The Algorithm for the Dynamic Dining Philosophers Problem

1 The Dynamic Dining Philosophers

Problem Statement. We need to develop and implement a protocol that allows nodes to join or leave a running dining philosophers network at runtime in an orderly fashion while maintaining an acyclic priority graph, ensuring safety, and preventing deadlock.

Our solution is based on a hiegenic solution discussed in class. To avoid deadlocks, we maintain a priority graphs, in which the person with a higher priority gets to eat first when two of adjacent philosophers are hungry. We use a fork mechanism to represent the priority graph in a distributed fashion. There is exactly one fork per each edge between two nodes, where nodes represent philosophers. Each fork can be either `dirty` or `clean`. A clean fork implies that the person holding it has a higher priority than the person at the other end of an edge. Conversely, a dirty fork implies that the person holding it has a lower priority. In this assignment, we extend this algorithm to allow philosphers to join and leave the table, while still maintain the acyclic priority graph, ensuring safey, and preventing deadlock.

2 The Algorithm

2.1 A set of states

There are 6 states:

(a) *joining*

(b) *thinking*

(c) *hungry*

(d) *eating*

(e) *leaving*

(f) *gone*

2.2 Information Stored by Each Process

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

```
1 philosophize(<state>, Node, Neighbors, ForksList)
```

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

(a) $p.state$ ($\langle$`state`$\rangle$): one of the six states outlined above.

(b) $p.id$ its process id (`Node`).

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

(d) $p.fork\_states$ (`ForkList`): a list $[f_{p,n_1}, f_{p,n_2}, f_{p,n_3}, \ldots, f_{p,n_k}]$ of fork states for each its neighbors, in the same order as the neighboring list. The fork state can be either 0 or 1, where 1 means the fork belongs to p, where as 0 means the fork belongs to its corresponding neigbor. For example, if $p.neighbors = [n_1, n_2, n_3]$ and $p.fork\_states = [0, 1, 0]$, then the fork f_{p,n_1} belongs to p; f_{p,n_2} belongs to n_2; and f_{p,n_3} belongs to n_3.

2.3 Message Types in the System

We categorize messages by its sender and its receiver:

2.3.1 From Philosophers to Philosophers

(a) a fork $f_{i,j}$: exactly one fork per one pair of neighboring processes i and j. A fork can be dirty (represented by 0) or clean (represented by 1).

(b) a fork request. The sender philosopher should only send this when it is in the *hungry* state and it does not hold a fork.

(c) a joining request. The sender philosopher, who should be in the joining state, once assigned by the external controller who to join, sends a joining request to each of its intended neighbors.

2.3.2 From External Controllers to Philosophers

(a) a `become_hungry` message. The philosopher should only receive this message while *thinking*; when it does, it transitions to *hungry*.

(b) a `stop_eating` message. The philosopher should only receive this message while *eating*; when it does, it transitions to *thinking*.

(c) a `leave` message. The philosopher can receive this message in any state other than *joining*; when it does, it should [TODO:Fill in the description of the protocol], then immediately leave the network.

2.3.3 From Philosophers to External Controllers

(a) an `eating` message. The philosopher should send this message, when it becomes *eating*, to the controller that trigger its transition to *hungry* (through the `become_hungry` message).

(b) a *gone* message. When it become *gone*, the philosopher should send this message to the controller that sent it the `leave` message that triggerred its departure.

2.4 Initial Distributions of Forks

(a) All forks are dirty.

(b) Initially, all forks are distributed to philosophers in an arbitrary way such that the corresponding graph is acyclic. We choose to use ids to decide which process has a fork initially; the process with a higher process id has a fork.

2.5 Actions before Transitions

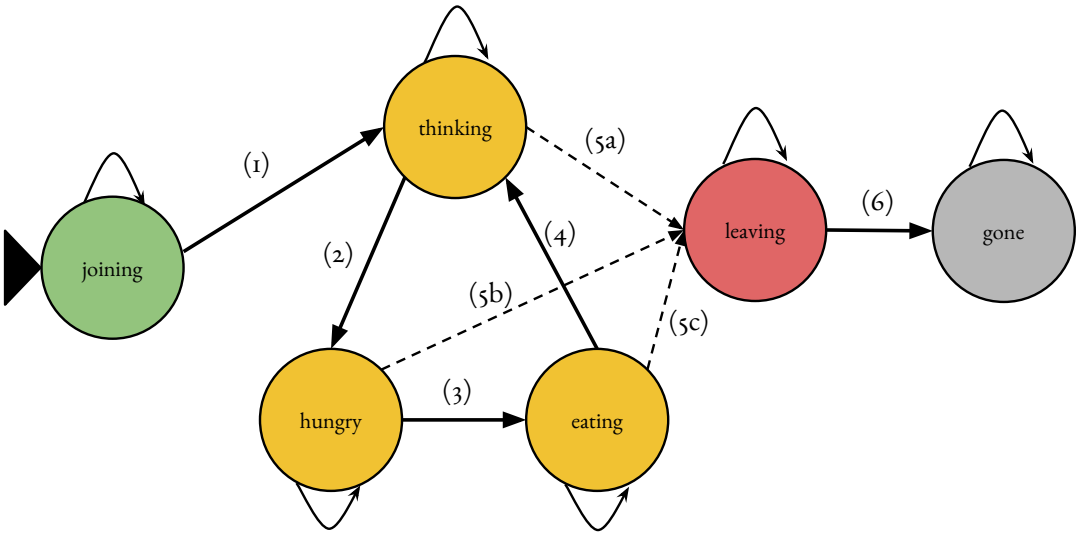

Figure 1: A diagram showing possible state transitions.

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

(1) $p.joining \rightarrow p.thinking$**:** After philosopher 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, all other philosophers are bound to hear p's request to join eventually and thus p is guaranteed to be given the state *thinking*.

(2) $p.thinking \rightarrow p.hungry$**:** The philosopher checks to see if it holds all forks. If so, performs the transition. If not, sends requests to all forks it does not have.

(3) $p.hungry \rightarrow p.eating$**:** When transitioning from $p.hungry$ to $p.eating$, the philosopher makes all forks dirty, thereby making its priority lowest among all its neighbors.

(4) $p.eating \rightarrow p.thinking$**:** The phisolopher sends any forks for which it received requests while eating.

(5) $(p.thinking \vee p.hungry \vee p.eating) \rightarrow p.leaving$**:** The philosopher can immediately go to the *leaving* state.

(6) $p.leaving \rightarrow p.gone$**:**

2.6 Incoming Request Messages

When a fork message is received, the philosopher marks the fork as clean.

During $p.hungry$ if the philosopher now holds all its forks, it transitions to *eating* state.

2.7 Incoming Fork Messages

3 Allowed Assumptions

3.1 External Controllers

E1 If a philosopher p informs an external controller that it is *eating*, the external controller will direct p to become either *thinking* or *leaving* within a bounded time.

E2 External controllers will not send duplicate or invalid control signals.

E3 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.

E4 Messages are never lost; sufficient time is allowed for a philosopher to bootstrap itself before other philosophers send it messages.

4 Proof of Correctness

4.1 Proof of Acyclic Priority Graph Invariance

The prioirity graph changes only in three possible cases.

Case 1: A new process joins the network. In this case, the new process has the lowest priority. Thus, all its neighbors will have arrow point to it. Thus, if there were a cycle, such cycle could not possibly pass this new process. Since the original graph was acyclic, there could not be cycles elsewhere as well. Thus, the resulting priority graph is still acyclic.

Case 2: A process leaves the network. A new priority graph is a subgraph of the original graph, and so the new graph cannot have cycles. If it were to have any cycles, these cycles would have presented in the original graph as well, which is not the case. Thus, the resulting priority network is acyclic.

Case 3: A process eats. In this case, based on our hiegenic algorithm, such process will then have the lowest priority among all its neighbors. Thus, all the arrow points to it. If there were a cycle, such cycle could not possibly pass this process. Since the original graph was acyclic, there could not be cycles elsewhere as well. Thus, the resulting priority graph is still acyclic.

4.2 Proof of Safety Properties

S1 **initially** *p.joining*

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

S2 *p.joining* **next** *(p.joining* $\vee$ *p.thinking)*

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

S3 $p.thinking$ **next** $(p.thinking \vee p.hungry \vee p.leaving)$

When thinking, the philosopher p can continue thinking, the external controller can issue the order to become hungry, or the controller may tell the philosopher to leave. No other "state" transitions are available to the philosopher at the *thinking* state.

S4 $p.hungry$ **next** $(p.hungry \vee p.eating \vee p.leaving)$

If a philosopher p is told by the external controller to become hungry, then it may be told to leave, it may become eating due to its hungry nature, or p may remain hungry (which would likely imply a failure in the system).

S5 $p.eating$ **next** $(p.eating \vee p.thinking \vee p.leaving)$

S6 $p.leaving$ **next** $(p.leaving \vee p.gone)$

S7 $p.gone$ **next** $(p.gone)$

S8 $p.eating \Rightarrow \langle \forall q | q \in p.neighbors \triangleright \neg q.eating \rangle$ (when a philosopher p is *eating*, none of its neighbors is *eating*)

S9 $(p.thinking \vee p.hungry \vee p.eating) \Rightarrow \langle \forall q | q \in p.neighbors \triangleright p \in q.neighbors \rangle$ (when a philosopher p is *thinking*, *hungry*, or *eating*, each of p's neighbors knows that p is one of its neighbors)

S10 $p.gone \Rightarrow \langle \forall q \triangleright \notin q.neighbors \rangle$ (when a philosopher p is *gone*, it is not in any other philosopher?s set of neighbors)

5 Proof of Progress Properties

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

After philosopher 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 E3, all other philosophers are bound to hear p's request to join eventually and thus p is guaranteed to be given the state *thinking*.

PG2 $p.hungry \rightsquigarrow p.eating$

When *hungry*, a philosopher p must be passed the forks once all neighbor philosophers are not eating. TODO [Articulate this...but this is the idea]: Based on the acyclic graph property and our algorithm to make just-finished-eating processes have the lowest priority among their neighbors, we guarantee that the a hungry philosopher will eventually eat.

PG3 $p.leaving \rightsquigarrow p.gone$

6 Appendix: Relevant Code

```
1 infinite_loop(Ref, Nodel, Neighbors) ->
2     philosophize, Node} ! {self(), Ref, become_hungry},
3     % {spelling, Node} ! {self(), Ref, stop_eating},
4     % {spelling, Node} ! {self(), Ref, leave},
5     receive
```

```
6         %{Ref, eating} ->
7         %print("~p is eating.~n", [Ref]);
8         {Ref, gone} ->
9         print("~p is gone.~n", [Ref]);
10        Reply ->
11        print("Got unexpected message: ~p~n", [Reply])
12        after ?TIMEOUT -> print("Timed out waiting for reply!")
13    end,
14    infinite_loop(Ref, Nodel, Neighbors)
15 end.
```

```
1  philosophize(Ref, joining, Node, ForksNeighborsList)->
2          print("joining"),
3          %philosophize(Ref, thinking, Node, Neighbors);
4      requestJoin(Ref, Node, ForksNeighborsList),
5      philosophize(Ref, thinking, ForksNeighborsList).
6
7  %requests each neighbor to join the network, one at a time,
8  %when joining there shouldn't be any other requests for forks or leaving going on
9  requestJoin(_,_,[])-> ok;
10 requestJoin(Ref, Node, ForksNeighborsList)->
11     try
12         io:format("Process ~p at node ~p sending request to ~n~s",
13             [self(), Node, hd(Neighbors)]),
14         io:format("After~n"),
15         {list_to_atom(hd(Neighbors)), Node} ! {self(), Ref, requestJoin},
16         receive
17             {Ref, ok} ->
18                 io:format("Got reply (from ~p): ok!",
19                           [Ref);
20             Reply ->
21                 io:format("Got unexpected message: ~p~n", [Reply])
22         after ?TIMEOUT -> io:format("Timed out waiting for reply!")
23         end,
24         requestJoin(Ref, Node, tl(Neighbors))
25     catch
26         _:_ -> io:format("Error getting joining permission.~n")
27     end.
```

```
1  philosophize(Ref, thinking, Node, ForksNeighborsList)->
2          receive
3             {self(), NewRef, leave} ->
4             print("leaving"),
5             philosophize(NewRef, leaving, ForksNeighborsList);
6             {self(), NewRef, become_hungry} ->
7                     print("becoming hungry"),
8                     philosophize(NewRef, hungry, ForksNeighborsList)
9    after ?TIMEOUT -> print("Timed out waiting for reply!")
10         end;
```

```
1  philosophize(Ref, hungry, Node, ForksNeighborsList)->
2          receive
3              {self(), NewRef, leave} ->
4                      print("leaving"),
5                      philosophize(NewRef, leaving, Node, ForksNeighborsList);
6          %   {self(), NewRef, Fork} -> AND/OR CHECK IF ALL NEIGHBORS ARE NOT EATING?
7                  %print("got fork"),
8                  % check if has all forks
```

```
9                % continue with philosophize(NewRef,
10
11          %end;
12
13          % want to receive all forks and then start eating
14          {controller, Node} ! {NewRef, eating},
15          print("eating"),
16          philosophize(Ref, eating, Node, ForksNeighborsList)
17          end;
```

```
1  philosophize(Ref, eating, Node, ForksNeighborsList)->
2           receive
3              {self(), NewRef, stop_eating} ->
4                   % handle forks and hygenity if doing that
5                   print("stopping eating"),
6                   philosophize(NewRef, thinking, Node, ForksNeighborsList);
7              {self(), NewRef, leave} ->
8                   print("stopping eating and leaving"),
9                   %get rid of forks
10                  philosophize(NewRef, leaving, Node, ForksNeighborsList)
11          after ?TIMEOUT -> print("Timed out waiting for reply!")
12          end;
```

```
1  philosophize(Ref, leaving, Node, ForksNeighborsList)->
2           %need to gather forks and then leave with them
3           {controller, Node} ! {Ref, gone}.
```