Derecho in DistAlgo

Vishnu Dutt Paladugu
CSE 523
Under Guidance of Prof. Annie Liu

Introduction

Derecho [1] is a library to help build replicated, fault tolerant services in the Cloud. It leverages the use of RDMA networking for low latency communication. It uses virtual synchrony for dynamic membership tracking.

Goal

The goal of this project is to implement the Derecho protocol in DistAlgo as per the pseudo-code in [1] under the sections Appendix - A.

Code

The code implemented as part of this project is hosted at https://github.com/unicomputing/derecho-distalgo/blob/master/derecho.da.

Input

The program takes in an integer as input to simulate the number of nodes in the system.

```
python -m da derecho.da 3
```

Output

The program simulates crash of one of the Nodes and outputs the constituents of the new view of the system after failure.

Implementation Design

The implementation follows the pseudo-code under Appendix-A in [1] for the most part, but takes a few assumptions for completeness of code. The DistAlgo code implementation contains the pseudo-code side-by-side line-to-line to compare for implementation changes, additions or deviations. The report below explains the Derecho protocol as per the pseudo-code and has the appendix section number against the sub-heading for reference.

RDMA Communication Simulation

To simulate RDMA communication for Shared State Table (SST) changes, a DistAlgo message based write utility *write_sst()* is written (by Prof. Annie Liu). Each call updates the local SST and then sends a message to all other nodes about the update. On receiving such an update message a Node updates its local SST copy.

Shared State Table (SST) (A.1)

Derecho uses a novel replicated data structure called the shared state table (SST). It serves as a tabular distributed shared memory abstraction. Each node in the cluster has write access to its row and reach access to all other rows. This eliminates write-write contention as at any time there is only a single writer. A node updates its local copy of its own row in the SST and then asynchronously broadcast the change as mentioned in above in the RDMA section.

```
class SSTRow:
  def __init__(self, nprocess, window_size):
    # suspected: Stores true if the Node suspects a failure of any other Node against its index
    self.suspected = [False] * nprocess
    # num_committed: total number of changes committed by the Node, as suggested by the Leader
    self.num_committed = 0
    # num_acked: total number of changes acknowledged by the Node, as suggested by the Leader
    self.num_acked = 0
    # received_num: The vector received_num holds counters of the number of messages received from each node
    self.received_num = [0] * nprocess
    # wedged: Stores true if the Node has wedged its SST
    self.wedged = False
    # changes: The node ids of the crashed/new nodes
    self.changes = []
```

A sample SST row declaration with few fields

Reducer Functions

The Derecho algorithm uses a bunch of reducer function over the SST which helps produce a summary of summary of certain projector's view of the entire SST. Aggregates such as min, max of a column are an example. The Distalgo code has implementation for all the required reducer function. Couple of them are listed here for example.

```
def min_not_failed(attr):
    '''
    - Calculates the min value for a column for non-failed Nodes in an SST
    '''
    return min({getattr(sst[row], attr) for row in range(len(sst)) if not sst[my_rank].suspected[row]})

def logical_and_not_failed(attr):
    '''
    - Calculates the logical AND of all values in a column for non-failed Nodes in an SST
    '''
    return all(getattr(sst[row], attr) for row in range(len(sst)) if not sst[my_rank].suspected[row])

  def logical_or(attr):
    '''logical OR of all values in a column in an SSTColumn to be passed as a parameter'''
    return some(v in sst, has=getattr(v, attr))
```

Examples of reducer functions

SST Multicast (SMC) (A.2)

SST Structure (A.2.1)

The SMC uses two fields used for message passing, *slots* and *recieved_num.* Slots is a vector of buffers of size window_size used to store the message. *Received_num* is a vector of integers to represent the latest message received from each node. The change of the received number at a particular index from k to k+1 indicates that a new message is available to be received from the node.

Initialization (A.2.2)

The recieved_num vector is initialized to -1 and the slots are initialized to null.

Sending (A.2.3)

Before a message can be sent, a slot is deduced which would hold the message. A slot can be reused if the previous message in the slot was received by everyone. This is derived by calculating the minimum value in the received_num field in the SST. Once decided, the slot's buffer is populated with the message and index incremented.

Receiving (A.2.4)

To receive a message a node has to look at the next slot in its index, increments the received_num variable and calls for a *recv* function (A.3.1).

Atomic Multicast Delivery in the Steady State(A.3)

Receive (A.3.1)

Upon receiving a message, the global index of the message is calculated and stored accordingly in the buffer (*msgs*). The node update the latest_received_index field in its SST against the sender of the message. Then updates the *global_index* field in the SST which implies all messages whose global index is lesser than the global_index are received by the node.

Stability and Delivery (A.3.2)

In a stable state the Node always checks for stable messages in its buffer (which is globally ordered) and then delivers them in a round robin fashion in order and updates the *latest_delered_index* field in the SST. Also this process clears the delivered messages from the buffer.View Change Protocol (A.4)

Failure Handling and Leader Proposing Changes for the Next View (A.4.1)

Crash Simulation

To simulate a crash of a Node, a simulator process - Sim is used. After start-up a Node is chosen at random to crash. The Sim process triggers a message to one of the remaining nodes about the failure. The receiver node marks the crashed node as 'suspected' and the system propagates this failure as per the algorithm.

Failure information propagation

Once a Node receives a '*failure'* message about a Node from the Sim process, it marks the suspected field for the failed process in the SST to true. The *suspect()* function propagates the 'suspected' status to all the nodes, which in turn update their local copy of SST to mark the failed process as suspected. The Derecho system throws a *derecho_partitioning_exception* exception if more than half of the nodes are suspected to have failed. The *failed* list in the current view is updated with the latest failed node and the current view is wedged.

Leader Election

All the nodes in the Derecho system are ordered using '*row_rank'* which is the row number in the SST each node represents. During leader election, the non-suspected node with the lowest row_rank is agreed upon as the leader by all the remaining nodes. Once the new leader is elected, it updates the current view that it is the leader.

View Change Proposal

Once failures are noted in the current view by the system, the leader initiates a new membership change proposal. The failed nodes are added to the *changes* list in the SST and committed.

Terminating old view and installing new view (A.4.2)

The follower nodes register that there are new changes available, and acknowledge and copy the change vector to their SST rows. The leader now commits the acknowledged proposals by all the nodes. The nodes on registering the new commited change proposals initiate an epoch termination protocol.

In the epoch termination protocol, the members of the new view is decided based on the current member list and the change list. The leader then cleans up the messages in its SST table due to the failure and sets the ragged_edge_computed flag to true. The follower nodes then copies the state as per the leader as part of the ragged edge cleanup protocol. Once the whole system has cleaned up, the next view is copied as the current view.

Further Work

The algorithm is implemented as per the pseudo-code provided in the appendix in [1]. But further work on evaluation and correctness needs to be undertaken for further understanding the algorithm. Also, the 'crash' feature of DistAlgo could be used to simulate the crash.

References

[1] **Derecho: Fast State Machine Replication for Cloud Services.** Sagar Jha, Jonathan Behrens, Theo Gkountouvas, Matthew Milano, Weijia Song, Edward Tremel, Robbert Van Renesse, Sydney Zink, and Kenneth P. Birman. 2019.ACM Trans. Comput. Syst. 36, 2, Article 4 (April 2019), 49 pages. DOI: ttps://doi.org/10.1145/3302258