Project Report

System Architecture:

The system used to design this project relies a lot on the singleton pattern. Specific work is split to different components which communicate with each other. All of these components are implemented as Singleton classes.

The following are the major components of the system.

SenderSocket

A class that can be used to send data to a host at a port. The socket is created and mostly closed right after the message is sent.

MessageDispatcher

A singleton class that lives for the duration of the program. The MessageDispatcher, as the name suggests dispatches messages through the socket to the respective processes. ALL UDP messages in the system are sent by the MessageDispatcher, which means that it is the only one accessing the SocketSender component of the program.

The MessageDispatcher maintains an internal queue of messages that it needs to send out. It also sends these messages on a different thread on a first come-first serve basis. ANY other component in the system need only call the .add_message_to_queue function.

The MessageDispatcher is a very simple component that ONLY waits and sends messages in its queue to their respective hosts. It does not even care about the delivery of these messages, once it “thinks” it has sent a message, it simply delegates the tracking of the reliability to the DeliveryTracker. It does not block other components.

DeliveryTracker

Similar to the MessageDispatcher, the delivery tracker is also a singleton class that lives for the duration of the program. This component tracks the delivery of all messages sent by the MessageDispatcher. It maintains its own state of the messages that have NOT been confirmed to be delivered yet.

This component is primarily contacted by

- MessageDispatcher after it shoots out a message to track the delivery of the message.

- MessageHandler when it receives a message. The handler notifies the tracker about the receipt of a message, at which point the tracker marks this message as delivered and deletes any data it had related to that message.

Detecting other process crashes:
Whenever the tracker receives a message from the dispatcher, it sets an arbitrary *expected_delivery_time* for the message. When the local time in the system exceeds this *expected_delivery_time* it adds this message back to the MessageDispatcher so that it gets sent again. The *expected_delivery_time* is set to vary for different types of messages.
It also has data to figure out the number of retries a particular message has been through, if the number of retries exceeds the *MAX_RETRY_COUNT*(set to 5), then it crashes the program assuming that the receiver of this message has also crashed. This is an assumption made by this system to predict/delete crashes on other processes.

The retry checks also happen on a different thread and do not block any other component.

ListenerSocket

This class is instantiated when the program begins and starts listening for packets on the specified port. This happens on a different thread. If, for some reason this socket fails, the program retries and ensures that the listener is back.

MessageQueue

This is a singleton class that lives for the duration of the program. It is responsive for the following:

- Managing the queue of ordered and unordered messages received by the system
- Marking appropriate messages as deliverable
- Processing the deliverable messages in order

The message queue can also be contacted to get the current state of the system, if a snapshot needs to be taken.

MessageHandler

All the messages received by the ListenerSocket is handed over to the MessageHandler. Infact, the ListenerSocket is designed in a way to require the message handler to be provided to it before it starts receiving messages. This class is instantiated by main and sent to the ListenerSocket.

The MessageHandler serves the single purpose of handling the messages received on the listener socket. It parses the message received and does the following:

- DataMessage:
    - Checks if its a duplicate by contacting the message queue, if so, ignores this message
    - Otherwise,
        - Adds sends this message to the MessageQueue.
        - Sends an ack for this message with the appropriate sequence to the MessageDispatcher, to be sent to the respective sender of the DataMesage.
        - Also updates the SeqCounter when sending the ack.
- AckMessage:
    - Infers that this scenario means that the data message sent to this host was received and therefore notifies the DeliveryTracker about the delivery of the respective DataMessage.
    - Checks if all the acks for the associated data message has been received with the help of DataMessageSeqTracker and if so,
        - with the help of the DataMessageSeqTracker figures out the max sequence for the DataMessage.
        - Sends a SeqMessage to be delivered to everyone to the MessageDispatcher.
- SeqMessage:
    - Infers that this scenario is possible because of the delivery of the DataMessage (if self sender) and the AckMessage and marks them appropriately in the DeliveryTracker.
    - Notifies the MessageQueue of the data that it received from the SeqMessage.
    - Updates the SeqCounter.
    - It then sends a SeqAck message to the MessageDispatcher to “ACK” this message. *(Note: The DeliveryTracker does not care about the delivery of this ACK and thus ignores this)*
- SeqAck
    - Infers that DataMessage, AckMessage and SeqMessage must have been received by the host and notifies the DeliveryTracker about the same.

Simulating network delays and message loss:

Apart from handling the messages, the message tracker also plays a useful role in simulating message drops and network delays. Based on the settings of the system, this component plays a useful role in simulating real world scenarios and testing them.

DataSeqTracker

This is a tightly bound component to the MessageHandler and helps in tracking the seq proposals received for the data messages. It maintains a note of proposals it has received by tracking the max proposal and a list of all proposing processes.

SeqProvider

This is a singleton class that lives for the duration of the program. It has a thread safe way of keep the local sequence number.

SockImpl

Is a helper class that makes handling sockets easy. This class is extended by both the ListenerSocket and the SenderSocket classes.

Log

Through the system, logs are handled by this class. It has static methods that help keeps logs within the system (print to the console). There is an option to set the log level for the system and track the respective log (error, verbose, info, debug, none)
***Usage**: Log::v("message") //v stands for verbose*

ProcessInfoHelper

This component keeps track of the information about the process, like its ID and hostname. Moreover, it also keeps a list of all other processes in the system. It is handy helper for the others when required.

All the components handle their own responsibilities with as little dependency on other components.

The following components are specific to the “part 2” of the project:

TCPListener

This class is similar to the SocketListener class and does the same thing, in that it maintains a connection and listens for events. The only difference between the two is that SocketListener uses UDP and this uses TCP.

TCPSender

Similarly, the TCPSender sends messages over TCP.

SnapshotHandler

This component is responsible to taking snapshots of the system.
Since this system uses TCP ONLY to send and receive markers, all messages received from the TCPListener are sent here. This handler then figures out if it has received a marker and if so, it executes the ChandyLamport Algorithm.

The markers are sent to the other over TCP using the TCPSender.

Moreover, the MessageDispatcher, also contacts the SnapshotHandler after sending everymessage. The SnapshotHandler is responsive for figuring out WHEN to initiate a snapshot.

Implementation Issues:

1. The system currently does not perform an initial handshake to ensure that all processes are up before sending the messages and just sleeps for 10 seconds before starting message dispatch. Although, the retry logic takes care of this aptly.
2. Part2: Tcp connection fails sometimes and just the markers are not received by all, and therefore the snapshot does not end in these scenarios.