ERLANG EVOLVES
FOR MULTI-CORE AND CLOUD ENVIROMENTS

Torben Hoffmann
Erlang Solutions Ltd.
@LeHoff

http://musings-of-an-erlang-priest.blogspot.dk/
Agenda

• Erlang fundamentals

• Challenges
Warning 1: The Truth

I will do a few simplifications in order to get the main points across.
Warning 1: The Truth

I will do a few simplifications in order to get the main points across.
Warning 1: The Truth

Will you tell the truth?

I will do a few simplifications in order to get the main points across.
Warning 1: The Truth

Will you tell the truth? Yes

I will do a few simplifications in order to get the main points across.
Warning 1: The Truth

Will you tell the truth? Yes
The whole truth?

I will do a few simplifications in order to get the main points across.
Warning 1: The Truth

Will you tell the truth? Yes
The whole truth? No

I will do a few simplifications in order to get the main points across.
Warning 1: The Truth

Will you tell the truth? Yes
The whole truth? No
So help you OTP?

I will do a few simplifications in order to get the main points across.
Warning 1: The Truth

Will you tell the truth?  Yes
The whole truth?  No
So help you OTP?  Yes

I will do a few simplifications in order to get the main points across.
Warning 2: Serious Love Ahead

I love Erlang!

It was part of a major career shift and I have never looked back. Apologies if I get too intense.
Realities Of Software Development

- Time-to-market pressure
- Utilisation of computing resources
- Scaling successes
- Maintenance burden
What Could Be...

3x productivity over C++/Java
Seamless scaling on multicore
Scaling nicely over machines
Less code per feature
The future is here today - it’s called Erlang!
What Could Be...

3x productivity over C++/Java
Seamless scaling on multicore
Scaling nicely over machines
Less code per feature
The future is here today – it’s called Erlang!
What Could Be...

3x productivity over C++/Java
Seamless scaling on multicore
Scaling nicely over machines
Less code per feature
The future is here today – it’s called Erlang!
What Could Be...

3x productivity over C++/Java
Seamless scaling on multicore
Scaling nicely over machines
Less code per feature
The future is here today – it’s called Erlang!
What Could Be...

3x productivity over C++/Java
Seamless scaling on multicore
Scaling nicely over machines
Less code per feature
The future is here today – it’s called Erlang!
What Could Be...

The future is here...

3x productivity over C++/Java
Seamless scaling on multicore
Scaling nicely over machines
Less code per feature
The future is here today – it’s called Erlang!
What Could Be...

The future is here...
The future is Erlang!

3x productivity over C++/Java
Seamless scaling on multicore
Scaling nicely over machines
Less code per feature
The future is here today – it’s called Erlang!
Erlang’s Original Requirements
Erlang’s Original Requirements

- Large scale concurrency
Erlang’s Original Requirements

- Large scale concurrency
- Soft real-time
Erlang’s Original Requirements

- Large scale concurrency
- Soft real-time
- Distributed systems
Erlang’s Original Requirements

- Large scale concurrency
- Soft real-time
- Distributed systems
- Hardware interaction
Erlang’s Original Requirements

- Large scale concurrency
- Soft real-time
- Distributed systems
- Hardware interaction
- Very large software systems
Erlang’s Original Requirements

- Large scale concurrency
- Soft real-time
- Distributed systems
- Hardware interaction
- Very large software systems
- Complex functionality
Erlang’s Original Requirements

- Large scale concurrency
- Soft real-time
- Distributed systems
- Hardware interaction
- Very large software systems
- Complex functionality
- Continuous operation for many years
Erlang’s Original Requirements

- Large scale concurrency
- Soft real-time
- Distributed systems
- Hardware interaction
- Very large software systems
- Complex functionality
- Continuous operation for many years
- Software maintenance on-the-fly
Erlang’s Original Requirements

- Large scale concurrency
- Soft real-time
- Distributed systems
- Hardware interaction
- Very large software systems
- Complex functionality
- Continuous operation for many years
- Software maintenance on-the-fly
- High quality and reliability
Erlang’s Original Requirements

- Large scale concurrency
- Soft real-time
- Distributed systems
- Hardware interaction
- Very large software systems
- Complex functionality
- Continuous operation for many years
- Software maintenance on-the-fly
- High quality and reliability
- Fault tolerance
Erlang’s Original Requirements

- Large scale concurrency
- Soft real-time
- Distributed systems
- Hardware interaction
- Very large software systems
- Complex functionality
- Continuous operation for many years
- Software maintenance on-the-fly
- High quality and reliability
- Fault tolerance

Sounds familiar?
Erlang’s Original Requirements

- Large scale concurrency
- Soft real-time
- Distributed systems
- Hardware interaction
- Very large software systems
- Complex functionality
- Continuous operation for many years
- Software maintenance on-the-fly
- High quality and reliability
- Fault tolerance

Sounds familiar?

Sounds good, right?
General vs Domain Specific

Small semantic gap

Telecom
General vs Domain Specific

Small semantic gap

Telecom

C++/Java

© 1999-2012 Erlang Solutions Ltd.
General vs Domain Specific

Small semantic gap

Telecom

C++/Java
General vs Domain Specific

Small semantic gap

Telecom

C++/Java

Erlang

© 1999-2012 Erlang Solutions Ltd.
General vs Domain Specific

Small semantic gap

C++/Java

Erlang

Telecom
General vs Domain Specific

Small semantic gap

C++/Java

Erlang

Telecom

Smaller gap = benefits!
Erlang was intended to deal with the control plane in telecom, which is all about orchestration of what goes on.

GUI and low-level things are not what Erlang was created for – hence Erlang has good support for integration with other languages.

Read the wonderful doctor thesis by Bjarne Däcker if you want to learn more: [http://](http://www.erlang.se/publications/bjarnelic.pdf)
Other Erlang Domains

- Messaging - XMPP et al
  - ejabberd, MongooseIM
- Webservers
  - Yaws, Chicago Boss
- Payment switches & soft switches
  - Vocalink, OpenFlow/LINC
- Distributed Databases
  - Riak, CouchDB, Scalaris
Other Erlang Domains

• Messaging - XMPP et al
  - ejabberd, MongooseIM

• Webservers
  - Yaws, Chicago Boss

• Payment switches & soft switches
  - Vocalink, OpenFlow/LINC

• Distributed Databases
  - Riak, CouchDB, Scalaris
Death propagates in shared memory unless you do a ton of defensive programming. Due to the actor model with no shared memory it is custom in Erlang to do fail–fast programming. No shared memory allows you to fail fast when suitable. Erlang uses message passing between processes to exchange information.
Death propagates in shared memory unless you do a ton of defensive programming. Due to the actor model with no shared memory it is custom in Erlang to do fail-fast programming. No shared memory allows you to fail fast when suitable. Erlang uses message passing between processes to exchange information.
Death propagates in shared memory unless you do a ton of defensive programming. Due to the actor model with no shared memory it is custom in Erlang to do fail–fast programming. No shared memory allows you to fail fast when suitable. Erlang uses message passing between processes to exchange information.
Death propagates in shared memory unless you do a ton of defensive programming. Due to the actor model with no shared memory it is custom in Erlang to do fail-fast programming. No shared memory allows you to fail fast when suitable. Erlang uses message passing between processes to exchange information.
Death propagates in shared memory unless you do a ton of defensive programming. Due to the actor model with no shared memory it is custom in Erlang to do fail–fast programming.
No shared memory allows you to fail fast when suitable. Erlang uses message passing between processes to exchange information.

To Share Or Not To Share
Death propagates in shared memory unless you do a ton of defensive programming. Due to the actor model with no shared memory it is custom in Erlang to do fail-fast programming. No shared memory allows you to fail fast when suitable. Erlang uses message passing between processes to exchange information.
Death propagates in shared memory unless you do a ton of defensive programming. Due to the actor model with no shared memory it is custom in Erlang to do fail-fast programming. No shared memory allows you to fail fast when suitable. Erlang uses message passing between processes to exchange information.
Death propagates in shared memory unless you do a ton of defensive programming. Due to the actor model with no shared memory it is custom in Erlang to do fail–fast programming.

No shared memory allows you to fail fast when suitable. Erlang uses message passing between processes to exchange information.
Death propagates in shared memory unless you do a ton of defensive programming. Due to the actor model with no shared memory it is custom in Erlang to do fail-fast programming. No shared memory allows you to fail fast when suitable. Erlang uses message passing between processes to exchange information.
Death propagates in shared memory unless you do a ton of defensive programming. Due to the actor model with no shared memory it is custom in Erlang to do fail-fast programming. No shared memory allows you to fail fast when suitable. Erlang uses message passing between processes to exchange information.
Death propagates in shared memory unless you do a ton of defensive programming. Due to the actor model with no shared memory it is custom in Erlang to do fail–fast programming.

No shared memory allows you to fail fast when suitable. Erlang uses message passing between processes to exchange information.
Death propagates in shared memory unless you do a ton of defensive programming. Due to the actor model with no shared memory it is custom in Erlang to do fail–fast programming. No shared memory allows you to fail fast when suitable. Erlang uses message passing between processes to exchange information.
Dealing With Failures

The ability to link processes and monitor them is the key to adopting fail-fast fully. Link & monitor works across machines!
Robust systems do not happen by accident – even in Erlang! You have to think about the consequences of a worker process that fails and let the supervisor take appropriate actions. Using the OTP library’s components makes it straightforward to implement the supervision tree, which has the added benefit that all things are started in the right order.
There is 1 scheduler per core.
The VM tries to load balance across the available cores.
Scales extremely well with the addition of extra cores – WITHOUT changing the programs!
Each instance of the Erlang runtime is called a node. There can be several nodes on one machine if you fancy that. Nodes detect when other nodes are not around any more – the programmer can then decide what to do. If you have the PID (Process Identifier) of a process you do not care which node it is on. You
Distribution Over Machines

Each instance of the Erlang runtime is called a node. There can be several nodes on one machine if you fancy that. Nodes detect when other nodes are not around any more – the programmer can then decide what to do.

If you have the PID (Process Identifier) of a process you do not care which node it is on. You
Each instance of the Erlang runtime is called a node. There can be several nodes on one machine if you fancy that. Nodes detect when other nodes are not around any more – the programmer can then decide what to do.

If you have the PID (Process Identifier) of a process you do not care which node it is on. You
Each instance of the Erlang runtime is called a node. There can be several nodes on one machine if you fancy that. Nodes detect when other nodes are not around any more – the programmer can then decide what to do.

If you have the PID (Process Identifier) of a process you do not care which node it is on. You
Each instance of the Erlang runtime is called a node.
There can be several nodes on one machine if you fancy that.
Nodes detect when other nodes are not around any more – the programmer can then decide what to do.
If you have the PID (Process Identifier) of a process you do not care which node it is on. You
As style changed a new group entered the top of the hip scale. For most it meant a serious restart or a stop altogether.
As style changed a new group entered the top of the hip scale. For most it meant a serious restart or a stop altogether.
As style changed a new group entered the top of the hip scale. For most it meant a serious restart or a stop altogether.
Staying Alive...

As style changed a new group entered the top of the hip scale. For most it meant a serious restart or a stop altogether.
Staying Alive Erlang Style

With Erlang you can survive upgrades without loosing service. Along with the code change signal you specify how the internal state of the process should be updated before continuing.
With Erlang you can survive upgrades without loosing service. Along with the code change signal you specify how the internal state of the process should be updated before continuing.
With Erlang you can survive upgrades without loosing service. Along with the code change signal you specify how the internal state of the process should be updated before continuing.
With Erlang you can survive upgrades without losing service. Along with the code change signal you specify how the internal state of the process should be updated before continuing.
With Erlang you can survive upgrades without loosing service. Along with the code change signal you specify how the internal state of the process should be updated before continuing.
Challenges
How Will I Know If It Really Scales?

A Scalability Benchmark Suite for Erlang/OTP.
A number of synthetic benchmarks plus real-world (dialyzer and scalaris). Extendable to test your own application.
A Scalability Benchmark Suite for Erlang/OTP.
A number of synthetic benchmarks plus real-world (dialyzer and scalaris).
Extendable to test your own application.
Who Is Doing What?

Core

Scheduler

Processes

Core

Scheduler

Processes

© 1999-2012 Erlang Solutions Ltd.
Who Is Doing What?

When is this process running?
Who Is Doing What?

- When is this process running?
- Who started this guy?
Percept: Erlang Concurrency Profiling Tool, utilizes trace informations and profiler events to form a picture of the processes's and ports runnability.

Percept2 is an extension of Percept (part of the OTP release).

Extensions: # of schedulers active, active functions, process migration, message passing stats, inter-node communication
Memory Alloc Previously

One central memory allocator for all schedulers on the same machine
One central memory allocator for all schedulers on the same machine
Upgrading Blocks

Current Code

Next Code

Load code  All schedulers blocked
Upgrade Without Blocking

Current Code

Next Code

Last Code

Load code

Each scheduler does it when needed

Coming in R16!
Erlang connects all nodes fully.
So you get a lot of connections.
A fully connected system might not be right for every problem. `s_groups` allows you to create clusters of nodes. Nodes inside a cluster are fully connected. Connections between clusters can be arbitrary.
Managing Erlang Systems

Basic Erlang has the ability to go in and monitor what is going on in any node you can attach yourself to. But no tool exists to manage a big number of nodes in a coherent fashion. This is no different from any other language/technology!
Managing Erlang Systems

- Provision machines

Basic Erlang has the ability to go in and monitor what is going on in any node you can attach yourself to. But no tool exists to manage a big number of nodes in a coherent fashion. This is no different from any other language/technology!
Managing Erlang Systems

- Provision machines
- Deploy Erlang application

Basic Erlang has the ability to go in and monitor what is going on in any node you can attach yourself to.
But no tool exists to manage a big number of nodes in a coherent fashion.
This is no different from any other language/technology!
Managing Erlang Systems

• Provision machines
• Deploy Erlang application
• Attach to node

Basic Erlang has the ability to go in and monitor what is going on in any node you can attach yourself to. But no tool exists to manage a big number of nodes in a coherent fashion. This is no different from any other language/technology!
Managing Erlang Systems

• Provision machines
• Deploy Erlang application
• Attach to node
• Dig out metrics

Basic Erlang has the ability to go in and monitor what is going on in any node you can attach yourself to.
But no tool exists to manage a big number of nodes in a coherent fashion.
This is no different from any other language/technology!
Managing Erlang Systems

- Provision machines
- Deploy Erlang application
- Attach to node
- Dig out metrics

Memory usage
CPU load
Process hierarchy

Basic Erlang has the ability to go in and monitor what is going on in any node you can attach yourself to. But no tool exists to manage a big number of nodes in a coherent fashion. This is no different from any other language/technology!
Erlang Solutions are building a tool as part of the RELEASE project to manage and operate big Erlang systems.

CCL = Cloud Computing Lace or Cloud Cuckoo Land depending on your mood.
Erlang And Parallelism
Erlang And Parallelism

- Created for
Erlang And Parallelism

• Created for
  - explicit concurrency
Erlang And Parallelism

• Created for
  - explicit concurrency
  - fault tolerance
Erlang And Parallelism

• Created for
  - explicit concurrency
  - fault toler ance
  - highly concurrent systems
Erlang And Parallelism

- Created for
  - explicit concurrency
  - fault tolerance
  - highly concurrent systems

- No direct support for
Erlang And Parallelism

- Created for
  - explicit concurrency
  - fault tolerance
  - highly concurrent systems
- No direct support for
  - matrix multiplication
Erlang And Parallelism

- Created for
  - explicit concurrency
  - fault tolerance
  - highly concurrent systems
- No direct support for
  - matrix multiplication
  - ray tracing
Erlang And Parallelism

• Created for
  - explicit concurrency
  - fault tolerance
  - highly concurrent systems

• No direct support for
  - matrix multiplication
  - ray tracing
  - coarse grained parallel problems
Intensional Parallelism

We are taking the good things from what has been learnt in the Haskell & Data-flow language communities and building a DSL which helps us leverage these types of parallel optimisations.
Intensional Parallelism

- Lucid like: demand-driven data computation
- Find short comings in the Erlang VM
- Variables are infinite streams of values

We are taking the good things from what has been learnt in the Haskell & Data-flow language communities and building a DSL which helps us leverage these types of parallel optimisations
Intensional Parallelism

- Lucid like: demand-driven data computation
- Find short comings in the Erlang VM
- Variables are infinite streams of values

```erlang
running_avg
  where
    sum = first(input) fby sum + next(input);
    n = 1 fby n + 1;
    running_avg = sum / n;
end;
```

We are taking the good things from what has been learnt in the Haskell & Data-flow language communities and building a DSL which helps us leverage these types of parallel optimisations
Intensional Parallelism

- Lucid like: demand-driven data computation
- Find short comings in the Erlang VM
- Variables are infinite streams of values

```
running_avg
where
  sum = first(input) fby sum + next(input);
  n = 1 fby n + 1;
  running_avg = sum / n;
end;
```

We are taking the good things from what has been learnt in the Haskell & Data-flow language communities and building a DSL which helps us leverage these types of parallel optimisations"
Going Forward
Going Forward

- Consider Erlang when the problem fits
Going Forward

- Consider Erlang when the problem fits
- More focus on right tool for the job
Going Forward

• Consider Erlang when the problem fits
• More focus on right tool for the job