**Logo, company name

Description automatically generated**

**School of Electrical Engineering and Computer Science**

**EE-353 Computer Networks**

**Assignment Two: Reliable Net**

**Instructor: Arsalan Ahmad**

**Submitted on 17th April 2022**

**BESE 11A, Spring 2022 Semester**

|  |  |
| --- | --- |
| Group Members | CMS ID |
| Ayesha Naime | 356760 |
| Fatima Hussain | 350328 |

BadNet is an unreliable channel for data transfer. To ensure Reliable Data Transfer, the **stop-and-wait protocol** was implemented.

**Badnet0.0: Reliable Transfer**

Is already reliable, therefore the client only has to send the data and server only had to receive the data.

**Badnet1.0 Loss Handling:**

This Badnet1.0 drops every 5th packet. Therefore, two concepts were introduced: **Timer** on the sender side and **ACK** from the Server side. The basic concept is that after sending a packet, the client socket will start a countdown. If the ACK for the sent packet is received within the countdown, it will transmit the next packet. If not, client will retransmit the same packet i.e., for which no ACK was received.

On the server side, if the ack is lost, then the client will simply assume that the packet was lost and retransmit.

**Implementation:**

* A Boolean variable **retransmission** is used to decide whether the current iteration of the while loop is transmission or retransmission.

Retransmission = False -> Transmission: New data will be extracted from the file, new packet will be created using makepkt(data, pktCount) and sent.

Retransmission = True -> Retransmission: No new data will be read, instead the last created packet will be resent.

* **Socket.settime(waitingTime)** is used to do the countdown. clientSocket.settime(2) is executed right before the **clientSocket.rcvfrom(bufferSize)** in the **try block**.
* **socket.timeout error** is caught by the **except clause** when no ACK is received, and retransmission is set to True. However, if the ack is received within the countdown, retransmission is set to False.

**Badnet2.0: Error handling:**

To handle corruption, **checksum field** was introduced in packet and ACKs. On the client/sender side checksum is calculated based on the data, inserted into the packet, and sent to the server/receiver. At the server side, it extracts the data and recalculates the checksum. If the received checksum and recalculated checksum are identical, correct ACK will be sent. If not, the previous ACK is sent, indicating there was some issue in the received packet. On the server-side checksum is calculated based on the sequence# included in ACK. When the client receives ACK, it recalculates checksum based on the ACK sequence# which is compared with the received checksum. If they are identical, correct ACK was received and packet was correctly transmitted. If not, client assumes there was an issue with the packet and retransmits the packet.

**Implementation:**

* function **corrupt(rcvpkt)** takes the received packet as argument. It extracts checksum and data from it. Using the data, it calculates the checksum and compares it with the received checksum. Returns **false** if packet is not corrupt and **true** otherwise.
* If corruption is discovered, **retransmission** is set to true so that packet is retransmitted in the next iteration instead of new packet transmission. If not, **retransmission** will remain false, and new packet will be generated and transmitted.

**Badnet3.0: Duplication handling:**

To handle duplication, **sequence number** are introduced in the packet and ACKs so that server can send the correct ACK to the client. When the server is waiting for packet of sequence#1 and the client duplicates the pkt#1, server will receive the first packet, send ack1 and start waiting for pkt of seq#0. However due to duplication, it will receive pkt1 again. In this case it will resend the last generated ack, which in this case is ack1. This ack1 is sent to stop the client from sending pkt1 and send the next packet, pkt0, instead. When the client finally sends pkt0, server will generate a new ack0 and send it to the client.

On the other side, when the server sends the ack, client will see if it is the correct seq#. If it is correct, then the next packet will be transmitted. If incorrect, the client will keep receiving until the countdown finishes, after which it will retransmit the packet.

**Implementation:**

* Integer variable **count** is used on the client side represents which packet is being sent. **Sequence number** can be extracted from count by using **count%2**
* **Integer variable waitingfor** is used on the server side to remember which pkt sequence number it is expecting from the client. Its value is alternates between 0 and 1.
* **hasCorrectSeqNo(rcvpkt, expectedSeqNo**) is used on the server side to check whether the received packet has the same seq# that we were expecting.
* If correct pkt is received, the **makepkt(sequenceNo)** generates the new ACKand sends it.
* On the client side, **isCorrectACK(ACKpkt, expectedSeqNo**) is used check whether the correct ack is received. This function will keep running in a **while loop**, until ACK of the correct seq# is received or the countdown finishes.

**Badnet4.0: Out of order handling**

Since we are implementing stop and wait, there will be no reordering as the client will not send the packet unless the previous one was received by the server and its ACK was received by the client.

**Implementation:**

When pkt5 is buffered instead of transmitted by **Badnet.transmit()**, the countdown will start and since the pkt was not sent, no ack will be received, causing the countdown to finish. Retransmission variable is set to true and in the next iteration, pkt5 is sent twice. This will be handled by the server as duplication.

**Server/receiver FSM:**

Diagram

Description automatically generated

**Client/sender FSM:**

Diagram

Description automatically generated