Why did we create Erlang?

Mike Williams
Ericsson AB
Stockholm
Sweden

mike@erix.ericsson.se

Maybe it didn’t happen exactly this way, but this is the way I think it should have happened.

☺

Problem Domain - Highly concurrent and distributed systems

- Thousands of simultaneous transactions
  - Light weight transactions
  - Greatest CPU load is implementing concurrency and communication not computation
- Many computers
  - different types (Bigendians, Littleendians, Intel, Sparc, PowerPC etc)
  - share nothing (no shared memory, different communication mechanisms (Ethernet, ATM, Proprietary))
- Many OS's
  - Solaris, VxWorks, Windows, pSOS, Linux, etc

Problem Domain - No down time

- Not allowed to have any planned or unplanned downtime
  - Acceptance criterion: five nines = 99.999% uptime or 5 minutes down time per year
- Recovery from software errors
  - Large systems will have software bugs
- Recovery from hardware failure
  - Network failure, processor failure, I/O failure
- Enable adding / deleting computers and other hardware at run time
- Update code in running systems

Problem Domain - Ease of programming

- Highly "expressive" programming language
- Easy portability between processor architectures
- Large scale development (tens or even hundreds of programmers)
- Incremental and exploratory programming
- Debugging and tracing - even in systems running at customer sites
- Easy to fix bugs (patches) and upgrade at all phases of design – even in systems running at customer sites

Solution Domain - Concurrency

- No existing industry quality OS or language offers light weight enough threads / processes
- Processes must be independent
  - No shared resources
  - One process must not be able to destroy another process
  - Reduce event/state matrix by selective message reception

Solution Domain – Concurrency & Distribution

- As we didn't want to modify or create and new OS, implementation of light weight, processes needed to be done in "middleware", I.e. on top of the OS.
- Making processes independent requires either control of the MMU or a language without pointers (or with safe pointers)
- Reducing the event/state matrix makes the signal / state model undesirable.
  - The signal state model requires a thread only suspending at the top level, not in a function/subroutine. This makes proper RPC's impossible.

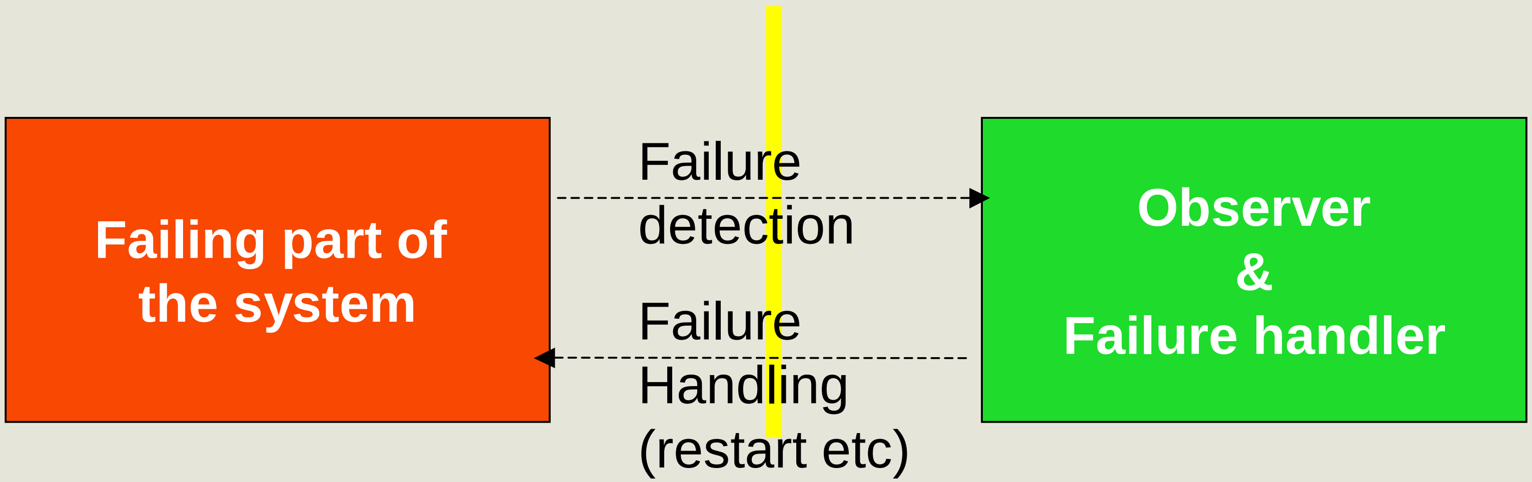
Solution Domain – Concurrency & Distribution: **Design decisions**

- Implement concurrency in a virtual machine on top of operating system.
- Use a language without explicit pointers.
- Use copying message passing as only interprocess communication mechanism.
- Implement selective message reception.
- Make communication between processes on different machines identical to communication between processes on same machine.
  - Type information retained at runtime enables automatic conversion of Erlang terms to an external format.

Solution Domain - No down time

- Principle for error detection: *It is unsafe to allow the failing part of the system to detect and correct failures itself*

No ability to crash
The observer



Solution Domain - No down time

- A software error in one process is best detected in another process
- Failure of one processor is best detected in another processor
- Frequently we want to be able to abort all the processes in a transaction if one of them fails for some reason
-

Solution Domain - No down time

Design Decisions:

- Create a concept of a “link” between processes. If a process fails, a special message (a signal) is sent to all the processes to which it has links.
- Default action of a process receiving a signal indicating failure of a process is to “die” and re-send on the signal to all linked processes.
- By setting a special flag, (trap_exit) a processor can override the default behaviour and receive the signal as an ordinary message.
- Links are bi-directional – (maybe a design mistake?)

ERICSSON

Solution Domain - No down time
Design Decisions:

- Two cases:
  - Server with a lot of clients. If a client fails server needs to take corrective action.
  - A lot of processes in a transaction – if one fails, all should fail.
- Link and Signal mechanism works across processor boundaries.
  - If a processor fails, signals will be sent to all processes which have links to processes in the failing processor.
- Error handling philosophy: “Let it crash” and let other processes clear up the mess.

Solution Domain - No down time

- Common design paradigm:
  - Let all active transactions be represented by groups of linked processes
  - Store inactive (steady state) transactions in replicated robust database (Mnesia)
  - Let resources needed by transactions be allocated by resource allocator processes which trap_exits and free up resources from failing transactions
  - Supervisor processes which trap_exits restart failing application on suitable processors. Data for these applications is the configuration data needed and the data for transactions in a steady state. (same mechanism used for replacing processors).

Solution Domain - No down time

Design Decisions:

- Design the virtual machine so new code can be loaded and processes can migrate to the new code.
- Ability to detect processes running old code.
- Design the standard design patterns (part of OTP) so that they can:
  - convert data to a new format if needed (e.g. when loading new code)
  - “hand over” to other processes in other processors when ordered to do so
- Application software needs to be aware of possible software updating and failure recovery, but with Erlang/OTP support the impact is minimised.

Problem Domain - Ease of programming (reminder)

- Highly "expressive" programming language
- Easy portability between processor architectures
- Large scale development (tens or even hundreds of programmers)
- Incremental and exploratory programming
- Debugging and tracing - even in systems running at customer sites
- Easy to fix bugs (patches) and upgrade at all phases of design – even in systems running at customer sites

Problem Domain - Ease of programming

Design Decisions:

- Use high level functional language with automatic memory handling and garbage collection
- Use execution of intermediate code by virtual machine to obtain easy portability between processor architectures
- Simple non/hierarchical module system
- Erlang shell allows testing of functions directly without any special test programs
- Virtual machine support for debugging and fault tracing
- Dynamic code replacement also very useful while developing / testing software

Comments

- We have frightened off a lot of people by using:
  - A functional language
  - A non O-O language
  - A non “C” like syntax
  - Recursion, single assignment etc
  - A virtual machine
- I.e. we have diverged a long way from industry mainstream. We are changing **very many** parameters at the same time.
  - Attitude changes in “mainstream” are possible
    - Remember what people said about Garbage Collection before Java?
    - Remember what people said about virtual machines before Java (UCSD Pascal ☺)

Comments

- The existing Armstong et al book is out of date!
  - The only “complete” book about Erlang and OTP which is available today is in French!
  - I have written a reasonably complete tutorial about Erlang
  - A complete Erlang Spec is available in the latest distribution
- **The use of Erlang is accelerating, the critical mass is about to be reached!**