Erlang

For building scalable, fault-tolerant applications

Joshua Miller
josh@joshinharrisburg.com
CPOSC 2009

Pain.

- How do I do multiple things at once?

Imagine if two threads, A and B, are executing the add method on the same instance of the Counter class. There is no way to know when the operating system switches between the two threads. The code is not executed as a single instruction by the Java virtual machine. Rather it is executed along the lines of:

```
get this.count from memory into register
add value to register
write register to memory
```

Observe what happens with the following mixed execution of threads A and B:

```
    this.count = 0;
A:  reads this.count into a register (0)
B:  reads this.count into a register (0)
B:  adds value 2 to register
B:  writes register value (2) back to memory. this.count now equals 2
A:  adds value 3 to register
A:  writes register value (3) back to memory. this.count now equals 3
```

http://tutorials.jenkov.com/java-concurrency/race-conditions-and-critical-sections.html

Immutability.

- Once you assign a variable, you can't change it.
- Better yet, neither can anyone else.

```
14:14 ~ $ erl
Erlang R13B (erts-5.7.1) [source] [smp:2:2] [rq:2] [async-threads:0] [kernel-poll:false]

Eshell V5.7.1  (abort with ^G)
1> X = 5.
5
2> X + 10.
15
3> X = 6.
** exception error: no match of right hand side value 6
4>
```

So, what about state?

- Recursion.

```
counter() ->
  counter(1).

counter(X) ->
  counter(X + 1).
```

Whoa, whoa, whoa.

Erlang syntax.

- Damien Katz, author of CouchDB:

Basic Syntax

Erlang is based originally on Prolog, a logic programming language that was briefly hot in the 80's. Surely you've seen other languages based on Prolog, right? No? Why not? Because Prolog sucks ass for building entire applications. But that hasn't deterred Erlang from stealing it's dynamite syntax.

The problem is, unlike the C and Algol based languages, Erlang's syntax does away with nested statement terminators and instead uses expression separators everywhere. Lisp suffers the same problem, but Erlang doesn't have the interesting properties of a completely uniform syntax and powerful macro system to redeem itself. Sometimes a flaw is really a strength. And sometimes it's just a flaw.

Because Erlang's expression terminators vary by context, editing code is much harder than conventional languages. Refactoring -- cutting and pasting and moving code around -- is particularly hard to do without creating a bunch of syntax errors.

http://damienkatz.net/2008/03/what_sucks_abou.html

Erlang syntax.

- In short: It's simple, but not "good."
- A sample, from CouchDB:

```
detuple_kvs([], Acc) ->
    lists:reverse(Acc);
detuple_kvs([KV | Rest], Acc) ->
    {{Key,Id},Value} = KV,
    NKV = [[Key, Id], Value],
    detuple_kvs(Rest, [NKV | Acc]).

expand_dups([], Acc) ->
    lists:reverse(Acc);
expand_dups([{Key, {dups, Vals}} | Rest], Acc) ->
    Expanded = [{Key, Val} || Val <- Vals],
    expand_dups(Rest, Expanded ++ Acc);
expand_dups([KV | Rest], Acc) ->
    expand_dups(Rest, [KV | Acc]).
```

Pattern matching.

- This is the good stuff.

```
detuple_kvs([], Acc) ->
    lists:reverse(Acc);
detuple_kvs([KV | Rest], Acc) ->
    {{Key,Id},Value} = KV,
    NKV = [[Key, Id], Value],
    detuple_kvs(Rest, [NKV | Acc]).

expand_dups([], Acc) ->
    lists:reverse(Acc);
expand_dups([{Key, {dups, Vals}} | Rest], Acc) ->
    Expanded = [{Key, Val} || Val <- Vals],
    expand_dups(Rest, Expanded ++ Acc);
expand_dups([KV | Rest], Acc) ->
    expand_dups(Rest, [KV | Acc]).
```

Stop! Demo Time.

Where were we?

- Oh right. State.

Sharing state.

- How do we find out the current count if we can't look at some changeable variable?

```
counter() ->
  counter(1).

counter(X) ->
  counter(X + 1).
```

Message passing.

- We'll just ask it.

```
counter(X) ->
  receive
    count -> io:format("X is now ~p~n", [X])
    after 1000 -> ok
  end,
  counter(X + 1).
```

```
16> CountPid = spawn(talk, counter, [0]).
<0.78.0>
17> CountPid ! count.
X is now 7
count
18> CountPid ! count.
X is now 10
count
```

Erlang processes.

- Easy.
- Cheap.
- Fundamental.
- Just call spawn()

Erlang processes.

- “Green,” but SMP is available.
- Virtuous consequence of immutability.

Erlang processes.

- Trivially distributable!

```
19> net_adm:ping(Node).
pong
20> spawn(Node, talk, counter, [0]).
```

Erlang processes.

Tradeoffs.

- You lose:
  - Mutable state.
  - Shared memory.
- You gain:
  - Trivial concurrency.
  - Trivial distribution.

Map Reduce

```
%% Applies the function Fun to each element of the list in parallel,
%% using the local node and any others attached
pmap(Fun, List) ->
  Parent = self(),
  Nodes = [node()] ++ nodes(),

  %% Build round-robin list of nodes
  NodeQueue = lists:map(fun(X) -> lists:nth(X rem lists:flatlength(Nodes) + 1, Nodes) end,
                        lists:seq(1, length(List))),
  ListWithNodes = lists:zip(List, NodeQueue),

  %% spawn the processes
  Refs =
      lists:map(
        fun(X) ->
                {Elem, Node} = X,
                Ref = make_ref(),
                spawn(Node,
                  fun() ->
                          Parent ! {Ref, Fun(Elem)}
                  end),
                Ref
        end, ListWithNodes),

  %% collect the results
  lists:map(
    fun(Ref) ->
          receive
              {Ref, Elem} ->
                  Elem
          end
    end, Refs).
```

That’s pretty cool, but...

- I’m going to spend my whole life writing recursive loops to keep track of things?

Open Telecom Platform

- OTP for short.
- Provides:
  - gen_server pattern (a generic server that tracks state and provides responses to predefined messages)
  - gen_fsm pattern (a finite state machine)
  - Supervisors and Application containers

Erlang Term Storage

- ETS for short - a dictionary of arbitrary Erlang terms.
- Like memcache, for Erlang.

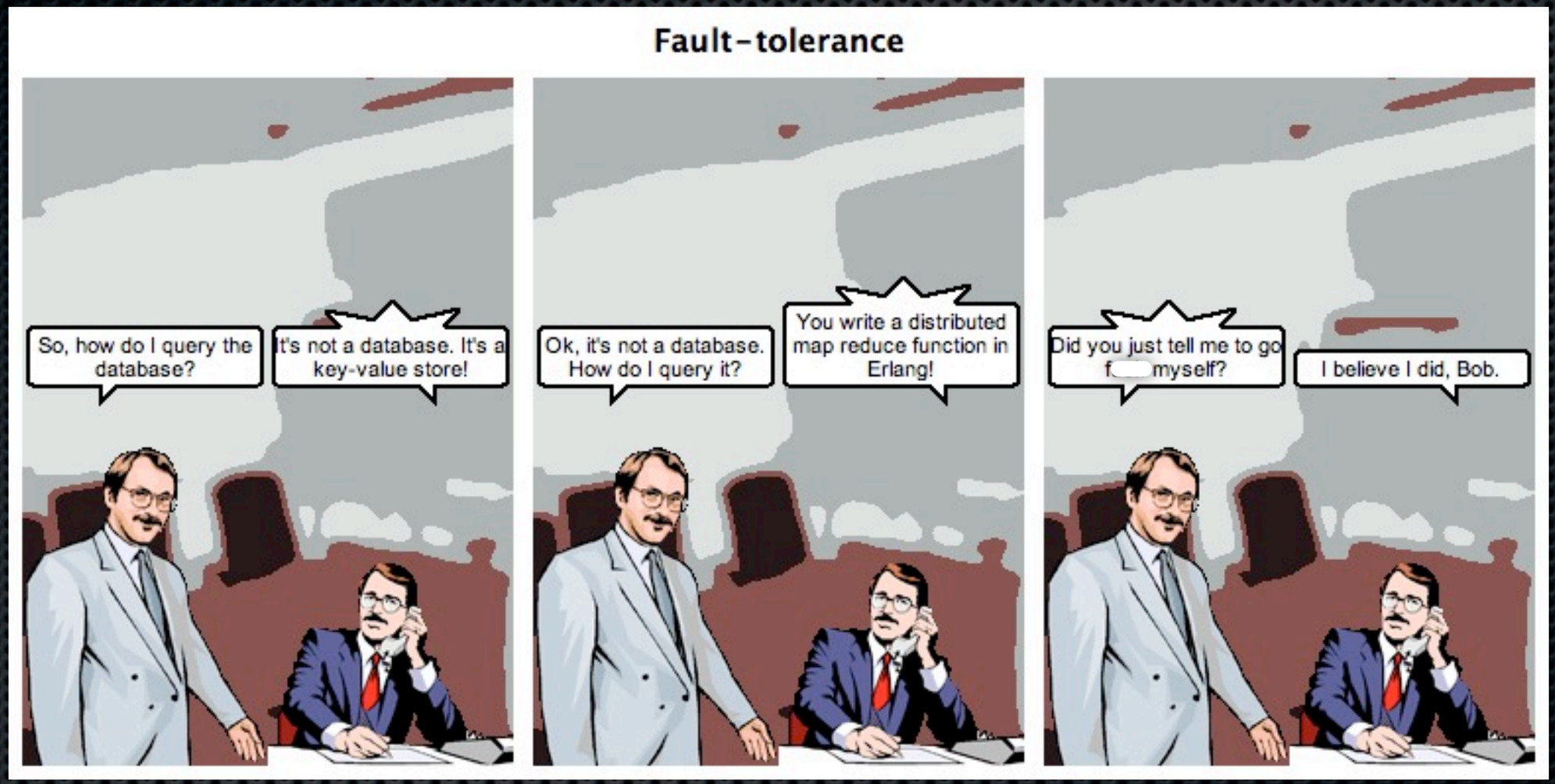
```
Eshell V5.7.1  (abort with ^G)
1> ets:new(monkeys, [bag,public,named_table]).
monkeys
2> ets:insert(monkeys, {josh, [{age, 12}, {likes_bananas, true}]}).
true
3> ets:lookup(monkeys, josh).
[{josh,[{age,12},{likes_bananas,true}]}]
4>
```

- D[isk]ETS - same thing, but persisted to disk.

Mnesia.

- All Erlang (stores records as Erlang terms).
- Distributable.
- Replicatable.
- How do you query it?

Mnesia.


Fault-tolerance
So, how do I query the database?
It's not a database. It's a key-value store!
Ok, it's not a database. How do I query it?
You write a distributed map reduce function in Erlang!
Did you just tell me to go f myself?
I believe I did, Bob.

Mnesia.

- (Query with regular old Erlang list comprehensions.)
- Like SQL, but with more job security.

And, of course...

- Typical SQL databases.
- Memcache.
- Anything that can talk HTTP.
- Or TCP.
- Or Unix sockets.
- Or C.

Where shouldn't I use Erlang?

- Where you're going to write lots of string-munging code.
- Where you need to build client-side GUIs.
  - (Of course, you can always just use Erlang for the back end)

So when I should I?

- When you need to do things in parallel.
- When you need to do things across machines.
- When you need to maintain complicated systems simply.

Resources

- *Programming Erlang*, Joe Armstrong
- *Erlang Programming* (O'Reilly), Francesco Cesarini & Simon Thompson
- OTP Design Principles - http://www.erlang.org/doc/design_principles/part_frame.html
- #erlang-otp on Freenode

Questions?

Joshua Miller
josh@joshinharrisburg.com
CPOSC 2009