****

**De La Salle University • College of Computer Studies**

**Machine Problem #2**

**A UDP-Based Reliable File Transfer**

Final Documentation Presented to

Ms. Franchesca Laguna

In Partial Fulfillment of

the Requirements for

NETWORK – S19

2nd term Academic Year 2014-2015

By:

Amadora, Angelo John

Aquino, Kurt Neil

Choy, Matthew Seaver

Tapales, Carlo Gabriel

December 5, 2014

**Milestone 1: A Stop and Wait Reliable Data Transfer (40%)**

In a stop and wait reliable data transfer, the sender sends one packet at a time. It then waits for an acknowledgement from the receiver before sending the next packet. The packets received are then assumed to be in order.

*Successful File Transfer at 0% loss. Make sure you terminate the file transmission. (10%)*

* A successful file transfer for a stop – and – wait data transfer is when there is not a single loss and also not a single packet was asked by the receiver to be resent. This is successful because given the smallest possible timeout window given to the receiver to wait for packets (for example purposes, let the smallest possible time to be 1 second) is 1 second, at most every packet must be sent within 1 second. It is also possible that the packet is sent before the 1 second mark, meaning that it is fast if there are no packet losses that have occurred during the data transfer.

*Implementation of sequence and acknowledgement numbers. (10%)*

* The implementation of stop – and – wait reliable data transfer in terms of sequence and acknowledgement numbers are as follows. First of all, the program reads the file that the user wants to send and converts it into bytes in order to form packets that are allowed to be sent. Once the conversion is done, we create a “sendPacket” thread (which can be found in the “sendPacket.java” file. which is like sending a packet itself. Once the thread is finished sending the bytes it returns an acknowledgement telling the sender that this thread is finished.

*Implementation of packet loss and timeout at both sender and receiver. (20%)*

* In stop – and – wait, the implementation of packet loss for the receiver part is when the receiver is waiting for the packet (which the thread is sending) but, not actually receiving anything, meaning its stuck waiting for a packet that might never come. In order to fix this, we have set a timeout or a length of time that the receiver will wait for the packet.

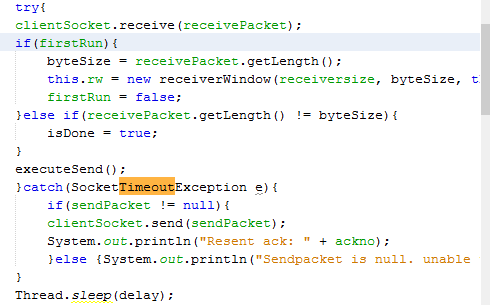


Figure 1. Timeout implementation

* The image above shows the concept of the timeout. The moment that it has reached that timeout exception found inside the try catch statement, meaning that the packet was lost, the sender must resend/rerun the thread containing the packet which was lost.
* The timeout may differ for each computer and it also may be set to different values depending on how long the user wishes to wait for the packet, this directly affects the sending time of the file depending on how long the user would wish to wait for each packet since the longer the window for waiting for the packet the longer the time it takes to wait for the rest of the packets sequentially. The sender also sends the packet one by one until the receiver tells it to resend the latest packet which is the one that was lost.

**Milestone 2: Implementation of Go-Back-N (85%)**

As compared to the stop and wait protocol, Go-Back-N is an example of a pipeline protocol which allows multiple packets to be sent at the same time. A sender window is then added to the system, hence adding more complexity to our reliable data transfer.

*The sender sends multiple packets without waiting for an acknowledgement. The receiver only accepts in-order packet. If a packet received is out of order it will discard that packet. (15%)*

* Prior to the implementation of the Receiver Window, the receiver only accepts packets in the correct order of sequence numbers, otherwise it discards the packet immediately. For every discarded packet, the same packet will again be sent until an acknowledgement has been received by the sender. Once the receiver receives all of the packets successfully, it then directly stores the packets locally before it finally writes to the file itself.

*The timer is implemented for the last unacknowledged packet (10%)*

* The timer is implemented by input, which means that it is set by the user. The timer is reset the moment the packet to be removed from the list of packets reaches the index 0. The moment the timer reaches it designated countdown it will resend all the packets in the group also known as all the unacknowledged packets.

*The sender processes cumulative acknowledgements. This can be seen when acknowledgements are lost. (10%)*

* Whenever a packet is being sent, a thread is made, and the amount of running packets increases. Before the next packet is sent, the program first checks if the latest acknowledgement number received is greater than the acknowledgement of the current thread. This is implemented in the checkOthers() and stopIf() methods, in which it checks the acknowledgment numbers of the previous packets sent. The acknowledgements for each packet changes whenever the expected sequence number of the packet is reached. For example if the expected sequence number is 39 and the sequence number of the sent packet is also 39, the new acknowledgment number to be sent will be 40, afterwards the program will then be notified that the latest packet is done sending and the thread should now be stopped. Otherwise the program will just send the previous acknowledgement number and the current packet will be discarded.

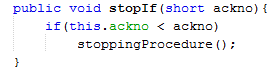
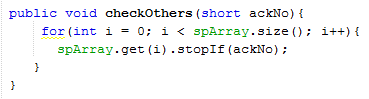


Figure 2. checkOthers() & stopIf() Functions

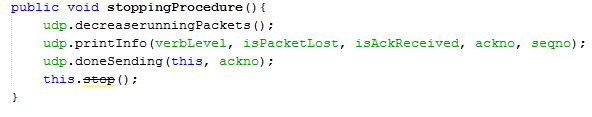


Figure 3. stoppingProcedure() Function (stops the thread)

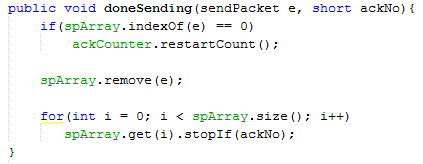


Figure 4. doneSending() Function

**Milestone 3: Implementation of TCP Reliable Data Transfer (105%)**

TCP is a hybrid of Go-Back-N and Selective Repeat. Hence, a receiver window is added to the system. In this case, the received packets are not necessarily in order anymore as long as the receiver window can still accommodate them.

*Implementation of the receiver window. When a packet is received, the receiver buffers it to the receiver window but sends an acknowledgment for the last in-order packet. (15%)*

* Before accessing the Receiver Window, its initial size must first be set in order for the program to determine the maximum number of packets it can accommodate. Regardless of the sequence number of the packet being received, it first checks if the corresponding packet is invalid or to be ignored, i.e. if it already exists or if it’s sequence number is less than the expected sequence number, or if it is valid, meaning that the sequence number of the packet is in order or if it is the start of a new file. In order for the program to determine which action it should perform (sending the acknowledgment or discarding the packet received), an addArray() function has been implemented to check all received packets in the Receiver Window prior to adding it to the list of packets to be appended to the file itself.

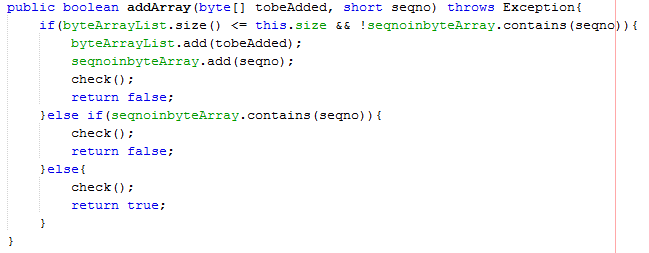
****

Figure 5. addArray() Function

*Implementation of Flow Control (5%)*

* Before accessing the Receiver Window, its initial size must first be set in order for the program to determine the maximum number of packets it can accommodate. Since the receiver window receives packets regardless of their order, a check() function has been implemented into the program. Every time the Receiver Window receives a packet, it first checks if the received sequence number of the packet is identical to the expected order number. After checking the corresponding sequence number, it then discards the packet if either there is already an existing sequence number saved in the ArrayList (sequence numbers saved for checking purposes) to avoid saving duplicate packets, or if the ArrayList is already full meaning the Receiver Window is overloaded. It only saves the packet into the ArrayList of Byte Arrays (packets to be appended into the actual file) if its sequence number matches the current order of the packets already received.

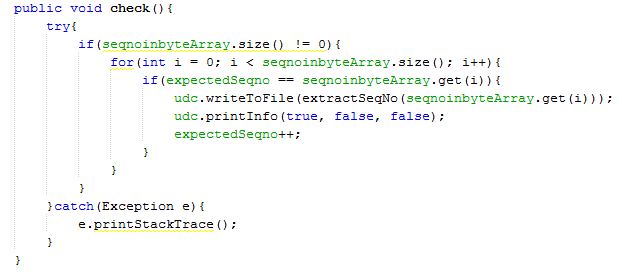


Figure 6. check() Function