fields

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         |A|S|F|                                               |
|         |C|Y|I|              Padding                          |
|         |K|N|N|                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|            Window             |           Checksum            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                              data                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

```
Source Port:  16 bits
  The source port number.

Destination Port:  16 bits
  The destination port number.

Sequence Number:  32 bits
  The sequence number of the first data octet in this segment (except
  when SYN is present). If SYN is present the sequence number is the
  initial sequence number (ISN) and the first data octet is ISN+1.

Acknowledgment Number:  32 bits
  If the ACK control bit is set this field contains the value of the
  next sequence number the sender of the segment is expecting to
  receive.

Control Bits:  3 bits (from left to right):
    ACK:  Acknowledgment field significant
    SYN:  Synchronize sequence numbers
    FIN:  No more data from sender

  Padding:  29 bits
    Must be filled with zeros.

  Window:  16 bits
    The number of data packets (4 bytes each) beginning with the one indicated in the
    acknowledgment field which the sender is willing to
    accept in a single sliding window.

  Checksum:  16 bits
    Last 16 bits of the MD5 hash of the packet header (excluding the checksum field)
and the data.
```

API Description

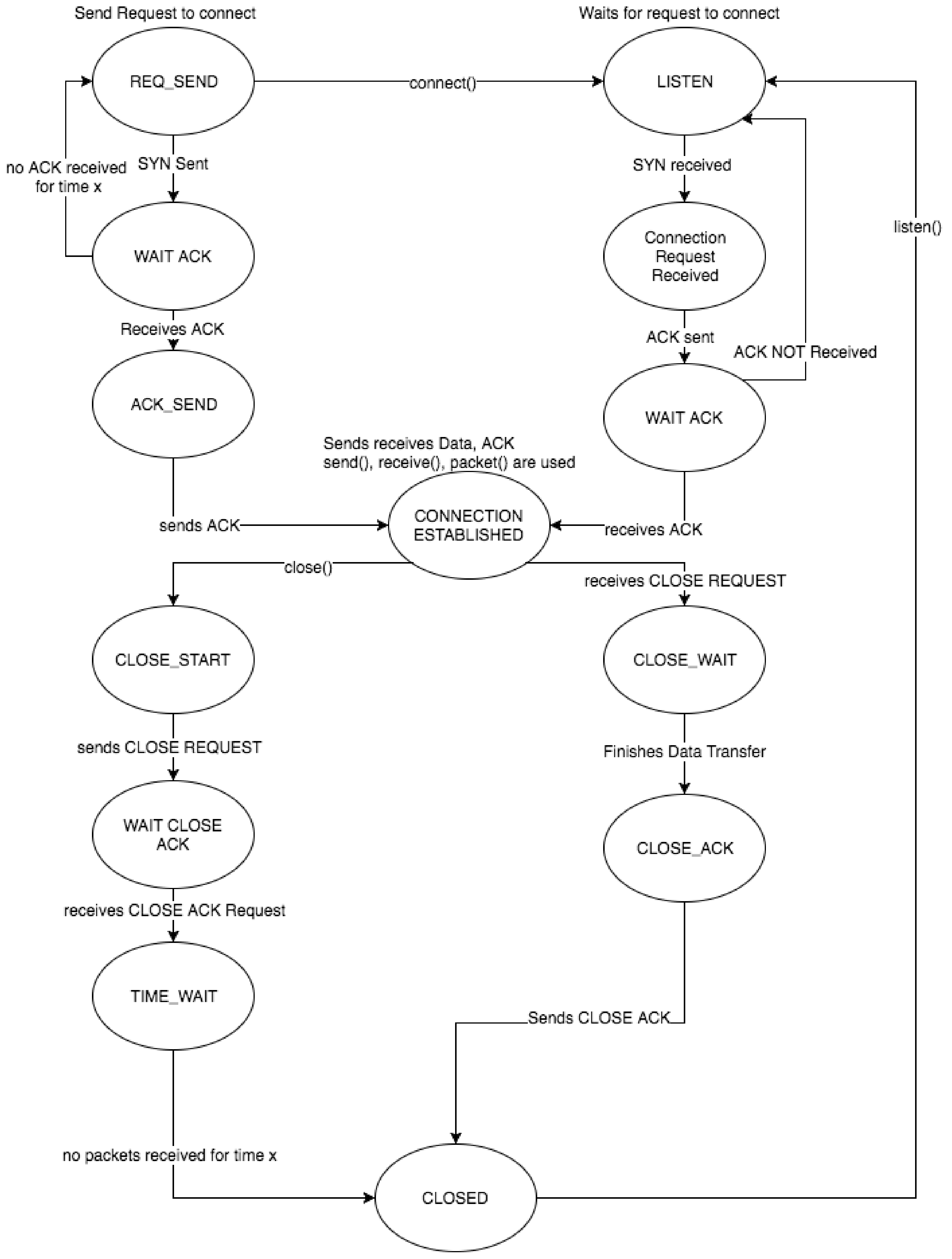
- **CRPsocket(port=8591, server=True, debug=False):** opens a socket; set **debug** flag to **True** in order to see debug messages in terminal; set **server** to **True** if socket is created by a server.
    - **.connect(server, port):** attempts a connection to a listening (server) socket, returns a **Connection** object instance. Supposed to be called by clients only.
    - **.accept():** waits for connections on an open socket, returns a **Connection()** object. Supposed to be called by servers only.
    - **.recv_window:** variable that indicates the sliding window size; intended to be modified by an application if needed.
- **Packet():** initializes a packet object.
    - **.from_bytes(b)**: forms a packet object from bytes **b**.
    - **.from_arguments(src_port, dst_port, syn=False, ack=False, fin=False, lst=False, seq_num=None, ack_num=None, window=None, data=0):** forms a packet from arguments.
- **Connection():** connection object used to send/receive data between two entities.
    - **.close():** closes the connection
    - **.recv():** checks the reception queue for incoming packets
    - **.send_data(data):** sends a bytes object **data**.
- Sample usage:
    - Client:
        - client = CRPSocket(port=8591, debug=debug)
        - conn = client.connect(server, port)
        - conn.send_data('POST'.encode('utf-8'))#sends POST command to the server

    - conn.send_data(fname.encode('utf-8'))#sends filename of the object to be sent
    - conn.send_data(data)#sends bytestream of the data
    - conn.close()#closes connection
  - Server:
    - server = CRPSocket(port=server_port, server=True, debug=debug)
    - conn = server.accept()
    - cmd = conn.recv().decode('utf-8')#receiving a command
    - fname = conn.recv().decode('utf-8')#after receiving the POST command, receives the filename
    - data = conn.recv()#receives data and performs necessary operations afterwards using the server application logic

Algorithmic descriptions for any non-trivial solutions

- Corrupted packets are detected using last 16 bits of MD5 checksum of the entire packet (minus the checksum field itself, of course). It has slight improvement over the way original TCP calculates the checksum, namely, the case when a packet is corrupted, but keeps the same bit count.
- Bidirectional data transfer allows for both parties to send each other data once the connection between them established without having to distinguish between server/client on the **Connection** object level.

Finite State Diagram



Error Resolutions

Lost Packets

- **Sender:** If no corresponding ACK packet is received from the receiver within a specified time interval, the sender resends the “lost” packet.
- **Receiver:** N/A

Duplicate Packets

- **Sender:** N/A
- **Receiver:** When a duplicate packet is received, which is detected using sequence number unique to that packet, it is ignored.

Incorrectly Ordered Packets

- **Sender:** Sends packets with sequence numbers inside the window range.
- **Receiver:** Uses sequence numbers to put packets in the correct order.

Corrupted Packets

- **Sender:** N/A
- **Receiver:** Corrupted packets (detected using checksum) are ignored, i.e., corresponding ACKs are not sent out, so that the sender is forced to behave the exact same way as if the packet was lost.

Extra Credit

Our team in planning to implement a bidirectional data transfer between client and the server, as well as sliding window that handles out of order packets, which is not trivial.