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.

Our snapshot algorithm is based on the Chandy-Lamport algorithm.

2 The Algorithm

2.1 A set of states

Each of the processes must be in one of the following 6 states:

(a) *joining*

(b) *available*

(c) *leaving*

(d) *gone*

(e) *red*

(f) *white*

2.2 Information Stored by Each Process

In the Erlang syntax, the `TODO` function's header represents the information stored by each process:

```
1 TODO(<state>, Node, Neighbors)
```

In other words, a process p contains the following information:

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

(b) $p.state$ (⟨`state`⟩): one of the sixstates 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 Lots of messages going on here.

2.3.2 From Storage Processes to Non-Storage Processes

M7 something.

2.3.3 From Non-Storage Processes to Storage Processes

M10

2.3.4 From Non-Storage Processes to Non-Storage Processes

M10 a `become_red` message.

2.3.5 From Processes to External Controllers

M10 a `stored` message.

M11 a `retrieved` message.

M12 a `result` message.

M13 a `failure` message.

2.3.6 From External Controllers to Processes

M10 a `store` message.

M11 a `retrieve` message.

M12 a `first_key` message.

M13 a `last_key` message.

M14 a `num_keys` message.

M15 a `node_list` message.

M16 a `leave` message.

2.4 Initial States of the System

(a) TODO

2.5 Actions before and after Transitions

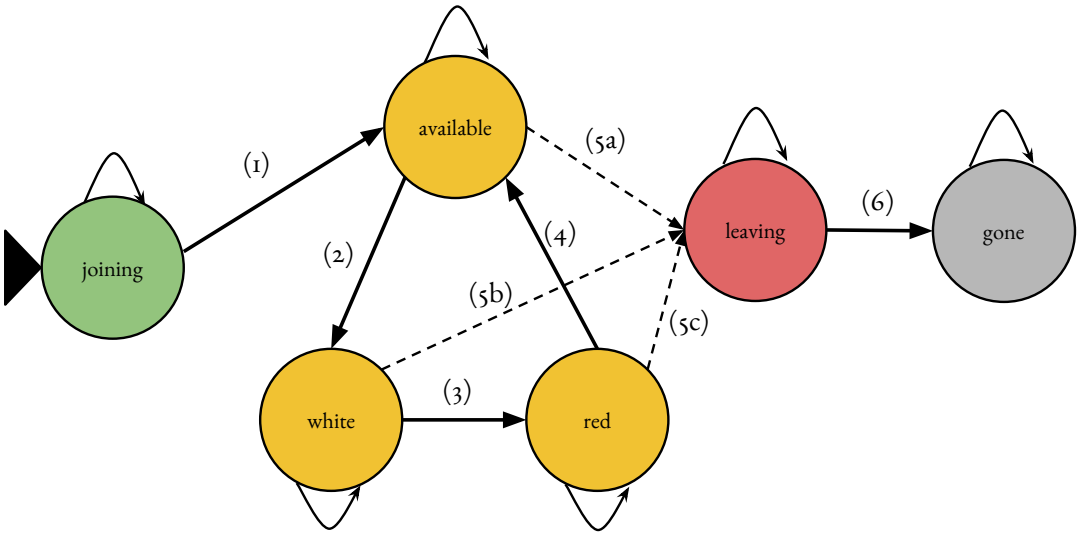

Figure 1: A diagram showing possible state transitions.

We will describe what would happen when a process receives a fork.

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

(5) $(p.leaving \vee p.red \vee p.white) \rightarrow p.leaving$**:** 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) $p.leaving \rightarrow p.gone$**:** 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.

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

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

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

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

Received in the *red* 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

2.7 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 *thinking* state.

Received in the *thinking* state: not possible.

Received in the *white* state: not possible.

Received in the *red* state: not possible.

Received in the *leaving* state: not possible.

Received in the *gone* state: not possible.

2.8 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 *thinking* state: removes the fork from the list and remove the neighbor from the list. It sends back `ok`.

Received in the *white* state: removes the fork from the list and removes the neighbor from the list. Then it checks if we have all the forks we need to eat. It sends back `ok`.

Received in the *red* state: removes the fork from the list and removes the neighbor from the list. Then it continues red. 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.

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

Received in the *joining* state: not possible.

Received in the *thinking* state: not possible.

Received in the *white* state: not possible.

Received in the *red* 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.

2.10 Actions for Incoming `become_white` Messages (M7)

Received in the *joining* state: not possible

Received in the *thinking* state: sends out requests to all neighbors for forks and then transition to white state.

Received in the *white* state: justs stay white.

Received in the *red* state: not possible.

Received in the *leaving* state: not possible.

Received in the *gone* state: not possible.

2.11 Actions for Incoming `stop_red` Messages (M??)

Received in the *joining* state: not possible.

Received in the *thinking* state: not possible.

Received in the *white* state: not possible.

Received in the *red* state: stops red, sends out the forks to all the processes that requested them, then transitions to the *thinking* state.

Received in the *leaving* state: not possible.

Received in the *gone* state: not possible.

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

Received in the *joining* state: not possible.

Received in the *thinking* 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 *white* state: transitions to *leaving*, which sends message to all neighbors that he's leaving. Once all processes give `ok`, then it transitions to gone.

Received in the *red* state: transitions to *leaving*, which sends message to all neighbors that he's 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.

2.13 How a new process informs its neighbors that it has joined the network

Sends joining requests to all of their neighbors.

2.14 How a new process knows that its neighbors are aware that it has joined the network

Waits for joining request acknowledgement messages from all of its neighbors.

2.15 How a process informs its neighbors that it is leaving the network

Sends leaving notifications to all of their neighbors.

2.16 How a process knows that its neighbors are aware that it is leaving the network

Waits for leaving notification acknowledgements from all of its neighbors.

3 Allowed Assumptions

3.1 External Controllers

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

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

A3 If a process p_1 receives a joining request from another process p_2, external controllers will not send signals to ask p_1 to leave until p_2 successfully joins the network.

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 Proof of Correctness

4.1 Proof of Safety Properties

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) $= 2^m$.

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.white* $\vee$ *p.leaving)*

When available, the process *p* can continue *available*, the external controller can issue the order to become white, or the controller may tell the process to leave. No other ”state” transitions are available to the process at the *available* state.

S5 *p.white* **next** *(p.white* $\vee$ *p.red* $\vee$ *p.leaving)*

If a process p is told by the external controller to become *white*, then it may be told to leave, it may become *red* due to its white nature, or p may remain *white*.

S6 *p.red* **next** *(p.red* $\vee$ *p.available* $\vee$ *p.leaving)*

If a process p is *red*, then only three cases are possible to the process. First, nothing may happen and the process will continue *red*. Second, the external controller can tell the process to `stop_ red`, in which p would become *available*. Third, the external controller can also tell the process to `leave`, in which p would become *leaving*. No other transitions are available at this stage.

S7 *p.leaving* **next** *(p.leaving* $\vee$ *p.gone)*

When a process p has been told to leave by the external controller, 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.

S8 *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*.

S9 $p.red \Rightarrow \langle \forall q | q \in p.neighbors \rhd \neg q.red \rangle$ (when a process p is *red*, none of its neighbors is *red*)

When a process p is *red*, then it holds all of its forks that it shares with its neighbors and since a process needs all of the forks it shares with its neighbors, that process with the forks will be the only one red.

S10 $(p.thinking \vee p.white \vee p.red) \Rightarrow \langle \forall q | q \in p.neighbors \rhd p \in q.neighbors \rangle$ (when a process p is *thinking*, *white*, or *red*, each of p's neighbors knows that p is one of its neighbors)

After joining, a leaving process q knows its neighbors and in each state *thinking*, *white*, or *red*, the neighbors list is updated if a neighbor leaves. Thus, each state has a real-time copy of the neighboring processes.

Before the process q can leave or remove a neighbor p who is either *thinking*, *white*, or *red* from *q.neighbors*, we guarantee that p remove q first.

S11 $p.gone \Rightarrow \langle \forall q \rhd p \notin q.neighbors \rangle$ (when a process p is *gone*, it is not in any other process's set of neighbors)

If p is *gone*, from our algorithm we know that $p.neighbors = \emptyset$. From this fact and from the safety property S10, we know that if q is *thinking*, *white*, or *red*, then p cannot be a neighbor q. If q is *leaving* or *gone*, $q.neighbors = \emptyset$ so $p \notin q.neighbors$. If q is *joining*, p cannot possibly be a neighbor of q due to our Assumption A3 that guarantees joining processes to be able to enter the network. Hence, we have covered all cases.

S12 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.

S13 $1 \leq n := \#(\text{nodes}) \leq 2^m$.

S14 $node_i.id \neq node_j.id$ if $i \neq j$ for all $0 \leq i, j \leq n-1$.

S15 $node_i.id \in \{0, 1, 2, \ldots, 2^m - 1\}$ for all $0 \leq i \leq n-1$.

S16 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 $node_n = node_0$.

S17 All storage processes are named in the global Erlang name registry.

S18 Each node must register at least one non-storage process (so that the node is always discoverable).

S19 If $\#(\text{nodes}) = n < 2^m$, then a new node can always join the system.

S20 A storage process can communicate to another storage process only if they are neighbors.

S21 A storage process can communicate to a non-storage process only if they are in the same node.

S22 A non-storage process can communicate to a storage process only if they are in the same node.

S23 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.

S24 Every message from the outside world (except for `leave`) must be responded to except for when relevant processes crash.

5 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 A3, 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)

6 Appendix: Relevant Code

```
1 TODO
```