

# The Algorithm for the Distributed Key-Value Storage Problem

## 1 The Distributed Key-Value Storage

---

**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 hygienic 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 philosophers to join and leave the table, while still maintain the acyclic priority graph, ensuring safety, and preventing deadlock.

## 2 The Algorithm

### 2.1 A set of states

Each of the philosophers must be in one of the following 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)  $p.state$  (`<state>`): one of the six states outlined above.

- (b) *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.
- (c) *p.neighbors* (Neighbors): a list of its neighboring processes  $[n_1, n_2, \dots, n_k]$ .
- (d) *p.fork\_states* (ForkList): a list  $[f_{p,n_1}, f_{p,n_2}, f_{p,n_3}, \dots, f_{p,n_k}]$  of fork states for each its neighbors, in the same order as the neighboring list. The fork is a 2-tuple  $\{holding, hygiene\}$ , where *holding* can be either 0 or 1, where 1 means the fork belongs to *p*, where as 0 means the fork belongs to its corresponding neighbor. The second element in the tuple is *holding*, which can be either CLEAN, DIRTY, or SPAGHETTI to establish the priority. 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. The value SPAGHETTI (or sometimes SPAGHETTI SAUCE) or something else indicates a placeholder. For example, if *holding* is 0 (not holding fork), *hygiene* is not meaningful anymore. That is, *hygiene* has a meaning only when the philosopher is holding a fork.

The corresponding global priority graph can be defined as follows. The nodes of the graph is the processes in the system graph, and there is a directed edge  $(p, q)$  from *p* to *q* if and only if *p* is a neighbor of *q* in the system graph and *p* has a higher priority than *q*.

For example, if  $p.neighbors = [n_1, n_2, n_3]$  and  $p.fork\_states = [\{0, -\}, \{1, CLEAN\}, \{0, -\}]$ , 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  $n_3$ . This means *p* has a higher priority than  $n_2$  but lower than  $n_1$  and  $n_3$ .

## 2.3 Message Types in the System

We categorize messages by its sender and its receiver:

### 2.3.1 From Philosophers to Philosophers

- M1 a fork: exactly one fork per one pair of neighboring processes. A fork can be DIRTY (the process holding it has lower priority) or CLEAN (the process holding it has higher priority).
- M2 a fork request. The sending philosopher should only send this when it is in the *hungry* state and it does not hold a fork.
- M3 a joining request. The sending 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. When the receiving philosopher receives this message, it adds the sender to its neighbors list.
- M4 a joining message acknowledgement. Once a joining philosopher receives an acknowledgement from a neighbor that it sent a joining request to, it knows that the sending philosopher are aware of its joining. Once it knows that all intended neighbors are aware of its joining, it can transition to the *thinking* state.
- M5 a leaving notification. The sending philosopher, who should be in either the *thinking*, *hungry*, or *eating* state, once signaled by an external controller to leave, it transitions to the leaving state and sends a leaving notification to all its neighbors.

M6 a leaving message acknowledgement. A philosopher, once received a leaving notification message, deletes the sender from its neighbors list, and immediately sends a leaving message acknowledgement to the sender.

### 2.3.2 From External Controllers to Philosophers

M7 a `become_hungry` message. The philosopher should only receive this message while *thinking*; when it does, it transitions to *hungry*.

M8 a `stop_eating` message. The philosopher should only receive this message while *eating*; when it does, it transitions to *thinking*. Just before transitioning to thinking, it will immediately handle any previous requests for forks.

M9 a `leave` message. The philosopher can receive this message in any state other than *joining*; when it does, it should ask for acknowledgment from all its neighbors to say that it is leaving, then immediately leaves the network by transitioning to *gone*.

### 2.3.3 From Philosophers to External Controllers

M10 an `eating` message. The philosopher should send this message, when it becomes *eating*, to the controller that triggered its transition to *hungry* through the `become_hungry` message.

M11 a `gone` message. When it becomes *gone*, the philosopher should send this message to the controller that sent it the `leave` message that triggered its departure.

## 2.4 Initial Distributions of Forks

(a) All forks are dirty.

(b) Initially, the first philosopher has no fork. When another philosopher requests to be its neighbor, it sends a joining request. The receiving philosopher will create a dirty fork. This prevents cycles since there is a possibility that if the fork was clean, he would never give it up.

## 2.5 Actions before and after Transitions

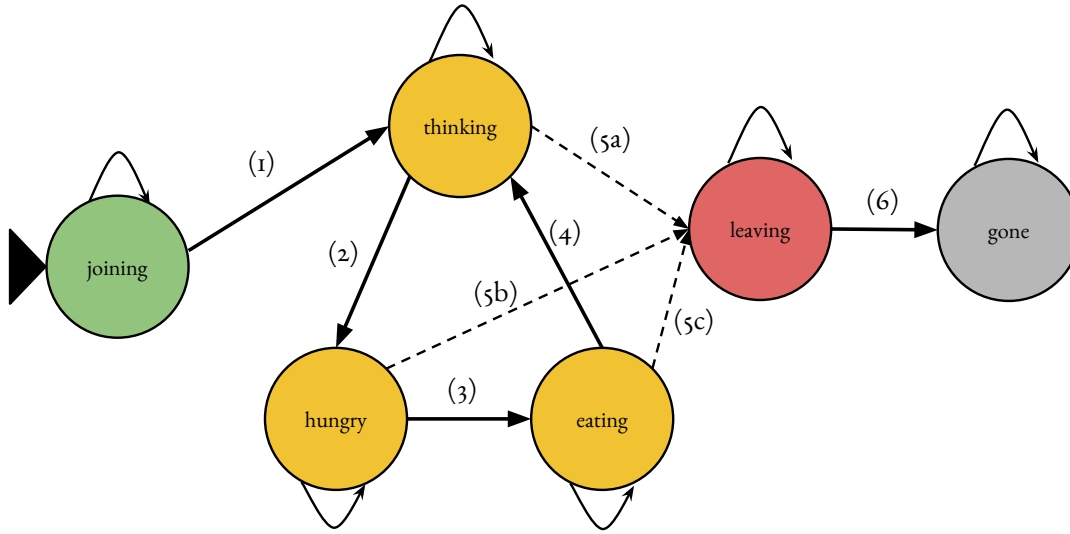


Figure 1: A diagram showing possible state transitions.

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

- (1)  $p.joining \rightarrow p.thinking$ : 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.
- (2)  $p.thinking \rightarrow p.hungry$ : check to see if has already received a `become_hungry` message from an external controller. If so, stores the identification of the external controller that triggered it to become *hungry*, then transitions to the *hungry* state. Once it becomes *hungry*, it sends a fork request to all neighbors that hold that forks that it needs.
- (3)  $p.hungry \rightarrow p.eating$ : checks to see if it holds all forks. If so, performs the transition and sends an `eating` message to the external controller that triggered it to become *hungry* and make all forks `DIRTY`, thereby making its priority lowest among all its neighbors. If not, it waits until it receives another fork and checks if he has them all again, until it actually has them all and transitions to *eating*, then sends an `eating` message to the external controller that triggered it to become *hungry* and make all forks `DIRTY`.
- (4)  $p.eating \rightarrow p.thinking$ : checks to see if it has already received a `stop_eating` message from an external controller. If so, performs the transition and sends any forks for which it received requests while eating.
- (5)  $(p.thinking \vee p.hungry \vee p.eating) \rightarrow p.leaving$ : checks to see if it has already received a `leave` message from an external controller. If so, the philosopher 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 philosopher  $p$  receives an incoming message depending on what state it currently is in.

## 2.6 Actions for Incoming Fork Messages (M1)

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

**Received in the *joining* state:** not possible.

**Received in the *thinking* state:** not possible.

**Received in the *hungry* state:** the process adds the fork to his list of forks marked as a `CLEAN` fork. It then checks if it has all the forks it needs to start eating, if not it will continue waiting for forks.

**Received in the *eating* state:** not possible, already had a fork.

**Received in the *leaving* state:** the process will ignore it since it send a message telling that process it's leaving. That process will eventually get the message and acknowledge the leaving process is leaving, removing that fork from the edge.

**Received in the *gone* state:** not possible.

## 2.7 Actions for Incoming Fork Request Messages (M2)

**Received in the *joining* state:** Since all the joining nodes has all dirty forks, it will give the requested forks to the requesting philosopher.

**Received in the *thinking* state:** sends the requested fork to the requesting philosopher.

**Received in the *hungry* state:** checks whether the fork is `DIRTY` or `CLEAN`. If the fork is `DIRTY`, it sends the requested fork to the requesting philosopher, and if it's `CLEAN`, it holds onto it, and makes a note that the requesting philosopher wanted it.

**Received in the *eating* state:** saves the request until it transitions to the *thinking* state. If the process leaves, the requesting process will get a leaving notification message and knows that that fork is no longer necessary to eat and delete that fork from its list.

**Received in the *leaving* state:** ignores.

**Received in the *gone* state:** not possible.

## 2.8 Actions for Incoming Joining Request Messages (M3)

**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 *hungry* state:** approves request and creates a dirty fork for the edge.

**Received in the *eating* 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

## 2.9 Actions for Incoming Joining Message Acknowledgement Messages (M4)

**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 *hungry* state:** not possible.

**Received in the *eating* state:** not possible.

**Received in the *leaving* state:** not possible.

**Received in the *gone* state:** not possible.

## 2.10 Actions for Incoming Leaving Notification Messages (M5)

**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 *hungry* 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 *eating* state:** removes the fork from the list and removes the neighbor from the list. Then it continues eating. 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.

## 2.11 Actions for Incoming Leaving Message Acknowledgement Messages (M6)

**Received in the *joining* state:** not possible.

**Received in the *thinking* state:** not possible.

**Received in the *hungry* state:** not possible.

**Received in the *eating* 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.

### 2.12 Actions for Incoming **become\_hungry** 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 hungry state.

Received in the *hungry* state: just stay hungry.

Received in the *eating* state: not possible.

Received in the *leaving* state: not possible.

Received in the *gone* state: not possible.

### 2.13 Actions for Incoming **stop\_eating** Messages (M8)

Received in the *joining* state: not possible.

Received in the *thinking* state: not possible.

Received in the *hungry* state: not possible.

Received in the *eating* state: stops eating, 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.

### 2.14 Actions for Incoming **leave** Messages (M9)

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 *hungry* 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 *eating* 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.

### 2.15 How a new philosopher informs its neighbors that it has joined the network

Sends joining requests to all of their neighbors.

### 2.16 How a new philosopher knows that its neighbors are aware that it has joined the network

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

### 2.17 How a philosopher informs its neighbors that it is leaving the network

Sends leaving notifications to all of their neighbors.

### 2.18 How a philosopher 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 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.

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 philosopher to bootstrap itself before other philosophers send it messages.

## 4 Proof of Correctness

### 4.1 Proof of Acyclic Priority Graph Invariance

The priority 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 hygienic 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 it 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*.

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

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

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

When a philosopher  $p$  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 philosopher could complete the *leaving* state and leave, successfully terminating and entering the *gone* state.

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

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

When a philosopher  $p$  is *eating*, then it holds all of its forks that it shares with its neighbors and since a philosopher needs all of the forks it shares with its neighbors, that philosopher with the forks will be the only one 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)

After joining, a leaving philosopher  $q$  knows its neighbors and in each state *thinking*, *hungry*, or *eating*, the neighbors list is updated if a neighbor leaves. Thus, each state has a real-time copy of the neighboring philosophers.

Before the philosopher  $q$  can leave or remove a neighbor  $p$  who is either *thinking*, *hungry*, or *eating* from  $q.neighbors$ , we guarantee that  $p$  remove  $q$  first.

S10  $p.gone \Rightarrow \langle \forall q \triangleright p \notin q.neighbors \rangle$  (when a philosopher  $p$  is *gone*, it is not in any other philosopher'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 S9, we know that if  $q$  is *thinking*, *hungry*, or *eating*, 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 philosophers to be able to enter the network. Hence, we have covered all cases.

## 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 A3, 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  $p$  sends fork requests to all its neighbors from which it needs forks. Due to our Assumption A1, we know that an *eating* philosopher will take a finite time. When a neighboring *eating* philosopher finished eating, it processes the fork request from  $p$ . Because our algorithm makes just-finished-eating processes have the lowest priority among their neighbors (in particular,  $p$ ) by making the fork `DIRTY`, they cannot hold on to that fork and must pass the fork to  $p$ . Thus, eventually  $p$  will have all the forks it needs, so it can eventually becomes *eating*.

PG3  $p.leaving \rightsquigarrow p.gone$  When  $p$  is in the *leaving* state, it sends leaving notifications to all its neighbors. Since messages are never lost, all of its neighbors will eventually get the messages and send acknowledgements back. Once it receives all acknowledgements, it can transition to *gone*.

## 6 Appendix: Relevant Code

```

1 %% CSCI182E - Distributed Systems
2 %% Harvey Mudd College
3 %% Dynamic Dining Philosophers
4 %% @author Tum Chaturapruek, Patrick Lu, Cory Pruce
5 %% @doc Think, hungry, and eat.
6 -module(philosopher).
7
8 %% =====
9 %%                                     Public API
10 %% =====
11 -export([main/1]).
12
13 %% =====
14 %%                                     Constants

```

```

15 %% =====
16
17 %% =====
18 %%                               Main Function
19 %% =====
20 % The main/1 function.
21 main(Params) ->
22     % try
23     % The first parameter is destination node name
24     % It is a lowercase ASCII string with no periods or @ signs in it.
25     NodeName = hd(Params),
26
27     % 0 or more additional parameters, each of which is the Erlang node
28     % name of a neighbor of the philosopher.
29     NeighborsList = tl(Params),
30     Neighbors = lists:map(fun(Node) -> list_to_atom(Node) end,
31         NeighborsList),
32     %% IMPORTANT: Start the empd daemon!
33     os:cmd("epmd -daemon"),
34
35     % format microseconds of timestamp to get an
36     % effectively-unique node name
37     net_kernel:start([list_to_atom(NodeName), shortnames]),
38
39     register(philosopher, self()),
40
41     %joining
42     philosophize(joining, Neighbors, dict:new()),
43
44     halt().
45
46
47 % This is a helper function to that sends forks to all
48 % processes in a list.
49 sendForks([], ForksList) -> ForksList;
50 sendForks(Requests, ForksList) ->
51     print("Sending the fork to ~p~n", [hd(Requests)]),
52     ForkList = dict:erase(hd(Requests), ForksList),
53     NewForkList = dict:append(hd(Requests), {0, "SPAGHETTI"}, ForkList),
54     {philosopher, hd(Requests)} ! {node(), fork},
55     sendForks(tl(Requests), NewForkList).
56
57 % Check to see if a process has all the forks
58 % it needs to eat
59 haveAllForks([], _) -> true;
60 haveAllForks(Neighbors, ForksList) ->
61     case (dict:find(hd(Neighbors), ForksList)) of
62         %Request the fork
63         {ok, [{0, _}]} -> print("Don't have all forks~n"),
64             false;
65         {ok, [{1, _}]} -> haveAllForks(tl(Neighbors), ForksList)
66     end.
67
68 % Constantly query neighbor philosophers to make sure that they are still
69 % there. If one is gone, delete fork to that philosopher and remove from
70 % neighbors list, sufficiently removing the edge. Otherwise, keep
71 % philosophizing.
72
73 check_neighbors([], _) -> ok;

```

```

74 check_neighbors([X|XS], ParentPid) ->
75     spawn(fun() -> monitor_neighbor(X, ParentPid) end),
76     check_neighbors(XS, ParentPid).
77
78 monitor_neighbor(Philosopher, ParentPid) ->
79     erlang:monitor(process, {philosopher, Philosopher}), %{RegName, Node}
80     receive
81         {DOWN', _Ref, process, _Pid, normal} ->
82             ParentPid ! {self(), check, Philosopher};
83         {DOWN', _Ref, process, _Pid, _Reason} ->
84             ParentPid ! {self(), missing, Philosopher}
85     end.
86
87 %requests each neighbor to join the network, one at a time,
88 %when joining there shouldn't be any other requests for forks or leaving going on
89 %If another process requests to join during the joining phase, hold onto it until
90 %successfully joined and then handle it using the erlang mailbox.
91 requestJoin([], ForksList) -> ForksList;
92 requestJoin(Neighbors, ForksList) ->
93     print("Process ~p at node ~p sending request to ~s~n",
94         [self(), node(), hd(Neighbors)]),
95     {philosopher, hd(Neighbors)} ! {node(), requestJoin},
96     receive
97         {Node, ok} ->
98             print("!Got reply (from ~p): ok!~n", [Node]),
99             ForkList = dict:append(Node, {0, "SPAGHETTI SAUCE"}, ForksList),
100             requestJoin(tl(Neighbors), ForkList)
101     end.
102
103 %% im pretty sure that the message passing should be the other way around since
104 %% we are only writing the philosophers' code and not the external controller's.
105 %% Was the infinite-loop intended to be a test controller? Also, should we keep using
106 %% NewRef or just use NewRef for eating?
107 requestForks([], _) -> print("No more neighbors to request ~n");
108 requestForks(Neighbors, ForksList) ->
109     %See if we have the fork from this edge
110     case (dict:find(hd(Neighbors), ForksList)) of
111         %Request the fork if 0
112         {ok, [{0, _}]} -> print("Requesting fork~n"),
113             {philosopher, hd(Neighbors)} ! {node(), requestFork};
114         {ok, [{1, _}]} -> ok
115     end,
116     requestForks(tl(Neighbors), ForksList).
117
118 % This is the joining state, initially the process will try to join
119 % by getting acknowledgement from all its neighbors. Only when it has
120 % acknowledgement can it transition to thinking
121 philosophize(joining, Neighbors, ForkList) ->
122     print("Joining~n"),
123     %philosophize(Ref, thinking, Neighbors);
124     ForksList = requestJoin(Neighbors, ForkList),
125     print("Requested to join everybody~n"),
126     %% spawn processes to monitor neighbors once joined
127     check_neighbors(Neighbors, self()),
128     %now we start thinking
129     philosophize(thinking, Neighbors, ForksList);
130
131 %When thinking, we can be told to leave, to become hungry,

```

```

132 %or get request for a fork
133 philosophize(thinking, Neighbors, ForksList)->
134   print("Thinking~n"),
135   receive
136     % Told by exteranl controller to leave
137     {NewNode, leaving} ->
138       print("~p left, removing him from lists", [NewNode]),
139       NewNeighbors = lists:delete(NewNode, Neighbors),
140       NewForkList = dict:erase(NewNode, ForksList),
141       philosophize(thinking, NewNeighbors, NewForkList);
142     % Told by another philosopher that he's leaving
143     {Pid, NewRef, leave} ->
144       print("Leaving~n"),
145       philosophize(leaving, Neighbors, ForksList, Pid, NewRef);
146
147     % Told to become hungry
148     {Pid, NewRef, become_hungry} ->
149       print("becoming hungry~n"),
150       %Send fork requests to everyone
151       requestForks(Neighbors, ForksList),
152       print("Sent requests for forks~n"),
153       case (haveAllForks(Neighbors, ForksList)) of
154         true -> Pid ! {NewRef, eating},
155               print("Have all forks!~n"),
156               philosophize(eating, Neighbors, ForksList, []);
157         false -> print("Don't have all forks :(~n"),
158               philosophize(hungry, Neighbors, ForksList, [], Pid, NewRef)
159       end;
160
161     % Another process requests to join
162     {NewNode, requestJoin} ->
163       print("~p requested to Join, accepting~n", [NewNode]),
164       NewNeighbors = lists:append(Neighbors, [NewNode]),
165       ForkList = dict:append(NewNode, {1, "DIRTY"}, ForksList),
166       {philosopher, NewNode} ! {node(), ok},
167       philosophize(thinking, NewNeighbors, ForkList);
168     % Another philosopher is checking if this philosopher is still running
169     {_Pid, missing, Who} ->
170       print("~p has gone missing!~n", [Who]),
171       NewNeighbors = Neighbors - [Who],
172       ForkList = dict:erase(Who, ForksList),
173       philosophize(thinking, NewNeighbors, ForkList);
174     % monitor alerting that a leaving philosopher has left
175     {_Pid, check, Who} ->
176       print("~p has left for sure, more SPAGHETTI for me!~n", [Who]),
177       NewNeighbors = Neighbors - [Who],
178       ForkList = dict:erase(Who, ForksList),
179       philosophize(thinking, NewNeighbors, ForkList);
180     % We get a request for a fork, which we send since we don't need it
181     {NewNode, requestFork} ->
182       print("sending fork to ~p~n", [NewNode]),
183       %delete the fork from the list send message
184       ForkList = dict:erase(NewNode, ForksList),
185       NewForkList = dict:append(NewNode, {0, "DIRTY"}, ForkList),
186       {philosopher, NewNode} ! {node(), fork},
187       philosophize(thinking, Neighbors, NewForkList)
188   end.
189
190

```

```

191 % Eating phase, the philosopher has all the forks it needs from its neighbors,
192 % eventually exits back to thinking or leaving, requests are handled once told
193 % to stop eating
194 philosophize(eating, Neighbors, ForksList, Requests) ->
195     print("eating!~n"),
196     receive
197         % Told by external controller to leave
198         {NewNode, leaving} ->
199             print("~p left, removing him from lists~n", [NewNode]),
200             NewNeighbors = lists:delete(NewNode, Neighbors),
201             NewForkList = dict:erase(NewNode, ForksList),
202             philosophize(eating, NewNeighbors, NewForkList, Requests);
203         % Told by another philosopher that he's leaving
204         {Pid, NewRef, leave} ->
205             print("leaving~n"),
206             philosophize(leaving, Neighbors, ForksList, Pid, NewRef);
207         % Told by external controller to stop eating
208         {Pid, NewRef, stop_eating} ->
209             print("stopped_eating~n"),
210             % send the forks to the processes that wanted them
211             ForkList = sendForks(Requests, ForksList),
212             Pid ! {NewRef, fork},
213             philosophize(thinking, Neighbors, ForkList);
214         % Another philosopher is checking if this philosopher is still running
215         {_Pid, missing, Who} ->
216             print("~p has gone missing!~n", [Who]),
217             NewNeighbors = Neighbors - [Who],
218             ForkList = dict:erase(Who, ForksList),
219             philosophize(thinking, NewNeighbors, ForkList);
220         % monitor alerting that a leaving philosopher has left
221         {_Pid, check, Who} ->
222             print("~p has left for sure, more SPAGHETTI for me!~n", [Who]),
223             NewNeighbors = Neighbors - [Who],
224             ForkList = dict:erase(Who, ForksList),
225             philosophize(thinking, NewNeighbors, ForkList);
226         % Another process requests to join
227         % Handle when another process wants to join us
228         {NewNode, requestJoin} ->
229             print("~p requested to join~n", [NewNode]),
230             % if we get a join request, just create the fork and give acknowledgement
231             NewRequests = lists:append(Neighbors, [NewNode]),
232             ForkList = dict:append(NewNode, {1, "DIRTY"}, ForksList),
233             {philosopher, NewNode} ! {node(), ok},
234             philosophize(eating, Neighbors, ForkList, NewRequests)
235     end.
236 % Is hungry, already requested all the forks
237 philosophize(hungry, Neighbors, ForksList, RequestList, Pid, Ref)->
238     print("Hungry, waiting for forks~n"),
239     receive
240         % Get the fork from the other process
241         {NewPid, NewRef, leave} ->
242             print("Leaving~n"),
243             philosophize(leaving, Neighbors, ForksList, NewPid, NewRef);
244         {NewNode, leaving} ->
245             print("~p left, removing him from lists~n", [NewNode]),
246             NewNeighbors = lists:delete(NewNode, Neighbors),
247             NewForkList = dict:erase(NewNode, ForksList),
248             case (haveAllForks(NewNeighbors, NewForkList)) of
249                 true -> Pid ! {Ref, eating},

```

```

250         philosophize(eating, NewNeighbors, NewForkList, RequestList);
251     false -> philosophize(hungry, NewNeighbors, NewForkList, RequestList,
        Pid, Ref)
252     end;
253     % Get a fork from a process, add it to the list and see if we have
254     % all of them to eat
255     {NewNode, fork} ->
256         print("Got fork from ~p~n",[NewNode]),
257         ForkList = dict:erase(NewNode, ForksList),
258         NewForkList = dict:append(NewNode, {1, "CLEAN"}, ForkList),
259         case (haveAllForks(Neighbors, NewForkList)) of
260             true -> Pid ! {Ref, eating},
261                 philosophize(eating, Neighbors, NewForkList, RequestList);
262             false -> philosophize(hungry, Neighbors, NewForkList, RequestList, Pid,
        Ref)
263     end;
264     % Another process joins, create the fork and give acknowledgement
265     {NewNode, requestJoin} ->
266         print("~p requested to join~n",[NewNode]),
267         %if we get a join request, just create the fork and give acknowledgement
268         dict:append(NewNode, [1, "DIRTY"], ForksList),
269         {philosopher, NewNode} ! {node(), ok};
270         % if someone requests a fork
271         % Another philosopher is checking if this philosopher is still running
272         {Pid, missing, Who} ->
273             print("~p has gone missing!~n",[Who]),
274             NewNeighbors = Neighbors - [Who],
275             ForkList = dict:erase(Who, ForksList),
276             philosophize(thinking, NewNeighbors, ForkList);
277     % monitor alerting that a leaving philosopher has left
278     {Pid, check, Who} ->
279         print("~p has left for sure, more SPAGHETTI for me!~n", [Who]),
280         NewNeighbors = Neighbors - [Who],
281         ForkList = dict:erase(Who, ForksList),
282         philosophize(thinking, NewNeighbors, ForkList);
283     %% Check the priority and give the fork only if they have
284     % higher priority
285     {NewNode, requestFork} ->
286         print("~p requested the fork~n",[NewNode]),
287         case (dict:find(NewNode, ForksList)) of
288             %Check status
289             {ok, [{1, "CLEAN"}]} -> print("My fork!, but I'll remember you wanted it~n"),
290                                     RequestsList = lists:append(RequestList, [NewNode]),
291                                     philosophize(hungry, Neighbors, ForksList,
        RequestsList, Pid, Ref);
292             {ok, [{1, "DIRTY"}]} -> print("~p Fine, I give it up~n",[NewNode]),
293                                     ForkList = dict:erase(NewNode, ForksList),
294                                     NewForkList = dict:append(NewNode, [0, "SPAGHETTI SAUCE"],
        ForkList),
295                                     {philosopher, NewNode} ! {node(), fork},
296                                     philosophize(hungry, Neighbors, NewForkList, RequestList,
        Pid, Ref)
297     end
298     end,
299     philosophize(hungry, Neighbors, ForksList, RequestList, Pid, Ref).
300
301
302 % Before leaving, the philosopher sends messages to all its neighbors

```

```

303 % telling them he's leaving and then leaves. Gone is not really a state ,
304 % just alerts controller that he successfully left
305 philosophize(leaving , [], -, Pid , Ref) -> Pid ! {Ref , gone} ,
306     print("I'm gone forever!\n") ,
307     halt() ;
308 philosophize(leaving , Neighbors , ForksList , Pid , Ref) ->
309     {philosopher , hd(Neighbors)} ! {node() , leaving} ,
310     philosophize(leaving , tl(Neighbors) , ForksList , Pid , Ref) .
311
312
313 % Helper functions for timestamp handling.
314 get_two_digit_list(Number) ->
315     if Number < 10 ->
316         ["0"] ++ integer_to_list(Number) ;
317     true ->
318         integer_to_list(Number)
319     end .
320
321 get_three_digit_list(Number) ->
322     if Number < 10 ->
323         ["00"] ++ integer_to_list(Number) ;
324     Number < 100 ->
325         ["0"] ++ integer_to_list(Number) ;
326     true ->
327         integer_to_list(Number)
328     end .
329
330 get_formatted_time() ->
331     {MegaSecs , Secs , MicroSecs} = now() ,
332     {{Year , Month , Date} , {Hour , Minute , Second}} =
333     calendar:now_to_local_time({MegaSecs , Secs , MicroSecs}) ,
334     integer_to_list(Year) ++ ["-"] ++
335     get_two_digit_list(Month) ++ ["-"] ++
336     get_two_digit_list(Date) ++ [" "] ++
337     get_two_digit_list(Hour) ++ [":"] ++
338     get_two_digit_list(Minute) ++ [":"] ++
339     get_two_digit_list(Second) ++ ["."] ++
340     get_three_digit_list(MicroSecs div 1000) .
341
342 % print/1
343 % includes system time.
344 print(To.Print) ->
345     io:format(get_formatted_time() ++ ": " ++ To.Print) .
346
347 % print/2
348 print(To.Print , Options) ->
349     io:format(get_formatted_time() ++ ": " ++ To.Print , Options) .

```