# CS 505 Project 2

## Short Overview:

The project is about implementation of the *ISIS total order multicast algorithm* to achieve a reliable total order delivery when a process multicast a message to a closed group of processes. In this algorithm the sender unicast a message to all the processes and also delivers its own message too. Upon receiving a data message the process sends ACK for the received message along with a proposed sequence number for the data message. It puts the data message along with the assigned proposed sequence number and wait for the final sequence number from the sender to deliver this message to the application. When the sender of a message receives ACK from all the processes in the group, it computes the maximum of all the proposed sequence number and sends that to all the processes as the final sequence number for the data message. Each process maintains a ready queue before delivering a message to the application. This ready queue has all the messages stored which are ordered based on the proposed sequence number of the message and sender's id in case of ties. All the messages from the top of this queue for which final sequence message has been received and is marked deliverable are delivered to the application in increasing order of sequence number, until a message which cannot be delivered is encountered. All the process will still remain alive after they have sent and received all the messages in the system.

## System Architecture:

The system consists of multiple process in distributed environment. All the processes can send and received the messages. There is a file named ***"hostfile.txt"*** which is shared among all the processes to know about each other. The hostfile contains the hostname of all the machines participating in the algorithm. The communication is duplex for each process and they have their own sender side and receiver side. The processes never terminates even after sending and receiving all the messages in the system. We are using UDP as the transport layer protocol and have implemented the ACK mechanism to make it reliable.

The sender side of the process unicast a data message to all the other process in the system. After sending the messages it stores this data message in its own queue with a proposed sequence number for this message. The sender also sets an alarm for 1 second as an ACK timer for its data message. After that it keeps waiting for all the ACK's to be received from the other processes. All the processes are either configured to send multiple data messages or none at all. Upon receiving a message the receiver side checks the type of the message and make the decisions as described below:

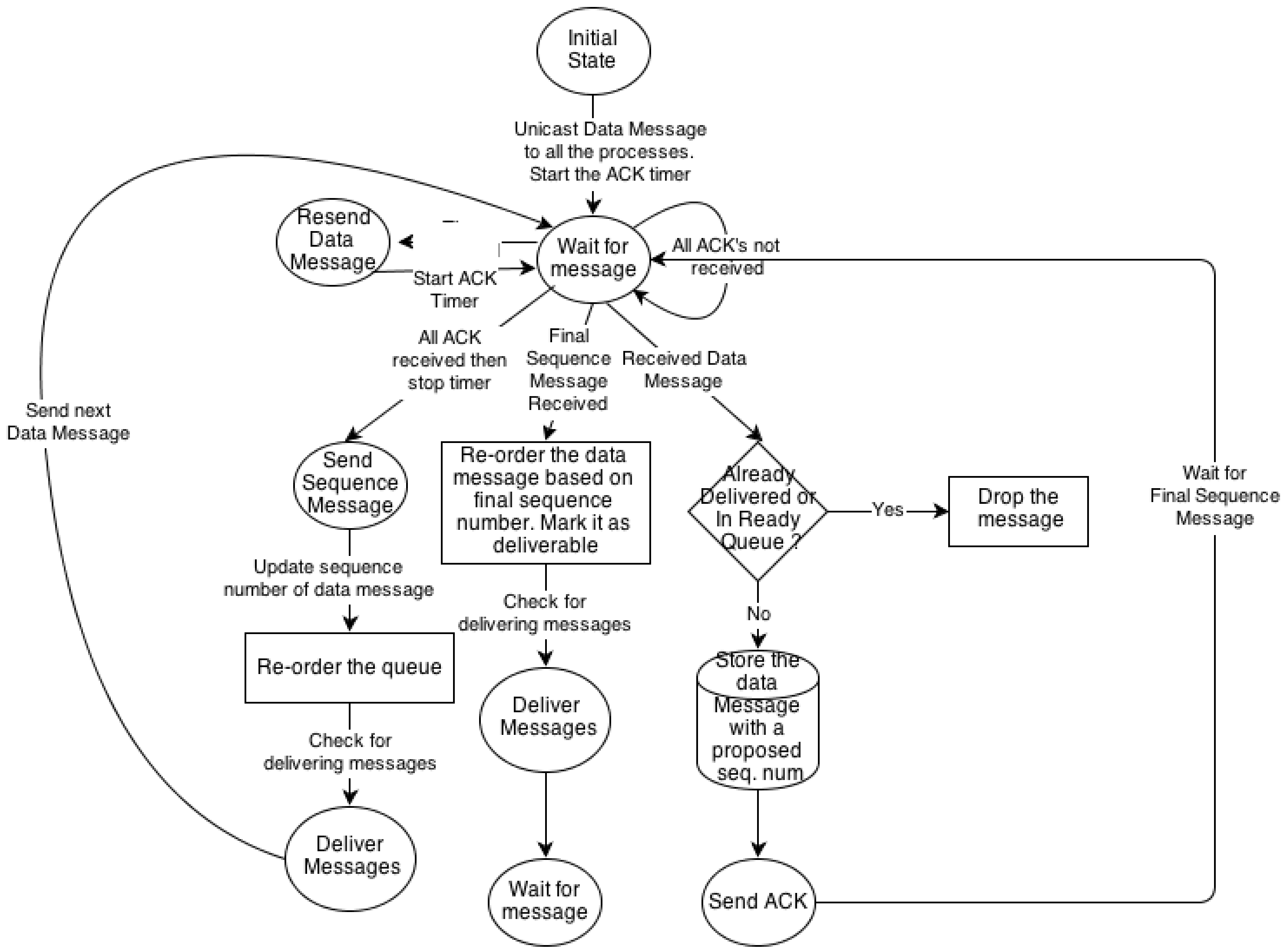
**a) ACK message:** If it's an ACK message then the receiver stores the sequence number received for its data message and check if all the ACK's has been received or not. If all the ACK's are not yet received it again waits for rest of the ACK. It removes the process ID from its expected ACK list so that upon ACK timer expiry the sender side only re-sends the data message to those processes from whom ACK is not received. If all the ACK's have been received then it calculate the max of the all received sequence number for its data message including its own and sends a sequence message with this final sequence number for its data message. It also updates its queue with this final sequence number and checks if there are some messages that can be delivered or not. Then it breaks from the receive loop and sends the next data message.

**b) Data Message:** If it's an Data Message then the receiver checks if the received data message has already been delivered or not. It also checks if the received data message is there in the ready queue or not If the received message is not found in any of the queue, then it stores the data message in its ready queue with a proposed sequence number. It sends an ACK message to the sender with this proposed sequence number too.

**c) Sequence Message:** If it's an sequence message then the receiver updates the data message sequence number for which sequence message has been received. It then re-orders the queue based on the new sequence number and checks if anything can be delivered from the ready queue. Then it again waits for receiving more messages.

## State Diagram:

The state diagram for the algorithm is as below:



Initial State
Unicast Data Message to all the processes. Start the ACK timer
Wait for message
Resend Data Message
Start ACK Timer
All ACK's not received
All ACK received then stop timer
Final Sequence Message Received
Received Data Message
Send next Data Message
Send Sequence Message
Update sequence number of data message
Re-order the queue
Check for delivering messages
Deliver Messages
Re-order the data message based on final sequence number. Mark it as deliverable
Check for delivering messages
Deliver Messages
Wait for message
Already Delivered or In Ready Queue ?
Yes
Drop the message
No
Store the data Message with a proposed seq. num
Send ACK
Wait for Final Sequence Message

## Design Decisions:

The program accepts command line arguments to configure the listening port specified by “-p” option, “-h” to specify the location of the hostfile and “-c” to specify the count of the data message a process needs to send.

Since its a duplex communication there is receiver and sender side of each process. I have made 4 structures such as: ***generalSystemInfo, senderInfo, receiverInfo and msgInfo.***

In structure ***generalSystemInfo*** I am storing the general info which will be used by both the sender and receiver state of a process. It contains information such as a map for hostname to ip address, ip address to id of the process. It keeps track of all the configurable parameters value.It also stores the information about the process hostname, last proposed sequence number, last proposed final sequence number and ID.

In structure ***senderInfo*** I am storing the information related to the sender side of the process. It keeps the sender socket, set containing the ID’s of the processes from which its expecting ACK for a data message. Data Message number which the sender has sent till now.

In structure ***receiverInfo*** I am storing the information related to the received side of the process. It keeps the receiver socket information, listening port number, a map ***(priorityToMsgList)*** which keeps all the data messages that needs to be delivered and a map to store information regarding all the delivered messages. The ***priorityToMsgList*** keeps all the messages sorted based on assigned sequence number and sender id of the data message in case of ties.

In the structure ***msgInfo*** I am keeping the received data message, sequence number corresponding to this message and a flag ***isDeliverable*** to keep track if it can be delivered or not. This structure is stored as a value for all the entries in the map ***priorityToMsgList.***

To keep the code modular I have written few methods whose functionalities are re-used. Functions like ***sendAckMessage***: To send an ack message after receiving a data message. I am sending the ack message only when I receive a data message for the first time.

***verifyAckMessage***: This function verifies the ack received by a process. If a valid ack is received then it updates the expected ack list in sender side. If an invalid ack is received then it just discards the message.

***handleTimerExpiry***:  This function helps to send the data messages to those processes from whom ACK has not been received. It is a callback function which is called when alarm is expired and SIGALARM interrupt is raised. After sending the data messages it again sets the alarm for 1 second.

***sendFinalSequenceNumberMsg***: This function helps to send the final sequence message to all the processes once all the ACK’s has been received for a data message. It also updates the sequence number for the message stored in the sending process queue.

***checkAndDeliverMsg***: This function check the ***priorityToMsgList*** and delivers the data messages which can be delivered. It also prints on the stdout the process id delivering the message, message id, sender id and the final sequence number of the delivered message.

***convertByteOrder, convertAckByteOrder, convertSeqByteOrder***: These functions are used to convert the message data’s from network to host byte order and vice-versa.

***checkIfDelivered***: This function checks if the received data message is already delivered to the application or not.

***checkIfQueued:*** This function checks if the received data message is already there in the ready queue or not.

***storeMessage***: This function helps to store the data message in the priorityToMsgList.

There are few other utility functions too which helps in verifying the command line parameters entered by the user. And few are used to print the message fields.

## Implementation Issues and Assumptions:

I am assuming few things which might cause some issues. The assumption are as follows:

a) There will no duplicate ACK’s received.

b) I have kept the retry timer or the ACK timer as 5 sec. It might be needed to increase the ACK timer in some scenarios cause small ACK timer will result in interrupt and control will take priority to re-send the messages rather than receiving any at that time. Due to this sometimes none of the process receives an ACK.

c) In data message I am passing constant value “1” as data.

d) I am using two flags “errorDebug” and “paramDebug” to print the debug messages. By default they will be turned off and the only output on stdout will be for delivered messages. In few error scenarios like a socket is not opened or a malloc is failed then I am printing the error message and exiting the application.

## References:

http://pages.cs.wisc.edu/~bart/739/papers/isis2.pdf