Erlang: Practical Functional Programming

Elixir Taiwan

May 16, 2016

Jake Morrison <jake@cogini.com>

Origins

Designed by Ericsson as a high level language to be used to build highly reliable telecom systems.

Erlang is functional for practical, not academic reasons.

Telephone switches with 10K simultaneous calls / ATM networking

Failure handling one customer should not affect other customers

Degrade gracefully when overloaded

Hardware failure should not stop the system

- Redundant hardware, two separate CPU boards
- Share call state between multiple servers

There is never a good time to do a software update

- Hot code updates

Monitoring, management and production debugging

Today

- "Internet scale" applications such as WhatsApp
- Financial services, e.g. high frequency trading
- Embedded systems

What is Erlang?

Languages

Erlang

Elixir

Lisp Flavoured Erlang

Prolog

Virtual Machine

- Similar model to Java
- Implements all the "hard stuff" needed to build scalable systems, e.g. networking, lightweight processes and message passing, so programmers can focus on the high level functionality of of the systems they are building.

OTP Framework

Building blocks for

- Network services
- Process supervision
- Finite state machines
- Event handling

Tools

- Monitoring
- Debugging

Philosophy

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.

Handling shared state

- If I don't have shared state, then I don't have to replicate it
  - Keep state in lightweight processes that don't talk with each other
- Separate state from processing
  - Resources
    - Concurrent processing
    - Limit to system capacity
  - Libraries
    - No state, pure functional calls
- Centralize handling of persistent state
  - Internal replicated in-memory database

Natural Concurrency

- One process for each truly independent activity
  - Not a web server that can handle 1M simultaneous requests
  - 1M web servers, each handling one request
- The world is asynchronous
  - Components communicate by sending messages to each other
  - Eventual consistency

Let it fail

It's ok if we can't handle a request, as long as we behave reasonably

Pattern matching

- If you see something wrong, log it and go on
- Invalid input
- Bugs = run time assertions

Overload

- Preserve resources to give good quality of service
- Pull based handling

Supervisors

- The solution is at a higher level than the problem

Logging

- Take action
- Get enough information to diagnose the problem

Production monitoring, maintenance and debugging

- Crash reports with enough information to replicate and fix the problem
- Full call stack
- Tracing of live systems
- Hot code updates

Next generation web apps

Requests are independent... unless they are not

- How do we efficiently manage shared state?
- Peer to peer communication

Next generation web apps

Real time web

- Web sockets
- Push messaging
- Mobile and web chat
- Mobile APIs
- Location based services

Next generation web apps

Single platform for everything

- Public web
- Back end admin
- Mobile APIs
- Gateway to 3rd party services
  - Payment gateway
  - Telephony systems

Next generation web apps

Scalability (= cluster)

Efficiency

Reliability

Manageability

The contenders

- Traditional web: PHP, Ruby on Rails
  - Easy to program
  - Lots of libraries
  - Poor scalability / concurrency
- New generation: Node.js, Golang
  - Better process model
  - No management tools
  - Low level
- Erlang / Elixir
  - Designed to scale
  - Easy development
  - Slightly mind bending

Questions?