Algorithms: The Dynamic Dining Philosophers

1 Introduction

1.1 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.

2 Allowed Assumptions

2.1 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.

2.2 Others

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

3 Solution

3.1 Overview

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.

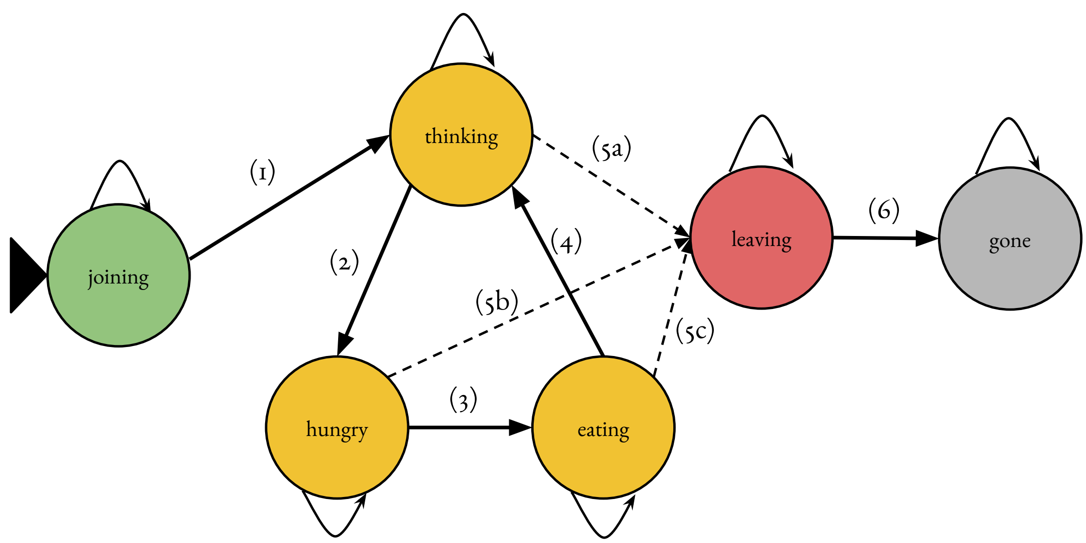
4 Algorithms

4.1 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.

4.2 Actions before 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 *leaving* state.

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

4.3 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.

4.4 Incoming Fork Messages

5 Proof of Correctness

5.1 Proof of Safety Properties

In order to show that the algorithm satisfies the Safety requirement, we must show that the relevant stages satisfy the Safety properties:

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.

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).

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.

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).

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

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

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

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

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

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

6 Appendix: Relevant Code

```
infinite_loop(Ref, Nodel, Neighbors) ->
    philosophize, Node} ! {self(), Ref, become_hungry},
   % {spelling, Node} ! {self(), Ref, stop_eating},
   % {spelling, Node} ! {self(), Ref, leave},
    receive
        %{Ref, eating} ->
        %print("~p is eating.~n", [Ref]);
        {Ref, gone} ->
        print("~p is gone.~n", [Ref]);
        Reply ->
        print("Got unexpected message: ~p~n", [Reply])
        after ?TIMEOUT -> print("Timed out waiting for reply!")
    end,
    infinite_loop(Ref, Nodel, Neighbors)
end.
```

```
philosophize(Ref, joining, Node, ForksNeighborsList)->
        print("joining"),
        %philosophize(Ref, thinking, Node, Neighbors);
    requestJoin(Ref, Node, ForksNeighborsList),
    philosophize(Ref, thinking, ForksNeighborsList).

%requests each neighbor to join the network, one at a time,
%when joining there shouldn't be any other requests for forks or leaving going on
requestJoin(_,_,[])-> ok;
requestJoin(Ref, Node, ForksNeighborsList)->
    try
        io:format("Process ~p at node ~p sending request to ~n~s",
            [self(), Node, hd(Neighbors)]),
        io:format("After~n"),
        {list_to_atom(hd(Neighbors)), Node} ! {self(), Ref, requestJoin},
        receive
            {Ref, ok} ->
                io:format("Got reply (from ~p): ok!",
                          [Ref]);
            Reply ->
                io:format("Got unexpected message: ~p~n", [Reply])
        after ?TIMEOUT -> io:format("Timed out waiting for reply!")
        end,
        requestJoin(Ref, Node, tl(Neighbors))
    catch
        _:_ -> io:format("Error getting joining permission.~n")
    end.
```

```
philosophize(Ref, thinking, Node, ForksNeighborsList)->
        receive
           {self(), NewRef, leave} ->
           print("leaving"),
           philosophize(NewRef, leaving, ForksNeighborsList);
           {self(), NewRef, become_hungry} ->
                   print("becoming hungry"),
                   philosophize(NewRef, hungry, ForksNeighborsList)
  after ?TIMEOUT -> print("Timed out waiting for reply!")
        end;
```

```
philosophize(Ref, hungry, Node, ForksNeighborsList)->
        receive
```

```
            {self(), NewRef, leave} ->
                    print("leaving"),
                    philosophize(NewRef, leaving, Node, ForksNeighborsList);
        %   {self(), NewRef, Fork} -> AND/OR CHECK IF ALL NEIGHBORS ARE NOT EATING?
                %print("got fork"),
                % check if has all forks
                % continue with philosophize(NewRef,

        %end;

        % want to receive all forks and then start eating
        {controller, Node} ! {NewRef, eating},
        print("eating"),
        philosophize(Ref, eating, Node, ForksNeighborsList)
        end;
```

```
philosophize(Ref, eating, Node, ForksNeighborsList)->
        receive
           {self(), NewRef, stop_eating} ->
                % handle forks and hygenity if doing that
                print("stopping eating"),
                philosophize(NewRef, thinking, Node, ForksNeighborsList);
           {self(), NewRef, leave} ->
                print("stopping eating and leaving"),
                %get rid of forks
                philosophize(NewRef, leaving, Node, ForksNeighborsList)
        after ?TIMEOUT -> print("Timed out waiting for reply!")
        end;
```

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