Functional Message Passing:
Getting started with Erlang

Arnab Sinha

COS 597C, Oct 21

materials drawn from:
http://www.erlang.org/download/getting_started-5.4.pdf

Overview

Background and Motivation

Sequential Erlang

Concurrent Erlang

Some Applications

History of Erlang

- Erlang either refer to Danish mathematician and engineer Agner Krarup Erlang, or alternatively, as an abbrev. of "Ericsson Language".
- Experiments with Erlang started in Ellemtel Computer Science Laboratory in 1987.
- First version was developed by Joe Armstrong.

Agner Krarup Erlang (1878-1929)
Inventor of the fields: traffic engineering and queueing theory (foundation of telecom network studies).

Cool Features of Erlang

- Started out as a concurrent Prolog.
- High reliability. (e.g. Ericsson Switch)
  - "You need mechanisms and primitives provided to the general programmer or the developer in order for the systems to have these properties, otherwise they tend to get lost in the way." – Lennart Ohman, Developer of Open Telecom Platform (OTP), MD of Sjoland & Thyselius Telecom AB in Sweden.
- Seamless scaling to large number of processes (literally 100,000 processes) through message passing communication.
- Functional programming (Seq.) + OO (Conc.).
- Concurrency for the sake of modularity, reliability etc.
- Library support (web-server, databases etc.)

Projects using Erlang

- Ericsson AXD301 switch (1998).
  - reported to achieve a reliability of nine “9”s (i.e. 1 sec of downtime in 1 billion seconds, roughly 30 years.)
- CouchDB, a document based database that uses MapReduce.
- ejabberd, instant messaging server
  - Facebook chat system based on ejabberd.

- Twitterfall, a service to view trends and patterns from Twitter

- …

Future of Erlang: What experts believe.

Joe Armstrong, author of Erlang

- Scalability: "Virtually all language use shared state concurrency. This is very difficult and leads to terrible problems when you handle failure and scale up the system...*Some pretty fast-moving startups in the financial world have latched onto Erlang*; for example, the Swedish www.kreditor.se." – Joe Armstrong, "Programming Erlang".

Ralph Johnson, co-author of the now-legendary book, "Design Patterns"

- Promise for multi-core apps: "I do not believe that other languages can catch up with Erlang anytime soon. It will be easy for them to add language features to be like Erlang. It will take a long time for them to build such a high-quality VM and the mature libraries for concurrency and reliability. So, Erlang is poised for success. *If you want to build a multicore application in the next few years, you should look at Erlang.*" – Ralph Johnson (UIUC), "Erlang, the next Java".

Sequential Programming: The Erlang Shell

```
% erl
Erlang (BEAM) emulator version 5.2 [source] [hipe]

Eshell V5.2  (abort with ^G)
1>

1> 2 + 5.
7
2>
```

Full-stop and carriage return

Numbers

On Ctrl-C

```
BREAK: (a)bort (c)ontinue (p)roc info (i)nfo (l)oaded
       (v)ersion (k)ill (D)b-tables (d)istribution

a
%
```

Another way to shutdown system: `halt().`

Sequential Programming: Modules and Functions

In Erlang the file name should be the module name: `tut.erl`

```
-module(tut).
-export([double/1]).

double(X) ->
    2 * X.
```

Function prototype declaration

Let's compile the file.

```
3> c(tut).
{ok,tut}
```

Let's run the program.

```
4> tut:double(10).
20
```

```
-module(tut1).
-export([fac/1]).

fac(1) ->
    1;
fac(N) ->
    N * fac(N - 1).
```

";" denotes "else" where as "." denotes "end"

Variables must start with capital letters, e.g. `Number, ShoeSize, Age`

Sequential Programming: Atoms

```
-module(tut2).
-export([convert/2]).

convert(M, inch) ->
    M / 2.54;

convert(N, centimeter) ->
    N * 2.54.
```

Atoms must start with small letters, e.g. `centimeter, inch, charles`

Let's compile and test

```
9> c(tut2).
{ok,tut2}
10> tut2:convert(3, inch).
1.18110
11> tut2:convert(7, centimeter).
17.7800
```

Let's try something which is not matching.

```
13> tut2:convert(3, miles).

=ERROR REPORT==== 28-May-2003::18:36:27 ===
Error in process <0.25.0> with exit value: {function_clause,[{tut2,convert,[3,miles]},{erl_eval,expr,3}
** exited: {function_clause,[{tut2,convert,[3,miles]},
                            {erl_eval,expr,3},
                            {erl_eval,exprs,4},
                            {shell,eval_loop,2}]} **
```

Atoms are analogous to elements of enumerated type or data-types in ML.

However:

- ✓ An atom consumes memory (*4 bytes/atom in a 32-bit system, 8 bytes/atom in a 64-bit system*).
- ✓ The *atom table is not garbage collected*, and so atoms will accumulate until the system tips over, either from memory usage or because 1048577 atoms were declared.

Sequential Programming:
Tuples and Lists

```
tut2:convert(3, inch).
```

Confusing!! Does this mean that 3 is in inches?
Or 3 cm needs to converted to inches?

```
-module(tut3).
-export([convert_length/1]).

convert_length({centimeter, X}) ->
    {inch, X / 2.54};
convert_length({inch, Y}) ->
    {centimeter, Y * 2.54}.
```

This is better!
Convert "X cm"

Although, tuple has a fixed number of parts, but it can contain any valid Erlang *term*.

```
{moscow, {c, -10}}
{cape_town, {f, 70}}
{paris, {f, 28}}
```

While tuples are enclosed within "{ }", lists are enclosed by "[ ]".

```
[{moscow, {c, -10}}, {cape_town, {f, 70}}, {stockholm, {c, -4}},
 {paris, {f, 28}}, {london, {f, 36}}]
```

A very useful way of looking at parts of list is by using "|" (like Prolog).

```
18> [First |TheRest] = [1,2,3,4,5].
[1,2,3,4,5]
19> First.
1
20> TheRest.
[2,3,4,5]
```

Sequential Programming: Record

Records are similar to structs.

```
-module(my_server).

-record(server_opts,
    {port,
    ip="127.0.0.1",
    max_connections=10}).
```

1st parameter: Name of the record.

2nd parameter: Tuple that contains the fields of the record and their default values.

Creating records.

```
Opts1 = #server_opts{port=80}.
Opts2 = #server_opts{port=80, ip="192.168.0.1"}.
```

Rest will take default values.

Accessing records.

```
Opts = #server_opts{port=80, ip="192.168.0.1"},
Opts#server_opts.port
```

Any time you want to access a record you have to include the record's name. Why? Because records aren't really internal data types, they're a compiler trick. Internal representation (map):

```
{server_opts, 80, "127.0.0.1", 10}
```

Source: http://20bits.com/articles/erlang-an-introduction-to-records/

Sequential Programming: Record

Updating records.

```
Opts = #server_opts{port=80, ip="192.168.0.1"},
NewOpts = Opts#server_opts{port=7000}.
```

Matching records.

```
handle(Opts=#server_opts{port=8080}) ->
    % do special port 8080 stuff
handle(Opts=#server_opts{}) ->
    % default stuff
```

Guard statement.

```
handle(Opts) when Opts#server_opts.port <= 1024 ->
    % requires root access
handle(Opts=#server_opts{}) ->
    % Doesn't require root access
```

Binding to ports below 1024 requires root access.

Source: http://20bits.com/articles/erlang-an-introduction-to-records/

Sequential Programming:
Writing Outputs to the Terminal

Function `format/2` takes 2 lists.

Each ~w is replaced by a term taken in order from the second list

```
32> io:format("hello world~n", []).
hello world
ok
33> io:format("this outputs one Erlang term: ~w~n", [hello]).
this outputs one Erlang term: hello
ok
34> io:format("this outputs two Erlang terms: ~w~w~n", [hello, world]).
this outputs two Erlang terms: helloworld
ok
35> io:format("this outputs two Erlang terms: ~w ~w~n", [hello, world]).
this outputs two Erlang terms: hello world
ok
```

`io:format` returns `ok` if there is no error.

Sequential Programming: Arity

Let's consider the following program for finding maximum number in a given list.

```
-module(tut6).
-export([list_max/1]).

list_max([Head|Rest]) ->
    list_max(Rest, Head).

list_max([], Res) ->
    Res;
list_max([Head|Rest], Result_so_far) when Head > Result_so_far ->
    list_max(Rest, Head);
list_max([Head|Rest], Result_so_far)  ->
    list_max(Rest, Result_so_far).
```

We have two functions with same name: `[list_max/1]` and `[list_max/2]`.

However, in Erlang they are regarded as entirely different functions. (`name/arity`).

Concurrent Programming: Processes

Ease of creation and communication among parallel threads make Erlang more appealing.

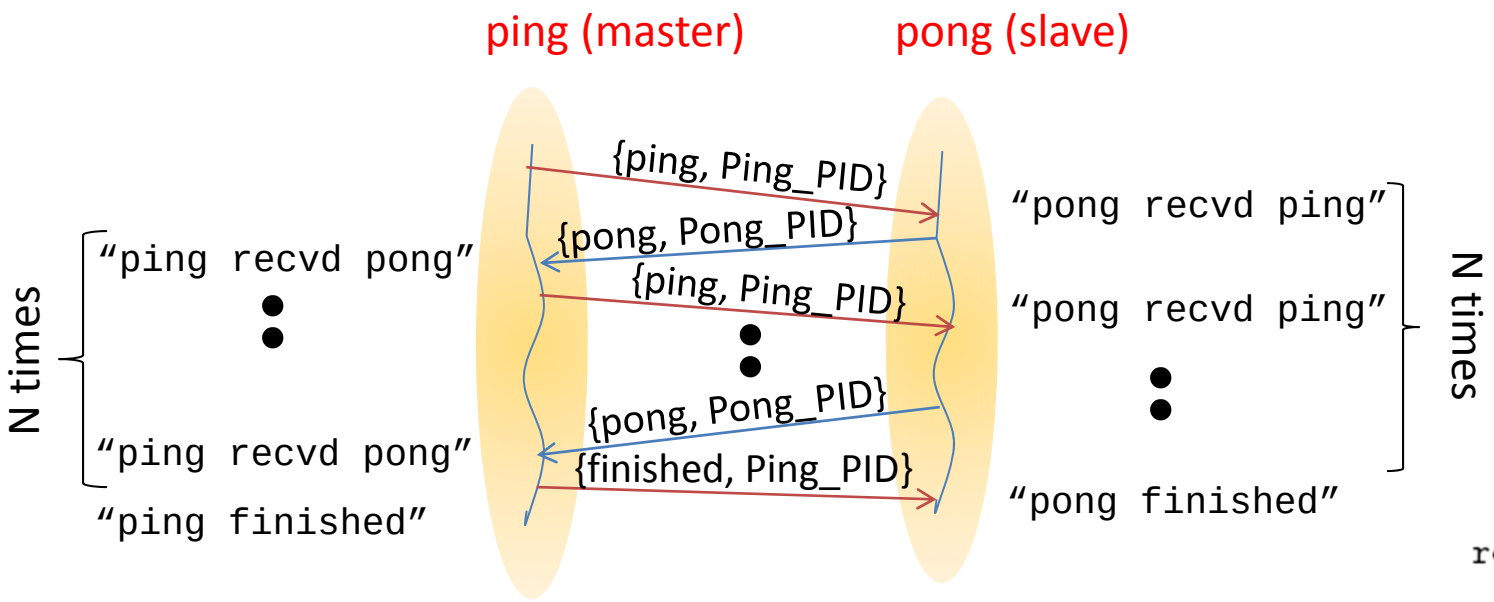
Threads of Erlang share no data – hence they are processes.

```
-module(tut14).

-export([start/1, say_something/2]).

say_something(What, 0) ->
    done;
say_something(What, Times) ->
    io:format("~p~n", [What]),
    say_something(What, Times - 1).

start() ->
    spawn(tut14, say_something, [hello, 3]),
    spawn(tut14, say_something, [goodbye, 3]).
```

```
spawn(Module,
Exported_Func.,
List of args)
```

Concurrent Programming:
Message Passing (Ping-Pong)

1. pong spawned with Pong_PID.
2. ping spawned with 'Pong_PID' as argument.



Send message construct:

```
Receiver_PID ! {message}
```

e.g. `Ping_PID ! Hello % Sent by Pong.`

Moreover, `my_pid = self() % returns self-PID`

Receive message construct:

```
receive
   pattern1 ->
       actions1;
   pattern2 ->
       actions2;
   ....
   patternN
       actionsN
end.
```

Concurrent Programming:
Message Passing (Ping-Pong)

```
-module(tu

-export([start/0, ping/2, pong/0]).

ping(0, Pong_PID) ->
    Pong_PID ! finished,
    io:format("ping finished~n", []);

ping(N, Pong_PID) ->
    Pong_PID ! {ping, self()},
    receive
        pong ->
            io:format("Ping received pong~n", [])
    end,
    ping(N - 1, Pong_PID).

pong() ->
    receive
        finished ->
            io:format("Pong finished~n", []);
        {ping, Ping_PID} ->
            io:format("Pong received ping~n", []),
            Ping_PID ! pong,
            pong()
    end.

start() ->
    Pong_PID = spawn(tut15, pong, []),
    spawn(tut15, ping, [3, Pong_PID]).
```

2. {ping, pid} sent to pong. "!" is the operator for send.

6. Ping prints if ping recvs 'pong'

7. Ping is the master and decides when to terminate

3. {ping, pid} recvd.
4. Send 'pong' to ping.

5. Goes back and waits for messages

1. Ping has the PID of pong

Concurrent Programming: register

What is ping and pong were created independently?

```
register(some_atom, Pid)
```

```
start() ->
    register(pong, spawn(tut16, pong, [])),
    spawn(tut16, ping, [3]).
```

Process pong registered as "pong".

Before:

Sent to PID

```
ping(N, Pong_PID) ->
    Pong_PID ! {ping, self()},
    receive
        pong ->
            io:format("Ping received pong~n", [])
    end,
    ping(N - 1, Pong_PID).
```

After:

Sent to reg. name

```
ping(N) ->
    pong ! {ping, self()},
    receive
        pong ->
            io:format("Ping received pong~n", [])
    end,
    ping(N - 1).
```

Concurrent Programming:
Distributed Programming

How about ping and pong were written on different computers?

Security Issue:

- *Magic cookie*.
- Having a file `.erlang.cookie` in the home directory of all the communicating machines.
- On Windows, directory pointed to by $HOME (env. variable) – might need to set it.

```
$ cd
$ cat > .erlang.cookie
this_is_very_secret
$ chmod 400 .erlang.cookie
```

Only owner can access it.
This is a requirement.

Any Erlang system which needs to communicate should have a name.

```
erl -sname my_name
```

A node called
"my_name" is created.

Erlang System:

- There can be multiple Erlang systems on the same computer but with different names.
- Each Erlang system running on a computer is called an Erlang node.

Concurrent Programming: Distributed Ping-Pong

```
-module(tut17).

-export([start_ping/1, start_pong/0,  ping/2, pong/0]).

ping(0, Pong_Node) ->
    {pong, Pong_Node} ! finished,
    io:format("ping finished~n", []);

ping(N, Pong_Node) ->
    {pong, Pong_Node} ! {ping, self()},
    receive
        pong ->
            io:format("Ping received pong~n", [])
    end,
    ping(N - 1, Pong_Node).
```

```
pong() ->
    receive
        finished ->
            io:format("Pong finished~n", []);
        {ping, Ping_PID} ->
            io:format("Pong received ping~n", []),
            Ping_PID ! pong,
            pong()
    end.

start_pong() ->
    register(pong, spawn(tut17, pong, [])).

start_ping(Pong_Node) ->
    spawn(tut17, ping, [3, Pong_Node]).
```

Let's assume two computers foo and bar.

```
kosken> erl -sname ping
Erlang (BEAM) emulator version 5.2.3.7 [hipe] [threads:0]

Eshell V5.2.3.7  (abort with ^G)
(ping@kosken)1>
        (ping@kosken)1> tut17:start_ping(pong@gollum).
        <0.37.0>
        Ping received pong
        Ping received pong
        Ping received pong
        ping finished
```

```
gollum> erl -sname pong
Erlang (BEAM) emulator version 5.2.3.7 [hipe] [threads:0]

Eshell V5.2.3.7  (abort with ^G)
(pong@gollum)1>

(pong@gollum)1> tut17:start_pong().
true
```

Concurrent Programming: Distributed Ping-Pong

PID vs registered name

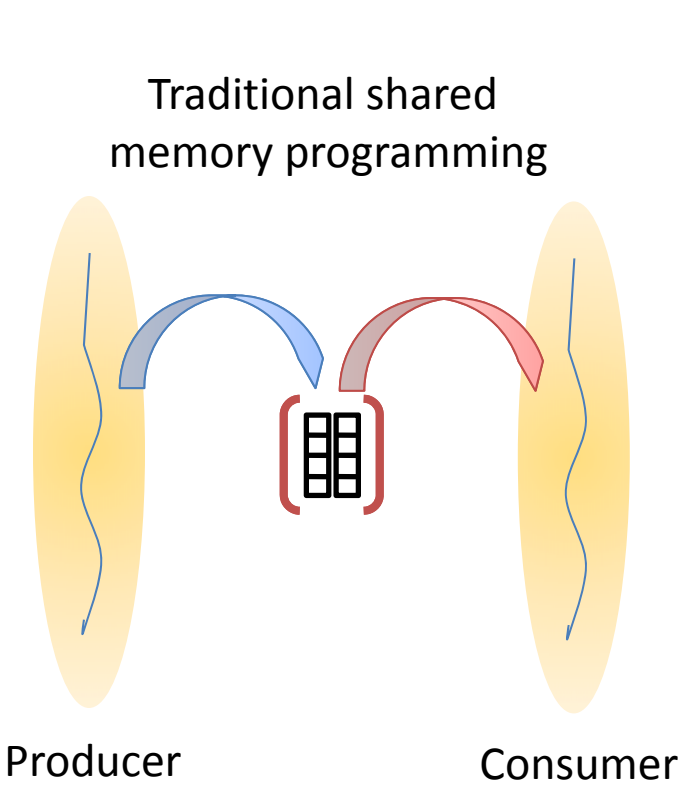
```
{ping, Ping_PID} ->
    io:format("Pong received ping~n", []),
    Ping_PID ! pong,
```

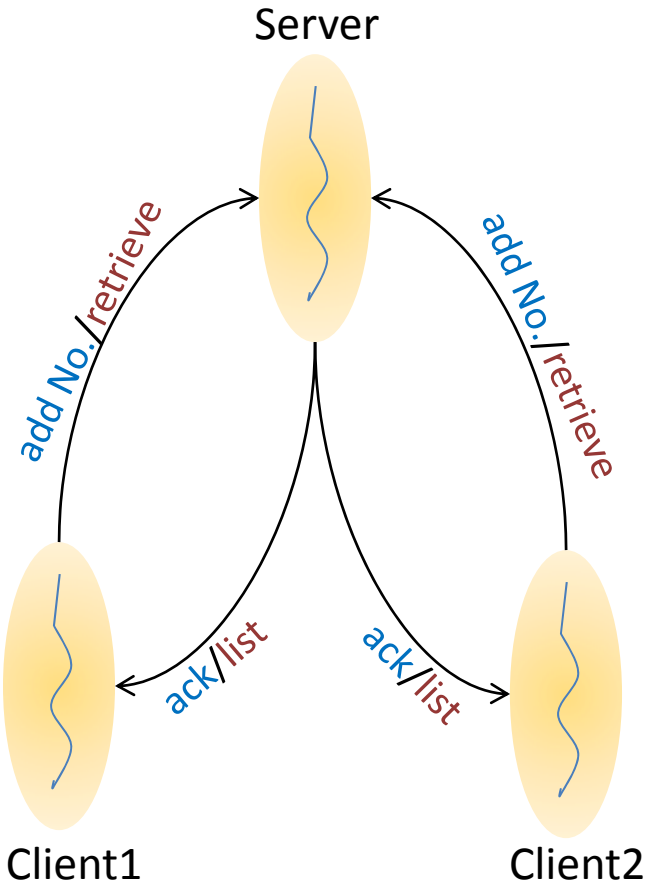
Erlang PIDs contain information about where the process executes , i.e. matter if the node is same or different.

However, for registered process (in a different node) we need `{registered_name, node_name}`.

```
{pong, Pong_Node} ! {ping, self()}
```

Some More Message Passing Examples





However, no locking is necessary in Erlang, as nothing is shared!

Hot Code Swapping

Acc. to page 355 in Joe Armstrong's book, there have been Erlang systems running out there for years with 99.9999999% reliability.

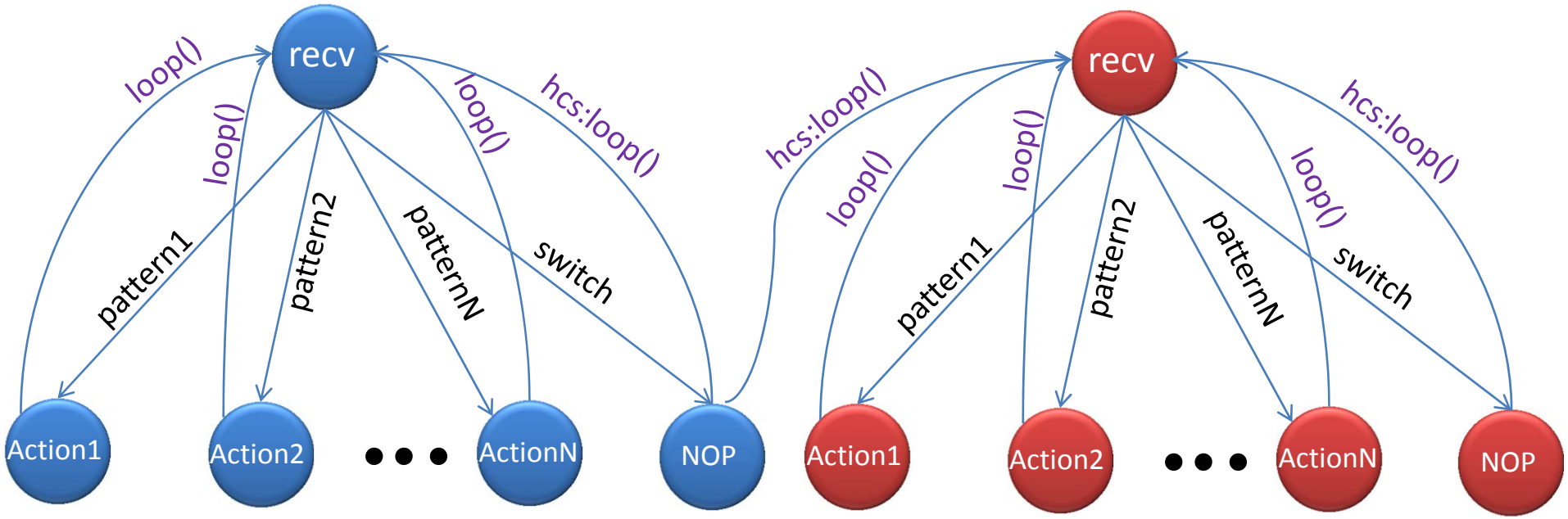
How to a fix a bug (traditional approach):

1. You find a bug in the system,
2. you fix it locally,
3. you take down the system,
4. upload the new code,
5. restart the system

How to a fix a bug (Erlang approach):

1. You find a bug in the system,
2. you fix it locally,
3. upload the new code. *System is fixed*.

Hot Code Swapping



`hcs:loop()` or `?MODULE:loop()` always points to the latest version.

Thank You!

Backup

Installation FAQ

- In Windows:
  - Download the distribution from here.
  - Download MS Visual C++ 2005 Redistributable Package (x86) from here.
  - Using winzip or winrar, extract the contents of the “vcredist_x86.exe” file to a folder, (which for me was vcredi~3.exe), then extract the contents of vcredi~3.exe file, and within that are vcredis1.cab and vcredist.msi.
  - While installation of the exe downloaded in step 1, point to the .msi file extracted in step 3.

Sys module

We use the sys module to suspend the registered process for our gen_server.
When a process is suspended, it will only respond to system messages.
Next the change_code/4 method is called:

```
5> sys:suspend(t).
ok
6> c(t).
{ok,t}
7> l(t).
{module,t}
8> sys:change_code(t,t,[],[]).
ok
9> sys:resume(t).
ok
10> t:print_state().
state: 3.0
ok
11> t:print_state().
state: 4.0
ok
```

```
change_code(Name, Module, OldVsn, Extra) ->
                ok | {error, Reason}
Types:
Name = pid() | atom() | {global, atom()}
Module = atom()
OldVsn = undefined | term()
Extra = term()
```