Introduction to Erlang

Erlang is a functional, concurrency-oriented, distributive, fault-tolerant programming language.

Philosophy of Erlang

- “Erlang was designed for writing concurrent programs that "run forever".” - Joe Armstrong
- Let it fail. Have another process deal with it.
- Fault tolerance requires at least *two* computers.

Erlang Mindset

- Lots of little tasks
- Tons of processes
- Each process does a task very well
- Ant Colony
- Ants are processes sending messages to each other

Why Erlang?

Erlang makes hard things easier.

Concurrency

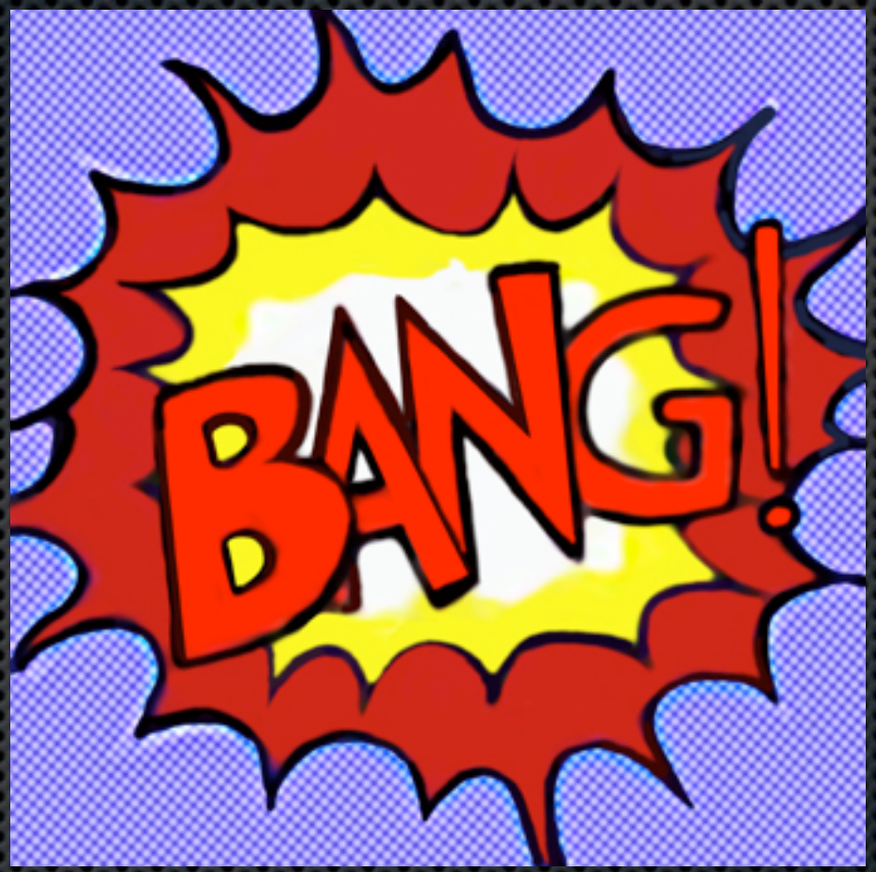
Spawning Processes

```
F = fun() ->
      io:format(“hi”)
    end,
spawn(F).
```

```
>> hi<0.34.0>
```

```
spawn(io, format, [“hi”]).
```

Message Passing


BANG!

```
Pid ! Message
```

```
Pid ! {do_task, run_home}
```

```
Pid ! 2
```

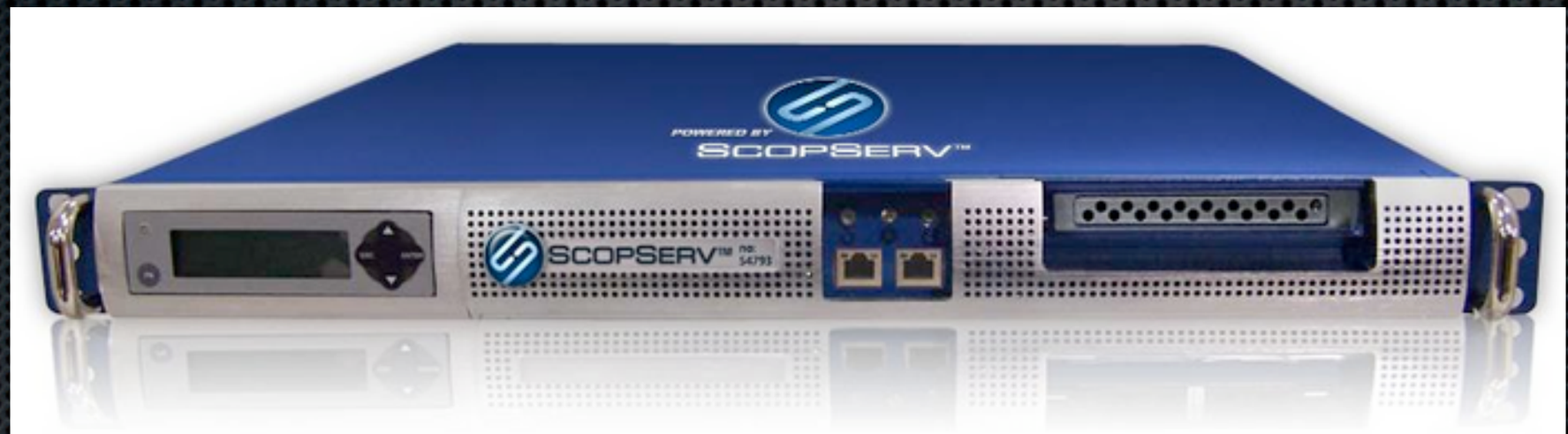
```
receive ... end
```

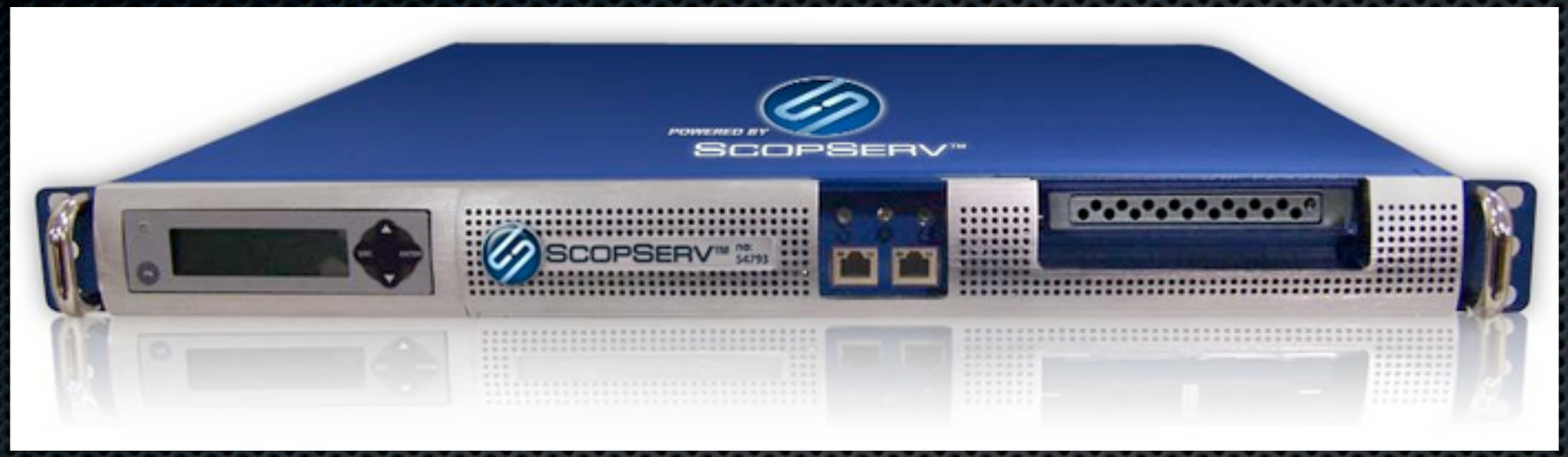
```
receive
  Pattern1 [when Guard1] ->
    Expressions1;
  Pattern2 [when Guard2] ->
    Expressions2
end.
```

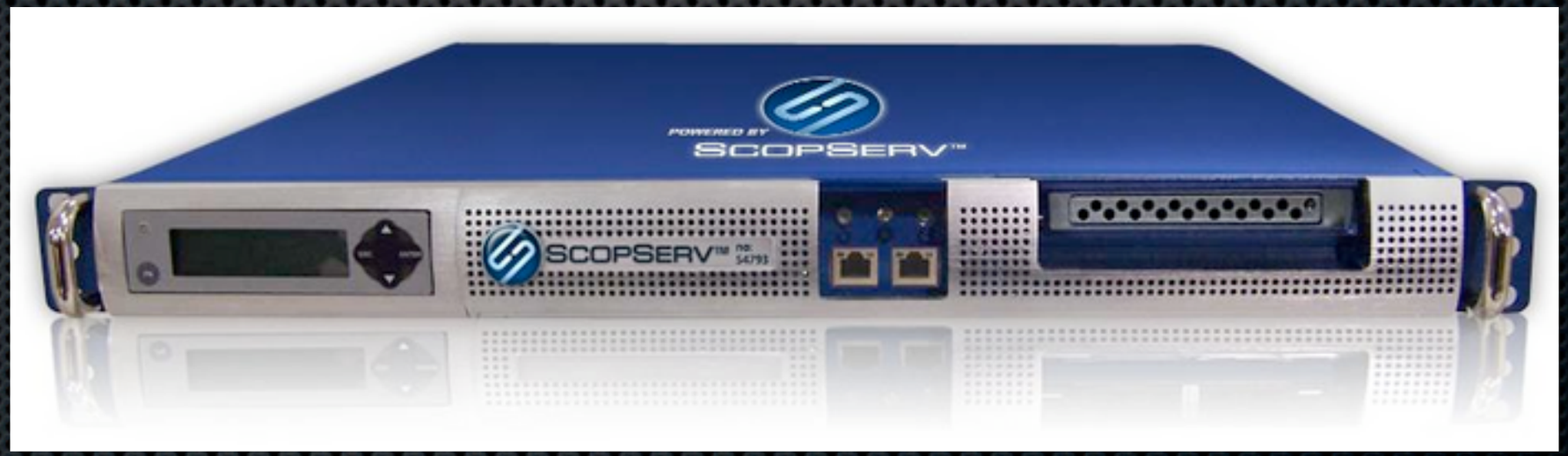
Messages are contained in a process's inbox.

```
receive
  _ ->
    ok
end.
```

Fault Tolerance

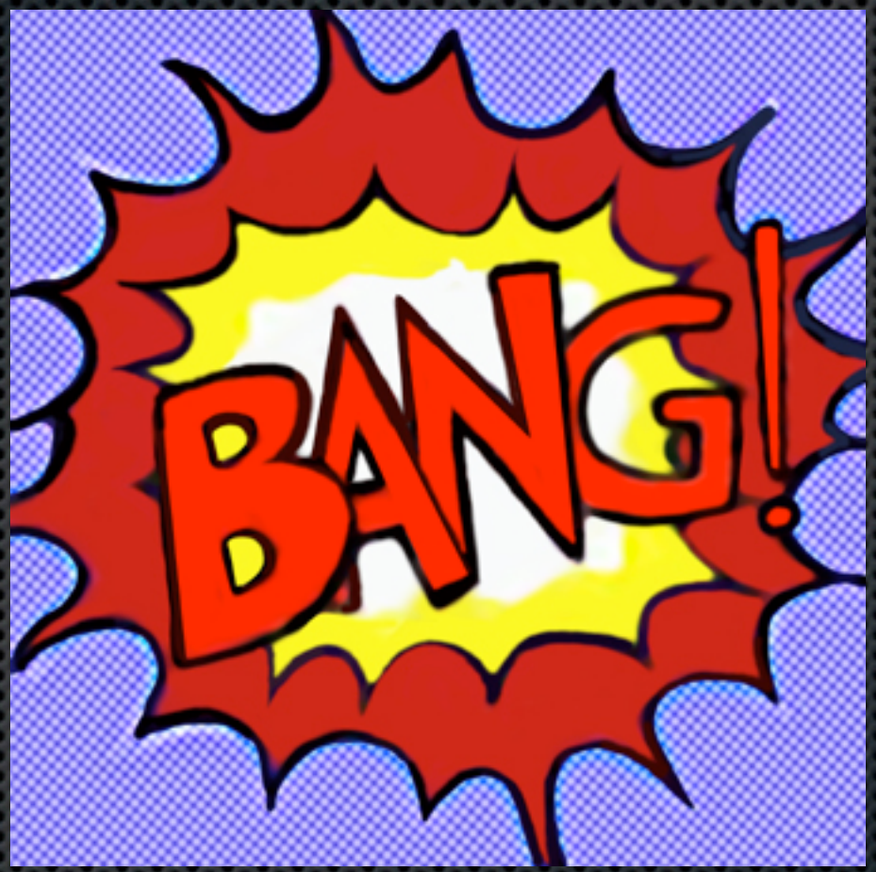

POWERED BY
SCOPSERV™
SCOPSERV™


POWERED BY
SCOPSERV™
SCOPSERV™


POWERED BY
SCOPSERV™
SCOPSERV™

which is why erlang was designed to be....

Distributive


BANG!

```
Pid ! Message
```

where Pid exists on another node.

Erlang Nodes

- Can only talk if the erlang ‘cookie’ is the same
- Can exist on the same machine or different machines
- Does have overhead, isn’t free
- Can have ‘C’ nodes (+ other node types that speak erlang)
- Communicate via Global Name Registry, Specific Pids, or by Node name

The Basics

Erlang is Functional

Periods, Commmas, Semicolons Oh My!

People think Erlang's syntax is ugly. It's actually very logical.

- Periods (.) end everything except when
- Semicolons (;) end clauses
- and Commas (,) separate expressions.

```
calculate_area({square, Size}) ->
  Size * Size;

calculate_area({circle, Radius}) ->
  Pi = 3.14,
  Pi * Radius * Radius;

calculate_area(_Unknown) ->
  ok.
```

```
case Expression of
  {do_task, run_home} ->
    Home = 123,
    spawn(run, home, [Home]);
  _ ->
    ok
end.
```

Data Types

- Integer - however big you can imagine
- Floats
- Atoms - similar to a ruby symbol, they are used to represent non-numerical constants (true, false, ok, error)
- Tuple - Fixed length collection of items ({one, two, three} or {one, two} etc)
- Lists - Stores a variable number of things ([1, 2, {one, two}, $a])

Missing Links...

- Strings - A string is just a list of integers in erlang.
- Booleans - Use atoms (true, false)
- Hash - Property List ([tuple()]) shown later

Pattern Matching

Once you pattern match, variable assigning feels dirty.

```
1 = 1      ( ok )
```

```
1 = 2      ( error thrown bad_match )
```

```
Value = {2, 4},
{Width, Height} = {2, 4}   ( ok )
```

```
[Head|Tail] = [1,2,3,4]     ( ok )
```

Variable Binding

or restricting infinite possibilities.

An unbound variable can be anything it needs to be (infinite possibilities)

A.

Pattern matching an unbound variable is like restricting what it can be.

```
A = 1.
```

```
A = 2.   (error bad_match)
```

Tuples

Single entity that holds a collection of items.

```
{1, 2, 3}.
```

```
{one, two}.
```

```
{user_id, 2, user_name, “asceth”}.
```

Lists

Variable sized container of items.

```
List = [1, 2, 3],
NewList = List ++ [4, 5, 6].
>> [1, 2, 3, 4, 5, 6]
```

```
[Head|Tail] = NewList.
>> Head -> 1
>> Tail -> [2, 3, 4, 5, 6]
```

[tuple()]

Property Lists or semi-hashes.

```
PList = [{key, value}, {key1, value1}].
```

```
case lists:keysearch(key, 1, PList) of
  false ->
    error;
  {value, {Key, Value}} ->
    Value
end.
```

- keydelete - deletes tuple with specified key in list of tuples and returns new list
- keymap - returns list of tuples where Nth element is replaced be specified fun
- keymember - tests for existence of key
- keymerge - combines two tuples with tuple1 taking priority
- ukeymerge, ukeysort - like keymerge but removes duplicates in tuple2

Functions

Pattern matching, Function Arities & more

- Functions with the same name can exist
- Functions with the same name and different arity can exist
- Erlang tells you if a function cannot be called due to order of definement
- Function order is very important

```erlang
contains(Key, []) ->
  false;
contains(Key, List) ->
  lists:any(fun(X) -> Key == X end, List).
```

```erlang
contains(1, [1, 2, 3]).
true

contains({home, 123}, [{apartment, 333}, {house, 000}]).
false
```

Anonymous

```
contains(Key, []) ->
  false;
contains(Key, List) ->
  lists:any(fun(X) -> Key == X end, List).
```

```
F = fun(X, Y) ->
      X1 = X * 2,
      Y1 = Y * X,
      X1 + Y1
    end,
F(2, 2).
>> 12
```

Looping

Tail recursion makes looping good again.

```
print([]) ->
  ok;

print([H|T]) ->
  io:format(“~p~n”, [H]),
  print(T).
```

Conditionals

If’s are okay, cases are cooler.

```
if
   A == 5 ->
      do_something();
   true ->
      do_something_else()
end.
```

```
case A of
  0 ->
    first();
  2 ->
    second();
  _ ->
    something()
end.
```

Resources

http://www.erlang.org

Technorati

The Pragmatic Programmers

Programming Erlang

Software for a Concurrent World

Joe Armstrong

Questions?