MP 6 - Actors, Threads, Concurrency and Parallelism

CS421 Agha - Spring 2015

rev. 1.2

Introduction

Assigned: April 26, 2015
Due: 11:59pm, May 6, 2015 **(the deadline is firm)**
Outline: This assignment covers material from lectures on Actors and Racket threads.
Grading: This assignment will account for 8% of your grade.

Submission and academic honesty instructions can be found here:

`https://wiki.cites.illinois.edu/wiki/display/cs421sp15/Assignments?src=contextnavchildmode`

Pay special attention to the policies regarding the use of other people's code.

Problems

In this MP, you will be using the racket threads library:

`http://docs.racket-lang.org/reference/threads.html`

Given a dataset of friend relationships, you will be answering queries of the sort "what are the common friends of Minas and Steven, within 2 friendship levels"? The file format is given by the following grammar:

`Friendships` ::= {`Name` : {`Name`}* ; }*

where `Name` is represented as a string of characters. A sample file in this format can be seen below:

```
Minas : Steven Sihan Alex ;
Steven : Minas Sihan Mario ;
Sihan : Minas Steven Peter ;
Peter : Sihan John ;
Alex : Minas ;
Mario : Peter ;
```

The above means that Minas' friends are Steven, Sihan and Alex. Steven's friends are Minas, Sihan and Mario and so on so forth. Please mind the friendship relation is not symmetric (for example, Mario is on Steven's list, but not vice versa).

The messages communicated between our system and yours will adhere to the following datatype:

```
(define-datatype message-type message-type?
  (query-msg
    (names (lambda (x)
             (and (pair? x)
                  (string? (car x))
                  (string? (cdr x)))))
    (depth integer?)
    (id integer?)
    (reply-to (lambda (x)
                (or (thread? x)
                    (place-channel? x)))))
  (filename-msg
    (filename string?))
```

This datatype is included in a message-type.scm file found under your mp6 directory, which you should use as is (do not change it in any way).

The `filename-msg` type of message contains the path of the file where you can read the dataset from. This message will be the first one we send, and it will only be sent once. This is the dataset you will be using to answer queries, represented by `query-msg`. Upon receipt of such a message, your system should compute the set of common friends of the people found in the `names` pair, going `level` levels deep. The response (result of the query) should be sent to the `reply-to` thread, in a `response-msg` message. It should contain the `id` of the original query, as well as the actual result itself, which will be a list of strings (names).

In order for us to be able to communicate with your system, you have to define a thread called "the-recipient", so that a call such as the following

```
(thread-send the-recipient (filename-msg "dataset"))
```

does not fail. Do not forget to `(provide the-recipient)` at the end of `mp6-1.scm`.

Assuming the above example file's name has already been sent to you through a `filename-msg` message, here are a few examples:

```
(define id 1)
(define level 1)
(define names (cons "Minas" "Sihan"))
(thread-send the-recipient (query-msg names level id (current-thread)))
(cases message-type (thread-receive)
  (response-msg (id result)
      (cons id result))
  (else 'undefined))
> (1 . ("Steven"))

(define id 1)
(define level 2)
(define names (cons "Minas" "Sihan"))
(thread-send the-recipient (query-msg names level id (current-thread)))
(cases message-type (thread-receive)
  (response-msg (id result)
      (cons id result))
  (else 'undefined))

> (1 . ("Steven" "Mario" "Peter" "Alex"))

(define id 1)
(define level 3)
(define names (cons "Sihan" "Peter"))
(thread-send the-recipient (query-msg names level id (current-thread)))
(cases message-type (thread-receive)
  (response-msg (id result)
    (cons id result))
  (else 'undefined))

> (1 . ("Minas" "Steven" "Mario" "Alex" "John"))
```

Mind that we will not care about the ordering of names in the returned list.

To clarify the way the `depth` parameter works: in the second example above, we need to find the friends Minas and Sihan have in common up to level 2. That means we need to find Minas' friends (Steven, Sihan, Alex) and the friends of each of those (Minas, Sihan, Mario, Minas, Steven, Peter, Minas). Merging and deleting duplicates we get (Minas, Sihan, Mario, Steven, Peter, Alex). Doing the same for Sihan we get (Minas, Steven, Peter, Sihan, Alex, Mario, John).

The answer to the query is the intersection of these two sets, sans "Minas" and "Sihan".

Notes: The supplied dataset will be in plain text format. You can parse it in any way you deem appropriate. Make sure you include the proper id in your response, so that the auto-grader can tell which response corresponds to which query. We strongly recommend you minimize your use of shared state in your code, and stick to the actor model of computation as much as possible. That should result in cleaner code, that is easier to optimize.

Extra Credit

Racket threads provide concurrency but not parallelism. To achieve performance gains, you may instead use *places*[1] If you choose to do so, you will be graded based on both correctness of the result and the performance of your code. The supplied file will be much larger than the above example, such that the use of parallelism will result in performance gains. The auto-grader will perform a stress test on your code, by sending a large number of task messages, with various values of the depth parameter. Depending on the performance gains vs a simple sequential solution, you will receive up to a 25% bonus for this MP.

Note that the interface to your code should still be a thread called `the-recipient`. That thread should still be capable of processing `query-msg` type messages. However, the embedded `reply-to` field will not be a thread; instead, it will be a `place-channel`. These channels have restrictions with regard to the kinds of values you can send/receive over them. Hence, your response should be a tuple constructed as such:

```
(define response (cons 'response-msg (cons id result)))
```

where `id` is still an integer, and corresponds to the id of the query, and `result` is a list of names, the actual solution to the query. You can send this response back using the `reply-to` field of the `query-msg` message as such:

```
(place-channel-put reply-to response)
```

If you go for the extra credit, please name your file parallel-mp6-1.scm. Otherwise, name it mp6-1.scm as usual.

[1] `http://docs.racket-lang.org/reference/places.html`