Goldy: a distributed game

Johan Montelius

October 2, 2016

Introduction

This seminar will serve two purpose; learning how to program a distributed application in Erlang and understanding why distributed applications are not as simple to control as it might first seam.

We will build a game called "Goldy" with a very simple structure. Each player will start the game and announce her position and also announce the positions of a set of gold nuggets. Each player can then start to move, trying to collect as many nuggets as possible. We will not pay any attention to if this is a fun game nor what the graphics look like but concentrate on how to keep each players state updated and hopefully synchronized.

1 The Implementation

To keep the design simple we implement each player as three processes: a multicast process, a game engine and a graphical user interface. The engine is the heart of the system and it will use the multicast process to communicate with other payers. The user interface will take care of user input and direct it to the engine and make changes to the graphical presentation of the game state. The interface process does not know very much about the game, it only acts on directives from the engine.

1.1 the multicaster

Let's start with the multicast process. Below is the skeleton code for the process. It is a very simple process that when created is given a master and a list of Peers. The peers are other multicast processes used by other players. We're doing one thing that looks strange, we're adding the process itself to the list of peers. We do this since we want to treat all messages the same.

The master can send two messages `stop` and `{send, Message}`. The process can also receive messages, `{msg, Message}`, from other multicast processes. These should be *delivered* to the master process. We include a catch all case to capture any strange messages, these can simply be logged to stdout and ignored.

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

```
start(Master, Peers) ->
    spawn(multicast, init, [Master, Peers]).

init(Master, Peers) ->
    loop(Master, [self()|Peers]).

loop(Master, Peers) ->
    receive
        {msg, Message} ->
            Master ! Message,
            loop(Master, Peers);
        stop ->
            %% our master wants us to terminate
             ok;
        {send, Message} ->
            %% our master wants us to send a message
            mcast(Message, Peers),
            loop(Master, Peers);
        Error ->
            io:format("strange message to multicast process ~w~n", [Error]),
            loop(Master, Peers)
        end.
```

To help you write the `loop/2` procedure it helps having a procedure to send a message to a list of peers. The `mcast/2` procedure below will do this for you.

```
mcast(_, []) ->
    true;
mcast(Msg, [Peer|Peers]) ->
    Peer ! {msg, Msg},
    mcast(Msg, Peers).
```

If you have written a hundred procedures that iterate over a list you will soon start to use the `lists` library to write more compact code. The following code does the same thing.

```
mcast(Msg, Peers) ->
    lists:map(fun(Pid) -> Pid ! {msg, Msg} end, Peers).
```

Now why did we include the own process identifier in the list of peers. The reason is that all engines should act only on those messages that are delivered by the multicast process. The engine is not allowed to make a move independently of other processes. It must multicast the message to everyone, including itself, and then handle all incoming messages in the order they arrive. We should not give our own messages higher priority than other messages; they all have to be queued before being handled.

1.2 the user interface

The user interface is for our purposes the least interesting module and we will not spend much time on it. You can take a look at the code given in the appendix an extend it any way you want. The important thing is that you understand the messages between the engine and the user interface.

The interface will report user interaction and there are only three type of messages that can be generated.

- `start` The user should start the game when she knows that all peers have been initialized.
- `stop` The user can stop the game at any point.
- `{move, Dir}` A directive to move the player one step in either direction: `up`, `down`, `left`, or `right`.

The engine will direct the user interface to create new object, remove objects and move objects. Each object is given an identifier when created and the engine uses the identifiers when modifying the objects. There are three different objects: `me`, `player`, and `gold`. The reason we want to have a separate object for our own player is that we should be able to easily identify it, all peers could be displayed the same.

- `{create, Obj, Id, Pos}` Create an object.
- `{move, Id, Pos}` Move the player (own or peer) to a position.
- `{remove, Id}` Remove an object (will be used when gold is taken).

Note that the user interface obviously knows where things are but we will never query the user interface about positions. Nor will we tell the user interface to move a player to the right or to the left. It's the engine that keep track of where objects are and we will always move object to an absolute position. This is a scenario how the user interface and engine interacts:

- the user will press a key to move an object left
- the interface sends a message to the engine
- the engine multicast the message
- the engine is delivered the message

- the engine updates its state
- the engine informs the interface of the new position

We will later see that not all moves are legal but this scenario will give you the general idea.

1.3 the state

Before implementing the engine process we define the representation and operations on the state. The state is simply a set of objects each with a unique identifier and a position. The players will have globally unique identifiers whereas gold nuggets will only have identifiers so that we can tell our local user interface process to remove them.

The operations that we will need for our state are as follows:

- `empty/0` an empty state
- `add/4` an object given type, identifier, and position
- `lookup/2` the position of an object given the identifier
- `request/3` to move a object to a position

The easiest way to implement this is to represent the state as a list of objects where each object is represented by the structure {`Type, Id, Pos`}. Creating an empty state, adding an object or looking up the position of an object now becomes a trivial exercise.

```
-module(state).

-export([empty/0, add/4, lookup/2, request/3]).

empty() ->

add(Id, Type, Pos, Objects) ->

lookup(Id, Objects) ->
```

Complete the code above and use the `lists:keysearch/3` library function. Also test your implementation with some simple examples.

To validate a request to move a player is slightly more tricky. We need to check if there is any other object at that position and decide if we should grant the move possibly updating the state. We will not allow a player to move to the same position as any other player but it should of course be allowed to move to the position of a gold nugget.

The `request(Player, Pos, State)` function should check if a player can move in a direction and return either:

- {`granted, Updated`}, if the move is legal
- {`gold, Gold, Updated`}, if the move is legal and hits a nugget, or
- {`denied, Reason`}, if there was an obstacle in the way.

If a gold nugget is found this nugget should be removed form the state and the identifier of the nugget is returned. Below you have some skeleton code to get you started. We can take for granted that a object that we want to move is actually present in the state.

```
request(Player, Next, Objects) ->
    case lists:keysearch(Next, 3, Objects)  of
        false ->
            {granted, .... };
        {value, ...}} ->
            {gold, ..., ...};
        {value, ...} ->
            {denied, "place taken"}
    end.

update(Id, Pos, Objects) ->
   [... | lists:keydelete(Id, 2, Objects)].
```

We now have enough support to start building the engine process.

1.4 the engine

When the engine is started it is given a unique identifier, a initial position, a list of gold nuggets and a list of peer nodes. When initialized it will first transform the list of peers to a list of *pids*. If the multicast process is registered under the name `cast` and the list of peers include a node `'gold@s1192.it.kth.se'` then the corresponding pid will be {`cast, 'gold@s1192.it.kth.se'`. The list of pids is given to the multicast process and the multicast process is registered under the same name. This will allow all multicast processes to find each other.

In the initialization phase we also create the user interface process before we move into a standby state. The standby state is needed since we do not want to start one player before all peers have created their multicasting processes. If a player is allowed to start in advance it will start to send messages into thin air.

```
-module(engine).

-export([start/4, init/4]).

-define(rows, 60).
-define(cols, 80).
-define(cast, cast).

start(Id, Pos, Gold, Peers) ->
   spawn(engine, init, [Id, Pos, Gold, Peers]).

init(Id, Pos, Gold, Peers) ->
    Pids = lists:map(fun(X) -> {?cast, X} end, Peers),
    Cast = multicast:start(self(), Pids),
    register(?cast, Cast),
    Gui = gui:start(self(), ?cols, ?rows),
    standby(Id, Pos, Gold, Cast, Gui).
```

In the standby state we are waiting for the user to hit the start button (or quit the game). When this happens we know that all peers are up and running. We will inform them about our position and where we have placed our gold nuggets.

```
standby(Id, Pos, Gold, Cast, Gui) ->
    receive
        start ->
            io:format("let's go~n", []),
            % inform everyone about our position
            Cast ! {send, {player, Id, Pos}},
            % and where we placed our gold nuggets
            lists:map(fun(XY) -> Cast ! {send, {gold, XY}} end, Gold),
            % our state is still empty
            State = state:empty(),
            loop(Id, State, Cast, Gui);
        stop ->
            Cast ! stop,
            io:format("terminating~n", []),
            ok
    end.
```

Notice that we start with an empty state. The state will only be filled once the multicast process delivers the messages. In this way we treat all messages the same, regardless if they come from our own engine or from some other engine.

It is now time to define the heart of the system, the engine loop.

```
loop(Id, State, Cast, Gui) ->
    receive
        {player, Id, Pos} ->
            % create me (I know its me since the second parameter matches Id)
            :
            loop(Id, Updated, Cast, Gui);

        {player, Player, Pos} ->
            % create another player
            :
            loop(Id, Updated, Cast, Gui);

        {gold, Pos} ->
            % create a gold nugget
            :
            loop(Id, Updated, Cast, Gui);

        {move, Player, Dir} ->
            % request to move a player in a direction
            :
            :

        {move, Dir} ->
            % directive from the gui to move me
            :
            loop(Id, State, Cast, Gui);

        stop ->
            % directive from the gui to stop executing
            :
            ok
    end.
```

The only complicated part of the engine should be how to handle the move message from the multicast process. The message asks us to move in a direction so we must of course find the current position of the object, calculate its new position (if it's even inside the board), and then check with the state if the request can be followed.

To calculate new positions we implement a function `move/2` that will give us the new position given a direction and a position. Let's represent a position as the record {`X, Y`}. The board is ?rows times ?cols so any valid position must be inside these boundaries.

```
move(right, {X, Y}) ->
```

```
    if
        X < ?cols-1 -> {X+1, Y};
        true -> false
    end;
:
:
```

With a bit of trial and error you will soon have your game up and running.

2 The Fun

To start the game we first need to start Erlang in distributed mode. You do this from a shell using two command line arguments, one that sets the name of the Erlang node and another that gives it a magic cookie. The cookie should of course be the same.

```
erl -name gold@130.237.212.139 -setcookie hello
```

If your name is registered in a DNS server you could use the domain name of the machine instead of the IP address. If you have any problems check your firewall setting.

When you have loaded your game you can start the game with the following command.

```
engine:start(jane, {4,4}, [{34,23},{12,34},{56,58}], ['gold@s1192.it.kth.se']);
```

The first argument is simply the name of this digger, the second its position and the this a list of gold nuggets. The last argument is the list of peers, in this case only one running in the Erlang shell `gold@s1192.it.kth.se`.

To see that it is all working you can run locally using only one machine but to really test the system you will need at least two machines.

3 The Problem

Work in groups and start a game with as many players as you have computers. Then try to play the game, first slowly to see that everything is working and then faster to see if the implementation can keep up with heavier load.

When you think it works, try using two or more computers and let the gold diggers bang into each other, head to head. Then try to stand next to a gold nugget. What is happening? Why?

Your task is to identify the problem and propose a solution. You should also write a four page (no not four pages of source code) paper where you describe the problem, how it can be solved and a outline how to implement it.

On the seminar you should be prepared to explain the problem and the solution you propose. You are allowed to work in groups but each member should write her own paper and be able to explain the solution.

You do not need to complete a implementation of your solution but in this case the devil is in the detail. You might think that you get it right but if you try to implement it you will most certainly run into problems that you did not think about at first.

Appendix A

gui.erl

```
-module(gui).

-export([start/3, init/3]).

-define(scale, 10).
-define(title, "Goldy").
```

The cols and rows define the size of the game. This is then scaled to make better graphical presentation. The process is the game engine to which user commands are sent.

```
start(Engine, Cols, Rows) ->
    spawn(gui, init, [Engine, Cols, Rows]).

init(Engine, Cols, Rows) ->
    Width = Cols * ?scale,
    Height = Rows * ?scale,
    Win = gs:window(gs:start(),
      [{map,true},{keypress, true},
       {title,?title},{width,Width+40},
       {height,Height}]),

    gs:frame(packer, Win,
      [{packer_x, [{fixed, 40}, {fixed, Width}]},
       {packer_y, [{fixed, Height}]}]),

    gs:frame(buttons,packer,
      [{packer_x,[{fixed,40}]},
       {packer_y,[{stretch,1,50},{stretch,2,50},{stretch,1,50}]},
       {pack_xy, {1,1}}]),
    gs:canvas(canv, packer,
      [{pack_xy,{2,1}}, {bg, blue},
       {width,Width}, {height,Height}]),

    gs:button(stop, buttons,[{label,{text,"Stop"}},{pack_xy,{1,1}}]),
    gs:button(start, buttons,[{label,{text,"Start"}},{pack_xy,{1,2}}]),

    gs:config(packer,[{width,Width+40},{height,Height}]),

    loop(Engine, empty()),
    gs:stop().

loop(Engine, Objects)->
    receive
        {gs, stop, click, _Data, _Args} ->
            Engine ! stop;
        {gs, start, click, _Data, _Args} ->
            Engine ! start,
            loop(Engine, Objects);
        {gs, _, destroy, _, _} ->
            Engine ! stop;
        {create, Obj, Id, Pos} ->
            % create a object at position Pos
            Ref = create(Obj, Pos),
            Updated = add(Id, Ref, Objects),
            loop(Engine, Updated);
        {move, Id, Pos} ->
            % move the object to C, R
            Ref = find(Id, Objects),
            move(Ref, Pos),
            loop(Engine, Objects);
        {remove, Id} ->
            % removing item
            Ref = find(Id, Objects),
            remove(Ref),
            Updated = delete(Id, Objects),
            loop(Engine, Updated);
        {gs, _, keypress, _, ['Up',_,_,_]} ->
            Engine ! {move, up},
            loop(Engine, Objects);
        {gs, _, keypress, _, ['Down',_,_,_]} ->
            Engine ! {move, down},
            loop(Engine, Objects);
        {gs, _, keypress, _, ['Right',_,_,_]} ->
            Engine ! {move, right},
            loop(Engine, Objects);
        {gs, _, keypress, _, ['Left',_,_,_]} ->
            Engine ! {move, left},
            loop(Engine, Objects);
        {gs, _, keypress, _, _} ->
            loop(Engine, Objects);
        Error ->
            io:format("gui: strange message ~w ~n", [Error]),
            loop(Engine, Objects)
    end.
```

All objects are small circles. Objects of diffent types are created with different colors.

```
create(Obj, {C, R}) ->
    Color = case Obj of
                me ->
                    red;
                player ->
                    grey;
                gold ->
                    yellow
            end,
    X = C*?scale,
    Y = R*?scale,
    gs:create(oval,canv,[{coords,[{X, Y}, {X+?scale, Y+?scale}]},
                         {fill,Color},{bw,2}]).


move(Ref, {C, R}) ->
    X = C*?scale,
    Y = R*?scale,
    gs:config(Ref,[{coords,[{X, Y}, {X+?scale, Y+?scale}]}]).


remove(Ref) ->
    gs:destroy(Ref).
```

The state is represented as a list of object references identified by unique identifiers. We assume that the objects are actually there when looking for them.

```
empty() ->
    [].

find(Id, Objects) ->
    case lists:keysearch(Id, 1, Objects) of
        {value, {Id, Ref}} ->
            Ref
    end.

add(Id, Ref, Objects) ->
    [{Id, Ref}|Objects].

delete(Id, Objects) ->
    lists:keydelete(Id, 1, Objects).
```