Implementation of Total-Ordered Multicasting:

Our chat application must deliver messages to the user in the same order on every node. This is total ordering. In order to implement total-ordered multicasting, we used a version of the ISIS Algorithm which consists of three different types of messages:

- CHAT — the initial message broadcast by a node to all other nodes
- PROPOSAL — a reply to a CHAT message sent with a suggested priority
- FINAL — a broadcast of the final priority assigned to a CHAT message, sent by its initial sender after receiving all PROPOSAL replies.

To cover all three types of messages plus additional information regarding each message that we might want to store, we abstracted this data into a `Message` class that has the following attributes. Note that not all attributes are relevant to each message type:

- `msgtype`: one of CHAT, PROPOSAL, or FINAL
- `origin`: the IP of the node the message originates from
- `msgid`: a unique ID for each message, created using Python's UUID module
- `text`: the body of the chat message
- `username`: the username associated with the node that sent the message
- `priority`: depending on `msgtype`, this is either proposed or final priority
- `deliverable`: whether this message is deliverable or not
- `proposals`: a set of all PROPOSAL messages received in response to this message so far, each with a proposed priority
- `proposals_mutex`*:* a lock used to protect access to the proposals set above
- `alive_set`: a set of all nodes that were alive when the message was initially sent

When broadcasting a new CHAT message via the `bcast_msg` function, the sending node assigns it its current counter value, and creates a copy of it to be stored in its own message queue, which is then sorted by priority.

When receiving a message, the node calls the the `handle_message` function which handles each message differently depending on its `msgtype`:

- CHAT: The receiver first prepares a new PROPOSAL message with its current counter value, then stores the message in its message queue and sorts it. Finally, it replies to the sender with this PROPOSAL message.
- PROPOSAL: Using the unique `msgid`, the receiver retrieves the original CHAT message from its queue and adds this PROPOSAL message to its Proposals set. It then computes the intersection of the original CHAT message's `alive_set` (the set of all nodes alive when it was sent) with the set of all nodes alive right now. If this intersection equals the set of all nodes that have delivered proposals for the original CHAT message, all

required proposals have been received. The receiver changes the original message's priority to the maximum of all proposals, marks it as deliverable, sorts the queue, and sends the FINAL message with the priority just decided.

- FINAL: Using the unique `msgid`, the receiver retrieves the original CHAT message from it queue, sets its priority to the final priority just received, marks it as deliverable, and sorts the queue.

Deliverable messages are displayed to chat via the `do_delivery` thread. It repeatedly checks the front of the node's queue for deliverable messages. If any are found, it displays them in order and pops them off the queue.

Failure Detection:

We depend on non-blocking TCP sockets to detect failures. The socket timeout is set to 1 second. TCP uses 4 messages upon a normal termination (when the user uses the `/quit` command). Each node sends a FIN packet to each other, and then send an ACK packet in response. For improper termination due to crashes, other processes will wait for a message (ACK or FIN) from that node and due to the 1 second timeout, we will detect failures within 1 second. This is similar to heartbeat failure detection protocol.

Upon failure detection, the `handle_crash` function removes the crashed node from the list of alive nodes and then goes through the message queue to do two things:

- firstly, it deletes all messages originating from the crashed node regardless of their deliverability status - to prevent delivering any message after notification of node failure,
- secondly, it checks the Proposals Set of every message sent by the calling node for proposals from the crashed node, which are now invalid. If any such invalid proposals are found, they are removed from the set. Then we recheck to see if the given message can now be marked deliverable. If so, we multicast a FINAL message to the remaining alive nodes.

Metrics:

In order to evaluate the performance of our distributed, we collected data on how the number of nodes active affected two different metrics: the average round-trip-time (RTT) taken between sending and receiving messages, and the failure detection time when a node crashed.

The values for RTT can be split further into two cases: the RTT between the sending of a CHAT message and the receiving of a PROPOSAL reply, and the RTT between the sending of

a PROPOSAL message and the receiving of the FINAL reply. For each number of nodes, we measured these RTTs for messages of different lengths - 2 bytes, 5 bytes and 20 bytes respectively - and averaged them to arrive at a better estimate of the round-trip time taken between messages.

It may be noted from our results that the RTT for a proposal-final exchange remains fairly constant at around 500000 us (with a few outliers taking about twice that time). The RTT for the chat-proposal, in contrast, varies greatly even between similar readings. This may be due to factors such as the pickling/unpickling of a long message text, the sorting of queues at either end, etc. Lastly, the results for failure detection time indicate that it is not significantly changed in any particular direction by the addition of more nodes to the system.

Round Trip Time						Failure Detection Time	
2 nodes:	2 bytes	5 bytes	20 bytes	Avg		2 nodes:	
chat-proposal	2496us	2460us	375846us	126934us		detect time	6281us
proposal-final	501242us	501074us	501103us	501140us			
						3 nodes:	
3 nodes	2 bytes	5 bytes	20 bytes	Avg		detect time	8099us
chat-proposal	523299us	1397973us	511841us	811038us			
proposal-final	502160us	500984us	500993us	501379us		4 nodes:	
						detect time	7119us
4 nodes	2 bytes	5 bytes	20 bytes	Avg			
chat-proposal	461325us	998467us	1570061us	1009951us		5 nodes:	
proposal-final	501091us	998901us	1001104us	833699us		detect time	7472us
5 nodes:	2 bytes	5 bytes	20 bytes	Avg		All times are recorded in microseconds(us).	
chat-proposal	1116663us	571369us	554239us	747424us			
proposal-final	997872us	569339us	551959us	706390us