Spring 2014 CSCI 182E—Distributed Systems
Homework 4
The Dynamic Dining Philosophers

Value: 20% of Homework Assignments Grade

Daniel M. Zimmerman
Harvey Mudd College

Assigned 26 February 2014
Due 13 March 2014, 23:59:59

This assignment is to be completed in teams of 2 or 3

The main purpose of this assignment is to design, and build an implementation of, a distributed mutual exclusion algorithm that solves a problem similar to the generalized dining philosophers problem (GDP) discussed in class. The main difference between GDP and the problem described here is that the topology of the system can change (by adding or removing philosophers) at runtime.

Note that this assignment is *not* about fault tolerance or crash recovery (except for the extra credit, described later); you will 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.

The system you implement will be structured as a *dynamic dining philosophers layer*, where the philosophers make some of their state changes in response to external stimuli rather than spontaneously or randomly "becoming hungry", "finishing eating", or "leaving the system". This has two advantages over a "standalone" dining philosophers implementation:

- It can be tested easily by causing different sequences of actions to occur and watching the results (you can always write test programs to test spontaneous/random behavior as well).
- It can be used as a building block for larger applications (such as a dynamic drinking philosophers implementation, generalized resource allocation, etc.).

You are strongly recommended to use Erlang and its distributed messaging for this assignment. You may choose to implement your programs in Java using Erlang messaging instead, but you will almost certainly find it easier to complete the assignment successfully in Erlang.

4.1 The Algorithm *40%*

In this part of the assignment you will design and argue for the correctness of an algorithm to solve the dynamic dining philosophers problem, which is as follows:

- A *network* consists of a number of *philosophers*, each of which has a (possibly empty) set of *neighbors*.
- A philosopher can be in one of the following 6 states: *joining*, *thinking*, *hungry*, *eating*, *leaving*, *gone*. The *thinking*, *hungry*, and *eating* states are as described in the original dining philosophers problem; in the *joining* state, a philosopher is in the process of joining the network; in the *leaving* state, a philosopher is in the process of leaving the network. The *gone* state is not a "real" state, and no process is ever running in state *gone*; it exists solely for the purpose of describing the algorithm, and represents a philosopher that has completely left the network and whose program has terminated.
- The following safety properties must hold for all philosophers p:
  - **initially** $p.joining$
  - $p.joining$ **next** $(p.joining \lor p.thinking)$
  - $p.thinking$ **next** $(p.thinking \lor p.hungry \lor p.leaving)$
  - $p.hungry$ **next** $(p.hungry \lor p.eating \lor p.leaving)$
  - $p.eating$ **next** $(p.eating \lor p.thinking \lor p.leaving)$
  - $p.leaving$ **next** $(p.leaving \lor p.gone)$
  - $p.gone$ **next** $p.gone$
  - $p.eating \implies \langle \forall\, q \mid q \in p.neighbors \rhd \neg q.eating \rangle$
    (when a philosopher p is *eating*, none of its neighbors is *eating*)
  - $(p.thinking \lor p.hungry \lor p.eating) \implies$
    $\langle \forall\, q \mid q \in p.neighbors \rhd p \in q.neighbors \rangle$
    (when a philosopher p is *thinking*, *hungry* or *eating*, each of p's neighbors knows that p is one of its neighbors)
  - $p.gone \implies \langle \forall\, q \rhd p \notin q.neighbors \rangle$
    (when a philosopher p is *gone*, it is not in any other philosopher's set of neighbors)
- The following progress properties must hold for all philosophers p:
  - $p.joining \leadsto p.thinking$
  - $p.hungry \leadsto p.eating$
  - $p.leaving \leadsto p.gone$

Note that there is no progress property to ensure that *eating* takes bounded time; this is because the transition from *eating* to *thinking* is controlled externally, as are the transitions from *thinking* to *hungry* and from many states to *leaving*. You may assume, for the purposes of developing and arguing for the correctness of your algorithm, that the external controller obeys the following rules related to both safety and progress:

- If a philosopher p informs the external controller that it is *eating*, the external controller will direct p to become either *thinking* or *leaving* within a bounded time.
- The external controller will not send duplicate or invalid control signals (e.g., it will not direct a philosopher p to become *hungry* when p has already been directed to become *hungry* and has not yet eaten; it will not direct a philosopher p to leave the system more than once, or to leave the system when p is *joining*; it will not direct a philosopher p to become *thinking* when p is not *eating*; etc.).

Devise an algorithm to solve this problem and argue for its correctness. You may wish to start from one of the solutions to the generalized dining philosophers problem discussed in class; the hygienic solution is probably a better starting point than the solution based on logical clocks. As a reminder, a distributed algorithm design like the ones we have done in class has the following attributes:

- A set of states (provided for you above).
- A description of the information stored by each process.
- A description of the message types in the system, what they mean, what information they carry, and what processes can exchange them. Some messages (such as control signals to trigger hunger or departure from the system) will only be sent from an external controller, somewhere outside the network of philosophers, to a philosopher; similarly, responses to such messages (such as notifications that a philosopher has become *eating* or *gone*) will only be sent from a philosopher to an external controller.
- A description, for each state, of the actions that a process takes when it receives a message of each possible type. In the worst case, in a system with k different message types, there would be $6k$ actions to describe; however, it is very likely that only a subset of the possible message types can be legitimately received in a given state (for example, forks cannot be received while *thinking* or *eating* in the hygienic solution as presented in class), so the actual number of actions to describe should be significantly smaller.

You do *not* need to describe a program for the external controller; just assume that it obeys the rules described above, triggering state transitions from *thinking* to *hungry*, from *eating* to *thinking*, and from various states to *leaving*, at appropriate times.

A good correctness argument will justify why your algorithm ensures the safety and progress properties described, similar to the way we justified the correctness of the other dining philosophers algorithms in class. It need not be (and probably *should not be*, because it would take far more effort than would be useful here) a formal proof. In fact, your argument will likely be quite similar to those arguments, with the added complexity of philosophers needing to enter and leave the network at runtime.

Hints

It may be helpful, for the purposes of uniformly describing the behavior of the philosophers, to write your algorithm such that each philosopher is started with a set of existing philosophers to which it must "attach" itself as a new neighbor. For example, to create a 3-cycle of philosophers, the first philosopher a would be started with no neighbors; the second philosopher b would be started with a as a neighbor; and the third philosopher c would be started with a and b as neighbors. In both the algorithm description and (later) the implementation, you may assume that sufficient time is allowed for a philosophers to bootstrap itself before other philsophers send it messages, so that messages are never lost.

Since there are arbitrary message delays, there will be some period of time in constructing a 3-cycle $\{a, b, c\}$ when a believes it is the only philosopher in the network, some period of time when a and b believe that they are the only two philosophers in the network, etc. Philosophers always act based on their current knowledge of the network. For example, when a is initialized it can eat immediately if so directed because it believes itself to be alone in the network; however, since b knows at initialization that a is a neighbor, it would not be reasonable for b to eat until it knows that a is not eating.

Some critical issues to address in the algorithm description are: how a new philosopher informs its neighbors that it has joined the network and how it knows that its neighbors are aware of its joining; how a philosopher informs its neighbors that it is leaving the network and how it knows that its neighbors are aware it has gone; and how to maintain an acyclic priority graph throughout the evolution of the network. You may find it necessary to delay adding new philosophers to the neighbor sets of existing philosophers until certain safe states are reached.

It is important to be sure that, if an external controller triggers a state transition (e.g., from *thinking* to *hungry*), the external controller is informed when important events (such as becoming *eating*) occur. This will prevent the external controller from sending control signals at inappropriate times, and is also important for building algorithms that use the dynamic dining philosophers layer.

4.2 The Implementation *60%*

In this part of the assignment you will implement the algorithm you developed in the previous part. In particular, you will write and test a *philosopher* program that implements a single dining philosopher capable of joining and leaving a network and carrying out the dining philosophers algorithm.

The requirements for the program are as follows:

- The module containing it must be called `philosopher` and must export a single function, `main/1`.
- It must take the following command line parameters (in this order):
  - A name to register itself with. This should be a lowercase ASCII string with no periods or `@` signs in it.
  - 0 or more additional parameters, each of which is the Erlang node name of a neighbor of the philosopher.

  For example, to start a 3-cycle of philosophers you might invoke the following commands (the different prompts indicate different machines—in this case `ash`, `elm`, and `oak`—on which you are running):

  ```
  {user@ash} erl -noshell -run philosopher main p1
  {user@elm} erl -noshell -run philosopher main p2 p1@ash
  {user@oak} erl -noshell -run philosopher main p3 p1@ash p2@elm
  ```

- It must use the name provided on the command line as the node short name to start the network kernel, as on Assignment 3 (don't forget to start EPMD as well!). Once the network kernel has been started, it should register itself with the name `philosopher`. This means that there will only be one philosopher process allowed per Erlang node (because they all have the same name).[1] You may make the assumption that nodes are uniquely named;[2] if you create a node called `foo@bar` with a philosopher, and that philosopher joins and later leaves a network, no subsequent philosopher on a node called `foo@bar` may join that same network.
- It may use any message formats you like for messages among the philosophers (they will presumably track those described in your algorithm quite closely), but must communicate with external controller programs using the following message formats. Erlang atoms are denoted by `typewriter` text, and strings appear

[1] The alternative would be to take twice as many command line parameters: a process name and a node name for each neighbor. For this assignment, we're going to avoid that.

[2] This assumption isn't really necessary to get a working solution, but it does eliminate the possibility of race conditions where multiple messages that are intended for the same philosopher are delivered to different ones.

in "quotes". *pid* is the Erlang process ID of the controller, used to allow the philosopher to properly address a reply. *ref* is a "unique" Erlang reference generated by the controller, used to allow the controller to match replies to control messages.

- {*pid*, *ref*, `become_hungry`} – sent by a controller to a philosopher. The philosopher should only receive this message while *thinking*; when it does, it transitions to *hungry*.
- {*ref*, `eating`} – sent by a philosopher, when it becomes *eating*, to the controller that triggered its transition to *hungry* (the *ref* should match the *ref* sent by the controller).
- {*pid*, *ref*, `stop_eating`} – sent by a controller to a philosopher. The philosopher should only receive this message while *eating*; when it does, it transitions to *thinking*.
- {*pid*, *ref*, `leave`} – sent by a controller to a philosopher. The philosopher can receive this message in any state other than *joining*; when it does, it should leave the network as soon as possible (given the protocol it is required to follow with respect to its neighbors).
- {*ref*, `gone`} – sent by a philosopher, when it becomes *gone*, to the controller that sent it the `leave` message that triggered its departure (the *ref* should match the *ref* sent by the controller).

Note that it is possible for more than one controller to send such messages to the same philosopher, and for controllers to send multiple messages at once; for example, a controller could send a `become_hungry` followed immediately by a `stop_eating` or even by another `become_hungry`. Dealing with this is straightforward in Erlang: in any given state, you can receive only the types of messages you are interested in using pattern matching. Since philosophers do not need to be fault-tolerant, you should not use timeouts in their receives or add "catch-all" patterns to catch unexpected messages.

Note also that these message formats are simple enough that you can easily send them by hand from an Erlang shell in order to manually test the behavior of philosophers in your system.

- It must generate sufficient console output to show exactly what is happening with respect to message sends/receives and state changes. Each message received or sent, and each state change, should be described in human-readable fashion on standard output. Each line of output must be timestamped to at least millisecond precision as required for previous assignments. In this way, since you are running on multiple machines on the same subnet (or multiple Erlang VMs on the same machine) and they are very closely synchronized in real time, you can use the outputs of multiple philosopher processes to satisfy yourself that the philosophers are behaving correctly.

In addition to writing the philosopher program, you must perform sufficient testing on it to convince yourself (and us!) that it works properly. There are many ways to do this:

- Create various networks of philosophers and exercise them manually by sending messages to the philosophers from an Erlang shell and by adding extra philosophers at runtime.
- Write test controller programs that automate the process of sending messages to philosophers in particular patterns that allow you to evaluate their functionality.
- Write shell scripts, invokable via `ssh` (having SSH keys and a working SSH agent will make this process much easier and prevent you from having to type your password many times), that allow you to automatically start multiple philosophers on multiple lab Macs from a single terminal window and interact witht them by sending controller messages.

The specific mechanism you choose for testing your philosopher program is up to you. Whatever you choose, you must provide us with sufficient information in your submission (descriptions of tests, logs, etc.) to convince us that your implementation works. Since the assignment has specific requirements for the philosophers' command line parameters, as well as specific communication requirements for controllers, we will also be able to test your submissions in a semi-automated fashion.

Hint: One way to write a well-structured state-transition program in Erlang is to write a different Erlang function for each state—each of which has the basic form "wait for message; update variables; maybe send messages; maybe change state"—and have the functions call each other (or themselves) to change (or stay in the same) state. There are also frameworks built into the Open Telecom Platform (OTP) for writing state machines, which you are certainly free to use, though they are perhaps overkill for this assignment.

4.3 Extra Credit *10%*

For 10% extra credit on the assignment, write your philosopher programs such that if a philosopher is terminated unexpectedly (such as with Control-C on the command line) its neighbors detect the unexpected termination and take appropriate actions. Effectively, this becomes an alternate method for a philosopher to leave the system (leaving its former neighbors to pick up the pieces, such as unneeded forks, old messages from that philosopher, etc., because no clean exit protocol is followed). In order to receive the extra credit, the unexpected termination of a process must be able to occur at any time (including before the process has transitioned from *joining* to *thinking*, and while the process is *eating*) without affecting the safety or progress of the rest of the network. You may, as in the regular implementation, make the assumption that nodes are uniquely named (i.e., once a philosopher on a node `foo@bar` is terminated, no subsequent philosopher on a node `foo@bar` joins the network).

Erlang has built-in facilities that make this substantially easier than it might be in other programming languages. In particular, you will likely find `erlang:monitor/2` useful should you attempt to complete this part of the assignment.

Submission

To submit your assignment, upload 3 files to the Assignment 4 page on Moodle:

- one of `algorithm.pdf` or `algorithm.txt`, a PDF or text file (respectively) containing a writeup of your algorithm and an accompanying correctness argument. You can use any document preparation software you choose to generate a PDF or you can submit a plain text document; no other document formats (LaTeX source, Word, Pages, RTF, etc.) will be accepted.
- `philosopher.zip`, a Zip file containing your Erlang source files (you may only have one, but you may also choose to put support routines in another module or to include Erlang-based testing code), details about your testing strategy (in a text file), any test programs/shell scripts/etc. you have written, and any other files (such as logging output) that you wish to provide as evidence that your program works.
- `readme.txt`, a text file containing a list of your team members (this is *extremely important*), any information you think the graders need to know about your code, and a brief description of your experiences implementing the homework assignment.

Only one team member needs to upload the files to Moodle; be *absolutely sure* that `readme.txt` includes the names of all team members!

Evaluation

Your algorithm will be evaluated on its correctness (75%) and on how convincing your correctness argument is (25%). Your program will be evaluated on its functional correctness (75%), determined from your testing strategy as well as from our testing, and on its code style/readability (25%).

If you put no effort into your `readme.txt` file, you may be penalized up to 10% of the value of the assignment. As long as you write some genuine reflections about your experiences working on the assignment, you will not incur a penalty.

If the names of all team members (including the one who submitted the files) are not listed in `readme.txt`, all team members will receive a 0 on the assignment.