Robert Virding

Principle Language Expert
Erlang Solutions Ltd.

Erlang Solutions Ltd.

Why Erlang is that it is (... and what is it?)

Overview

- **A bit of history**
- A bit of philosophy
- A few examples

Ancient history

- All started in Ericsson Computer Science Lab
- “Everybody” wrote POTS programs to make phones ring on our MD110 in lab
- 1986: First reference to "Erlang" in paper at Logic conference describing writing telecom apps in concurrent logic
- Joe started programming telephony in Smalltalk based on communicating processes with ideas from CSP
  - and then started using Prolog

Early history

- Mike and I join the team
- First Erlang implementations of Erlang in Prolog
- We worked out suitable concurrency and error detection/handling models
  - Lots of discussions about this
- Erlang “wanders over” from Prolog to a functional language
  - Unwanted properties of Prolog
    - backtracking and logical variables

Middle ages

- 1990: ISS and “The Movie”
  - Erlang first presented to the world
- Need more speed for potential product
- First Erlang VM, the JAM, developed
  - Could now implement dynamic code loading
- Erlang more or less now complete as to basics

Some reflections

- We thought a lot about the problem
- Basic “specification” for Erlang system were taken from AXE10 and PLEX
  - For example need for, and type of, error handling
  - Safe language
  - BUT use conventional hardware and OS
- Very few initial goals as to details of Erlang
  - It just "became" a functional language
  - Concurrency and error handling more natural as part of the language

Overview

- A bit of history
- **A bit of philosophy**
- A few examples

Basic principles/requirements

- Lightweight, massive concurrency
  - Asynchronous communication
- Process isolation
- Error handling
- Continuous evolution of the system
  - Dynamic code updating
- Soft real-time
- Distribution

Secondary principles/requirements

- Simple high-level language
- "Safe" language
- Provide tools for building systems, not solutions
  - Too limited
  - (and we usually got them wrong)

Overview

- A bit of history
- A bit of philosophy
- **A few examples**

How Erlang does it

Sequential Language

- Simple functional language
  - With a “different” syntax
- It is safe!
  - For example no pointer errors
- It is reasonably high-level
  - At least then it was
- Dynamically typed

Sequential Language

- Typical features of functional languages
  - Immutable data
  - Immutable variables
  - Extensive use of pattern matching
  - Recursion rules!

Concurrency

- Light-weight “green” processes
  - Millions of Erlang processes possible on one machine
    - and running in a product
- Processes are used for everything
  - Concurrency
  - Managing state
- Processes are isolated!
- No global data!

Concurrency: message passing

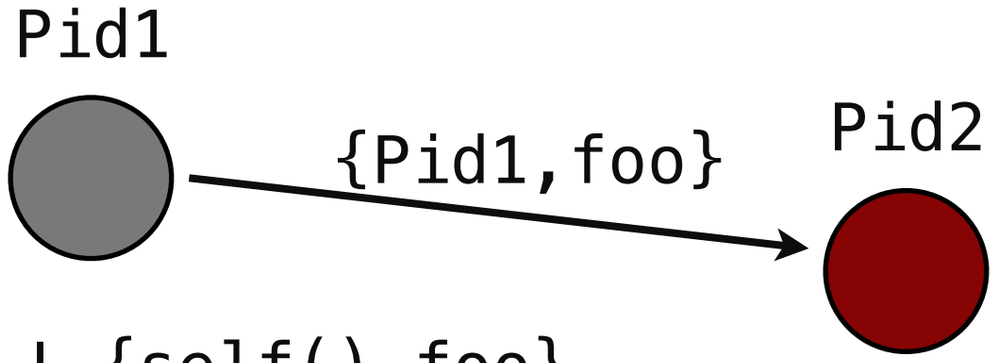
- Only provide basic primitives
- Very cheap asynchronous message passing
  - Send a message to a process
  - Selective receive
    - Limits combinatorial explosion in non-deterministic systems
- More complex operations built using send/receive
  - Synchronous messages built from 2 sends
  - Error handling complicates matters

Concurrency: message passing


```
receive
  start -> ...
  stop  -> ...
  {Pid,foo} ->
    ...
end
```

- Messages are sent using the **Pid ! Msg** expression
- Received messages are stored in the process's mailbox
- Messages are received using the **receive … end** expression
- Messages can be matched and selectively retrieved
- Mailboxes are scanned sequentially.

Concurrency: message passing

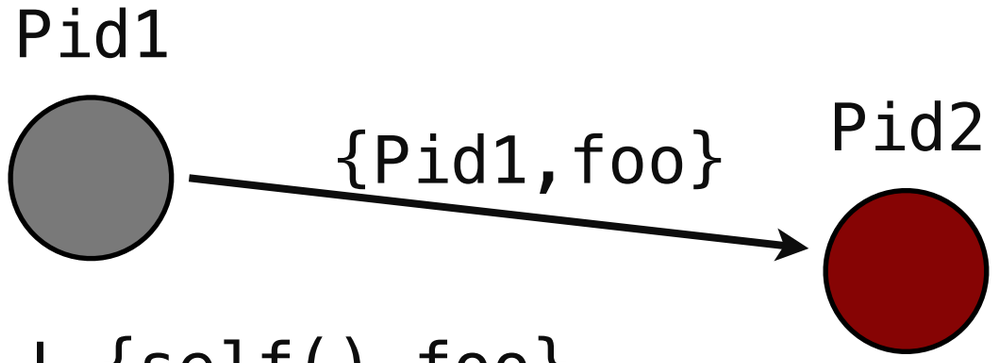



```
Pid2 ! {self(),foo}
```

```
receive
  start -> ...
  stop  -> ...
  {Pid,foo} ->
    ...
end
```

- Messages are sent using the **Pid ! Msg** expression
- Received messages are stored in the process's mailbox
- Messages are received using the **receive … end** expression
- Messages can be matched and selectively retrieved
- Mailboxes are scanned sequentially.

Concurrency: message passing




```
Pid2 ! {self(),foo}
```

```
receive
  start -> ...
  stop  -> ...
  {Pid,foo} ->
    ...
end
```

- Messages are sent using the **Pid ! Msg** expression
- Received messages are stored in the process's mailbox
- Messages are received using the **receive … end** expression
- Messages can be matched and selectively retrieved
- Mailboxes are scanned sequentially.

Error handling

ERRORS WILL ALWAYS OCCUR!

Error handling

The system must never go down!

Parts may crash and burn

BUT

The system must never go down!

Error handling

System must be able to

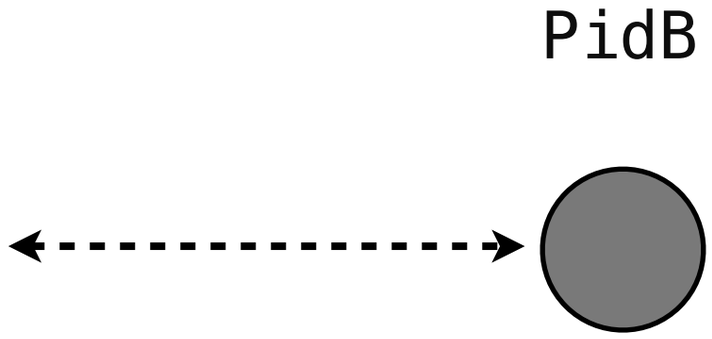
- Detect
- Contain
- Handle
- Recover from

errors

Error handling

- **Links** between processes
- **Exit Signals** are sent along links when processes terminate abnormally
- The process receiving the signal will exit
- Then propagate a new signal to the processes to which it is linked

Error handling



- **Links** between processes
- **Exit Signals** are sent along links when processes terminate abnormally
- The process receiving the signal will exit
- Then propagate a new signal to the processes to which it is linked

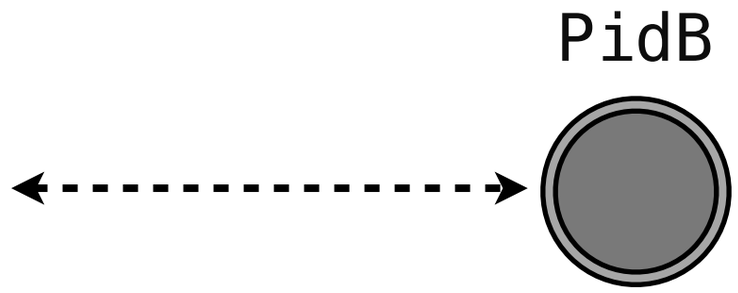
Error handling

- **Links** between processes
- **Exit Signals** are sent along links when processes terminate abnormally
- The process receiving the signal will exit
- Then propagate a new signal to the processes to which it is linked

Error handling

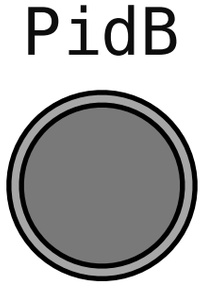
- Processes can **trap** exit signals.
- Exit signals will be converted to messages
- They are saved in the process mailbox
- If an exit signal is trapped, it does not propagate further

Error handling



- Processes can **trap** exit signals.
- Exit signals will be converted to messages
- They are saved in the process mailbox
- If an exit signal is trapped, it does not propagate further

Error handling



- Processes can **trap** exit signals.
- Exit signals will be converted to messages
- They are saved in the process mailbox
- If an exit signal is trapped, it does not propagate further

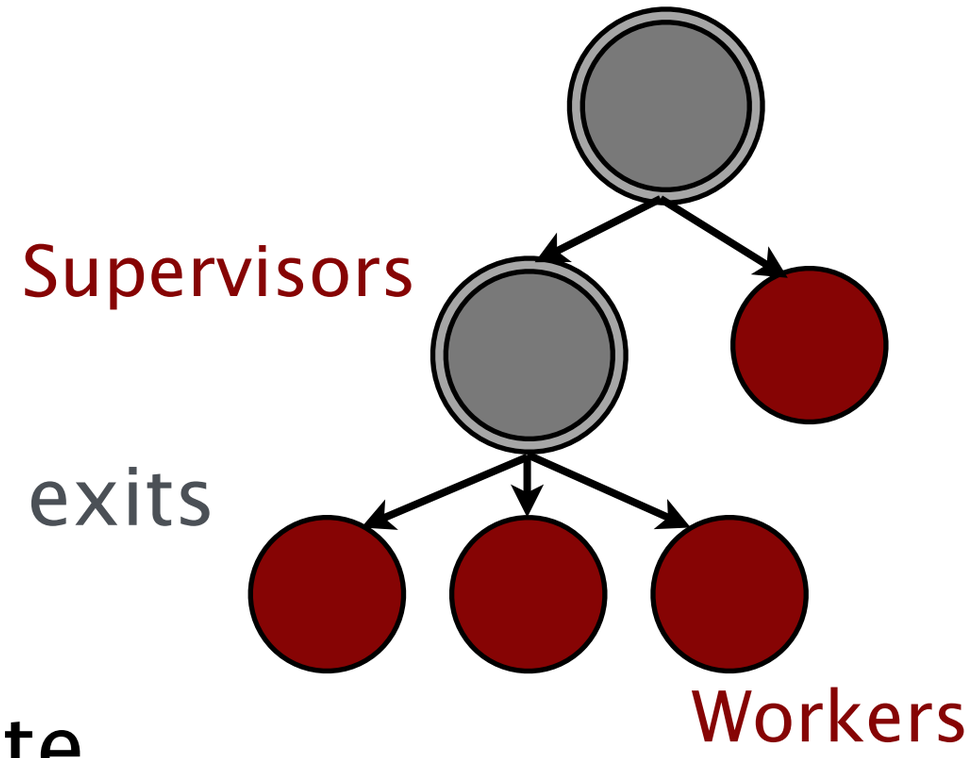
Robust systems

How do you build robust systems?

- You need to ensure
  - Necessary functionality always available
  - System cleans up when things go wrong
- Must have at least two machines!
  - Need distribution

Robust systems: Supervision trees

- Supervisors will start child processes
  - Workers
  - Supervisors
- Supervisors will monitor their children
  - Through links and trapping exits
- Supervisors can restart the children when they terminate



Robust systems: Monitor processes

- Servers monitoring clients
  - Clean-up after then if they crash
- Processes monitoring co-workers
- Groups of co-workers dying together

Code handling

- Module is the unit of all code handling
  - No inter-module dependencies
    - Causes problems with static typing
- Have two versions of each module
  - Old and current
  - Allows controlled take-over
- Well defined behaviour with respect to code
  - You **KNOW** what happens when you call a function
  - You **KNOW** what happens when you load a module