Specification and Runtime Checking of Derecho, A Protocol for Fast Replication for Cloud Services

Kumar Shivam

Vishnu Paladugu

Yanhong A. Liu

Stony Brook University

{kshivam,vpaladugu,liu}@cs.stonybrook.edu

May 17, 2023

Abstract

Reliable distributed systems require replication and consensus among distributed processes to tolerate process and communication failures. Understanding and assuring the correctness of protocols for replication and consensus have been a significant challenge. This paper describes the precise specification and runtime checking of Derecho, a more recent, sophisticated protocol for fast replication and consensus for cloud services.

A precise specification must fill in missing details and resolve ambiguities in English and pseudocode algorithm descriptions while also faithfully following the descriptions. To help check the correctness of the protocol, we also performed careful manual analysis and increasingly systematic runtime checking. We obtain a complete specification that is directly executable, and we discover and fix a number of issues in the pseudocode. These results were facilitated by the already detailed pseudocode of Derecho and made possible by using DistAlgo, a language that allows distributed algorithms to be easily and clearly expressed and directly executed.

keywords: replication and consensus protocols, executable specification, runtime checking

1 Introduction

Reliable distributed systems require replication and consensus among distributed processes to tolerate process and communication failures. Many algorithms and variations have been proposed for replication and consensus, starting from Virtual Synchrony (VS) by Birman and Joseph [BJ87], and Viewstamped Replication (VR) by Oki and Liskov [OL88], including the well-known Paxos algorithm by Lamport [Lam98], among many others, e.g., see [VRA15, CL21]. However, understanding and assuring the correctness of these algorithms have remained a significant challenge, especially as more sophisticated algorithms are being developed. This paper. This paper describes the precise specification and runtime checking of Derecho [JBG⁺19a], a more recent, sophisticated protocol for fast replication and consensus for cloud services. Derecho provides state machine replication and dynamic membership tracking, especially for replicating large data with non-blocking pipelines, and is shown to be significantly faster than comparable widely used, highly-tuned, standard tools. It employs a lock-free distributed shared memory called a shared-state table (SST) for sharing protocol-control information, especially suitable for running on remote direct memory access (RDMA).

Our specification is written in DistAlgo [LSL17], a language that allows distributed algorithms to be easily and clearly expressed and directly executed. It provides all three benefits enabled by DistAlgo: (1) distributed processes and communications, both synchronous and asynchronous, are expressed at a high level as pseudocode, (2) the specification is completely precise, supported by formal operational semantics of DistAlgo, and (3) the specification is directly executable in distributed environments, supported by the DistAlgo compiler that is built on top of the Python compiler.

A precise specification must fill in missing details and resolve ambiguities in English and pseudocode algorithm descriptions while also faithfully following the descriptions. Our specification is especially facilitated by Derecho's already detailed pseudocode and descriptions [JBG⁺19a, Appendix A], as well as Derecho's active team of experienced researchers and developers. Derecho pseudocode uses exact fields of structures for information kept in the system state, especially including for the SST, and provides in detail the key steps in both steady-state execution and view change protocols.

To help check the correctness of the Derecho specification, we also performed careful manual analysis and increasingly systematic runtime checking. The clarity of the specification allows some issues to be noticed by quick manual inspection, while automatic running and checking allow more subtle issues to be discovered.

We specified and checked well-established safety properties as well as various progress queries and results. These specifications and automatic checking are enabled by a general framework for runtime checking of safety and liveness properties [LS20] supported by DistAlgo. As a result, the properties are specified at a high level as logical statements and are checked automatically by a checker process while the protocol runs, without changes to the specification of the protocol that can obscure the clarity of the protocol specification.

There has been a significant amount of related research, as discussed in Section 7. Our work contains three main contributions:

- We develop a rigorous specification of Derecho that corresponds closely to the pseudocode and is complete, precise, and directly executable.
- We discover and fix a number of issues in the Derecho pseudocode, e.g., [JBG⁺19b, Errata, page 50], and helped improve the pseudocode [Jha22, page 72].
- We demonstrate through Derecho a practical method for developing a rigorous and improved specification through not only manual inspection but also automated runtime checking.

Note that the bugs and fixes we found are for the Derecho pseudocode [JBG⁺19a], and have been checked and confirmed by the Derecho team [JBG⁺19b, Jha22, Jha23]. In all cases, Derecho developers have also checked and confirmed that the bugs are not in their implementation in C++ [Jha22, Jha23].

Bugs in protocol pseudocode are quite normal, simply because pseudocode is manually created and there is no way to run or check other than by staring at it. DistAlgo is exactly for expressing protocols easily and precisely at such pseudocode level, and then running them for testing and for systematic runtime checking of desired properties. A complete specification of Derecho in DistAlgo can be found at [der].

2 Derecho and specification language

2.1 Derecho overview

Derecho [JBG⁺19a] is a replication protocol for coordinating distributed actions. The protocol supports state machine replication by utilizing specialized hardware technology, specifically RDMA. RDMA enables direct access to remote memory without involving the CPU, using hardware such as Network Interface Card (NIC), resulting in higher throughput and lower latency through avoidance of context switching.

The protocol employs group multicasting to order client request messages and supports atomic multicast and total-ordered message delivery. In this context, a group is defined as a set of member processes referred to as nodes. Atomic multicast ensures that messages sent by a member node are either delivered to all member nodes or none at all, while total-ordered message delivery guarantees that messages are delivered in the same order to all member nodes in the group.

Each node in a group maintains a copy of SST, one row for each node. Each node updates its own row in the SST and propagates the update to other nodes using RDMA.

The protocol has two main parts.

(1) Steady-state execution. Derecho employs the SST multicast (SMC) protocol for small message multicast, as described in Section 4. To initiate a multicast, one of the nodes in the group stores the incoming request message in SST's ring buffer data structure, which is propagated to all nodes in the group. Each node buffers the message upon receiving it until it knows all the nodes in the group have received it. Atomic multicast delivery of a message happens when all the previous messages have been delivered on all nodes, and the current message has been received on all nodes.

(2) View change. Derecho employs virtual synchrony [BJ87] to track dynamic membership in a process group computing style. Process groups allow members to join and leave the group while the application is active, triggering a membership change. Each membership change initiates a new epoch (view), and the protocol progresses through a series of epochs, each with its own membership. A two-phase commit pattern is used to carry out a membership change, with information exchanged via the SST.

2.2 Language for precise specification

For precise executable specification of Derecho at a high level that corresponds to algorithm pseudocode, we use DistAlgo [LSLG12, LSL17]. DistAlgo supports the following four main concepts of distributed programming by building on an object-oriented programming language, Python.

(1) Distributed processes that can send messages. A type P of processes is defined by

process P: stmt

The body *stmt* may contain, among usual definitions,

- a setup definition for setting up the values used in the process,
- a run definition for running the main flow of the process, and
- receive definitions for handling received messages.

A process can refer to itself as **self**. Expression **self**. *attr* (or *attr* when there is no ambiguity) refers to the value of *attr* in the process.

- ps := n new P creates n new processes of type P, and assigns the new processes to ps.
- ps.setup(args) sets up processes ps using values of args.
- *ps*.start() starts run of *ps*.

new can have an additional clause, **at** *node*, specifying remote nodes where the created processes will run; the default is the local node.

A process can easily send a message m to processes ps:

send \boldsymbol{m} to \boldsymbol{ps}

(2) Control flow for handling received messages. Received messages can be handled both asynchronously, using receive definitions, and synchronously, using await statements.

• A receive definition is of the following form:

receive m from p: stmt

It handles, at yield points, un-handled messages that match m from p. A yield point is of the form --l, where l is a label; it specifies a point in the program where control yields to handling of un-handled messages, if any, and resumes afterwards. There is an implicit yield point before each **await** statement, for handling messages while waiting. The **from** clause is optional. • An await statement is of the following form:

await $cond_1: stmt_1$ or ... or $cond_k: stmt_k$ timeout t: stmt

It waits for one of $cond_1$, ..., $cond_k$ to be true or a timeout after period t, and then nondeterministically selects one of $stmt_1$, ..., $stmt_k$, stmt whose conditions are true to execute. Each branch is optional. So is the statement in await with a single branch.

(3) High-level queries for synchronization conditions. High-level queries can be used over message histories, and patterns can be used to match messages.

• Histories of messages sent and received by a process are kept in sent and received, respectively. sent is updated at each send statement, by adding each message sent. received is updated at the next yield point if there are un-handled messages, by adding un-handled messages before executing all matching receive definitions.

Expression sent m to p is equivalent to m to p in sent. It returns true iff a message that matches m to p is in sent. The to clause is optional. Expression received m from p is similar.

• A pattern can be used to match a message, in sent and received, and by a received definition. A constant value, such as "release", or a previously bound variable, indicated with prefix =, in the pattern must match the corresponding components of the message. An underscore _ matches anything. Previously unbound variables in the pattern are bound to the corresponding components in the matched message.

For example, received("release",t3,=p2) matches every triple in received whose first component is "release" and third component is the value of p2, and binds t3 to the second component.

A query can be an existential or universal quantification, a comprehension, or an aggregation over sets or sequences.

• An existential quantification and a universal quantification are of the following two forms, respectively:

some v_1 in s_1 , ..., v_k in s_k has cond each v_1 in s_1 , ..., v_k in s_k has cond

They return true iff for some or each, respectively, a combination of values of variables that satisfies all v_i in s_i clauses, *cond* holds.

• A comprehension is of the following form:

 $\{e: v_1 \text{ in } s_1, \ldots, v_k \text{ in } s_k, cond\}$

It returns the set of values of e for all combinations of values of variables that satisfy all v_i in s_i clauses and condition *cond*.

• An aggregation is of the form *agg* s, where *agg* is an aggregation operator such as count or max. It returns the value of applying *agg* to the set value of s.

• In all query forms above, each v_i can be a pattern.

Other operations, such as set union and sequence concatenation, can also be used.

(4) Configuration for setting up and running. Configuration for requirements such as the use of logical clocks and the use of reliable and FIFO channels can be specified in a main definition. For example, configure channel = fifo specifies that fifo channels are used and TCP is used for process communication.

DistAlgo also supports automatic visualization of replays forward and backward, making it much easier to understand protocol runs.

DistAlgo compiler, Python syntax, queries, and extended sent and received. To allow anyone with Python to run DistAlgo directly, DistAlgo compiler supports the Python syntax [LLS17]. For example, send m to p is written as send(m, to=p), and each sent m to p has cond is written as each(sent(m, to=p), has=cond); in patterns, =var is written as _var.

While Derecho pseudocode does not use high-level set queries, these queries are critical to specify the many reducer functions used. Also, slightly extended p.sent and p.received, denoting the process p's sent and received sequences, respectively, is critically helpful for specifying the properties to be checked.

In our specification in DistAlgo, the following convention for comments are used: (1) comments after **#** are text or pseudocode copied from the Derecho paper [JBG⁺19a], except when noted as from email with Sagar Jha; (2) comments after **##** (or no comments) describe code we had to fill in; and (3) comments after **###** describe changes to the pseudocode in paper.

3 Specifying system state

Information maintained in a system is essential for specifying the system. We define classes with fields that allow the algorithm steps in DistAlgo to match the corresponding steps in pseudocode exactly. Fig. 1 shows the complete precise specification in DistAlgo.

The key information maintained by Derecho is the SST, specified as a list of SSTRow objects. Fig. 1 (lines 7-32) shows the definition of class SSTRow with its fields. For instance, field slots (line 13) is a list for a ring buffer, with reusable slots (lines 1-6) for request messages and related metadata.

In addition to the SST, Derecho needs a View object to hold information about an epoch (used interchangeably with view in Derecho), such as the leader and members in the view. This object is critical in the membership-change protocol, which is triggered when a member joins or leaves the group. Fig. 1 shows the definition of class View (lines 33-47) with its fields and methods to add and remove members.

Function write_sst in Fig. 2 (lines 1-5) specifies an update to the SST. It updates the local SST and sends an rdma_write_sst message to other nodes. Upon receiving the message,

```
1 class Slot:
              ""A slot stores a client request message. A vector of slots is a field in SST"""
         """ A slot stores a citent request measure in the store a citent request measure in the store a citent request measure in the store and the store in the store and the store in the st
 3
 6
      class SSTRow:
              ""A shared state table (SST) row that stores info about a node.
 8
                A SST has a SSTRow for each member node and is stored in each member"""
          def __init__(self, n, window_size): ## initialize SST columns for the row; all used in pseudocode but the last two
# n: ## number of member nodes in the group
# window_size: ## length of vector of slots for storing client req msgs received by the node, directly or indirectly
10
              self.slots = [Slot() for _ in range(window_size)] # (p.33) vector of window_size slots ## for client request msgs
self.received_num = [-1] * n # (p.33) number of messages received from each node ### number-1
13
14
                                                                                                                               ## initialized in Node.init(), and set in receive_req()
15
                                                                                           # ## global index of last message received from the most lagging node
# ## min of self.global_index over all members
16
              self.global_index = -1
17
              self.latest_delivered_index = -1
              self.latest_received_index = [-1] * n # ## index of latest msg received from each node, set in recv to received_num
18
                                                                                             ## i.e., self.received_num-1 ### redundant, but not clear with null msgs
# ## for each node, min of latest_received_index over all rows in SST
# ## for each node, whether that node is suspected to have failed
19
20
              self.min_latest_received = [-1] * n
21
              self.suspected = [False] * n
22
                                                             # ## true when any node is suspected
              self.wedged = False
                                                                # ## blue when any node is suppressed
# ## list of nodes suspected (or added from joins) to proposed as changes by the leader
# ## number of nodes in self.changes, i.e., length of self.changes
23
              self.changes = []
self.num_changes = 0
24
              25
26
27
28
29
30
31
                                                                  ### could use logical or over sst[my_rank].suspected or even just own suspected
33 class View():
           """A view that holds the information of an epoch. An epoch is the duration of a view."""
34
          def __init__(self, n, epoch=0, leader_rank=0):
    # n: ## number of members in the view
35
36
              37
38
              self.fedder_failed = [None] * n
self.failed = [False] * n
                                                                                  # ## list of member nodes in the view
39
                                                                             # ## 11St of member nodes in the task as suspected and thus considered failed # ## for each node, whether that node is suspected and thus considered failed
40
41
          def add_member(self, node):
                                                                                      ## add member to the view
42
              self.members.append(node)
                                                                                      ## append node to members
                                                                                      ## add the failed attribute corresponding to the added node
43
              self.failed.append(False)
44
          def remove member(self. node):
                                                                                      ## remove node from members of the view
              index = self.members.index(node)  ## get index of node in members of
del self.failed[index]  ## get index of node in members
45
46
47
              self.members.remove(node)
                                                                                      ## remove node from members
```

Figure 1: Specification of system state.

```
def write_sst(row, col, val, index=None): ## write SST entry at row and col, at index if col is a list, with val
      2
3
4
      send(msg_tagged, to= others)
                                                                  ## send the message to other nodes
    6
7
8
9
        output('received rdma_write_sst msg for different epoch, current: ', curr_view.epoch, ' and msg: ', epoch)
10
        return
      if col == "slots" and any(curr_view.failed): # (p.12) ## if msg is a req and curr view has failed members, ignore
11
12
        output("Failure detected, new data messages dropped")
13
        return
14 \\ 15
      if row in freeze:
                                                # (p.16) ## if msg is for a row in freeze, ignore; should do at write
        output ("ignored rdma_write_sst msg because of frozen row: ", row, " node: ", G[row])
16
17
      wt_local_sst(row, col, val, index)
                                                ## write local SST
18
    def wt_local_sst(row, col, val, index=None): ## write local entry at row and col, at index if col is a list, with val
19
      if index is None:
                                              ## if index is None, meaning col is not a list
## just update SST entry with val
        setattr(sst[row], col, val)
set
20
21
22
      else:
                                              ## col is a list
        entry = getattr(sst[row], col)
entry[index] = val
setattr(sst[row], col, entry)
                                              ## retrieve SST entry at row and col
                                              ## update the element at row and col with updated entry
23
24
```

Figure 2: Specification of function for writing to SST.

```
1 def LogicalOr(col):
2 """logical 'OR' of all values in column col in sst"""
3 return any(getattr(sst[row], col) for row in range(len(sst)))
```

Figure 3: Specification of an example reducer function.

nodes execute a receive handler (lines 6-17), which calls function wt_local_sst to update the SST row corresponding to the sender node (lines 18-24).

The protocol uses a number of reducer functions on the SST. These functions read SST entries and compute aggregation information. These reducer functions are specified to compute exactly as stated in the pseudocode. Fig. 3 shows an example, LogicalOr.

4 Specifying steady-state execution

Steady-state execution describes atomic multicast delivery of a client request message across the nodes in a group. The messages are delivered in a round-robin fashion according to a global message order captured by global_index, where each node sends one multicast in each delivery cycle.

Fig. 4 shows a complete precise specification of steady-state execution in DistAlgo. It consists of three main parts: sending messages, receiving messages, and in-order delivery of messages.

The first part consists of functions send_req and get_buffer. Upon receiving a request message from a client, the receive handler (lines 1-8) is executed. We added a check to avoid processing duplicate requests (lines 4-7). Subsequently, function send_req is called, which uses function get_buffer (line 23) to obtain a slot in field slots to hold the request. Function get_buffer returns a pointer to the ring buffer if the request message previously in that slot has been successfully received on all the nodes in the group, and nullptr otherwise. However, in our specification, we return the slot number if the reservation is successful

```
def receive(msg=('request', req)): ## receive request from client, and send req to the group
       client, req_id, _ = req  ## request is of form (client, req_id, cmd)
output("request: ", req, " received from client: ", client)
if some(sent(('response', _req_id, res), to= _client)): ## if a response to req was already sent
output("Duplicate request received: ", req, ". Sending result: ", res, " back.")
send(('response', req_id, res), to= client)  ## send the response again
 2
 3
 6
          return
 8
       send_req(req)
                                           \ensuremath{\texttt{\#\#}} send request by putting it in the next slot, if a slot is available
     9
       11
12
                                                            completed_num = Min{sst[*].received_num[my_rank]};
if (sent_num - completed_num < window_size) {</pre>
13
14
15
                                                                          " completed_num: ", completed_num)
16
17
                                                           sst[my_rank].slots[slot].size = msg_size;
return sst[my_rank].slots[slot].buf; } ### return slot
18
       sst[my_rank].slots[slot].size = msg_size
19
                                                       #
       return slot
     20
21
22
       23
24
                                                              (p.34) ### added
                                                       # slot = (sent_num + 1) % window_size; ### redundant; what if None?
## # (p.34) the application writes the message contents in the buffer
# sst[my_rank].slots[slot].index++;
25
26
       sst[my_rank].slots[slot].buf = req
27
       sst[my_rank].slots[slot].index += 1
       28
29
     30
31
34
                                           slot = (sst[my_rank].received_num[i]+1)%window_size
                                     #
         # Slot = (sst[my_rank].received_num[i]+1) // window_size + 1:
# if (sst[i].slots[slot].index == (sst[my_rank].received_num[i]+1)/window_size+1) {
# if (sst[i].slots[slot].index == (sst[my_rank].received_num[i]+1)/window_size+1) {
35
36
           37
38
39
40
                                             recv(M(i, sst[my_rank].received_num[i])); }}
     # (p.34) A.3 Atomic Multicast Delivery in the Steady State
41
     # Å.3.1 Receive. ## received request msg, update msgs and related indices, but first send null msgs if needed
def recv(req, i, k): # on recv(M(i,k)) {
42
43
44
       ### the if-block below is added to avoid stalls by slow senders in delivery of received message,
45
       ### pseudocode here as in email from Sagar Jha to Vishnu Paladugu on 11/29/19, quoting an email by him dated 7/29/18
# if (I am a sender && this subgroup is not in unordered mode) { ### ignored
46
       if my_rank < i and k > sent_num:
    for _ in range(k - sent_num):
        output("sending no-op, case 1")
47 \\ 48
                                               # if (my_rank < i && I have not sent M(my_rank, k)) { ### used sent_num
## for every missing msg: ### do all at once to be more efficient</pre>
49
50 \\ 51
       for _ in range (k-1 - sent_num):
    output("sending no-op, case 2")
52
                                              ## for every missing msg: ### do all at once to be more efficient
53 \\ 54
            send_req(None)
                                                     send a null message }}
       msgs[gi(i, k)] = req # msgs[gi(M(i,k))] = M(i,k);
55
56
       57
58
59
       min_index_received, lagging_node_rank = min_and_idx(sst[my_rank].latest_received_index)
                                  (min_index_received, lagging_node_rank) =
    (min,argmin) i sst[my_rank].latest_received_index[i];
                              #
60
                              _ , .____recolved + 1 * len(G) + lagging_node_rank - 1)
# sst[my_rank].global_index = (min_index_received + 1) * |G| + lagging_node_rank - 1;
# }
61
       write\_sst(my\_rank, "global\_index", (min\_index\_received + 1) * len(G) + lagging\_node\_rank - 1)
62
                              #
63
64
     # (p.34) A.3.2 Stability and Delivery.
65
      ## deliver consecutive msgs that have been received by all nodes, in order of global index of the msgs
       66
     def stability_delivery():
67
68
       sorted kevs = sorted(msgs.kevs())
                                                 ## sort because msgs must be delivered in increasing global index
69
       for g_idx in sorted_keys:
                                                     for (msg : msgs) {
                                                 #
            g_idx <= stable_msg_idx:
70
71
                                                       if (msg.global_index <= min_stable_msg_index) { ### min_ deleted</pre>
         if
           deliver_upcall(g_idx, msgs[g_idx]) #
                                                          deliver_upcall (msg); ### add first argument, to see global_index in
         output
72
73
74
75
76
            del msgs[g_idx]
                                                         msgs.remove(msg.global_index); }}
                                                 #
       if sorted_keys and min(sorted_keys) <= stable_msg_idx:</pre>
                                                 ## if there stored reqs are less than received reqs
                                                     ### added test to not multicast unnecessarily as this is in an always
         write_sst(my_rank, "latest_delivered_index", stable_msg_idx)
                                                     sst[my_rank].latest_delivered_index = stable_msg_index }
```

Figure 4: Specification of steady-state execution.

1 2 3 4	<pre>def deliver_upcall(global_idx, req): output("in deliver_upcall(), gi: ", global_idx, if req is None: return (client, req_id, _) = req</pre>	<pre>## execute request req, at decided global index " and req: ", req) ## if request is a null msg for no-op, return ## request is form (client, req_id, cmd)</pre>
5 6 7 8	<pre>if not some(sent(('response', _req_id, _), to= . state, res = execute(global_idx, req, state) send(('response', req_id, res), to= client) output("response sent to the client/sim proces</pre>	<pre>client)): ## if request has not been responded to before ## execute request at global_idx in state ## send response to client s, index: ", global_idx, "response: ", res)</pre>
9 LO L1	<pre>def execute(global_idx, req, state): (_, req_id, _) = req return (state+[(global_idx, req)], req_id)</pre>	<pre>## execute the command in req in given state ## request is of form (client, req_id, cmd) ## return call history and req_id as new state and result</pre>

Figure 5: Specification of functions for delivering and executing requests.

and None otherwise (lines 16 and 19). The message is then written in all nodes by calling write_sst (line 28).

The second part uses functions receive_req and recv to receive incoming messages from other nodes. Function receive_req checks for new messages from other nodes in the group (lines 31-40). It is run continually to simulate "always" in the pseudocode, by using a nondeterministic random choice to select a function to run in node's run function's top-level infinite loop. Upon receiving a message, function recv is called, which stores the message in a dictionary data structure along with calculating global_index, which represents the highest global index of the message that can be safely delivered based on the local computation (lines 43-63).

The third part uses stability_delivery to deliver messages, in order of their global indexes, that have been received on all nodes (lines 66-77). The minimum of the global_index across nodes, known as the stable_msg_idx (line 67), is the index up to which messages can be safely delivered.

The pseudocode lacks a definition of function deliver_upcall. We added it, as shown in Fig. 5. In our specification, we have abstracted the atomic multicast delivery of a request message from its execution. Function deliver_upcall takes global_index and request and signifies the delivery of the request (lines 1-8). If a corresponding response has not been sent for a request, it calls the function execute (lines 9-11) to signify execution of the request. This approach helps prevent execution of duplicate requests that may arise due to client resends.

5 Specifying view change

Upon encountering the failure of a node, the group undergoes a membership change by employing a two-phase commit. A key part of the algorithm is leader selection. Fig. 6 shows its complete precise specification in DistAlgo.

The leader-selection algorithm is specified by two functions, find_new_leader (lines 1-4) and an "always" running function leader_selection (lines 6-25). Function find_new_leader selects the first non-suspected node as the leader. In function leader_selection, if a new leader selected is different from the current leader (line 8), it waits until all non-suspected nodes recognize it as the leader before continuing (lines 12-23).

```
def find_new_leader(r): # find_new_leader(r) {
        for i in range(len(curr_view.members)): # for (int i = 0; i < curr_view.max_rank; ++i) { ### max_rank replaced
if sst[r].suspected[i]: continue # if (sst[r].suspected[i]) continue;
else: return i # else return i }}
 2
 3
 4
        if new_leader == my_rank:
# all_others_agree = True
      # (p.35) ## update the current view, at the end, with the new leader
 5
 6
      def leader_selection():
         if new_leader == my_rank:
    # all_others_agree = True
                                                                      # if (new_leader != curr_view.leader_rank && new_leader == my_rank)
 9
                                                                    ### split 2 conjuncts, to add the else-branch for the second
# bool all_others_agree = true ### moved into while-loop
10
           ### if not moved, if it becomes False in for-loop below, it stays False, and the while-loop never stops
while find_new_leader(my_rank) == my_rank: # while (find_new_leader(my_rank) == my_rank) {
    --receive_messages  ## yield to receive msgs
11
12
13
                 ### needed to receive updates to SST which may result in new leader selection ### break atomicity
14
15
                all_others_agree = True
                                                                     ### moved here from outside while-loop, as explained above
16 \\ 17
                for r in range(len(sst)):
    if not sst[my_rank].suspected[r]:
                                                                      #
#
                                                                             for (r: SST.rows) {
                                                                                if (sst[my_row].suspected[r] == false)
18
19
                      all_others_agree = all_others_agree and (find_new_leader(r) == my_rank)
                                                              # all_others_agree &&= (find_new_leader(r) == my_rank) }
# if (all_others_agree) {
20
                if all_others_agree:
21
22
                                                                               curr_view.leader_rank = my_rank;
                   curr_view.leader_rank = my_rank
                                                                      #
                   output("I am the new leader!!!")
23
                                                                      #
                                                                               break; }}}
                   break
                                                                      ## else: ### added else-branch, for when new leader is not self
## set current view's leader to be new leader
24
           else:
25
              curr_view.leader_rank = new_leader
```

Figure 6: Specification of leader selection.

6 Runtime checking and analysis

6.1 Manual inspection and automated checking

To help ensure the correctness of the protocol specification, we perform careful manual inspection automated testing, and increasingly systematic runtime checking of safety and progress properties, and repeat this process for each anomaly and improvement discovered until all inspections, tests, and checks pass. This approach led to a complete precise specification in DistAlgo, after filling in missing details in the English and pseudocode description and resolving additional issues.

The testing and checking methodology consists of configuring and executing the protocol with varying numbers of requests and member nodes, ring buffer size, etc., as well as introducing random node failures. The systematic runtime checking was enabled by a general framework in DistAlgo for runtime checking of safety and liveness properties without touching the complete protocol specification [LS20].

6.2 Properties checked

For property checking, the following messages are used:

- p.sent('deliver_upcall', i, req, t) for p calling deliver_upcall(i, req) at time t
- p.sent('execute', i, req) for p calling execute(i, req)
- p.receive('request', req) for message ('request', req) received by p

An important property discussed in the paper [JBG⁺19a] is the round-robin message delivery.

Delivery ordering. "Derecho uses a simple round-robin delivery order: Each active sender can provide one multicast per delivery cycle, and the messages are delivered in round-robin manner. The global index of M(i, k), gi(M(i, k)) is the position of this message in the round-robin ordering."

```
each p.sent('deliver_upcall', i1, req1, t),
    p.sent('deliver_upcall', i2, req2, t2)
    has (not i2>i1 or t2>t)
```

We also check the following well-known properties, taken and quoted exactly from Paxos-SB [KA08], except that an update in Paxos-SB corresponds to a client request, and a server is a node process in Derecho.

Validity. "Only an update that was introduced by a client (and subsequently initiated by a server) may be executed."

```
each p.sent('execute', i, req)
     has some p1.received('request', _req)
```

Agreement. "If two servers execute the ith update, then these updates are identical."

Uniform integrity. "If a server executes an update on sequence number i, then the server does not execute the update on any other sequence number i' > i."

each p.sent('execute', i, req) has not some =p.sent('execute', i2, =req) has i2>i

Additionally, we use aggregation queries to specify and check important progress properties, e.g., the total number of executed request equals the total number of client requests.

```
count(req: p.receive('request', req)
= count(req: p.sent('execute', i, req)
```

6.3 Issues found and fixed

Our specification and checking approach—by following the pseudocode exactly, facilitated by the already detailed pseudocode of Derecho—has also led to finding and fixing some issues in the pseudocode. Many of these issues were difficult to identify in the original pseudocode due to usual problems with pseudocode, compounded with the complexity of Derecho, but became evident after the specification in DistAlgo. While most of the issues were easy to find and fix, others were not.

Some initial issues (e.g., typos) and more were already addressed in an errata [JBG⁺19b, Errata, page 50] and a dissertation [Jha22], and some others (e.g., adding null messages to prevent stalls, Fig. 4 lines 47-54) were resolved with help from Derecho developers [Jha19, Jha23]. In all cases, Derecho developers have checked and confirmed that these bugs are not in their implementation in C++.

The first bugs [JBG⁺19b, Errata, page 50] were found mostly by manual inspection, when writing and examining the DistAlgo specification by following the text description of the logic and the pseudocode of the protocol and cross checking. The rest were essentially all discovered first by automated testing and checking and then by manual inspection, adding tests and checks, and running again to confirm. The fixes proposed passed all tests, checks, and inspections.

We discuss two examples issues discovered and fixed, for steady-state execution and view change, respectively. Despite being minor in hindsight, such issues were tricky to discover due to complex control flows from high nondeterminism. Both helped improve Derecho's pseudocode [Jha22, pages 144 and 149].

Overwriting in ring buffer. In steady-state execution, field slots in SST stores requests in a ring buffer of size window_size (Fig. 1, line 13). To prevent overwriting a slot, get_buffer should return nullptr if the number of pending messages equals the buffer size, i.e., sent_num (number of messages sent by the node) - completed_num (number of sent messages that have been received by all nodes) = window_size (Fig. 4 line 13). Instead, the check uses "<" originally [JBG⁺19a, Sec. A.2.3, page 33] and ">" in a later errata [JBG⁺19b, Errata, page 50]).

Note that the improved Derecho pseudocode uses \geq [Jha22, page 144], which is also correct and was how we first proposed to fix. Using "=" is more precise, because for the protocol to be correct, ">" should never happen.

This is clearly a minor bug, and finding the "fix" in the errata was relatively quick. However, discovering more issues after that and debugging them were highly involved, but solved with help by both manual inspection and automatic checking. The bug manifested as a deadlock in function receive_req (Fig. 4, line 35) for nodes that did not receive the previous message in the slot. It only happens when the ring buffer is full, after a long execution trace of member nodes handling many client requests, even for a small buffer size.

Deadlock in leader selection. Leader selection (Fig. 6) selects the first non-suspected node (lines 1-4) as the new leader and uses variable all_others_agree to track a logical conjunction checking if all nodes agree with the new leader (lines 10, 18, and 20). We discovered that if a node did not initially agree, i.e., find_new_leader returned a different leader, all_others_agree would be set to False (line 18) on the new leader's node, and cause a deadlock because all_others_agree can never be set to True again.

To fix this, we (1) move the first write to all_others_agree from outside (line 10) to inside (line 15) of the while, to reset it in each iteration, and (2) add an else branch to update the new leader for non-leader nodes (lines 24-25).

6.4 Resulting specification and direct execution

Table 1 shows the size of specification of Derecho in DistAlgo.

DistAlgo specifications can be run directly. For example, Derecho specification in a file derecho.da (Appendix A) can be executed with Python 3.7 by simply running pip install pyDistAlgo to install DistAlgo and then running python -m da derecho.da. To run the system with a failure of a node injected, set test_failure to True in method main.

The specification with steady-state and failure-induced membership change runs smoothly.

Table 1: Specification size (in number of lines, including output lines, excluding empty or comment-only lines) for Derecho specification in DistAlgo (derecho.da in Appendix A excluding method main and class Sim).

Protocol component	Size
state and helper functions	95
steady-state execution, incl. delivering&executing reqs	63
view change	132
imports, helper, choices in run	14
total	304

For instance, with 1000 requests, three nodes, one client, window-size 400 and no failure, the average time over 10 runs was 4.8 seconds, measured on a 2.6 GHz 6-Core Intel Core i7 CPU with 16 GB RAM running macOS Ventura and Python 3.7.12. A complete sample run for three nodes, one client, and ten requests, with window-size as ten can be found in Appendix B.

Note that this is essentially the same speed for runtime checking. Our specification is not currently optimized for efficiency; it directly runs many reducer functions aggregations, sorting, and output constantly, each taking linear time or worse. These high-level functions are essential for ensuring correctness. The efficiency of such expensive functions can be improved drastically, asymptotically, by using incrementalization, as in [LSL17]. Our goal in this work is to develop a complete, precise, and correct specification.

7 Related work and conclusion

Significant efforts have been devoted to specifying, testing, analyzing, checking, and verifying distributed algorithms, as evidenced by works such as [HHK⁺15, CLS16, PLSS17]. Furthermore, various specification languages and verification tools, such as TLA+ Toolbox [Lam94, Lam02, Mic] and Ivy [MP20], have been developed to aid in this task. Another area of focus is verifying the executable specifications of distributed consensus protocols, as demonstrated by projects like IronFleet [HHK⁺15] and Verdi [WWP⁺15]. For example, Verdi [WWA⁺16] provides a verified implementation of Raft in Coq [Coq] with 50,000 lines of proof. Nonetheless, the challenge remains in the significant development efforts and expertise required for such verification.

Runtime verification (RV) is a useful tool for ensuring the correct functioning of complex distributed algorithms and their executable specifications, especially for real-world applications written in general-purpose programming languages, where manual verification can be prohibitively difficult. Several RV tools have been developed, including MoP [CR05, CR07] and ELarva [CFG12]. However, they either have not been applied to general distributed systems or have more complex property specification and checking processes. Although WidsChecker [LLPZ07] found bugs in Paxos' IO automata specification, it is tightly coupled with programs developed using WidsToolkit, is not publicly available, and requires complex specification scripts. In contrast, our runtime verification framework [LS20] enables clear and precise high-level specifications of properties, allowing for the detection of subtle bugs in protocol pseudocode and earlier specifications.

Previous research has explored different approaches for producing executable code from formal specifications, such as from process algebras [HCS01], I/O automata [GLMT09], and various high-level languages, like Dedalus [AMC⁺11], Bloom [ACHM11], EventML [Bic09], and DAHL [LNRS10]. However, DistAlgo [LSLG12, LSL17] stands out as a language specifically designed for easy and precise expression and direct execution of distributed algorithms. Additional work using DistAlgo includes many distributed algorithms specified in DistAlgo, e.g., [LSL17] and on github (https://github.com/search?q=distalgo), automatic transformations of Event-B models [Gra20] into DistAlgo, as well as uses in various BS and MS theses, e.g., [Wid20, Laz21, Shi22].

For our study, we selected Derecho due to its importance in various high-speed data replication in intelligent IoT edge applications [SYL⁺22], with optimization techniques like Spindle [JRB22] improving bandwidth utilization for small messages. Nevertheless, compared to multi-Paxos [VRA15], with a corresponding executable specification in DistAlgo [LCS19], Derecho is much more complex with many more pieces of information maintained in much more sophisticated control flows and with less complete pseudocode. This made it challenging to determine some of the missing details in the specification.

Our directly executable specification closely corresponds to the protocol pseudocode, and this helped us better understand the protocol, leading us to identify missing details required for a complete, precise, executable specification. We use runtime checking to check important safety and progress properties of the protocol. Future work includes the incorporation of various fault-injection testing methods with the current runtime verification framework, use of the specification in DistAlgo to help with proof development [RJB21] for both safety and liveness, as well as automated ways to correlate formal specifications in DistAlgo with efficient implementations in lower-level languages such as C++.

Acknowledgments

We thank the Derecho team, Ken Birman and Sagar Jha in particular, for their prompt replies to our inquiries and their detailed and helpful explanations about the Derecho protocol. We thank Thejesh Arumalla for careful study of work by the Derecho team and greatly helpful questions and discussions. This work was supported in part by NSF under grant CCF-1954837 and ONR under grant N00014-21-1-2719.

References

[ACHM11] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. Consistency analysis in bloom: a calm and collected approach. In *Conference* on Innovative Data Systems Research, 2011.

- [AMC⁺11] Peter Alvaro, William R. Marczak, Neil Conway, Joseph M. Hellerstein, David Maier, and Russell Sears. Dedalus: Datalog in time and space. In Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Sellers, editors, *Datalog Reloaded*, pages 262–281, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
- [Bic09] Mark Bickford. Component specification using event classes. In Grace A. Lewis, Iman Poernomo, and Christine Hofmeister, editors, *Component-Based Software Engineering*, pages 140–155, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
- [BJ87] Kenneth P. Birman and Thomas A. Joseph. Exploiting virtual synchrony in distributed systems. *SIGOPS Oper. Syst. Rev.*, 21(5):123–138, November 1987.
- [CFG12] Christian Colombo, Adrian Francalanza, and Rudolph Gatt. Elarva: A monitoring tool for Erlang. In Sarfraz Khurshid and Koushik Sen, editors, *Runtime Verification*, pages 370–374, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
- [CL21] Saksham Chand and Yanhong A. Liu. Brief announcement: What's live? understanding distributed consensus. pages 565–568, July 2021.
- [CLS16] Saksham Chand, Yanhong A. Liu, and Scott D. Stoller. Formal verification of multi-paxos for distributed consensus. In John Fitzgerald, Constance Heitmeyer, Stefania Gnesi, and Anna Philippou, editors, *FM 2016: Formal Methods*, pages 119–136, Cham, 2016. Springer International Publishing.
- [Coq] Coq, a formal proof management system. https://coq.inria.fr/.
- [CR05] Feng Chen and Grigore Roşu. Java-mop: A monitoring oriented programming environment for java. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, 2005.
- [CR07] Feng Chen and Grigore Roşu. Mop: An efficient and generic runtime verification framework. In Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOP-SLA '07, page 569–588, New York, NY, USA, 2007. Association for Computing Machinery.
- [der] Derecho distalgo github repository. https://github.com/unicomputing/derechodistalgo.
- [GLMT09] Chryssis Georgiou, Nancy A. Lynch, Panayiotis Mavrommatis, and Joshua A. Tauber. Automated implementation of complex distributed algorithms specified in the ioa language. International Journal on Software Tools for Technology Transfer, 11:153–171, 2009.

- [Gra20] Alexis Grall. Automatic generation of distalgo programs from event-b models. In Alexander Raschke, Dominique Méry, and Frank Houdek, editors, *Rigorous State-Based Methods*, pages 414–417, Cham, 2020. Springer International Publishing.
- [HCS01] D. Hansel, R. Cleaveland, and S.A. Smolka. Distributed prototyping from validated specifications. In *Proceedings 12th International Workshop on Rapid System Prototyping. RSP 2001*, pages 97–102, 2001.
- [HHK⁺15] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. Ironfleet: Proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 1–17, New York, NY, USA, 2015. Association for Computing Machinery.
- [JBG⁺19a] Sagar Jha, Jonathan Behrens, Theo Gkountouvas, Matthew Milano, Weijia Song, Edward Tremel, Robbert Van Renesse, Sydney Zink, and Kenneth P. Birman. Derecho: Fast state machine replication for cloud services. ACM Trans. Comput. Syst., 36(2), April 2019.
- [JBG⁺19b] Sagar Jha, Jonathan Behrens, Theo Gkountouvas, Matthew Milano, Weijia Song, Edward Tremel, Robbert van Renesse, Sydney Zink, and Kenneth P. Birman. Derecho: Fast state machine replication for cloud services. ACM Trans. Comput. Syst., 36:4:1–4:49, 2019. with Errata, page 50, Nov. 2019. https://www.cs.cornell.edu/ken/derecho-tocs.pdf.
- [Jha19] Sagar Jha. Re: Null sends, November 27 2019. Email, with Vishnu Paladugu, forwarding an email dated Jul 29, 2018.
- [Jha22] Sagar Jha. RDMA-accelerated state machine for cloud ser-Cornell University, vices. PhD thesis. Ithaca, NY, 122022.https://www.cs.cornell.edu/projects/Quicksilver/public_pdfs/dissertation.pdf.
- [Jha23] Sagar Jha. Re: Understanding the derecho's view-change algorithm, April 30 2023. Email with Kumar Shivam.
- [JRB22] Sagar Jha, Lorenzo Rosa, and Ken Birman. Spindle: Techniques for optimizing atomic multicast on rdma. In 2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS), pages 1085–1097, 2022.
- [KA08] Jonathan Kirsch and Yair Amir. Paxos for system builders: An overview. In Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware, LADIS '08, New York, NY, USA, 2008. Association for Computing Machinery.

- [Lam94] Leslie Lamport. The temporal logic of actions. ACM Trans. Program. Lang. Syst., 16(3):872–923, may 1994.
- [Lam98] Leslie Lamport. The part-time parliament. ACM Trans. Comput. Syst., 16(2):133–169, may 1998.
- [Lam02] Leslie Lamport. Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley Longman Publishing Co., Inc., USA, 2002.
- [Laz21] Aleksandar Lazic. The library of distributed protocols. Master's thesis, University of Fribourg, 2021.
- [LCS19] Yanhong A. Liu, Saksham Chand, and Scott D. Stoller. Moderately complex paxos made simple: High-level executable specification of distributed algorithms. In Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming, PPDP '19, New York, NY, USA, 2019. Association for Computing Machinery.
- [LLPZ07] Xuezheng Liu, Wei Lin, Aimin Pan, and Zheng Zhang. WiDS checker: Combating bugs in distributed systems. In 4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07), Cambridge, MA, April 2007. USENIX Association.
- [LLS17] Yanhong A. Liu, Bo Lin, and Scott Stoller. DistAlgo Language Description. http://distalgo.cs.stonybrook.edu, March 2017.
- [LNRS10] NUNO P. LOPES, JUAN A. NAVARRO, ANDREY RYBALCHENKO, and ATUL SINGH. Applying prolog to develop distributed systems. *Theory and Practice of Logic Programming*, 10(4-6):691–707, 2010.
- [LS20] Yanhong A. Liu and Scott D. Stoller. Assurance of distributed algorithms and systems: Runtime checking of safety and liveness. In Jyotirmoy Deshmukh and Dejan Ničković, editors, *Runtime Verification*, pages 47–66, Cham, 2020. Springer International Publishing.
- [LSL17] Yanhong A. Liu, Scott D. Stoller, and Bo Lin. From clarity to efficiency for distributed algorithms. ACM Transactions on Programming Languages and Systems, 39(3):12:1–12:41, May 2017.
- [LSLG12] Yanhong A. Liu, Scott D. Stoller, Bo Lin, and Michael Gorbovitski. From clarity to efficiency for distributed algorithms. In Proceedings of the 27th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications, pages 395–410, 2012.
- [Mic] Microsoft research. the tla toolbox. http://lamport.azurewebsites.net/tla/toolbox.html.

- [MP20] Kenneth L. McMillan and Oded Padon. Ivy: A multi-modal verification tool for distributed algorithms. In Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part II, page 190–202, Berlin, Heidelberg, 2020. Springer-Verlag.
- [OL88] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of* the Seventh Annual ACM Symposium on Principles of Distributed Computing, PODC '88, page 8–17, New York, NY, USA, 1988. Association for Computing Machinery.
- [PLSS17] Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. Paxos made epr: Decidable reasoning about distributed protocols. Proc. ACM Program. Lang., 1(OOPSLA), oct 2017.
- [RJB21] Lorenzo Rosa, Sagar Jha, and Ken Birman. DerechoDDS: Efficiently leveraging RDMA for fast and consistent data distribution. In CARS 2021 6th International Workshop on Critical Automotive Applications: Robustness & Safety, Münich, Germany, September 2021.
- [Shi22] Kumar Shivam. Specification and runtime checking of algorithms for replication and consensus in distributed systems. Master's thesis, Stony Brook University, 2022.
- [SYL⁺22] Weijia Song, Yuting Yang, Thompson Liu, Andrea Merlina, Thiago Garrett, Roman Vitenberg, Lorenzo Rosa, Aahil Awatramani, Zheng Wang, and Ken Birman. Cascade: An edge computing platform for real-time machine intelligence. In Proceedings of the 2022 Workshop on Advanced Tools, Programming Languages, and PLatforms for Implementing and Evaluating Algorithms for Distributed Systems, ApPLIED '22, page 2–6, New York, NY, USA, 2022. Association for Computing Machinery.
- [VRA15] Robbert Van Renesse and Deniz Altinbuken. Paxos made moderately complex. ACM Comput. Surv., 47(3), feb 2015.
- [Wid20] Roland Widmer. Byzantine-fault tolerant algorithms in DistAlgo. bachelors thesis, 2020.
- [WWA⁺16] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. Planning for change in a formal verification of the raft consensus protocol. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2016, page 154–165, New York, NY, USA, 2016. Association for Computing Machinery.
- [WWP⁺15] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: A framework for implementing

and formally verifying distributed systems. SIGPLAN Not., 50(6):357-368, jun 2015.

A Derecho executable specification in DistAlgo

1 # This is a DistAlgo implementation of Derecho, as described in Sagar Jha, Jonathan Behrens, Theo Gkountouvas, Matthew Milano, Weijia Song, 3 # Edward Tremel, Robbert van Renesse, Sydney Zink, and Kenneth P. Birman. 4 # "Derecho: Fast State Machine Replication for Cloud Services", 5 # ACM Transactions on Computer Systems, Vol. 36, No. 2. Article 4. March 2019. 6 # http://www.cs.cornell.edu/ken/derecho.pdf 7 # In the code below, the following convention for comments are used: 8 # 1. comments after # are text or pseudocode copied from the paper, 9 # except a block from an email from Sagar Jha as noted in function recv. 10 # 2. comments after ## (or no comments) indicate code we had to fill in. $11\,$ # 3. comments after ### indicate changes to pseudocode in paper. 12 import sys ## for taking command line arguments ## for taking a random choice among multiple actions 13 import time 14 import random 15 import os ## for getting a bytestring of random bytes of a given size 16 def min_and_idx(l):
17 """min of list 1 and index of first min element""" $m = \min(1)$ return m, l.index(m) 19 20~ # (p.12-13) 3.4 Shared State Table: The SST (par.2) 21~# Derecho uses protocols that run on a novel replicated data structure 22~# that we call the shared state table, or SST. 23 # The SST offers a tabular distributed shared memory abstraction.
24 # Every member of the top-level group holds ## top-level not in key protocol steps
25 # its own replica of the entire table, in local memory. 26 # Within this table, there is one identically formatted row per member. 27 # A member has full read/write access to its own row but $28\,$ # is limited to read-only copies of the rows associated with other members. 29 # (p.32) Appendix PSEUDO-CODE FOR KEY PROTOCOL STEPS (p.32-38) 30 # A 31 # A.1 Notation 32 # A.1.1 SST 33 # column_name ->string|string[int] // e.g. wedged or latest_received_index [3] 34 # sst_row->sst[row_rank] ## row_rank is index of the row in sst 35 # row_rank->int ## index of row is in 0..len(sst) 36 # sst_column ->sst[*].column_name 37 # sst_entry->sst_row.column_name // e.g. sst[0].stable_msg_index[0] 38 # A rank of a member is the index of its row in the SST. 39 # The code shown below is run by every process, but each has a distinct rank (referred to as my_rank). 40 # (p.33) A.2 SMC 41 # In what follows, we begin by presenting the SST multicast (SMC), 41 # In what follows, we objand by proceeding the test of the second the membership management protocol that follows, we obtain a full Paxos. 44 # $45~\mbox{\sc \#}$ RDMC could be similarly formalized but is omitted for brevity. 46 # 47# A.2.1 SST Structure. SMC uses two fields, slots and received_num 48 # slots is a vector of window_size slots, 49 # each of which can store a message of up to max_message_size characters. 50 # The index associated with a slot is used to signal that 51 # a new message is present in that slot: 52 # For example, if a slot's index had value k and transitions to k + 1, 53 # then a new message is available to be received. 54 # The vector received num holds $55~{\rm \#}$ counters of the number of messages received from each node. 56 class Slot: """A slot stores a client request message. A vector of slots is a field in SST""" 57""A Slot stores a creat request message. A vector of slots is a first is in the first is a first is a first is a first initialized in Node.init()
self.buf = None # (p.33) index associated with a slot ## initialized in Node.init()
self.size = 0 # ## size of the request, set in get_buffer() ### defined but not used 58 def 59 60 61 62 class SSTRow: 63 """A shared state table (SST) row that stores info about a node. 64 A SST has a SSTRow for each member node and is stored in each member""" 65 def __init__(self, n, window_size): ## initialize SST columns for the row; all used in pseudocode but the last two 66 # n: # n: ## number of member nodes in the group
window_size: ## length of vector of slots for storing client req msgs received by the node, directly or indirectly 67 self.slots = [Slot() for _ in range(window_size)] # (p.33) vector of window_size slots ## for client request msgs self.received_num = [-1] * n # (p.33) number of messages received from each node ### number-1 ## initialized in Node.init(), and set in receive_req() 68 70 # ## global index of last message received from the most lagging node
min of self.global_index over all members $71 \\ 72$ self.global_index = -1 self.latest_delivered_index = -1 73 74 self.latest_received_index = [-1] * n # ## index of latest msg received from each node, set in recv to received_num ## i.e., self.received_num-1 ### redundant, but not clear with null msgs

self.min_latest_received = [-1] * n # ## for each node, min of latest_received_index over all rows in SST 76 77 78 79 80 self.num_acked = 0 # ## number of nodes in self.changes acknowledged 81 ### set in 1 place by us, using num_changes self.num_committed = 0 # ## min of self.num_acked over not suspected nodes 82 self.num_installed = 0 # ## number of nodes installed (added/removed) by the node, as proposed by the leader 83 84 self.ragged_edge_computed = False # ## true for leader calling terminate_epoch or others after leader did; ## the call happens when leader's num_committed > self's num_installed 85 self.active = False # (p.40) ## true when the epoch is active, only used at start ### not in pseudocode 86 87 ### could use logical or over sst[my_rank].suspected or even just own suspected 88 class View(): """A view that holds the information of an epoch. An epoch is the duration of a view.""" def __init__(self, n, epoch=0, leader_rank=0): # n: ## number of members in the view 89 90 91 92 93 self.members = [None] * n
self.failed = [False] * n 94 # ## list of member nodes in the view # ## for each node, whether that node is suspected and thus considered failed 95 96 def add_member(self, node): ## add member to the view 97 self.members.append(node) ## append node to members ## add the failed attribute corresponding to the added node 98 self.failed.append(False) 99 def remove member(self. node): ## remove node from members of the view index = self.members.index(node) ## get index of node in members 100 del self.failed[index] ## remove failed entry for node
remove node from members self.members.remove(node) 103 class Node(process): 104 ## simulate Derecho's write to SST using RDMA: ## each node owns a row in the SST and is the only writer of the row; ## each write to local SST must be multicasted to all other nodes to update their copies of the row. 106 def write_sst(row, col, val, index=None): ## write SST entry at row and col, at index if col is a list, with val 108 109 111 def receive(msg = (_, 'rdma_write_sst', row, col, val, index, epoch)): ## _ ignores tag 'data' or 'control'
output("received: ", ('rdma_write_sst', row, col, val, index, epoch))
if epoch != curr_view.epoch: # (p.40) ## if msg is for a different epoch, ignore
output('received rdma_write_sst msg for different epoch, current: ', curr_view.epoch, ' and msg: ', epoch) 112 113 114 return if col == "slots" and any(curr_view.failed): # (p.12) ## if msg is a req and curr view has failed members, ignore 117 output("Failure detected, new data messages dropped") 118 119 return 120 if row in freeze: # (p.16) ## if msg is for a row in freeze, ignore; should do at write output("ignored rdma_write_sst msg because of frozen row: ", row, " node: ", G[row]) wt_local_sst(row, col, val, index) ## write local SST 124 def wt_local_sst(row, col, val, index=None): ## write local entry at row and col, at index if col is a list, with val ## if index is None, meaning col is not a list
just update SST entry with val 125if index is None: setattr(sst[row], col, val) 126 127 ## col is a list ## retrieve SST entry at row and col
update the element at index of entry with val
update SST entry at row and col with updated entry 128 entry = getattr(sst[row], col) 129 entry[index] = val setattr(sst[row], col, entry) 130 131 ## functions below are called in pseudocode; they are here because sst, G, and my_rank are defined in Node # (p.32, under A.1.1) reducer function, for example, def Min(col, index=None): # (p.32) Min(sst_column) represents the minimum of all the entries of the column. 133 134 136 return min(getattr(sst[row], col) for row in range(len(sst))) else return min(getattr(sst[row], col)[index] for row in range(len(sst))) def NotFailed(): # (p.32) NotFailed is a filtering function that removes the rows that are suspected, from the column
return [row for row in range(len(sst)) if not sst[my_rank].suspected[row]] 139 140 def MinNotFailed(col): # (p.32) MinNotFailed(sst_column) is a Min(NotFailed(sst_column)) ### direct Min not work 141 "min of all values of column col for non-suspected nodes in sst"" 142143 return min(getattr(sst[row], col) for row in NotFailed()) 144 def LogicalAndNotFailed(col): """logical 'AND' of all values in column col for non-suspected nodes in sst""" return all(getattr(sst[row], col) for row in NotFailed()) 145 146 147def LogicalOr(col):

148 """logical 'OR' of all values in column col in sst"""

```
149
           return any(getattr(sst[row], col) for row in range(len(sst)))
150
       def Count(col, val): # (p.32) Count(sst_column, value) counts the number of entries that are equal to value.
               count of rows in sst where column col has value val""
         return len([row for row in range(len(sst)) if getattr(sst[row], col) == val])
                                                                                                                         ### this follows the English but
152 #
           return len([i for i in range(len(sst)) if getattr(sst[my_rank], col)[i] == val]) ### this is intended from use
153
154
        def min_with_val(col, val):
           """min rank of row in sst where column col has value val"""
return min(row for row in range(len(sst)) if getattr(sst[row], col) == val)
156
157
        def max_gi(): # (at call) max over n of (sst[my_rank].min_latest_received[n] * |G| + n)"""
            """max value of the global index of all the messages in the group,
158
           return max((sst[my_rank].min_latest_received[i] * len(G) + i) for i in range(len(G)))
        160
          """request msg in vector of slots given sender's rank i and sender's message index k"""
return sst[i].slots[k%window_size].buf
163
        def gi(i, k): # The global index of M(i, k), gi(M(i, k)) is the position of this message in the round-robin ordering.
165
           ""global message index of a message given sender's rank i and sender' message index k return k * len(G) + i \# gi(M(i, k)) = i * |G| + k \# \# bug fixed, in ERRATA of paper
166
167
168
        # (p.33) A.2.2 Initialization
        def initialize(): ## initialize SST and other fields, using G and window_size
self.n = len(G)  # ## number of nodes in group G
           self.n = len(G)  # ## number of nodes in group G
self.sst = [SSTRow(n, window_size) for _ in range(n)]
171
172
                                      # for i in 1 to n {
                                           for j in 1 to n {
                                      #
173
174
                                              sst[i].received_num[j] = -1; }
                                           for k in 1 to window_size {
   sst[i].slots[k].buf = nullptr
176
                                              sst[i].slots[k].index = 0 }}
177
           self.sent_num = -1 # sent_num = -1 ## number of messages sent by this node - 1
self.msgs = {} # ## dict of requests received but not yet executed, indexed by global index
self.freeze = set() # (p.16 ln 1, p.41 B.3 ln 3-6) SST rows of failed members ### is a call in pseudocode
178
179
180
181
           ## set of rows of members sensed or suspected failed ### could use sst[my_rank].suspected
self.others = set(G)-{self} ## set of other nodes in the group
182
           return n, sst, sent_num, msgs, freeze, others
183
        def setup(G, my_rank, window_size, max_msg_size, state):
    # G:    ## list of nodes in the group
    # my_rank:    ## index of this node in the list of nodes
    # window_size:    ## size of the vector of slots for SST
    # max_msg_size:    ## max message size, in number of bytes
    ## of the vector of slots.
184
185
186
187
188
189
           ## state: history of states of the application
           self.n, self.sst, self.sent_num, self.msgs, self.freeze, self.others = initialize()
self.curr_view = View(n)  # ## current view
self.curr_view.members = G  # ## members of current view, set to G
190
191
192
           output('initial group: ', G)
193
194
        def run():
          write_sst(my_rank, "active", True) # (p.40) ## mark the epoch
await(LogicalAndNotFailed("active")) # (p.40) ## wait for the members to be active
195
196
197
           while True:
198
              --receive_messages ## yield to receive messages
              choice = random.choice(['recv', 'stable', 'suspect', 'elect', 'other'])
if choice == 'recv': receive_req()  # always
elif choice == 'stable': stability_delivery()  # always
199
200
201
              elif choice == 'suspect': suspect()
elif choice == 'elect': leader_selection()
202
                                                                               # alwavs
203
                                                                               # always
              # (p.36) A.4.2 Terminating old view and installing new view after wedging.
204
205
              elif sst[curr_view.leader_rank].num_changes > sst[my_rank].num_acked:
                                                                    # when (sst[leader_rank].num_changes > sst[my_rank].num_acked) {
    # when (sst[leader_rank].num_changes > sst[my_rank].num_acked) {
    ## ### added definition needed for uses below
    # if (curr_view.leader_rank != my_rank) {

206
                leader_rank = curr_view.leader_rank
207
208
                 if leader_rank != my_rank:
                   i leader_rank := my_rank: # 1 (curr_view.leader_rank != m
output("leader: ", G[leader_rank], " proposed a new change")
output("changes list received from leader is: ", sst[leader_rank].changes)
209
210
211
                   write_sst(my_rank, "num_changes", sst[leader_rank].num_changes)
                                                                                  sst[my_rank].num_changes = sst[leader_rank].num_changes;
212
                                                                         #
213
                   write_sst(my_rank, "changes", sst[leader_rank].changes)
214
                                                                                  sst[my_rank].changes = sst[leader_rank].changes;
                  215
216
                                                                                  curr_view.wedge(); ### not defined
sst[my_rank].wedged = true;
                                                                          #
218
                  write_sst(my_rank, "wedged", True)
                                                                         #
                                                                          # }}
219
220
                write_sst(my_rank, "num_acked", sst[leader_rank].num_changes)
                                                                          ### acknowledge the changes ### missing but needed to terminate
### added based on email from Sagar Jha to Shivam Kumar on 4/30/23
              elif (curr_view.leader_rank == my_rank and # when (curr_view.leader_rank == my_rank and
223
224
                      MinNotFailed("num_acked") > sst[my_rank].num_committed):
```

```
23
```

225MinNotFailed(sst[*].num_acked) > sst[my_rank].num_committed) { # 226 output("commit_proposal_leader: ", G[curr_view.leader_rank])
write_sst(my_rank, "num_committed", MinNotFailed("num_acked")) 227 228 # sst[my_rank].num_committed = MinNotFailed(sst[*].num_acked);
} 230 elif sst[curr_view.leader_rank].num_committed > sst[my_rank].num_installed: 231 # when (sst[my_rank].num_committed[leader_rank] > sst[my_rank].num_installed[my_rank]) { output("leader: ", G[curr_view.leader_rank], " committed a new membership change") 232 # curr_view.wedge(); ### not defined # sst[my_rank].wedged = true; 233 234 write_sst(my_rank, "wedged", True) await(LogicalAndNotFailed("wedged")) # when (LogicalAndNotFailed(sst[*].wedged) == true) { 235236 terminate_epoch() terminate_epoch(); }} 237def receive(msg=('request', req)): ## receive request from client, and send req to the group 238 client, req_id, _ = req ## request is of form (client, req_id, cmd)
output("request: ", req, " received from client: ", client) if some(sent(('response', _req_id, res), to= _client)): ## if a response to req was already sent 240output("Duplicate request received: ", req, ". Sending result: ", res, " back.")
send(('response', req_id, res), to= client) ## send the response again 241 242 243 244send reg(reg) ## send request by putting it in the next slot, if a slot is available 245 # (p.33) A.2.3 Sending. First the sending node reserves one of the slots: def get_buffer(msg_size):
 assert msg_size <= max_msg_size</pre> 246247 completed_num = Min{sst[*].received_num[my_rank]}; if (sent_num - completed_num < window_size) {</pre> 248 249251252 253254# return sst[my_rank].slots[slot].buf; } ### return slot 255 # (p.34) After get_buffer returns a non-null buffer, # the application writes the message contents in the buffer and calls send 256257258 def send_req(req): 260 # slot = (sent_num + 1) % window_size; ### redundant; what if None? ## # (p.34) the application writes the message contents in the buffer 261 sst[my_rank].slots[slot].buf = req
sst[my_rank].slots[slot].index += 1 262# sst[my_rank].slots[slot].index++; 263 264 265 266 267268 slot = (sst[my_rank].received_num[i]+1) % window_size 269270 slot = (sst[my_rank].received_num[i]+1)%window_size # 271 272273 274 ++sst[my_rank].received_num[i]; # recv(M(i, sst[my_rank].received_num[i]), i, sst[my_rank].received_num[i])
recv(M(i, sst[my_rank].received_num[i])); }} # (p.34) A.3 Atomic Multicast Delivery in the Steady State 277# Å.3.1 Receive. ## received request msg, update msgs and related indices, but first send null msgs if needed def recv(req, i, k): # on recv(M(i,k)) { 278 279 ### the if-block below is added to avoid stalls by slow senders in delivery of received message, 280 ### pseudocode here as in email from Sagar Jha to Vishnu Paladugu on 11/29/19, quoting an email by him dated 7/29/18 # if (I am a sender && this subgroup is not in unordered mode) { ### ignored 281 282 283 if my_rank < i and k > sent_num:
 for _ in range(k - sent_num): # if (my_rank < i && I have not sent M(my_rank, k)) { ### used sent_num
for every missing msg: ### do all at once to be more efficient</pre> 284output("sending no-op, case 1") 285286 287 288 output("sending no-op, case 2") 289 290 send_req(None) send a null message }} 291 msgs[gi(i, k)] = req # msgs[gi(M(i,k))] = M(i,k);292 293 294 min_index_received , lagging_node_rank = min_and_idx(sst[my_rank].latest_received_index) (min_index_received, lagging_node_rank) = 295# (min, argmin) i sst[my_rank].latest_received_index[i]; 296 297 write_sst(my_rank, "global_index", (min_index_received + 1) * len(G) + lagging_node_rank - 1) 298# 299 300 # (p.34) A.3.2 Stability and Delivery. 301 302 def stability delivery(); 303

304 ## sort because msgs must be delivered in increasing global index sorted_keys = sorted(msgs.keys()) for g_idx in sorted_keys: if g_idx <= stable meg f</pre> 305 # for (msg : msgs) { g_idx <= stable_msg_idx:</pre> if (msg.global_index <= min_stable_msg_index) { ### min_ deleted deliver_upcall(msg); ### add first argument, to see global_index in 306 307 deliver_upcall(g_idx, msgs[g_idx]) # output 308 del msgs[g_idx] msgs.remove(msg.global_index); }} 309 if sorted_keys and min(sorted_keys) <= stable_msg_idx:</pre> ## if there stored reqs are less than received reqs ### added test to not multicast unnecessarily as this is in an always 310 311 write_sst(my_rank, "latest_delivered_index", stable_msg_idx) 312 sst[my_rank].latest_delivered_index = stable_msg_index } 313 314 # (p.35) A.4 View Change Protocol # A.4.1 Failure Handling and Leader Proposing Changes for Next View. def receive(msg=('failure', r)): # every 1 millisecond { 315 # post RDMA write with completion to every SST row that is not frozen 317 # if (no completion polled from row r) { ### receive failure of row r, simulating polling on SST for not receiving completion from row r for a time period 318 319 sst.freeze(r); ### not defined; added field in Node to track this report_failure(r); }} freeze.add(r) report_failure(r) 321 # # (p.35) ## update the suspected field upon noticing a node failure 322 323 324 325 327 328 raise Exception("ERROR: derecho_partitioning_exception") # throw derecho_partitioning_exception; }} def find_new_leader(r): # find_new_leader(r) { 329 for i in range(len(curr_view.members)): #
 if sst[r].suspected[i]: continue # 330 331 332 else: return i # else return i }] 333 # (p.35) ## update the current view, at the end, with the new leader 334 def leader_selection(): # always { ### made function and called in run new_leader = find_new_leader(my_rank)
if new_leader != curr_view.leader_rank:
 if new_leader == my_rank:
 # olleader 335 # new_leader = find_new_leader(my_rank)
if (new_leader != curr_view.leader_rank && new_leader == my_rank) 336 ### split 2 conjuncts, to add the else-branch for the second # bool all_others_agree = true ### moved into while-loop 337 # all_others_agree = True 338 ### if not moved, if it becomes False in for-loop below, it stays False, and the while-loop never stops 339 while find_new_leader(my_rank) == my_rank: # while (find_new_leader(my_rank) == my_rank) {
 --receive_messages ## yield to receive msgs 340 341 ### needed to receive updates to SST which may result in new leader selection ### break atomicity 342 all_others_agree = True 343 ### moved here from outside while-loop, as explained above # 344 for r in range(len(sst)): for (r: SST.rows) { # if not sst[my_rank].suspected[r]: # if (sst[my_row].suspected[r] == false)
all_others_agree = all_others_agree and (find_new_leader(r) == my_rank) 345 346 347 # all_others_agree &&= (find_new_leader(r) == my_rank) } if (all_others_agree) {
 curr_view.leader_rank = my_rank; 348 if all others agree: # 349 curr_view.leader_rank = my_rank # 350 output("I am the new leader!!!") # 351 break break: }}} ## else: ### added else-branch, for when new leader is not self set current view's leader to be new leader 353 curr_view.leader_rank = new_leader ## 354 # (p.36) ## poll field suspected in SST and propagate the status to all members in the group # always { ### made function and called in run
for (every row r and s) { 355 def suspect(): 356 for r in range(n): for s in range(n): 357 358 if sst[r].suspected[s]: if (sst[r].suspected[s] == true) { # 359 ## if self is suspected: (p.?) if my_rank == s: 360 output("I am suspected and will be removed, shutting down myself now") not sst[my_rank].suspected[s]:
write sst(my_rank]." 361 exit() shut down self ## Shut down self
added test to not multicast repeatedly 362 if # }}
j}
for (s=0; s < num_members; ++s) { ### num_member=n
if sst[my_rank].suspected[s] == true and comm
split 2 conjuncts</pre> write_sst(my_rank, "suspected", True, s) # 363 364 365 for s in range(n): if sst[my_rank].suspected[s]: 366 367 if not curr_view.failed[s]: 368 freeze.add(s) report_failure(s) curr_view.failed[s] = True # curr_view.failed[s] = true # curr_view.failed[s] = true write_sst(my_rank, "wedged", True) # sst[my_rank].wedged = true if curr_view.leader_rank == my_rank and G[s] not in sst[my_rank].changes: # if (curr_view.leader_rank == my_rank and set[mw_rank].wedged = true 369 370 371 372 373 374 (curr_view.leader_rank == my_rank and sst[my_rank].changes.contains(s) == false) { output("changes list updated with: ", G[s]) next_change_index = sst[my_rank].num_changes - sst[my_rank].num_installed; ### omitted, position is not needed to add to Python list 377 378 changes = sst[my_rank].changes changes.append(G[s]) 379 write_sst(my_rank, "changes", changes) # sst[my_rank] 380 381 sst[my_rank].changes[next_change_index] = id of node owning s

```
382
                num_changes = sst[my_rank].num_changes + 1
383
                write_sst(my_rank, "num_changes", num_changes)
# sst[my_rank].num_changes++;
384
                                          3333
385
386
       # (p.37) ## continue A.4.2 in second half of run()
           terminate_epoch():
387
                                          # terminate_epoch() {
388
         leader_rank = curr_view.leader_rank
389
         committed_count = sst[leader_rank].num_committed - sst[leader_rank].num_installed
390
                                               committed_count = sst[leader_rank].num_committed - sst[leader_rank].num_installed;
         next_view = View(n, curr_view.epoch + 1)  ##
next_view.members = curr_view.members.copy() #
for change_idx in range(committed_count): #
                                                           ## create a new
391
392
                                                                next_view.members = curr_view.members
393
                                                                for (change_index=0; change_index<committed_count;change_index++) {</pre>
394
           node = sst[my_rank].changes[change_idx]
                                                            #
                                                                   node_id = sst[my_rank].changes[change_index];
                                                                 if (curr_view.contains(node_id) == true)
395
           if node in next_view.members:
396
             next_view.remove_member(node)
                                                                    new_view.members.remove(node_id); } ### new_view -> next_view
                                                            #
397
             output("Removed node from next_view's group: ", node)
                                                                  else {
398
           else:
             ._____ueu_member(node)  # next_view.members.append(node_id); }}
output("Added node to next_view's group: ", node)
300
            next_view.add_member(node)
400
401
         if leader_rank == my_rank:
                                                             #
                                                                 if (leader_rank == my_rank) {
402
           leader_ragged_edge_cleanup()
                                                                    leader_ragged_edge_cleanup();
403
         else:
                                                                 else {
                                                                  when (sst[leader_rank].ragged_edge_computed == true) {
           await sst[leader_rank].ragged_edge_computed #
404
405
           non_leader_ragged_edge_cleanup()
                                                                     non_leader_ragged_edge_cleanup(); }}
         curr_view = next_view
resetup(sst[my_rank], committed_count)
406
                                                                 curr_view = next_view;
                                                             #
                                                             ## re-setup the system, reinitialize SST and other fields
407
         output ('new view installed with members: ', curr_view.members)
408
409
                                                             # }
410
       # (p.37-38) ## clean up ragged edge
           411
       def leader_ragged_edge_cleanup():
412
413
             # Let rank be s.t. sst[rank].ragged_edge_computed is true
pr i in range(len(G)): # for (n = 0; n < |G|; ++n) {
write_sst(my_rank, "min_latest_received", sst[rank].min_latest_received[i], i)</pre>
           #
for i in range(len(G)): #
414
415
416
417
                                             sst[my_rank].min_latest_received[n] = sst[rank].min_latest_received[n]; }
sst[my_rank].ragged_edge_computed = true; } ### lifted outside if
                                       #
418
                                       419
         else:
           for i in range(len(G)): #
420
             write_sst(my_rank, "min_latest_received", Min("latest_received_index", i), i)
421
                                     #
422
                                                sst[my_rank].min_latest_received[n] = Min(sst[*].latest_received_index[n]);
                                             sst[my_rank].ragged_edge_computed = true; } ### lifted outside else
                                       #
423
424
         write_sst(my_rank, "ragged_edge_computed", True)
         deliver_in_order()
425
                                       #
                                           deliver_in_order(); }
426
       # (p.38) ## clean up ragged edge
       def non_leader_ragged_edge_cleanup(): # non_leader_ragged_edge_cleanup() {
427
428
         leader_rank = curr_view.leader_rank
         for i in range (len(G)): # for (n = 0; n < |G|; ++n) {
429
           430
431
         432
433
434
435
       # (p.38) ## after clean up of ragged edge, deliver pending
       def deliver_in_order(): # deliver_in_order() {
436
437
         curr_g_idx = sst[my_rank].latest_delivered_index
         # curr_global_index = sst[my_rank].latest_delivered_index;
# curr_global_index = sst[my_rank].latest_delivered_index;
max_g_idx = max_gi() # max_global_index = max over n of (sst[my_rank].min_latest_received[n] * |G| + n);
438
439
         for g_idx in range(curr_g_idx + 1, max_g_idx + 1):
# for (global_index = curr_globa
440
           441
442
443
444
             # if (sender_index <= sst[my_rank].min_latest_received[sender_rank]) {
    output("delivering message with global index: ", g_idx, " and message: ", msgs[g_idx])
    deliver_upcall(g_idx, msgs[g_idx]) # deliver_upcall(msgs[global_index]); }}</pre>
445
446
             ## not del in msgs and write_sst as in stability_delivery becaue this is failure case, to rest msg to {}
448
449
       # (p.40) Next, Derecho creates a new SST instance for the new epoch and
       # associates an RDMC session with each sender for each subgroup or shard
# (thus, if a subgroup has k senders, then it will have k superimposed RDMC sessions: one per sender).
450
451
       # The epoch is now considered to be active
452
       def resetup(old_sst_row, num_changes_installed): ## re-setup system after next view with membership is prepared
self.G = curr_view.members  ## new group members in current view
453
454
455
         self.my_rank = curr_view.members.index(self)
                                                              ## set my_rank based on the index in the new_view
456
         initialize()
                                                              ## reinitialize the system
457
         cur_epoch = curr_view.epoch
         send(('view_change', cur_epoch), to= others) ## send completion of view change to all other nodes
output("view_change message sent to all other nodes; waiting to receive the same from others")
await each(p in others, has=some(received(('view_change', _cur_epoch), from_=_p))) ## wait to receive same from all
458
459
460
          others
461 #
         await each(p in others, has=received(('view_change', cur_epoch), p)) ## wait to receive same from all others
```

```
26
```

```
462
           output("view_change message received from all other nodes; starting the epoch")
463
           write_sst(my_rank, "changes", old_sst_row.changes[num_changes_installed:])
464
                                                                                                  ## copy uninstalled/pending changes to new group's SST
            write_sst(my_rank, "num_installed", old_sst_row.num_installed + num_changes_installed)
465
466
                                                                                                 ## update num_installed with new changes installed
           ## update num_installed w
write_sst(my_rank, "num_changes", old_sst_row.num_changes) ## copy old num_changes
write_sst(my_rank, "num_committed", old_sst_row.num_committed)## copy old num_committed
write_sst(my_rank, "num_acked", old_sst_row.num_acked) ## copy old num_ack
write_sst(my_rank, "active", True) ## (p.40)
467
468
469
470
471
        def deliver_upcall(global_idx, req):
                                                                             ## execute request req, at decided global index
          472
           Output('In deliver_update(), g-, g-, ## if request is a null msg for no-op, return
(client, req_id, _) = req ## if request is form (client, req_id, cmd)
if not some(sent(('response', _req_id, _), to= _client)): ## if request has not been responded to before
state, res = execute(global_idx, req, state) ## execute request at global_idx in state
send(('response', req_id, res), to= client) ## send response to client
extruct("response sent to the client/sim process, index: ", global_idx, "response: ", res)
473
474
475
476
478
479
        def execute(global_idx, req, state):
                                                                              ## execute the command in req in given state
480
          (_, req_id, _) = req
return (state+[(global_idx, req)], req_id)
                                                                              ## request is of form (client, req_id, cmd)
481
                                                                             ## return call history and req_id as new state and result
482 ## The rest is for simulation, testing, and performance measurements

      482 ## The rest is is is is is is in start

      483 class RequestMetrics:

      484 """object holding metrics for a request"""

      484 """object holding metrics for a request""

        486
487
488
           self.run_start_time = run_start_time  ## system time at the time of sending request
self.run_end_time = 0.0  ## system time at the time of receiving the response for the request
489
490
491 class Sim(process):
492 """simulator for failure injection and client requests"""
403
        def setup(nodes, num_requests, test_failure, msg_size):
494
           ## nodes; list of nodes in the system
## num_request; total number of requests to send
495
           ## test_failure; whether to test a node failure scenario
## msg_size; size of the client request messages
self.metrics = {} ## stores the metrics per request in
496
497
                                      ## stores the metrics per request id
## number of responses received
498
499
           self.num_resp = 0
500
        def run():
501
           failure_injected = False
                                                                ## whether a failure message has been sent to the system
502
           for tid in range(num_requests):
              metrics[tid] = RequestMetrics(time.clock(), time.time())
503
              node = random.sample(nodes,1)  ## randomly select a node to send the request
output("sent request: ", tid, " to node: ", node)
504
505
              send(('request', (self, tid, os.urandom(msg_size))), to= node)
506
507
              # Uncomment to use sim as an external failure detector
508
              # External failure detector
              if test_failure and not failure_injected and tid == num_requests/2: ## failing last node
509
                send(('failure', 0), to= nodes[1]) ## send index of the last node as a failed node to all nodes
send(('failure', len(nodes)-2), to= nodes[1])
output("Sent failure message to first node")
511 #
513
                 nodes.remove(nodes[0])
514
                 failure_injected = True
           output('done sending all requests')
516
           while True:
             if await (len(setof(tid, received(('response', tid, res)))) == num_requests):
    output("---- all responded")
517
518
                 send(('metrics', list(metrics.values())), to= parent())
519
520
                 break
              elif timeout(10):
                 output("----
                                     --- timeout!!!"
                 for tid in range(num_requests):
524
                   if not metrics [tid].success:
                         = random.sample(nodes,1)
                      output('sending message: ', tid, ' to node: ', n)
send(('request', (self, tid, os.urandom(msg_size))), to=n)
526
528
           send('done', to= parent())
529
         def receive(msg= ('response', tid, res), from_= p):
530
           if not metrics[tid].success:
              cur_metric = metrics[tid]
              cur_metric.cpu_end_time = time.clock()
cur_metric.run_end_time = time.time()
533
534
              cur_metric.success = True
              out_metric.ising
num_resp += 1
output("---- Received tid: ", tid, " from: ", p, " with num_resp: ", num_resp) ## output response received
536
```

```
537 def main():
        config(channel is fifo, handling is all, clock is lamport, visualize is {
538 #
                # colors: override message and process colors, defaults to random (to fix random
# supports any valid CSS color value
539 #
540 #
                # https://developer.mozilla.org/en-US/docs/Web/CSS/color_value
541 #
                # examples: Transparent, Yellow, DarkRed, rgb(255, 255, 0),
# rgba(255, 255, 0, 0.1), hsl(210, 100%, 50%)
'colors': {
542 #
543
544 #
545 #
                   # processes
#'Sim': 'aquamarine',
546
                    # messages
'request': 'purple',
'response': 'blue',
'data': 'lime',
547 #
548
549 #
550 #
                     'control': 'red'
552 #
                    'rdma_write_sst': 'gray'
553 #
               }
          })
554 #
555
       config(channel is {fifo,reliable}, clock is lamport, handling is all)
                    = int(sys.argv[1]) if len(sys.argv) > 1 else 3
556
       num_nodes
                                                                            ## number of nodes in the system
       num_requests = int(sys.argv[1]) if len(sys.argv) > 2 else 10 ## number of requests from applications
nreps = int(sys.argv[3]) if len(sys.argv) > 3 else 1 ## number of repetitions, for calculating metrics
557
558
       msg_size
559
                     = int(sys.argv[4]) if len(sys.argv) > 4 else 1
                                                                            ## size of request messages
       560
561
       test_failure = False
                                                                            ## whether to test a failure scenario
562
563
       throughputs = []
564
        or _ in range (nreps):
t1 = time.time()
       for
565
566
         nodes =
                 sorted(list(new(Node, num= num_nodes))) ## create Node processes, ordering only for ease of tracing
567
         state = []
                                                    ## history of state of the application
         for rank, node in enumerate(nodes): ## setup Node processes
568
569
           setup(node, (nodes, rank, window_size, max_msg_size, state))
         start (nodes)
         sim = new(Sim, (nodes, num_requests, test_failure, msg_size)) ## create and set up Sim process
         start(sim)
         await(received(('done'), from_=sim)) ## wait to receive 'done' from sim
574
         throughputs.append(round(num_requests/(time.time() - t1)))
                    ----- time:", time.time() - t1)
         print (
         end(sim)
         end(nodes)
578
       cpu_times = []
       run_times = []
579
580
       total runs = 0
       for metrics in listof(metric, received(("metrics", metric))):
581
582
        for metric in metrics:
583
           total runs += 1
584
           if metric.success:
             cpu_times.append(metric.cpu_end_time - metric.cpu_start_time)
585
586
              run_times.append(metric.run_end_time - metric.run_start_time)
587
       output("AVG cpu_times", round(sum(cpu_times)/len(cpu_times), 5))
       output("AVG run_times", round(sum(run_times)/len(run_times), 5))
output("AVG throughput", round(sum(throughputs)/len(throughputs), 5))
588
589
```

Listing 1: Complete specification in DistAlgo.

B Derecho sample output

[319] derecho.Sim<Sim:b4c05>:OUTPUT: sent request: 4 [<Node:b4c02>] to node: $\frac{20}{21}$ [319] derecho.Sim<Sim:b4c05>:OUTPUT: sent request: [<Node:b4c03>] 5 to node: [319] derecho.Sim<Sim:b4c05>:OUTPUT: sent request: 6 to node: [<Node:b4c03>] 22[320] derecho.Sim<Sim:b4c05>:OUTPUT: sent request: [<Node:b4c02>] to node: derecho.Sim<Sim:b4c05>:OUTPUT: sent request: 23[320] 8 to node: $[\leq Node \cdot b4c02 >]$ 24derecho.Sim<Sim:b4c05>:OUTPUT: sent request: 9 [320] to node: [<Node:b4c03>] 25derecho.Sim<Sim:b4c05>:OUTPUT: done sending all requests [320] 26[322] derecho.Node <Node: b4c02 >: OUTPUT: sending no-op, case 1 27 derecho.Node<Node:b4c02>:OUTPUT: request: (<Sim:b4c05>, 3, b'\x14') received from client: [325] <Sim:b4c05> 28 29 derecho.Node <Node: b4c02>: OUTPUT: request: [325] (<Sim:b4c05>, 4, b'\xbf') received from client: <Sim:b4c05> (<Sim:b4c05>, 7, b'\x06') (<Sim:b4c05>, 8, b'\xb3') [325] derecho.Node<Node:b4c02>:OUTPUT: request: received from client: <Sim:b4c05> 30 [326] derecho.Node <Node: b4c02>: OUTPUT: request: received from client: <Sim: b4c05: 31[341] derecho.Node <Node: b4c03 >: OUTPUT: request: (<Sim:b4c05>, 5, b'\x83') received from client: <Sim:b4c05> derecho.Node<Node:b4c03>:0UTPUT: request: (<Sim:b4c05>, 6, b'\xca') received from client: derecho.Node<Node:b4c03>:0UTPUT: request: (<Sim:b4c05>, 9, b'0') received from client: <</pre> 32 [342] <Sim:b4c05> 33 [342] <Sim:b4c05> $\frac{34}{35}$ [344] derecho.Node<Node:b4c04>:OUTPUT: sending no-op, case 2 [345] derecho.Node<Node:b4c03>:OUTPUT: in deliver_upcall(), gi: 0 and req: None derecho.Node<Node:b4c03>:OUTPUT: in deliver_upcall(), gi: 1 and req: None 36 [345] [346] derecho.Node<Node:b4c03>:OUTPUT: in deliver_upcall(), gi: 2 and req: (<Sim:b4c05>, 0, b'd') [346] derecho.Node<Node:b4c03>:OUTPUT: response sent to the client/sim process, index: 2 response: 0 37 38 ./derecho.da:598: DeprecationWarning: time.clock has been deprecated in Python 3.3 and will be removed from Python 3.8: 39 use time.perf_counter or time.process_time instead cur_metric.cpu_end_time = time.clock() 40 [346] derecho.Sim.Sim.b4c05>:OUTPUT: ---- Received tid: 0 from: <Node:b4c03> with num_resp: 1 [347] derecho.Node<Node:b4c04>:OUTPUT: sending no-op, case 2 [351] derecho.Node<Node:b4c03>:OUTPUT: in deliver_upcall(), gi: 3 and req: None 41424344derecho.Node<Node:b4c03>:OUTPUT: in deliver_upcall(), gi: 4 and req: [351] (<Sim:b4c05>, 2, b'+') [351] derecho.Node<Node:b4c04>:0UTPUT: in deliver_upcall(), gi: 0 and req: None [351] derecho.Node<Node:b4c04>:0UTPUT: in deliver_upcall(), gi: 0 and req: None [351] derecho.Node<Node:b4c04>:0UTPUT: response sent to the client/sim process, index: 4 response: 2 [351] derecho.Node<Node:b4c04>:0UTPUT: in deliver_upcall(), gi: 1 and req: None [351] derecho.Node<Node:b4c04>:0UTPUT: in deliver_upcall(), gi: 2 and req: (<sim:b4c05>, 0, b'd') [351] derecho.Node<Node:b4c04>:0UTPUT: in deliver_upcall(), gi: 2 and req: (<sim:b4c05>, 0, b'd') 45464748 4950[352] derecho.Node<Node:b4c03>:OUTPUT: in deliver_upcall(), gi: 5 and req: (<Sim:b4c05>, 1, b'\xe1') derecho.Node:Node:b4c0d>:OUTPUT: response sent to the client/sim process, index: 5 response: derecho.Node<Node:b4c03>:OUTPUT: response sent to the client/sim process, index: 5 response: 51[352] 0 52[352] derecho.Node<Node:b4c04>:0UTPUT: in deliver_upcall(), gi: 3 and req: None derecho.Node<Node:b4c04>:0UTPUT: in deliver_upcall(), gi: 4 and req: (<Sim:b4c05>, 2, N derecho.Sim:Sim:b4c05>:0UTPUT: ---- Received tid: 1 from: <Node:b4c03> with num_resp: $\frac{53}{54}$ [352] [352] (<Sim:b4c05>, 2, b'+') 55[352] derecho.Node<Node:b4c04>:0UTPUT: response sent to the client/sim process, index: 4 response: 2
derecho.Node<Node:b4c04>:0UTPUT: in deliver_upcall(), gi: 5 and req: (<Sim:b4c05>, 1, b'\xe1')
derecho.Node<Node:b4c04>:0UTPUT: response sent to the client/sim process, index: 5 response: 1 $\begin{array}{c} 56 \\ 57 \end{array}$ [352] [352] $\frac{58}{59}$ [353] [354] derecho.Node <Node: b4c04 >: OUTPUT: sending no-op, case 2 derecho.Node<Node:b4c03>:OUTPUT: in deliver_upcall(), gi: 6 and req: (<Sim:b4c05>, 3, b'\x14') 60 [355] [356] derecho.Node<Node:b4c03>:OUTPUT: response sent to the client/sim process, index: 6 response: 3
derecho.Sim<Sim:b4c05>:OUTPUT: ---- Received tid: 3 from: <Node:b4c03> with num_resp: 4
derecho.Node<Node:b4c04>:OUTPUT: in deliver_upcall(), gi: 6 and req: (<Sim:b4c05>, 3, b'\x14') 6162 [356] 63 [357] derecho.Node<Node:b4c04>:OUTPUT: response sent to the client/sim process, index: 6 response: derecho.Node<Node:b4c02>:OUTPUT: in deliver_upcall(), gi: 0 and req: None derecho.Node<Node:b4c02>:OUTPUT: in deliver_upcall(), gi: 1 and req: None 64 [357] 65 [367] 66 [367] 67 [367] derecho.Node<Node:b4c02>:OUTPUT: in deliver_upcall(), gi: 2 and req: (<Sim:b4c05>, 0, b'd') derecho.Node<Node:b4c02>:OUTPUT: response sent to the client/sim process, index: 2 response: 0 68 [367] derecho.Node:b4c02>:OUTPUT: in deliver_upcall(), gi: 3 and req: None derecho.Node<Node:b4c02>:OUTPUT: in deliver_upcall(), gi: 4 and req: (<Sim:b4c05>, 2, b'+') derecho.Node<Node:b4c02>:OUTPUT: response sent to the client/sim process, index: 4 response: 69[367] 70[367] 71[368] 72 73 74 derecho.Node <Node: b4c02>: OUTPUT: [368] in deliver_upcall(), gi: 5 and req: (<Sim:b4c05>, 1, b'\xe1') derecho.Node<Node:b4c02>:0UTPUT: response sent to the client/sim process, index: 5 response: derecho.Node<Node:b4c02>:0UTPUT: in deliver_upcall(), gi: 6 and req: (<Sim:b4c05>, 3, b'\x1 [368] [368] (<Sim:b4c05>, 3, b'\x14 derecho.Node<Node:b4c02>:OUTPUT: response sent to the client/sim process, index: 6 response: 3 derecho.Node<Node:b4c02>:OUTPUT: in deliver_upcall(), gi: 7 and req: (<Sim:b4c05>, 5, b'\x83') $\frac{75}{76}$ [368] [368] 77 78 79 derecho.Node<Node:b4c02>:OUTPUT: response sent to the client/sim process, index: 7 response: 5 [368] derecho.Node<Node:b4c02>:OUTPUT: in deliver_upcall(), gi: 8 and req: None
derecho.Node<Node:b4c02>:OUTPUT: in deliver_upcall(), gi: 9 and req: (<Sim:b4c05>, 4, b
derecho.Sim<Sim:b4c05>:OUTPUT: ---- Received tid: 5 from: <Node:b4c02> with num_resp: [368] [368] (<Sim:b4c05>, 4, b'\xbf') 80 [369] derecho.Node<Node:b4c02>:OUTPUT: response sent to the client/sim process, index: 9 response: 4
derecho.Node<Node:b4c02>:OUTPUT: in deliver_upcall(), gi: 10 and req: (<Sim:b4c05>, 6, b'\xca') 81 [369] 82 [369] derecho.Node<Node:b4c02>:0UTPUT: in deliver_upcall(), gi: 10 and req: (<Sim:b4c05), 6, b'\xca') derecho.Sim<Sim:b4c05>:0UTPUT: response sent to the client/sim process, index: 10 response: 6 derecho.Node<Node:b4c02>:0UTPUT: ---- Received tid: 4 from: <Node:b4c02> with num_resp: 6 derecho.Node<Node:b4c02>:0UTPUT: in deliver_upcall(), gi: 11 and req: None derecho.Node<Node:b4c02>:0UTPUT: in deliver_upcall(), gi: 12 and req: (<Sim:b4c05>, 7, b'\x06') 83 [369] 84 [369] 85 [369] 86 [369] derecho.Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node
(Node)
(Node
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
(Node)
 87 [369] 88 [369] 89 [370] derecho.Node<Node:b4c02>:0UTPUT: in deliver_upcall(), gi: 13 and req: (<Sim:b4c05>, 9, b'0') derecho.Node<Node:b4c02>:0UTPUT: response sent to the client/sim process, index: 13 response: 90 [371] 91[371] derecho.Node<Node:b4c02>:OUTPUT: in deliver_upcall(), gi: 14 and req: None derecho.Node<Node:b4c02>:OUTPUT: in deliver_upcall(), gi: 15 and req: (<Sim:b4c05>, 8, b'\xb3') derecho.Sim<Sim:b4c05>:OUTPUT: ---- Received tid: 9 from: <Node:b4c02> with num_resp: 9 92 [371] [371] 93 [371] derecho.Sim
Sim:b4c05>:OUTPUT: ---- Received tid: 9 from: <Node:b4c02> with num_resp: 9
[371] derecho.Sim<Sim:b4c05>:OUTPUT: response sent to the client/sim process, index: 15 response:
[372] derecho.Sim<Sim:b4c05>:OUTPUT: ---- Received tid: 8 from: <Node:b4c02> with num_resp: 10
[372] derecho.Sim<Sim:b4c05>:OUTPUT: ---- all responded
[374] derecho.Node_<Node_:4ec01>:OUTPUT: AVG com time a context of the client of the cl 94958 96 97 derecho.Node_<Node_:4ec01>:0UTPUT: AVG run_times 0.0058 derecho.Node_<Node_:4ec01>:0UTPUT: AVG run_times 0.05636 98 99 [374] [374] derecho.Node_<Node_:4ec01>:OUTPUT: AVG throughput 55.0 [374] da.api<MainProcess>:INFO: Main process terminated. ------ time: 0.18188166618347168 100 101

Listing 2: Derecho output for group consisting of 3 nodes, 1 client, and 10 requests per client

with window-size as 10.