Garby: a distributed garbage collector

Johan Montelius

October 2, 2016

Introduction

In this exercise you will learn how to implement a snap-shot algorithm. As an example we will try to detect garbage in a distributed computing. This is tricky and as you will see, a quite expensive operation. Not taking the snap-shot in itself but how to interpret the snap-shot and how to make the most use of the gained information.

We will first set up a network of workers that do very little useful things but have an obsession of sending remote references to each other. They all have a simple memory and build tree structures that they more than willingly give a references to. They have a limited set of registers so once in a while they will loose their own reference to the structures thus generating garbage. This is taken care of by the Erlang garbage collector but we might have an external reference to the structure and must then keep it as long as this reference is still needed.

A coordinator is given the pleasure of initiating a snap-shot and collecting all the states. From this information it should determine which externally referenced data structures that are dead. It will inform the workers and the workers will then remove these from their set of externally references structures.

1 The Worker

The worker is a process that waits for incoming messages but after a while decides to send a request by its own. We will first build the worker and then see if we can have it to participate in a snap-shot.

The state of a worker is:

- **Name**: the symbolic name of the worker, used for debugging
- **Peers**: a list of process identifiers of other workers
- **Regs**: a register implemented as a list of objects
- **External**: a list objects with references that we have given to other workers
- **Recorder**: a structure used to record things during a snap-shot

Let's look at the objects a bit closer. The are simple tree structures on the form {`tree, Left, Right`} or `nil`. We will not create circular structures so we will not be in danger of going into an endless loop if we traverse a tree.

If we ask another worker for an object, it will pass us a unique reference. The references are created with the built-in procedure `make_ref/0` and will when printed look something like `#Ref<0.0.0.1930>`. To make this presentation easier to follow we will print them simply as `#19`. If we are given a reference we can include it in a tree as one of the branches. To make sure that we can recognise it properly we enclose it in a tuple like this: {`remote, #19`}. A branch of a tree could thus be `nil`, a remote reference {`remote, #19`} or another tree.

If we're asked for an object we create a new reference, associate this reference to one of our objects and pass the reference as a reply. The associating is kept in the list of external references. The list will have entries that looks like this {`#19, Obj`}, where `Obj` is one of our data structures.

Let's carry on. This is how we would initiate the worker.

```
-module(worker).

-export([start/2, init/2]).

start(Name, Seed) ->
    spawn(worker, init, [Name, Seed]).


init(Name, Seed) ->
    Regs = [nil, nil, nil],
    External  = [],
    Recorder = false,
    random:seed(Seed,Seed,Seed),
    receive
        {peers, Peers} ->
            work(Name, Peers, Regs, External);
        stop ->
            ok
    end.
```

As you see we start with a list of three registers, all with data structures set to `nil`. There are of course so far no external references and the recorder is set to false. We use the same trick as we have seen before and wait for a list of peer processes before going into the main loop of the process.

The main loop will look something like this, fill in the dotted lines.

```
work(Name, Peers, Regs, External, Recorder) ->
  Idle = random:uniform(1000),
  receive
      {request, From} ->
          %% they want a random reference
          .... = make_ref(),
          .... = random(Regs),
          From ! {remote, self(), ....},
          work(Name, Peers, Regs, [....|External]);
       {remote, From, Remote} ->
          %% an external ref, let's build a new data structure
          .... = allocate(Regs, Remote),
          Updated = store(Regs, ....),
          work(Name, Peers, Updated, External);
       status ->
          io:format("~w: length of external ~w~n", [Name, length(External)]),
          work(Name, Peers, Regs, External);
       stop ->
          ok;
       Error ->
          io:format("~w received strange message ~w~n", [Name, Error]),
          work(Name, Peers, Regs, External)
   after Idle ->
        %% This is just here to simulate an execution.
        random(Peers) ! {request, self()},
        Updated = dosomething(Regs),
        work(Name, Peers, Updated, External)
  end.
```

To simulate an execution we set a timeout and then requests a remote reference from a randomly selected peer. We also the `dosomething/1` procedure that will replace some of the values in the list of registers with `nil`. This is to generate garbage; if we remove a reference to a data structure that is not linked to by another living data structure nor by a external reference it becomes garbage.

To complete the worker you only have to implement four procedures:

- **random(List)**: randomly select one element in a list
- **allocate(Regs, Remote)**: build a tree structure, use objects from the registers and the remote reference, remember to include the remote reference in a tuple {`remote, Remote`}
- **store(Regs,Obj)**: place the object in one of the registers, return the new set of registers, this means that we will loose a reference to another object

- **dosomthing(Regs)**: randomly replace some of the registers with `nil` to simulate an execution, give the registers a 20 percent chance of surviving

To see if the worker is actually doing something we can do a small test run.

```
-module(garby).

-export([test/0, stop/0, kill/1]).

test() ->
    Rick = worker:start(rick, 34),
    Roy =  worker:start(roy, 23),
    Rachel = worker:start(rachel, 24),
    Rick ! {peers, [Roy, Rachel]},
    Roy ! {peers, [Rick, Rachel]},
    Rachel ! {peers, [Rick, Roy]},
    register(rick, Rick),
    register(roy, Roy),
    register(rachel, Rachel).

stop() ->
    rick ! stop,
    roy ! stop,
    rachel ! stop.

status() ->
    rick ! status,
    roy ! status,
    rachel ! status.

kill(Name) ->
    case whereis(Name) of
       undefined ->
         ok;
       Pid ->
         exit(Pid, kill)
    end.
```

The `kill/1` procedure could come in handy if things go wrong. Do a test run and see how the list of external references is growing.

2 Garbage collection

Garbage collection is partly done by the Erlang runtime systems. Objects that we do longer have access to will be detected as garbage and reclaimed. The problem is that we keep our list with external references. As long as an object is references from this list it will remain part of our execution state. How can we detect that a reference is no longer needed? This is the problem of distributed garbage collection and we will solve it.

2.1 the coordinator

In our snap-shot solution we will make use of a dedicated coordinator. This is often a role taken by one of the workers but it's easier to implement this as a separate process.

```
-module(coordinator).

-export([gc/1]).

gc(Workers) ->
    [Wrk|_] = Workers,
    Wrk ! {snap, self()},
    collect(Workers),
    io:format("collector: snap-shots collected", []),
    ok.

collect([]) ->
    ok;
collect(Workers) ->
    receive
        {shot, From, _, _} ->
            Rest = lists:delete(From, Workers),
            collect(Rest);
        Error  ->
            io:format("collector: strange message ~w~n",[Error]),
            error
    after 10000 ->
            timeout
    end.
```

The coordinator will do very little to start with, it will send a {`snap, self()`} message to one of the workers and then wait for snap-shots from all of the workers. Note that we must pass proper process identifiers to the collector, if we simply give it registered names then the sending will work but the collection will not work.

2.2 markers

Now we return to the worker. The coordinator will send one of the workers a `snap` message. The worker that receives the message should initiate the snap-shot algorithm and send markers to all of its peers. It must then keep track of that which peers it has received markers from.

A worker that receives a marker for the first time should return a marker from the sender and then pass markers to its other peers. In order to keep track of which markers we are still waiting for we extend the state of the worker to hold a `Recorder`. The recorder will be simple to start with but will later include more and more information that we need. The recorder will initial be set to `false` so that we know if we are in the recording state or not.

Extend the worker and include the following two messages:

```
{snap, Coordinator} ->
       io:format("~w: initiate snapshot~n", [Name]),
       New = snapshot(Recorder, Coordinator, Peers, Regs, External),
       work(Name, Peers, Regs, External, New);

{marker, From, Coordinator} ->
       io:format("~w: received marker~n", [Name]),
       Recorded = marker(Recorder, From, Coordinator, Peers, Regs, External),
       work(Name, Peers, Regs, External, Recorded);
```

This is the implementation of the snapshot and marker procedures. We will extend them later but this will do for now. The snapshot will check that we are not in a recording session all ready and then create a new recorder. We could actually extend the system to handle multiple concurrent recorders but this will do for now

```
snapshot(false, Coordinator, Peers, Regs, External) ->
    snap(Coordinator, Peers, Regs, External);
snapshot(Recorder, _, _, _, _) ->
    % we ignore this but we could have started a concurrent recorder
    Recorder.
```

The marker procedure will do two things: if we are not in a recording session a recorder is created, otherwise we close the open channel from which the marker was received. The open channels are represented by a list of pids so it's a simple step to see if the sender is still in the list. If we have no more open channels the initial coordinator is sent our computed state.

```
marker(false, From, Coordinator, Peers, Regs, External) ->
    From ! {marker, self(), Coordinator},
```

```
    snap(Coordinator, lists:delete(From, Peers), Regs, External);
marker({Coordinator, Needed, Optional, Open}, From, _, _, _, _) ->
    case lists:delete(From, Open) of
        [] ->
            %% this was the last marker, recording is done
            Coordinator ! {shot, self(), Needed, Optional},
            false;
        Rest ->
            {Coordinator, Needed, Optional, Rest}
    end.
```

The `snap/4` procedure simply creates a structure holding the coordinator, the needed references, optional references and a list of all peers representing the open channels. We use dummy values for needed and optional references for now, this will be changed later.

```
snap(Coordinator, Peers, Regs, External) ->
    lists:map(fun(Pid) -> Pid ! {marker, self(), Coordinator} end, Peers),
    Needed = needed,
    Optional = optional,
    {Coordinator, Needed, Optional, Peers}.
```

Do some test runs to see that the makers are passed around in the network. Add some print instructions to trace the execution. If it works your almost done, only the tricky part left.

2.3 the needed references

What we want to capture is which of the external references that we still need so that we can remove any garbage references from our list of external references. The information wee need to send to the coordinator is thus which external references we are still interested in; these will be called the needed references. The needed references can be collected from the register of the worker; anything that is reachable from the registers should of course survive a garbage collection.

```
reachable([], Reachable)->
    Reachable;
reachable([Ref|Refs], Reachable) ->
    case Ref of
        nil ->
            reachable(Refs, Reachable);
        {remote, Rem} ->
            reachable(Refs, [Rem|Reachable]);
        {tree, Ri, Rj} ->
```

```
            reachable([Ri,Rj|Refs], Reachable)
    end.
```

How should this procedure be called to construct a list of all remote references accessible from the registers? Where should it be called?

We are a bit lucky in our little experiment. If you examine how structures are created you will see that we can not create circular structures. If this was the case the reachable procedure would have to be more carefully in its traversal of data structures. We can also not that the objects we traverse can not contain a reference of our own. If we play around with how new data structures are created we could have a more complex state but the limited system we have now will serve our purposes.

2.4 the optional references

We are one step closer to the answer but we're not done. The problem is that if we only look at our registers we might forget structures that we have given remote references to but that are no longer reachable from our registers. We must include these in our report but they are of course optional, if the external reference is no longer needed nor are the references reachable from that external reference (if they are not reachable by other means).

```
optional([], Optional) ->
    Optional;
optional([{Ref, Obj}|Refs], Optional) ->
    optional(Refs,[{Ref, reachable([Obj], [])}|Optional]).
```

Where and how should the optional procedure be called?

2.5 collection

Now let's return to the coordinator and see what we can do with the information. From each worker we will receive two things, a list of references that definitely are alive and a list of optionally alive references. The list of optional references also gives us information about which external references the workers have. We would like to determine which one of these that are no longer needed. We do this in two procedures called `filter/2` and `dead/1`.

```
gc(Workers) ->
    [Wrk|_] = Workers,
    Wrk ! {snap, self()},
    {...., .....l} = collect(Workers, ..., ...),
    Rest = filter(....., .....),
    Dead = dead(....),
    ok.
```

The filter procedure should run through the list of needed references and trim the list of optional references. If a needed reference is found in the list of optional references, the associated references should also be treated as needed and further filter the list of optional references. Complete the following skeleton using procedures from the lists library.

```
filter([], Optional) ->
    ....
filter([Ref|Needed], Optional) ->
    case ...........  of
        {value, .... } ->
            Rest = ......
            filter(Needed, filter(..., Rest));
        false ->
            filter(...., Optional)
    end.
```

If we're successful we will no have a filtered the list of optional references. If we quickly run through this and collect the external references, ignoring the associated, we will have a list of dead references.

```
dead(Optional) ->
    lists:map(fun({R,_})-> R end, Optional).
```

Now let's make some use of this information.

2.6 dead entries

The list of dead entries compiled by the coordinator should now be sent to all of the workers. A worker that receives the list will match this against it list of external references and remove entries that are no longer needed.

```
  {dead, Dead} ->
    Keep = keep(External, Dead),
    work(Name, Peers, Regs, Keep, Recorder);
```

The keep procedure can be written quite easy using the filter procedure from the lists library.

```
keep(External, Dead) ->
    lists:filter(fun({R,_}) -> not lists:member(R, Dead) end, External).
```

Add some print statements to see how many dead entries are detected and if the lists of external references shrink at all. The total number of external references might still grow and grow but at least we have removed the dead entries... we haven't removed any live entries have we?

2.7 recording

We now return to the tricky part in the snap-shot procedure. We have successfully sent markers around the network and collected the state of each worker but there is one thing that we have not done. We have not recorded incoming messages on channels that are still open, channels where we have sent a marker but still not received one.

Examine the incoming messages of the worker, which messages (if any) do we need to record? What information is important and what does it actually mean?

There is not a lot of code that we have to add to our worker to solve this but it does require some thinking. Once you have identified the important information we can make use of a recording procedure.

```
recording(false, _, _) ->
    false;
recording(Recorder, From, Remote) ->
    {Coordinator, Needed, Optional, Open} = Recorder,
    case lists:member(From, Open) of
        true ->
            {Coordinator, [Remote|Needed], Optional, Open};
        false ->
            Recorder
    end.
```

If you do it right you need not change the coordinator at all.

Do some experiments and smile: you have implemented a distributed garbage collector using the snap-shot algorithm. Can you form a network of nodes and see that is actually working?