CS-3251-A
Homework 3: Reliable Data Transfer Protocol Design Report
October 29th, 2016
Shaohui Xu (902987595) shaohuixu13@gmail.com
DU Xinnan (903266560) xinnan@gatech.edu

GT-CRP

A Protocol for Reliable Communication

1. Overview

a. General

GT-CRP is a TCP like, pipelined reliable data transfer protocol that provide mechanism for handling lost, duplicated, out-of-order and corrupted packets. It also support bi-directional data transfer and non-trivial checksum.
GT-CRP is implemented using the best-effort connectionless services of UDP sockets.

b. Features

- **Is your protocol non-pipelined (such as Stop-and-Wait) or pipelined (such as Selective Repeat)?**
  For **GT-CRP**, we will apply a pipelined approach (Selective Repeat, more details can be found in later parts for lost and corrupted packets handling) based on fixed size packets. A double queuing system is utilized here between client and server to handle all the communication.
  The first queue is a **sending queue**, we will use this queue to send fixed size packets. We will set the packet size when we first start the **GT-CRP** connection. Thus during the whole process of data transfer, we will have a standard data denomination that helps better keep track of all the packets.
  The second queue is the **not-acked queue**, when the sender send a packet using the sending queue, we will also add the same packet to the not-acked queue and once we received ACK from receiver for this specific packet we can then remove it from the not-acked queue.

- **How does your protocol handle lost packets?**
  GT-CRP handles lost packets by using a selective repeat and fast retransmit approach that utilizes the double queueing system.
  When we send any data from client(sender) to server(receiver), we will pass a sequence number at the same time. When the receiver received the packet, it will send a packet with ACK flag set to 1 and acknowledge number set to the sequence number + 1. When the receiver receives an ACK for a particular sent packet, the packets is then deleted from the not-acked queue, if the packet is the smallest unacked packet, move the window forward to the next unacked queue number. To handle packets loss, we define a timeout for each packet, which is 2-second initially and is subject to later change. If the receiver doesn't get any response associated with the packet from the server, the sender will simply retransmit it. The timeout time can be changed dynamically depends on the network. For our implementation, we use similar technique as TCP does.

    Timeout = EstimatedRTT + 4*DevRTT

  Where EstimatedRTT is defined as:

EstimatedRTT = (1-α)*previous EstimatedRTT + α*SampleRTT

We use alpha = 0.1 in CRP as we assume the network relatively stable.

The DevRTT provides a safety margin for timeout interval, as we want the interval be large if the network is less stable, the definition of DevRTT is:

DevRTT = (1 - beta)*DevRTT + beta*|SampleRTT - EstimatedRTT|

We will use the typical value 0.25 for beta.

However, timeout is usually long and may cause inefficiency if there are a lot of duplicate ACKs, thus we use fast retransmit to accelerate data transmission. The mechanism of fast retransmit is simple, we simply retransmit a packet if we receive 3 ACKs with Acknowledge number equal to the packet's sequence number. Thus for most of the time, we don't need to wait until timeout.

Another thing we need to deal with packets loss is flow control. Our flow control algorithm consist of two parts

1. Receiver side: we will set the receiver buffer size to 1KB in the beginning and dynamically adjust the buffer size until it reaches 4KB. When the buffer size is smaller than 4KB, whenever the buffer is full, we will double the buffer size. When the buffer size is its maximum, we will send a packet with value k to notify the sender to shrink its window size.
   k = max receive window size - (largest sequence number in the receive window - smallest sequence number in the receive window)

   This is not simply the amount of space left in the buffer as it prioritizes getting packets in the correct order. We follow the following logic to promote correct packet order receipt in the receive buffer: we constantly check the value of K, if K is less or equal to 1, the sender will stop sending, shrink the window size to half, and wait for 200ms.

2. The Sender first set the window size to a relatively small value and increases gradually. When the sender receives the packet indicating that the receiver buffer is full, the sender just shrink the window size to half. The sender may repeat similar operation over and over again.

- **How does your protocol handle corrupted packets?**

  GT-CRP uses **Fletcher-16** checksum to handle corrupted packets. The checksum is calculated before the packet is sent, then it's encoded in the CRP header. When the receiver receive the packet, it will also runs the same Fletcher-16 checksum algorithm on the received packet. If the calculated value matches the one in the packet's header, it means the packet is intact, if not, we will send a **NACK** to notify the sender to move the packet from Not-ack-queue to sending queue and retransmit it.

  More details on **Fletcher-16** can be found in later sections.

- **How does your protocol handle duplicate packets?**
  The CRP protocol handles duplicate packets using sequence number. If sender received two or more packets with the same sequence number, it just keep the earliest one and drop all the later ones.

- **How does your protocol handle out-of-order packets?**
  On the receiver side, we resemble packets into order to handle out-of-order or lost packets.
  We will implement the buffer on the receiver side using a queue. We fill up this buffer using data from packets in the order of corresponding sequence numbers, the max size of the queue equals the window size. We keep poping the received packets until we see the first gap in sequence number when we receive a packet. We also have a variable ‘s’ to keep track of starting sequence number in the queue to make sure we only receive packets with sequence number larger than s and smaller than s+window size.

- **How does your protocol provide bi-directional data transfers?**
  GT-CRP uses the strategy of **piggybacking** to handle bi-directional data transfers. We combine data packets and ACK packets together. Hence we can easily switch the role of sender and receiver during data transfer without adding more queues.
  Whenever we receive a packet, receiver will validate that with checksum, and then about to send an ACK. There are two possible cases at this state:
  1. There are packets in sending queue with ACK sets to 0, then we piggyback this ACK to that packet and deliver it in next window when current data packets about to go out.
  2. There’s no packets in sending queue with ACK sets to 0, then we directly send out an ACK packet.

- **Does your protocol use any non-trivial checksum algorithm (i.e., anything more sophisticated than the IP checksum)?**
  Other than the trivial IP checksum algorithm, **GT-CRP** implements **Fletcher-16** checksum algorithm.
  The Fletcher-16 checksum is the extension of simple checksum. The simple checksum is just sum up all the short blocks’(size n) value, then calculate the modulus of the sum. For example, if the block size is 16, what we will do is adding ap all the 16-bits values and divide it by 65526 and keeping only the remainder. However, the simple checksum is not sensitive to the order of blocks and the universe of its value is not enough, it’s possible that two different packets hold the same checksum. Thus, we need to compute the second value when computing the simple checksum. For each block of the data word, taken in sequence, the block's value is added to the first sum and the new value of the first sum is then added to the second sum. By adapting this method, the

checksum becomes sequence sensitive and the universe of possible checksum values is now the square of the value for the simple checksum. The Fletcher-16 algorithm use block size of 8 and combine the two sum together, namely, the second sum is multiplied by 255 and added to the simple checksum.

- Does your protocol have any other special features for which you request extra credit? Explain.
    - **GT-CRP** will include a **Dynamic Buffer** to better improve the space utilization: Dynamic buffer size to handle flow control, max 4K. Initially set to 1K at the beginning of the connection,we double the size of the buffer when the buffer is full. If the buffer reaches its maximum, it will be the same as TCP's flow control.
    - **PiggyBacking** for bidirectional data transfer.
    - **Fletcher-16** checksum algorithm.
    - Better handling of connection: It's possible that during a connection, one of the hosts accidentally shut down, then the sender or receiver will repeatedly send packets to the other side. To handle this case, we add a timeout for the whole connection, the host starts the timer every time it receives response from another side, when the timer timeout, the host will automatically close the connection.

2. Detailed Specification

a. a high-level description of how CRP works and of any special features you have designed

- **Connection Establishment**

  Connection establishment is very similar to TCP's 3-handshake procedure. The sender first initiate connection, sends packet A with SYN bit set to 1 and sequence number x. Then the receiver respond with a packet B whose ACK bit and SYN bit equals 1, acknowledge number set to x+1 and sequence number set to y. The maximum buffer size (4KB) is encoded in the 32-bit data field of B. Then, upon receiving the packet, sender sends receiver a packet C with ACK bit = 1 and ACK num = y+1. We will send the first datagram along with the fixed packet size in the data field of C. Then the sender and receiver established connection.

- **Data Transmitting:**

  Sender:

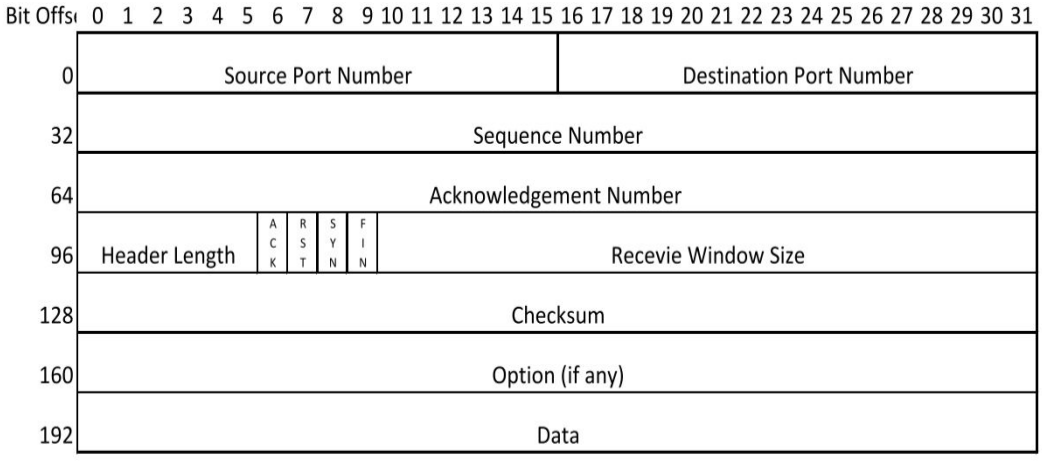
  Whenever the sender receives data from application above, it pushes it to the sending queue. At the very beginning of data transfer, the sender will send n (window size) packets and pushes them to the not-ACKed queue. If the timer timeout or we received a NACK from the receiver, CRP will push that packet to the front of the sending queue and retransmit it. If an ACK with acknowledge number k is received, CRP will pop packets whose sequence number is smaller or equal to k from the not-ACKed

queue and push packets from the sending queue until the not-ACKed queue is full.

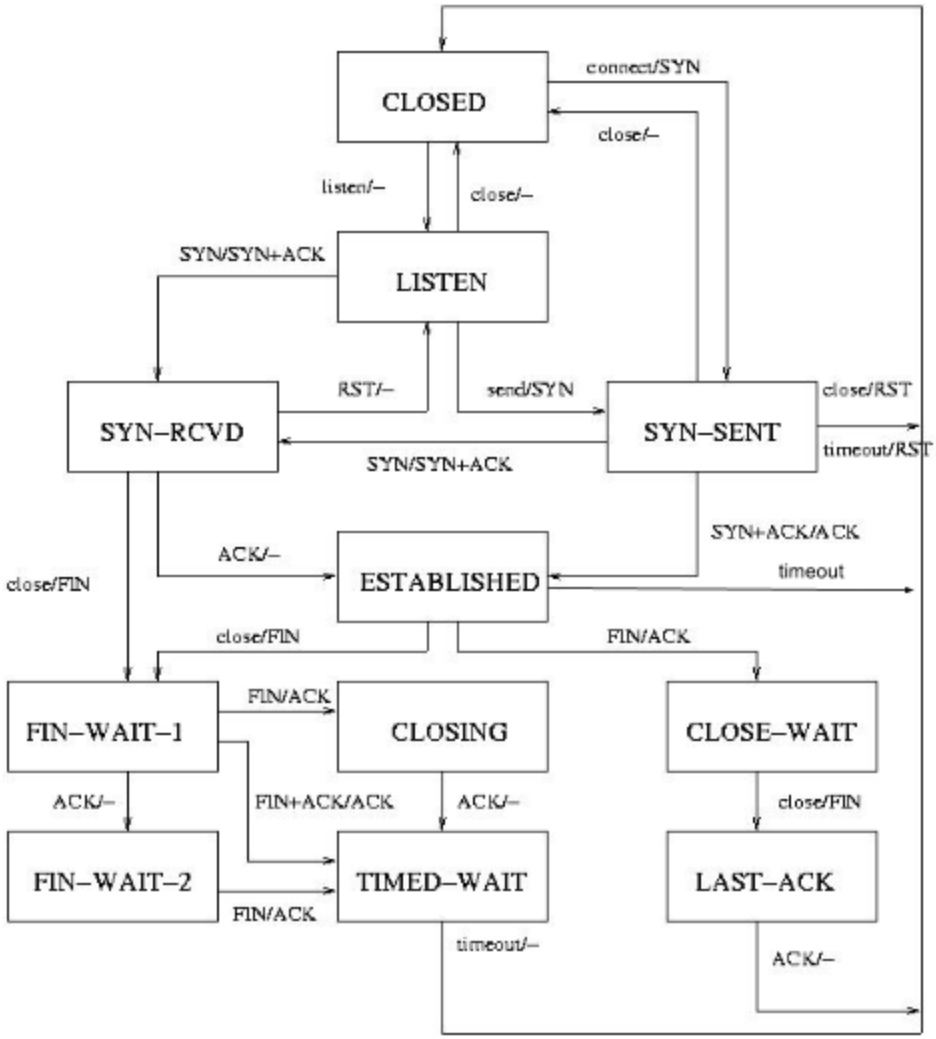
Receiver:
When the packet arrives, we first use Fletcher 16 checksum to detect broken packets and send a NACK if broken packet is found.
If the receiver receives a in-order segment with expected sequence number, and all packets up to that sequence number have been ACKed, the CRP receiver wait up to 200ms(500ms for TCP) for next packets, if no segment, send ACK.
If the receiver receives an in-order packets with expected sequence number and there is a packets waiting for sending ack, the receiver will immediately send ACK with Acknowledge number equal to the sequence number + 1.
If an out-of-order packet (sequence number higher than what we expected), immediately send **duplicate** ACKs (as TCP does) with Acknowledge number equal to the expected sequence number.
When there is an arrival of a packet that fills the lower end of the gap, send ACK immediately.
Both Host can act as senders and receivers because we use piggybacking two-way data transmission.

- **Connection Termination:**
  CRP handles close of connection in the same way as TCP does. In our CRP protocol, there is another scenario which will lead to connection termination. One host will close the connection if it waits enough time and can't get any response from another side.

b. Header Structure and header Fields



c. **Finite State-machine diagrams for the two CRP end-points**



d. **A formal description of your API (the functions it exports to the application layer)**
   - **Server**
     - setupServer(port)
       - Set up **GT-CRP** server with port number
     - sendPacket(data)
       - Send data to the other endpoint
     - readData()
       - Read data sending from the other endpoint
     - close()
       - Close the connection
   - **Client**
     - connectTo(clientPort, serverIP, serverPort)

        - Connect to **GT-CRP** server at serverIP:serverPort that has already been set up
        - Bind the client socket to self port
    - sendPacket(data)
        - Send data to the other endpoint
    - readData()
        - Read data sending from the other endpoint
    - close()
        - Close the connection

e. **Algorithmic descriptions for any non-trivial algorithms in CRP**

The sample c code of the **Fletcher 16** checksum is as follows:

```
uint16_t Fletcher16( uint8_t *data, int count ){
    uint16_t sum1 = 0;
    uint16_t sum2 = 0;
    int index;

    for( index = 0; index < count; ++index ){
        sum1 = (sum1 + data[index]) % 255;
        sum2 = (sum2 + sum1) % 255;
    }

    return (sum2 << 8) | sum1;
    }
```