Snapy: the search for dead marbles

Johan Montelius

October 2, 2016

Introduction

In this exercise you will learn how to implement a snap-shot algorithm. We will use a very simple scenario with a set of workers that create and share *marbles* with each other. The problem is to find out which marbles are alive so that references to dead marbles can be removed. It's in a sense a simplified garbage collection problem. The problem is simplified by the fact that the data structures, the marbles, are atomic and that we do not create duplicates of marbles. We could have solved the problem using a simpler solution but why not play around with a snap-shot algorithm.

1 The Worker

A worker will keep a state of marbles that it holds, called the *Alive* set, and marbles that it has created and passed to other workers, the *Exported* set. It will also keep a set of *Peers* that are the other workers in the network.

To wisualize the worker we will use a simple gui where the marbels of a worker are shown. The marbles in the alive set are shown as red dots and marbels in the exported set as greeen dots.

A marble is owned by the creating worker but could be held by any worker. We can look at a marble and identify the worker that created the marble. We will use this in order to check if the owner actually keeps track of its marbles. As long as a marble is held by a worker the owner should keep a record of the marble. One can see the marble as a pointer to some shared resource and as long as a workers has access to the pointer the resource must be available. If there is no pointer in the system to the resource the resource is of course garbage and can be thrown away.

1.1 the life of a worker

The worker is a process that is waiting for incoming messages but after being idle for a while decides to do some work. The work consist of checking that a randomly selected marble is still available (`ping(Alive)`), request a new marble from one of its peers (`req(Peers)`) and possibly throw one or two marbles away (`trow_away(Alive)`). We will implement these functions later.

```
worker(Peers, Gui, Alive, Exported) ->
```

```
Idle = random:uniform(?delay),
receive

    {request, From} ->
        {Alv, Exp} = request(From, Gui, Alive, Exported),
        worker(Peers, Gui, Alv, Exp);

    {marble, Marble} ->
        Alv = incoming(Gui, Marble, Alive),
        worker(Peers, Gui, Alv, Exported);

    {ping, Marble} ->
        check(Gui, Marble, Exported),
        worker(Peers, Gui, Alive, Exported);

    quit ->
        quit

after Idle ->
        %% Check that a marble still exists
        ping(Alive)

        %% Send a request for a marble
        req(Peers)

        %% Throw some marbles away.
        Rest = throw_away(Gui, Alive),

        worker(Peers, Gui, Rest, Exported)
end.
```

The messages a worker can receive are either from other workers or from a managing process that want to terminate the execution. The messages from other workers are the following:

- {request, From}: a request is received from a peer worker. Send one of the marbles, randomly selected, in the `Alive` set or create a new marble. If we send an existing marble then we delete it from the `Alive` set. If a new marble is created this marble must be added to the set of exported marbles.
- {marble, Marble}: a marble that is received from a peer. Add the marble to the `Alive` set.
- {ping, Marble}: another worker wants to know if a marble that we

have created still exists. We should have a record of this in our set of `Exported` marbles. If the marble is not found we will log an error.

The implementation of the work procedures is uncomplicated. We need to implement a set of functions to construct and access marbles and decide on how to represent the alive and exported sets. The sets can simply be represent as list and a marble can be represented as a tuple {`marble, Ref, Pid, Pos`} where the `Ref` is a unique reference, `Pid` the process identifier of the creator of the marble and `Pos` a position used by the gui.

Sending a request is trivial using a function `pick_one` to select a randomly selected element from the list of peers.

```
req(Peers) ->
    {value, Peer} =  pick_one(Peers),
    Peer ! {request, self()}.
```

The incoming request can be served either with one of the marbles we have in our alive set or a marble that we create. If we select one from the alive set we remove it from the set and also send a message to the gui. If we create a new marble it is added to the exorted set; this is the only way marbles can be added to this set.

```
request(From, Gui, Alive, Exported) ->
    N = length(Alive),
    if
        N > 4 ->
            {value, Marble} =  pick_one(Alive),
            From ! {marble, Marble},
            delete(ref(Marble), Gui),
            {lists:delete(Marble, Alive), Exported};
        true ->
            Marble = marble(),
            From ! {marble, Marble},
            exported(ref(Marble), pos(Marble), Gui),
            {Alive, [Marble|Exported]}
    end.
```

When the message with the marble is retured to the requesting worker the marble is simply added to the alive set.

```
incoming(Gui, Marble, Alive) ->
    alive(ref(Marble), pos(Marble), Gui),
    [Marble|Alive].
```

When we throw things away need to check that we actually have something to throw away. Below is an implementation that only throws a marble away if there are more than four marbles in the `Alive` set.

```
throw_away(Gui, Alive) ->
    N = length(Alive),
    if
        N > 4 ->
            {value, Marble} =  pick_one(Alive),
            delete(ref(Marble), Gui),
            lists:delete(Marble, Alive);
        true ->
            Alive
    end.
```

The function `pick_one/1` can be implemented using `lists:nth/1` and a call to `random:uniform`. We only have to know how many marbles there are and make sure that we do not try to select something from an empty list.

The `alive/3` procedure will add a red marble to the gui and `exported/3` adds a green marble. The `delete/2` procedure will send a message to the gui to remove a marble.

2 The first experiment

Run the workers and see that they are actually doing something. Note how the number of exported marbles increase. This is of course obvious since we now and then create new marbles that we pass on to peers but once they have been added to the set of exported marbles there is no way to remove them from this set.

Before starting on the snap-shot solution you should think of how this could easily be solved. What would happen to your solution if we where allowed to make a copy of an existing marble and pass the copy to one of our peers? What would happen if we allowed marbles to hold references to other marbles?

3 A solution - not

Instead of trying to figure out locally if a marble is garbage we leave this to an external process, a controller. The controller will send a message to each worker and have them report back which marbles it has in the alive set and which marbles it has exported. We only have to update the worker with one extra message handler.

```
        {snap, Cntrl} ->
            snap(Cntrl, Peers, Alive, Exported),
            worker(Peers, Gui, Alive, Exported);
```

In compiling the answer we only need to send the marble references since we do not need to know who actually created them. We do however send our process identifier so that the controller know which worker that might be interested in what information.

```
snap(Cntrl, _Peers, Alive, Exported) ->
    Cntrl ! {report,
             self(),
             lists:map(fun(M) -> ref(M) end, Alive),
             lists:map(fun(M) -> ref(M) end, Exported)}.
```

Now the controller has the pleasure of sending a `snap` request to all workers and collect one reply from each one. It should then try to deduce if there are any exported marbles that are no longer alive. Note that the workers will only send us references of marbles since this is the only thing that we need to have.

```
gc(Workers) ->
    lists:map(fun(W) -> W ! {snap, self()} end, Workers),
    collect(Workers, [], []).
```

It can first collect the replies and add all the alive marbles into one list and but keep the exported marbles associated to each worker in another list. When all responses have been received it's time to filter the exported sets using the set of alive marbles.

```
collect([], Alive, Extported) ->
    lists:map(fun({W, Opt}) -> W ! {dead, filter(Opt, Alive)} end, Extported);
collect(Waiting, Alive, Extported) ->
    receive
        {report, Worker, Alv, Exp} ->
            collect(lists:delete(Worker, Waiting),
                    lists:append(Alv, Alive),
                    [{Worker, Exp}| Exported])
    end.
```

An exported marble need only remain in the list if the marble is in the alive set. If we can filter out the references that are not alive we can use this information and send it back to the worker in a message `{dead, Dead}`. Filtering the list is of course easy using a the higher order function `lists:filter/2`.

```
filter(Exported, Alive) ->
    lists:filter(fun(Exp) -> not lists:member(Exp, Alive) end, Exported).
```

We now need to add another message handler to the worker so that it can receive the message and filter its own set of exported marbles.

```
        {dead, Dead} ->
            Filtered = filter(Gui, Exported, Dead),
            worker(Peers, Gui, Alive, Filtered);
```

The filtering is easily expresses using some folding. The higher order function `lists:foldl/2` will apply a function to a element taken from a list, a dead marble, and a starting value, the list of exported marbles. The resulting value, where we have removed the dead marble from the list of exported marbles, is then used as the new starting value. When we have applied the function to all elements of the dead list we have only marbles that are alive left.

```
filter(Gui, Exported, Dead) ->
    lists:foldl(fun(D, Exp) ->
                  case lists:keysearch(D, 2, Exp) of
                      {value, Marble} ->
                          delete(ref(Marble), Gui),
                          lists:keydelete(D, 2, Exp);
                      false ->
                          Exp
                  end
          end,
          Exported, Dead).
```

3.1 some experiments

Does this actually work? Do some tracing of the number of exported marbles after each garbage collection. Do we find any dead entries and do we manager to decrease the number of exported marbles. Since an exported marble should be alive somewhere the total number of alive marbles should be close to the total number of exported marbles. One exception is of course when a marble has been created and inserted into a set of exported marbles but it has not yet been inserted at the receiving side. Also if a marble is thrown away it will still be present at the exported side before we do a garbage collection.

When you have done some experiment you can add the following code. In the working phase we also want to ping a randomly selected marble. Assuming we have a function to randomly select an element from a list the `ping/1` procedure can be implemented as follows.

```
ping(Alive) ->
    case pick_one(Alive) of
        {value, Marble} ->
            owner(Marble) ! {ping, Marble};
        false ->
            ok
    end.
```

If we send a {`ping, Marble`} message we also need to add a message handler. We simply check that the Marble actually does exist in our set of alive marbles.q

```
        {ping, Marble} ->
            check(Marble, Exported),
            worker(Peers, Alive, Exported);
```

If we do not find the marble in the list we write an error message to the terminal.

```
check(Marble, Exported) ->
    case lists:member(Marble, Exported) of
        true ->
            ok;
        false ->
            io:format("error: marble not found~n", [])
    end.
```

Do you have any error messages? Can you insert delays by using calls to `timer:sleep/1` to increase the risk of errors? What is the root of the problem?

4 Snap shot

A proper snap shot can not be taken just by collecting the states of each node, we also need to take carer of the messages in transit. In order to do this we have to change the implementation. We start by slightly changing how the controller works. Instead of sending a snap shot message to every worker it will only send a message to one of the workers. It will still receive reports from all workers and the calculation of dead marbles is the same.

```
gc(Workers) ->
    [W|_] = Workers,
    W ! {snap, self()},
    collect(Workers, [], []).
```

The workers must change more. Only one worker receives the snap shot message and should initiate the snap sot by sending markers to the other workers. When a snap shot is initialized we should start recording incoming messages from workers that have not yet sent us a marker. Our first change will be to add a recorder as an additional state of the worker, the initial recorder is set to `na` (not available) but will be set to a tuple

$$\{\texttt{recorder}, \texttt{Cntrl}, \texttt{Peers}, \texttt{Alive}, \texttt{Exported}\}$$

containing everything we need once we receive a snap message or our first marker.

- Cntrl: the process identifier of the controller in order to know where to send the snap shot
- Peers: the peers from which we have not received a marker.
- Alive: references of marbles that we need
- Exported: references of marbles that we have exported

When we receive the snap message or our first marker (we know it it's the first since the recorder will be set to `na`) we can easily create the initial recorder. The question now is what incoming messages that changes this state and how we know when we have seen all marker.

It turns out that it is only one incoming message that is of interest, the message that contains a marble. This marble should be added to the set of alive marbles.

Since we only should record information from workers that have not sent us a marker message we need to keep track of which workers that have sent us markers and which messages that we need to record we need to change the format of the `marble` message.

```
{marble, From, Marble} ->
    Alv = incoming(Gui, Marble, Alive),
    Rec = rec(From, Marble, Recorder),
    worker(Peers, Gui, Alv, Exported, Rec);
```

Once we have received all markers we will send the same message as before to the controller. The controller, working as before with out any changes, should now have more correct information to work with.