The Algorithm for the Distributed Key-Value Storage Problem

1 The Distributed Key-Value Storage

Problem Statement. We need to develop and implement an algorithm for the distributed key-value storage system, which involves distributed computations such as snapshot algorithms when the external world asks a question about the entire system. The system must be fault tolerant, correct, highly available, and reasonably fast.

2 The Algorithm

2.1 A set of states

Each of the processes must be in one of the following 4 states:

(a) *joining*

(b) *available*

(c) *leaving*

(d) *gone*

2.2 Information Stored by Each Storage Process

A process p contains the following information:

(a) $p.m$ (m): the parameter m.

(b) $p.state$ (⟨`state`⟩): one of the 4states outlined above.

(c) $p.node$ its process node (`Node`), represented as a lowercase ASCII string with a machine name, separated by an @ symbol. (For example, `p1@ash`). We can always find out our nodename with node(), so it is not passed around.

(d) $p.neighbors$ (`Neighbors`): a list of its neighboring processes $[n_1, n_2, \ldots, n_k]$.

2.3 Message Types in the System

We categorize messages by its sender and its receiver:

2.3.1 From Storage Processes to Storage Processes

M1 a {*pid*, *ref*, `first_key_for_the_next_k_processes_inclusive`, *lookahead*, *num_lookahead*} message, sent from a storage process i to a storage process j to ask j to compute the first key among storage processes $j, j+1, \ldots, j+lookahead-1$. The value *num_lookahead* is an actual number that the process j will make further call, which is a much smaller number than *lookahead*, since it will call helper functions in a similar fashion.

M2 a {*ref*, `first_key_result_for_the_next_k_processes_inclusive`, *result*} message, sent from a storage process j to a storage process i which asked j to compute the first key among storage processes $j, j+1, \ldots, j+lookahead-1$.

M3 a {*pid*, *ref*, `last_key_for_the_next_k_processes_inclusive`, *lookahead*, *num_lookahead*} message, sent from a storage process i to a storage process j to ask j to compute the last key among storage processes $j, j+1, \ldots, j+lookahead-1$. The value *num_lookahead* is an actual number that the process j will make further call, which is a much smaller number than *lookahead*, since it will call helper functions in a similar fashion.

M4 a {*ref*, `last_key_result_for_the_next_k_processes_inclusive`, *result*} message, sent from a storage process j to a storage process i which asked j to compute the last key among storage processes $j, j+1, \ldots, j+lookahead-1$.

M5 a {*pid*, *ref*, `num_keys_for_the_next_k_processes_inclusive`, *lookahead*, *num_lookahead*} message, sent from a storage process i to a storage process j to ask j to compute the number of keys among storage processes $j, j+1, \ldots, j+lookahead-1$. The value *num_lookahead* is an actual number that the process j will make further call, which is a much smaller number than *lookahead*, since it will call helper functions in a similar fashion.

M6 a {*ref*, `num_keys_result_for_the_next_k_processes_inclusive`, *result*} message, sent from a storage process j to a storage process i which asked j to compute the number of keys among storage processes $j, j+1, \ldots, j+lookahead-1$.

2.3.2 From Storage Processes to Non-Storage Processes

M7 We haven't done the detail of this part, but we think that it will need to send a `node_name` message to know which node the storage process is running in. This will be used in the `node_list` request.

2.3.3 From Non-Storage Processes to Storage Processes

M8 The reply of what a node name the storage process is running in.

2.3.4 From Non-Storage Processes to Non-Storage Processes

M9 A request to compute `node_list`. Alternatively, we can make a linked list of the nodes. That is, each node knows its sucessor node. It passes the `node_list` request among the circle until it reaches the original requester. Then we get a complete node list and report back to the external world.

2.3.5 From Processes to External Controllers

M9 a {*ref*, `stored`, *key*, *value*} message, sent to the pid from a store request, with the ref from the request, to confirm that the store operation took place and overwrote the previously stored value old-value. If there was no previously stored *value*, old-value should be the atom `no_value`. world to store value for key.

M10 a {*ref*, `retrieved`, *value*} message, sent to the pid from a retrieve request, with the ref from the request, to indicate that value is stored for the requests *key*; if no value is stored for *key*, value should be the atom `no_value`.

M11 a {*ref*, `result`, `result`} message, sent to the pid from a `first_key`, `last_key`, `num_keys`, or `node_list` request, with the ref from the request, to communicate the result. In the case of `first_key`, `last_key` and `node_numbers` the result will be a list; in the case of `num_keys`, it will be an integer.

M12 a {*ref*, `failure`} message, sent to the pid from a request to indicate that the request failed. This is an optional, polite way to tell the outside world that a request failed rather than simply letting the outside world time out.

2.3.6 From External Controllers to Processes

M13 {*pid*, *ref*, `store`, *key*, *value*} message, sent by the outside world to store value for key.

M14 a `retrieve` message – as stated in the assignment.

M15 a `first_key` message – as stated in the assignment.

M16 a `last_key` message – as stated in the assignment.

M17 a `num_keys` message – as stated in the assignment.

M18 a `node_list` message – as stated in the assignment.

M19 a `leave` message – as stated in the assignment.

2.4 Actions before and after Transitions

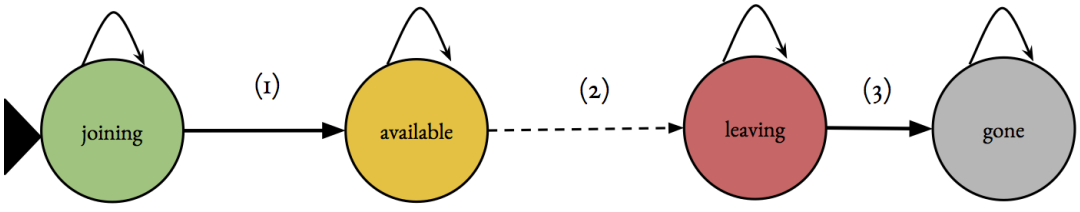

Figure 1: A diagram showing possible state transitions.

We will describe what would happen between states.

(1) *p.joining* → *p.available***:** sends a joining request to all of its neighbors. Once it receives all joining message acknowledgments from all its neighbors, transitions to the *available* state.

(5) *p.leaving* → *p.leaving***:** checks to see if it has already received a `leave` message from an external controller. If so, the process transitions to the *leaving* state and sends leaving notifications to all of its neighbors.

(6) *p.leaving* → *p.gone***:** once it receives leaving notification acknowledgements from all of its neighbors to whom it sent leaving notifications, it transitions to the *gone* state. At this stage, it knows that all of its neighbors are aware of its leaving and already deleted it from their neighbors list. With this knowledge, it then deletes all its neighbors from its neighbors list.

Now we need to define what should happen when a process p receives an incoming message depending on what state it currently is in.

2.5 Actions for Incoming Joining Request Messages (M??)

Received in the *joining* state: holds onto the request until it successfully joins and transitioned to available. This prevents odd joining circles that lead to deadlocks.

Received in the *available* state: approves request and creates a dirty fork for the edge.

Received in the *leaving* state: the assignment specifies that we do not have to handle this situation, the external controller should know better.

Received in the *gone* state: not possible

2.6 Actions for Incoming Joining Message Acknowledgement Messages (M??)

Received in the *joining* state: adds that neighbor to the forklist, saying you don't have the fork. Then it tries to contact the rest of your neighbors. Once all neighbors have sent it an `ok`, it can transition to the *available* state.

Received in the *available* state: not possible.

Received in the *leaving* state: not possible.

Received in the *gone* state: not possible.

2.7 Actions for Incoming Leaving Notification Messages (M??)

Received in the *joining* state: not possible, otherwise this problem is impossible. This is due to our Assumption

Received in the *available* state: removes the fork from the list and remove the neighbor from the list. It sends back `ok`.

Received in the *leaving* state: removes the fork from the list and removes the neighbor from the list. It sends back `ok`.

Received in the *gone* state: not possible.

2.8 Actions for Incoming Leaving Message Acknowledgement Messages (M??)

Received in the *joining* state: not possible.

Received in the *available* state: not possible.

Received in the *leaving* state: Try to contact the other neighbors, remove that neighbor from the list of neighbors.

Received in the *gone* state: not possible.

2.9 Actions for Incoming `leave` Messages (M??)

Received in the *joining* state: not possible.

Received in the *available* state: transitions to *leaving*, which sends messages to all neighbors to say that it is leaving. Once all processes give `ok`, then it transitions to gone.

Received in the *leaving* state: stay leaving.

Received in the *gone* state: not possible.

3 Allowed Assumptions

3.1 External Controllers

A1 Only storage processes will receive requests from the outside world.

A2 External controllers will not send duplicate or invalid control signals.

A3 More assumptions are listed on page 6 of the assignment.

3.2 Misc.

A4 Messages are never lost; sufficient time is allowed for a process to bootstrap itself before other processes send it messages.

A5 There is ample time to rebalance when a node leaves or joins.

4 Communication

For a storage process i to send to storage process j, it sends the message through the following path. Assume $i < j$. If $j < i$ we can replace j with $2^m + j$ and do computation in modulo 2^m.

Let

$$j - i = 2^{a_t} + 2^{a_{t-1}} + \ldots + 2^{a_0},$$

where $a_t > a_{t-1} > \cdots > a_0 \geq 0$. This is equivalent to express the number $j - i$ in base 2. Then we design the message to go through the following path:

$$i \longrightarrow i + 2^{a_t} \longrightarrow i + 2^{a_t} + 2^{a_{t-1}} \longrightarrow \cdots \longrightarrow i + 2^{a_t} + 2^{a_{t-1}} + \ldots + 2^{a_0} = j,$$

which is a valid path because each sender sends a message to its neighbor. Thus, for each step, storage process r computes $j - r = 2^{a_s} + 2^{a_{s-1}} + \ldots + 2^{a_0}$ (where $0 \leq s \leq t$) and sends to its neighbor that has id $r + 2^{a_s}$. Note that the value a_s can be computed by

$$a_s = \lfloor \log_2(j - r) \rfloor,$$

where $\lfloor \cdot \rfloor$ is the floor function, which gives the largest integer that is less than or equal to the given number. For example, if $i = 2$ and $j = 13$, the following path happens:

$$2 \longrightarrow 2 + 2^{\lfloor \log_2(11) \rfloor} = 2 + 2^3 = 11 \longrightarrow 11 + 2^{\lfloor \log_2(2) \rfloor} = 11 + 2^1 = 13.$$

5 First Key, Last Key, Num Keys

These three commands from the outside world invoke distributed computations. The algorithm is a snapshot-like and uses the idea of devide-and-conquer. Consider the neighbor list to be its fingers:

```
1 Neighbors = [(Id + round(math:pow(2, K))) rem TwoToTheM || K <- lists:seq(0, M - 1)].
```

The idea is, each storage process will break a task to subtasks and ask its neighbors to help them compute each subtask. The pseudocode is as follows:

Algorithm 1 Abstract Distributed Algorithms

1: **function** ABSTRACT_DISTRIBUTED_ALGO(*pid*, *ref*, *task*)
2: Call ABSTRACT_DISTRIBUTED_ALGO_SUBTASK(*pid*, *ref*, *sub_task*, 2^m, $m + 1$) via sending.
3: Wait for a response.
4: Format the output as Result and report back to *pid* with the message $\{ref, \texttt{result}, Result\}$.
5: **end function**

where *task* can be `first_key`, `last_key`, or `num_keys`.

Algorithm 2 Abstract Distributed Algorithm Helper Functions

1: **function** ABSTRACT_DISTRIBUTED_ALGO_SUBTASK(*pid*, *ref*, *sub_task*, *lookahead*, *num_lookahead*)
2: **if** *num_lookahead* $= 1$ **then** ▷ Need no further communication
3: DO_TASK on the table in this storage process.
4: **else**
5: Create a new table to store partial results.
6: DO_TASK on the table in this storage process and insert the result into the table.
7: create a list L of tuples $\{id + 2^k, 2^k, k + 1\}$ for $k = 0, 1, \ldots, numLookAhead - 2$.
8: **for** $\{neigborId, lookAhead, numLookAhead\}$ in L **do**
9: send a global message to storage process with id $id + 2^k$ (mod 2^m) to compute $ABSTRACT\_DISTRIBUTED\_ALGO\_SUBTASK(pid, ref, lookahead, num\_lookahead)$
10: **end for**
11: Wait for all results for all subtasks to come.
12: Combine the results and return.
13: **end if**
14: **end function**

where *DO_TASK* corresponds to the given task. In particular, for the `first_key` message, it will compute the first key in the table stored in this process. The combine part will take the first key of the aggregated results. For the `last_key` message, it will compute the last key. The combine part will take the last key of the aggregated results. Finally, for the `num_keys`, it will compute the number of keys in the table in stored in this process. The combine part will take the sum of the aggregated results.

Algorithm 3 Implementation pf Abstract Distributed Algorithms and Helper Functions

```
1   {Pid, Ref, first_key_for_the_next_k_processes_inclusive, LookAhead, NumLookAhead}
        ->
2     println("~s:~p > Received first_key_for_the_next_k_processes_inclusive command
          "
3       ++ "with lookahead (including self) of ~p and num lookahead of ~p.",
4       [GlobalName, Ref, LookAhead, NumLookAhead]),
5     Result = case NumLookAhead of
6       1 ->
7         ets:first(Table);
8       _ ->
9         % The summary table is more like a list, but we use an
10        % ordered_set, duplicate_bag ets table for convenience.
11        % each element will be a singleton tuple
12        SummaryTable = ets:new(summary_table, [ordered_set, duplicate_bag]),
13        % start with the first key from this process
14        ets:insert(SummaryTable, {ets:first(Table)}),
15        NeighborsWithLookAhead = [
16            {
17              % a tuple of size 3
18              (Id + round(math:pow(2, K))) rem TwoToTheM,
19              round(math:pow(2, K)),
20              % the number of processes to lookahead (including self)
21              K + 1
22            }
23            % we already lookahead at itself. So we will look ahead using
24            % the parameters [0, 1, 2, ..., NumLookAhead - 2],
25            % which has the total number of things in it being NumLookAhead - 2.
26            || K <- lists:seq(0, NumLookAhead - 2)
27        ],
28        println("~s:~p > Plan to send subcomputation requests to storage processes
              with id ~p",
29          [
30            GlobalName,
31            Ref,
32            lists:map(fun({A, _, _}) -> A end, NeighborsWithLookAhead)
33          ]
34        ),
35        % send a request to compute first key for the next LookAhead processes
36        lists:map(
37          fun({ProcessId, ProcessLookAhead, NumProcessesLookAhead}) ->
38            TargetName = getStorageProcessName(ProcessId),
39            println("~s:~p > Sending subcomputation for the first_key request "
40              ++ "to ~p with lookahead (including self) of ~p and the number "
41              ++ "of processes (including self) to lookahead of ~p",
42              [GlobalName, Ref, TargetName, ProcessLookAhead, NumProcessesLookAhead
                    ]),
43            global:send(
44              TargetName,
45              {self(), make_ref(), first_key_for_the_next_k_processes_inclusive,
46                ProcessLookAhead, NumProcessesLookAhead}
47            )
48          end,
49          NeighborsWithLookAhead
50        ),
51        % expect the table to eventually have LookAhead elements
52        wait_and_get_the_first_key(GlobalName, self(), Ref, SummaryTable,
              NumLookAhead)
```

For example, in Fig. 5, we let $m = 3$. Suppose the storage process 0 receives the `first_key` message. It will call a helper function $first\_key\_helper(self, 2^m, m+1)$ because it needs to look for the first key among the total of 8 processes, and $m + 1 = 4$ is the number of the processes it actually needs to call a helper function on (including itself). It will call $first\_key\_helper(id = 1, lookahead = 2^0, num\_lookahead = 1)$, $first\_key\_helper(id = 2, lookahead = 2^1, num\_lookahead = 2)$, and $first\_key\_helper(id = 4, lookahead = 2^2, num\_lookahead = 3)$. Each call is done by global message passing. Once the process 0 gets results from process 1, 2, and 4, it finds the first key among the first keys reported back from process 1 (covering those keys in process 1), process 2 (covering those keys in storage processes 2, 3), process 4 (covering those keys in storage processes 4, 5, 6, 7), together with the first key in storage process 0.

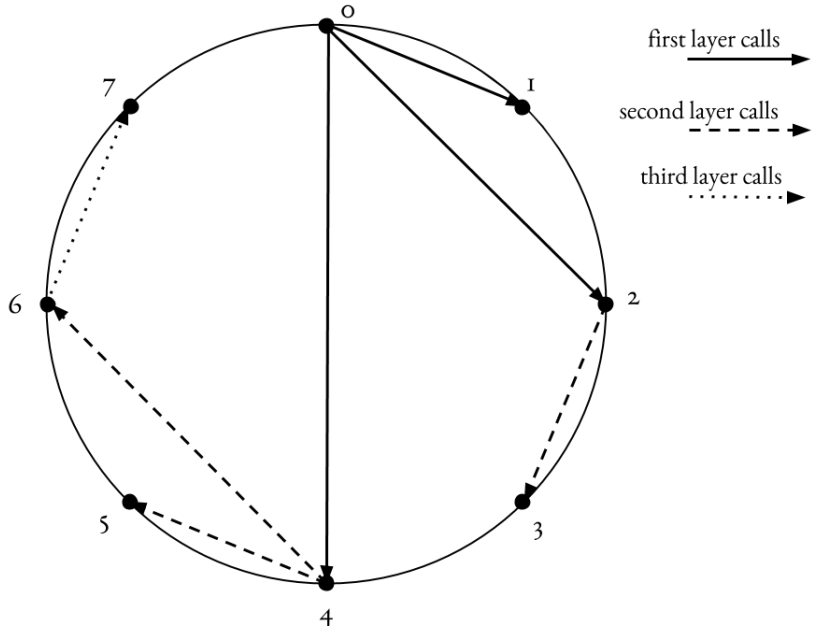

Figure 2: A diagram showing how storage process communicates. In this example, storage 0 sends messages to storage process 1 requesting it to find the first key among storage process 1; to storage process 2 to find the first key among storage processes 2 and 3; and to storage process 4 to find the first key among storage processes 4, 5, 6, and 7. Then the storage process 0 combines these results and determines the gloabally first key.

6 Hash Function

We use a simple hash function that adds all the characters of the key string together and returns the modulus of the sum with respect to 2^m.

Algorithm 4 Hash function

```
%% hash function to uniformly distribute among storage processes.
hash(Str, M) when M >= 0 -> str_sum(Str) rem round((math:pow(2, M)));
hash(_, _) -> -1.   %% error if no storage processes are open.

%% sum digits in string
str_sum([]) -> 0;
str_sum([X|XS]) -> X + str_sum(XS).
```

7 Fault Tolerance

We run out of time to write this part, but we have that implemented. Basically one table is duplicated to store among all its neighbors.

8 Proof of Correctness

8.1 Safety Properties and Proofs

Suppose the parameter m, which is a nonnegative integer, is given.

S1 **initially** *p.joining*

p is given the state of *joining* in which the process *p* is requesting to join the group and cannot possibly gain any other state until granted acceptance.

S2 #(storage processes) $= 2^m$.

Initially, the number of the storage processes spawned by the first node is 2^m. After more nodes join, storage processes are differently distributed among nodes, but the total number does not change.

S3 *p.joining* **next** *(p.joining* $\vee$ *p.available)*

This requirement is satisfied because *p* can be constantly trying to join the party but may be waiting infinitely or it can be granted the state of available (which is the only initial state in the party).

S4 *p.available* **next** *(p.available* $\vee$ *p.leaving)*

When available, the process *p* can continue *available* , or *leaving* . No other "state" transitions are available to the process at the *available* state.

S5 *p.leaving* **next** *(p.leaving* $\vee$ *p.gone)*

When a process *p* has been told to leave by the external controller or crashes, it is destined to leave thus may continue its cleanup and remain in the *leaving* stage or the process could complete the *leaving* state and leave, successfully terminating and entering the *gone* state.

S6 *p.gone* **next** *(p.gone)*

When a process *p* is *gone*, the process may not join again (implying it may not reach anymore states) and thus is in the fixed state of *gone*.

S7 When a storage process p receives a store/retrieve message with key k, if $k = p.id$, then it performs an action. If not, then it forwards the request to an apppropriate process.

This is true due to our algorithm.

S8 $1 \leq n := \#(\text{nodes}) \leq 2^m$.

This is true due to our algorithm.

S9 $node_i.id \neq node_j.id$ if $i \neq j$ for all $0 \leq i, j \leq n-1$.

This is true due to our algorithm.

S10 $node_i.id \in \{0, 1, 2, \ldots, 2^m - 1\}$ for all $0 \leq i \leq n-1$.

This is true due to our load balancing algorithm.

S11 Without loss of generality, we can let $0 \leq node_1.id < node_2.id < \cdots < node_n.id \leq 2^m - 1$. Then, for $i = 0, 1, \ldots, n-1$, $node_i$ hosts the storage processes with ids in the set $\{node_i.id, node_i.id + 1, node_i.id + 2, \ldots, node_{i+1}.id - 1\}$ in modulo 2^m calculation and with the notation $node_n = node_0$.

This is true due to our load balancing algorithm.

S12 All storage processes are named in the global Erlang name registry.

This is true due to our algorithm.

S13 Each node must register at least one non-storage process (so that the node is always discoverable).

This is true due to our algorithm.

S14 If $\#(\text{nodes}) = n < 2^m$, then a new node can always join the system.

S15 A storage process can communicate to another storage process only if they are neighbors.

This is true due to our algorithm.

S16 A storage process can communicate to a non-storage process only if they are in the same node.

This is true due to our algorithm.

S17 A non-storage process can communicate to a storage process only if they are in the same node.

This is true due to our algorithm.

S18 A non-storage process in node i can communicate to a non-storage process only if it is in the nodes that host neighbors of the storage processes hosted by node i.

This is true due to our algorithm.

S19 Every message from the outside world (except for `leave`) must be responded to except for when relevant processes crash.

This is true due to our algorithm.

S20 If storing is succesful, num_keys should not change.

S21 If storing is not succesful, num_keys should increase by one.

S22 retrieve should not change num_keys.

S23 num_keys is correct.

This relies on each key is stored in exactly one table (not counting backup), which is true because of each hash function gives a unique value.

S24 first_key and last_key are correct.

Since we go through all elements effectively, and we find the first and last keys, we should get the globally first and last key, respectively.

9 Proof of Progress Properties

PG1 $p.joining \rightsquigarrow^* p.available$ (* if its neighbors remain in the network long enough)

After process p has started up, it has given itself the *joining* state. Assuming that the neighbors in which p knows about are running correctly and the network runs as expected and Assumption A**??**, all other processes are bound to hear p's request to join eventually and thus p is guaranteed to be given the state *available*.

PG2 $p.leaving \rightsquigarrow p.gone$ When p is in the *leaving* state, it sends leaving notifications to all its neighbors. Since messages are never lost, all of its neigbors will eventually get the messages and send acknowledgements back. Once it receives all acknowledgements, it can transition to *gone*.

PG3 $p.gone \rightsquigarrow$ (all other nodes detect and rebalance)