Total-order Reliable Multicast and Chandi-Lamport Global Snapshot Systems

by Thien Nguyen

Total-order Reliable Multicast

Algorithms overview

- We implement the reliable multicast protocol that guarantees the total ordering of the sent messages (i.e. each process will have the same ordering of the delivered messages). This implementation can tolerate message drop/delay (by resending acknowledgement/data message after a set timeout) but does not tolerate process failure/crashing.
- Total ordering is achieved by implementing the total order algorithm which uses sequence numbers (starting from 1 and increments everytime an event happens) to order incoming Data Message. Each process has a delivery queue that holds message along with their sequence number. In this project, each process spawns a receiving thread that performs a specific task depending on what type the message it receives is:
    - If it's a Data Message, it replies with an Acknowledgement Message along with a sequence number that is used for final ordering.
    - If it's an Acknowledgement (ACK) Message, it stores it in a map that keeps track of the received acknowledgement so far. If it receives enough acknowledgements (i.e. from all other processes), it picks the max sequence number and broadcast that to everybody (with a Sequence Message) signifying the final sequence choice for such message.
    - Finally, if the incoming message is a Sequence Message, the process reorders the delivery queue based on this final sequence. Then it delivers as many messages (in front of the queue) with a final sequence number as possible. Messages that do not have a final sequence number yet (arise from said process sending out Data Message but with a smaller sequence number) can block the delivery of messages with sequence numbers already.
- It can be shown that this numbering scheme of messages (with tie-breaking using proposer id) provides both total-ordering and agreement of the messages' sequence. In which the process of delivering messages through a priority queue guarantees that the delivery is monotonically increasing (w.r.t. the sequence number/sender id).

Handling message dropped and delayed

- We use "watchdog" threads with timeout to handle message drops and delays.
- There are three possible places where messages can be dropped:
    1. The sending of data messages
    2. The sending of ack messages
    3. The sending of seq messages
- For sending Data Messages, associating with each Data Message is a thread keeping track of whether an ACK from a certain process has been received. If an ACK from a certain process p in the group has not been received after a certain timeout, it's either that process p has never received the Data message (case 1) or that the Ack was dropped (or just delayed). In either case, we resend the Data message in which the receiving process can finally receive the data message and respond with an ack, OR resend the old ACK if it's a duplicate Data Message.
- Now, associating with each ACK is another watchdog process waiting for a corresponding sequence message. If after a certain timeout, the watchdog thread notices that it hasn't seen a corresponding sequence message for such message, it assumes that either the Ack was dropped or the sequence message was dropped. In either case it resends the ACK until it receives a sequence message, where the process receiving a duplicate ACK simply resends the sequence message.

Program outline and implementation details

UDP as communicator

- We created a class called networkagent.h to handle all setting up and communication of UDP. This greatly simplifies the code and provides much needed encapsulation for programs of this size.
- Since UDP is unreliable, the drop/delay/non-fifo assumptions above holds. However, the advantage is that it's much faster than TCP.

Startup and multicasting messages

- First each container waits for the other container to connect to the network (so nobody send until every process specified in the Hostfile is "ready" i.e. have sent and received I'm alive messages from all other processes).
- Then after making sure everyone is up, the program begins sending DataMessages with arbitrary data (in this project, a function of the msg_id).
- Then the algorithm above is executed along with watchdog threads tracking message drops and received.
- **In this project, the sender does not wait until the previous message is finalized before sending the next message. Hence, this detail makes the input non-trivial in which the algorithm would have to deal with NON-FIFO messages.**

Implementation of simulated drops and delays

- For delays, we simply sleep that amount before sending a message (implemented through the communicator class above).
- For drops, we generate a uniformly random real number between 0 and 1. If that number is less than the drop rate, then we drop the message (i.e. not calling the send function).

Delivery-queue, ACK-History, and Delivered-List

- The delivery-queue holds pending messages along with their sequence number, proposer, sender, data and whether if they are deliverable or not.
- The delivered list is simply a vector holding (in-order) the messages that was delivered from the delivery-queue.
- The ACK-History is a map that maps a message (sent out) along with the list of ACKS it has received. The first ACK would always be the self-ACK that contains the sending-process' current sequence number.

Chandi-Lamport Global Snapshot

We implement the Chandi-Lamport Global Snapshot algorithm as an add-on service to the reliable multicast program above. It is a service in terms that it disrupts the program above to the minimum. It's implemented as a detached-thread from the program above that's only awoken when called into service (a daemon style program). The only major modification to the program above is setting a flag to when to send messages passing through its communication channels to the snapshot thread.

The algorithm

- One process will be an initiator. It will first take a snapshot of its local state and then send a marker out to other snapshot processes telling that it wants to take a snapshot. At that moment it turns on recording for incoming and outgoing messages per communication channel from the main program that it's keeping track of (in this case the Reliable Multicast program).
- Then when a non-initiating process receives its first marker from the initiator, it wakes up and asks the host program to send a local snapshot as well as to turn on recording of messages for all channels **that it hasn't received any marker from** . So for this process receiving its first marker, it would turn on channel recording for all processes except the initiator. Then it sends its marker to everybody.
- Now, when any process receives a second marker onwards from some process p, it finalizes that channel recording for that process p. Then the snapshot program finishes taking a snapshot after it has received a marker from everybody (hence closing all channel recording).
- The local snapshot would include a local state and a channel state.
- The initiator can then collect all the local snapshots to make a global snapshot or in our case, we output each local snapshot into an output folder which can be combined later as wish.

Marker receiving rules

- Marker receiving rule (for this process j):
    - if it's the first marker ever (say from process i), record local state
        - turn on recording channel for all channels except (i), and send out markers
    - We need to keep a list of markers already received say alreadyReceivedProc
    - When we receive a message (from the object we manage):
        - we check if it's from a process that we have received a marker before
        - if we have received a marker from that process, we ignore the incoming msgs
        - otherwise, we record it in a channel state
        - When we receive a marker (say from process k) after recording our state:
        - We add process j to alreadyReceivedProc and finalize its channel state
        - If we have received all n-1 markers, we send our local snapshot to the initiator

Communication between the snapshot daemon and the main program

- We simply use a shared datastructure (accessible by making the classes "friends") and use a mutex to ensure atomiticity of read/write operations.
- For signalling when to start recording communication channels, we also use a shared flag along with a mutex to ensure in-order operation.

Communication between snapshot daemons

- Since the model of the original CL Glboal Snapshot algorithm assumes a reliable communication channel, TCP is a natural choice for snapshot daemons to communicate. We open a TCP communication channel for every pair of process.

Spawning a snapshot daemon

- Each multicast program will create a snapshot object. This object will be spawned as a background process (i.e. daemon) ready to either initiate a snapshot or response to one.
    - They are setup as a listening TCP connection. Then when the initiator wants to initiate a snapshot, it will prompt those daemons to accept the connection in which a channel is created for communicating.
    - It's then when each receiving daemon will receive a marker (from the initiator)
    - It will then invoke the object that it's keeping track of to send a snapshot of the state
    - It would also inform the object to record incoming messages on its channel
    - Then finally it sends out a marker to everybody.

A note on channel state in snapshot

- Since the snapshot algorithm is implemented with TCP (with no delay/dropping), when we run the total-order multicast algorithm with delay and drop-rate, we will not observe (most of the time) any messages in the channel state of the snapshot. The reason for this is due to the snapshot algorithm finishes way before any messages from the reliable multicast algorithm can be sent out (since they have delays and there's no delay in the snapshot algorithm). This can be adjusted to add delays to the snapshot algorithm.

Running the code

Naming requirement (important)

- This program assumes that each container name is uniquely identified by the ending numbers. So valid container names can be: `container1, container2, container3, ...` but invalid container names are like `cat, dog, elephant, ...`

Compiling and setting up containers

- Below is an example of a Dockerfile to compile. We require g++ and the Pthread flag to compile.

```
FROM ubuntu

RUN apt-get update && \
    apt-get install -y g++

ADD ./*.cpp /app/
ADD ./*.h /app/
ADD Hostfile /app/
ADD rprj1.sh /app/
ADD countdelaydroprate.sh /app/
RUN mkdir playground

WORKDIR /app/


RUN g++ -pthread networkagent.cpp waittosync.cpp CL_global_snapshot.cpp reliable_multicast.cpp main.cpp -o prj1
```

- Then we can run `docker build . -t prj1` to compile.
- Then running `docker run -it --name <container_name> --network <networkname> prj1` will create an interactive node. This is where we can run each node and observe its output (along with any debug messages).
- We can run the program following instructions of the section below.

Turning on and off DEBUG messages

- Debug messages are displayed when the `#define DEBUG` line in `reliable_multicast.h` is uncommented. We would need to recompile in order to add debug messages (this includes watchdog information, message drop information, delivery_queue, and any error/recovery messages).

Adjusting parameters

- Parameters like watchdog-timeout and maximum number of timesouts (until declaring a process has failed) can be changed through tweaking `#define` field in `reliable_multicast.h`.
- Note this program spawns `total message count * number of processes` threads total. If this become problematic, one can adjust the `MAX_NUM_THREADS` parameter in `reliable_multicast.h`.

Running the program

- The usage is specified as `./prj1 -h Hostfile -c <count> [ -t <delay_in_ms> -d <droprate> -X <take_snapshot_after>]` where `<count>` is the number of messages for the running process to multicast to the other processes.
- Hence, by setting count to be either 0 or a positive integer, we can **specify whether a process is a sender/receiver or purely a receiver**. This program supports any arbitrary number of senders at the same time.

Running multiple containers

- This was written to be run interactively on the terminal. So it's best to run each container separately and observe the output separately. For each terminal (say from using Tmux or iTerm) that we spawn, after building, we can run the following to enter the interactive shell: `docker run -it --name <container_name> --network <networkname> prj1`
- Then once we are in the interactive shell, we can run each container as detailed in the next section.

Full run command with simulated message drops and delays

- The full run command is: `./prj1 -h Hostfile -c <count> [ -t <delay_in_ms> -d <droprate> -X <take_snapshot_after>]` .
- Adjusting the message drops and delays can be done by setting the `-t` and `-d` parameters in the usage.

Specifying which process to send

- This can be done by setting the count of receiving processes to 0 and count of sending processes to a positive integer.