CDHT REPORT

The program design consists of 4 threads (userv, tserv, pinger, and filerequest) that work concurrently to respond to ping messages from other peers, respond to tcp messages associated with file-requests as well as with successor updates, ping successors every 10 seconds (this interval was set to 3 seconds in the youtube video), and monitor requests from stdin input. These functions operate as threads using structures from the pthread library.

The above functions can be considered the core of the program design. However, several helper functions help drive this core, such as filesend, filereceive, digits_only, hasfile, and randgen. The first two helper functions are used to transfer a file, using the TCP transport protocol, from one peer to another. The digits_only function checks if a string consists of digits only. This is essential for differentiating between the message protocols for file requests, and file responses. The hasfile function is the hash function of the program, that is, it takes in a filename and returns its hash. The randgen function generates a float in the range [min, max].

The progression of the program design was iterative. Thus as I coded for further steps I was constantly adapting various parts of my program, with the code for the preceding step as a template. For the pinging aspect of the program, I decided to create two functions, pinger and userv (short for udp server) which are used concurrently to send pings to and from each peer using UDP. The pinger function sends a ping to its immediate successor after t seconds, sleeps for t seconds, and then sends a ping to its second successor (t can be changed by adjusting two values in the loop of the function, however it has been default set to 10). It recycles these steps, incrementing a global variable pseq every time a packet is sent to either the first or second successor. Moreover, the userv function responds to requests from other peers by sending a response message to such peers, and acknowledges responses to its corresponding peer (printing a message to stdout). The message design defined for these ping messages is "Request|Response\nPeerNo\nSeqNo\n" based on whether the message is a request, or response.

Moreover, the userv function keeps track of the acknowledgement numbers, global variables lastrec1 and lastrec2, sent from pings of the corresponding peers two successor peers. This means it can keep track of the difference between the sequence numbers (SeqNo) of pings it sends and acknowledgement of these pings, which is crucial in determining whether a peer is alive as determined in Step 5. The program uses this mechanism to determine whether or not to update the successor peers in the case that a certain successor/peer ungracefully leaves. Certain sequences of steps are carried out based on whether it is the peers first successor, or the second successor that ungracefully left. In the case that the first successor ungracefully leaves, the peer immediately updates its first successor to be its second successor by assigning the global variable succ1addr (initialised to the first successor) to succ2addr. Then, it sends to succ1addr a TCP message asking for the corresponding peers 1st successor and updates its second successor based on this information. If the second successor ungracefully leaves, the peer leaves its first successor peer as is. It only updates its second successor by sending a TCP message to its first successor asking for their next successor (note that the first predecessor of the ungracefully departing peer updates before the second predecessor).

The file transfer aspect of the program executes via the filesend, filereceive, filerequest, hasfile and tserv functions. First of all, the user inputs a request into the terminal of the peer, which is then picked up by the filerequest thread. This thread then opens a request message and sends this to its successor peer (using TCP). This peer then checks if it owns the file by executing the hasfile function (i.e. checking if its peer number is equivalent to the hash of the filename). If it does, it sends a response message directly back to the file requesting peer (again over TCP). Otherwise it forwards the message to its own successor. The design used for these messages are “peernorequestor\nfilename\npeernosender\n for a request, and “response\npeernoholder\nfilename\n for a response. Note that the peernosender header is important for computations involving two adjacent peers in the hasfile function. Then, the binary file is transferred over UDP from the peer owning the file to the requesting peer. The protocols for these messages is “seqno\nackno\nsegmentsize\nflag\ncontent”, for data being sent by the sender, and “seqno\nackno\nflag\n for data being acknowledged by the receiver. Seqno is the first byte of data sent by the sender, and ackno is the 1st byte of data that is next expected by the file requestor. Flag is 0 or 1 depending on whether the packet is the last in the sequence of packets that are sent. ‘segmentsize’ is the amount of ‘content’ that is sent in a particular packet.

A graceful departure of a successor is executed using the filerequest, and tserv threads. When a user inputs quit in the terminal, the filerequest thread sends two departure messages over TCP to the predecessors of its corresponding peer. These predecessors are recorded in global variables pred1, and pred2, which are recorded earlier in execution by requests taken in the userv program. These messages are of the format “dpeerno\nsucc1\nsucc2\n”. dpeerno is a concatenation of 'departure' and the peer number of the terminal peer, whilst succ1 and succ2 are the peer numbers of its two successors. These two messages are then picked up by these predecessor peers both of which update their global variables succ1addr, and succ2addr, corresponding to the socket addresses of their successors. Both of these predecessors then send an acknowledgement back (TCP), either of the form “ack1”, or “ack2”, after which the terminal peer dies.

The code implemented can be made more efficient since there are many extraneous variables and sequences presented. These are present however to make the code more readable, and thus easier to deconstruct/debug when necessary. For example all parts of the header for a file transfer are identified by a variable, however, one could expedite constructing this header by not allocating space for all the variables and instead inputting the corresponding values directly into the segment. Another improvement that can be made is potentially the number of functions that are used. This is since many of the functions such as filesend, and tserv may be lengthy to read, and thus could be further deconstructed into two or three functions that are clearer to piece together.

Note that the code used for setsockopt (which renders bound sockets reusable) as well as the code used for digits_only have been referenced from stackoverflow via the following sites:
https://stackoverflow.com/questions/14422775/how-to-check-a-given-string-contains-only-number-or-not-in-c
https://stackoverflow.com/questions/24194961/how-do-i-use-setsockoptso-reuseaddr/25193462.

Attached is the link for the screencast demo as required:
https://youtu.be/hiPTn6nzaFQ.