Overview

Reliable file transfer mechanism traditionally involves hand shaking procedures. It inherently runs over Transmission Control Protocol (TCP). TCP has a lot of overhead processing because of the handshaking procedures and congestion control. Congestion control creates a number of issues because it needs to get acknowledgement for every packet sent and it does not increase the window size unless it is sure that all the packets in that window have been acknowledged. In order to achieve faster transmission and reception, we need to remove most of the overhead processing and send packets quickly and efficiently. For doing so, we went ahead with User Datagram Protocol (UDP).

UDP has simple transmission model with minimal overhead processing. It runs over Internet Protocol and hence can be routed. Since it does not have any overhead processing like TCP, it is unreliable and has losses. UDP is a connectionless protocol and packets are in the form of datagrams. Since, it does not involve overhead processing, it is very fast and is used in applications where loss is acceptable but are sensitive to time. To make the protocol reliable and transmit data at a higher rate than what Secure Copy (SCP) normally gives, we have incorporated some features that are explained in the design decisions.

Design Decisions

Our goal was to achieve at least 20Mbps for a reliable transmission. We also have to prove the files transferred are same by checking the MD5 checksum. Initially, we were just sending packets to the receiver on the link without any limitation on the sender. We observed a lot of packet drop on the link as well as at the receiver buffer. We were not able to transmit even half of the file size on our side to the receiver.

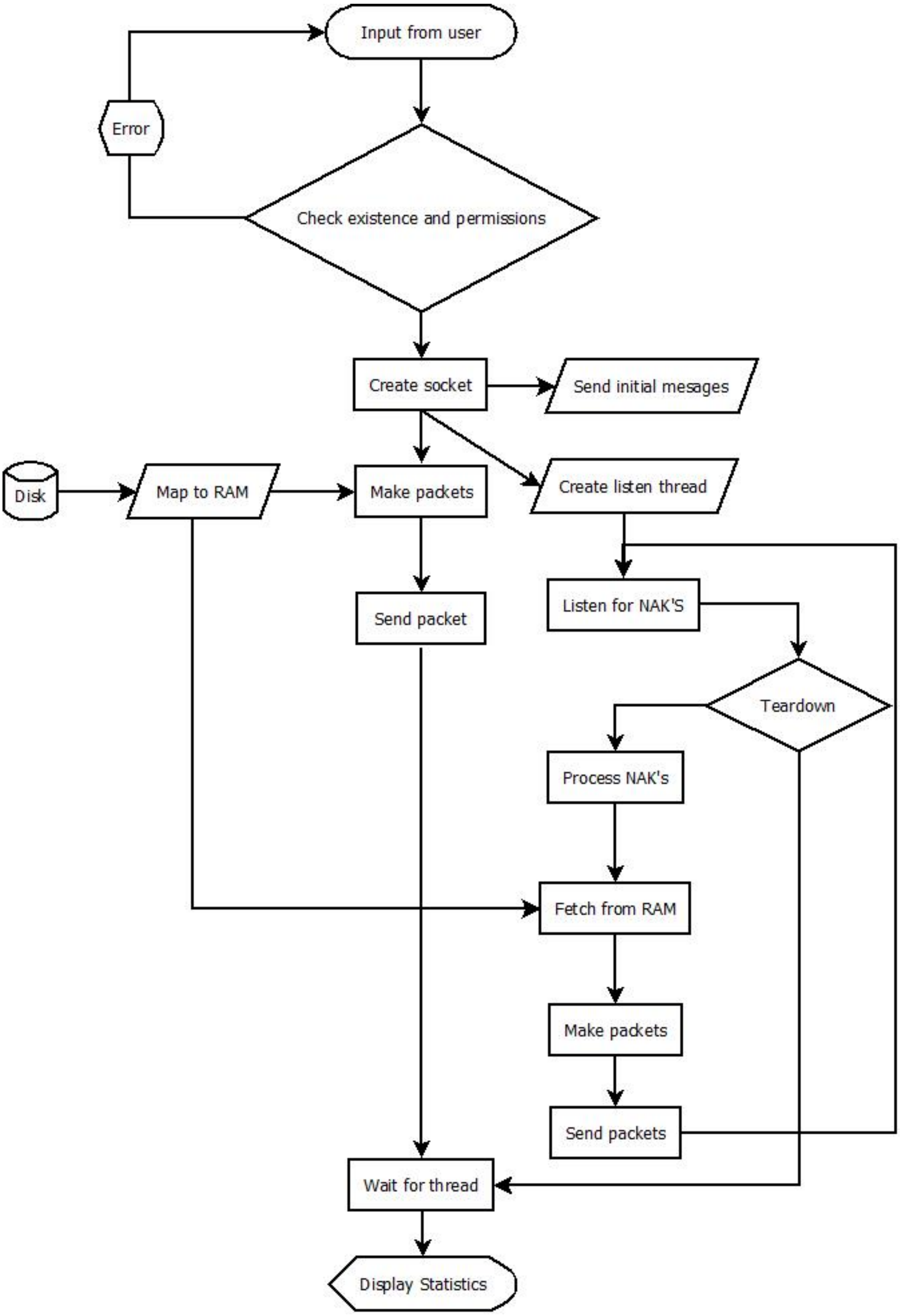
Initial exchange of messages involves getting the file size that the receiver is going to get. The receiver then calculates the maximum sequence number it is going to receive from the sender.

Next, we decided to implement negative acknowledgements in order to fill in the holes created while transmitting. For doing so, we used a multi-threaded approach where there is a main thread which will keep sending on the sender side and on the receiver, the main thread will just receive on recvfrom and write packets to the file. We sent sequence number with each packet in order to aid retransmission. The receiver will create an integer array and is initially set to '0'. The receiver when it receives a packet, will parse the sequence number and put a '1' at that index in the array. This way, we keep track of all the packets received and missing. The other thread on the receiver will just read this array and create a singly inked list which will store the index of the array where there is a '0'. The second thread will also send the missing sequence numbers for retransmission. It keeps reading the array at regular intervals which is controlled by the sequence number and updates the linked list and then sends the missing sequences using sendto to the sender. On the sender side, there is a thread that will keep listening for negative acknowledgements and will directly send the packets with those sequence numbers by reading from memory.

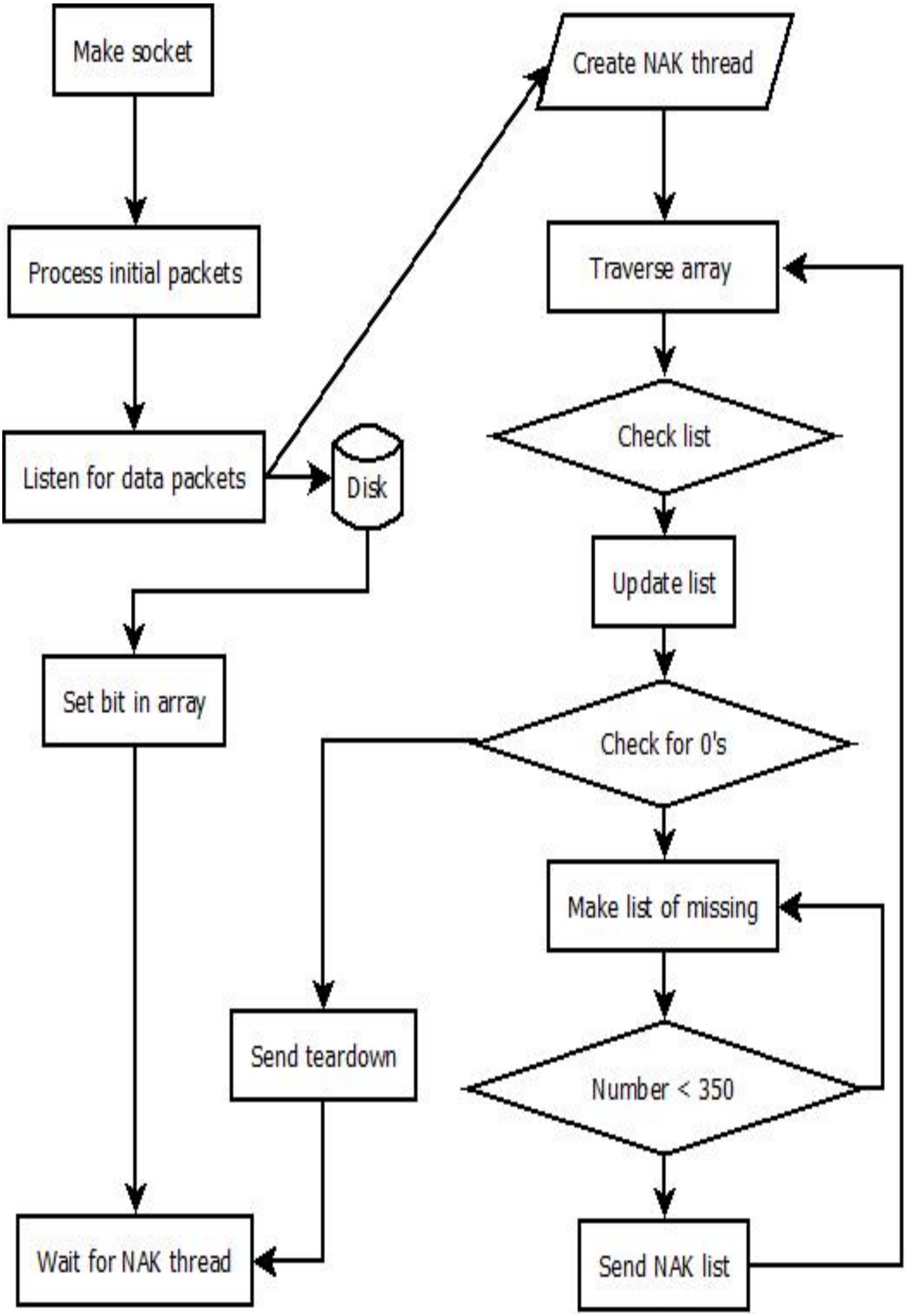
For reading file from the disk, we used mmap to map the file to the RAM using virtual memory. It will give us a base address and form the base address we move ahead based on the packet size. Using memcopy, we put it in the packet. On the receiver side, we use fwrite to read from the buffer and write to disk.

Algorithm

Client side:



Server side:


Make socket
Process initial packets
Listen for data packets
Disk
Set bit in array
Wait for NAK thread
Create NAK thread
Traverse array
Check list
Update list
Check for 0's
Send teardown
Make list of missing
Number < 350
Send NAK list

Execution:

1. Copy the files on to the required directories.
2. Compile Server code on one node and client code on another node.
3. Use the command “gcc <FILENAME> -g -W –o <OBJECT> -lpthread -lrt” to compile and create objects.
4. Use the command ./<SERVEROBJECT> -i <IPADDRESSOFCLIENT> -p<PORTNUMBER> -r<RECEIVEFILENAME>.
5. Use the command ./<CLIENTOBJECT> -i <IPADDRESS> -p<PORTNUMBER> -f<FILETOSEND> -r<RECEIVEFILENAME> to run client.

Results and Analysis:

LINK SPEED	RTT	LOSS	FILE SIZE	TIME	THROUGHPUT
100Mbps	10ms	1%	100MB	29.88s	55.357Mbps
100Mbps	200ms	20%	1GB	328.89s	52.176Mbps