Introduction to Erlang

Paulo Sérgio Almeida

Grupo de Sistemas Distribuídos
Departamento de Informática
Universidade do Minho

Erlang

- Created by Ericsson; after mathematician A. K. Erlang
- Dynamically typed;
- Purely functional and strict kernel;
- Ultra-lightweight processes managed by the Erlang VM;
- Concurrency by asynchronous message passing (actor model);
- Persistent and transactional database for global state;
- emphasizes reliability; always-on systems;
- emphasizes previsibility; soft-realtime systems.

Erlang: functional languages can be used in the real world

- Even though simple and elegant, Erlang is not a toy-language;
- One of the more successful functional languages:

AXD 301:	Ericsson ATM switch; 1.7 Mlocs Erlang; 99.999999999% reliability (31ms/year);
GPRS:	76% code in Erlang;
ejabberd:	Jabber/XMPP server, adopted by jabber.org; 250000 simultaneous users in a domain; 10000 users per host;
Yaws:	web server; worked well with 80000 connections in a test where Apache died at 4000;
RabbitMQ:	implementation of messaging standard AMQP (Advanced Message Queuing Protocol);
Riak	highly scalable, fault-tolerant distributed database;
WhatsApp	messaging, voice, video, . . .

Erlang: functional kernel

- Functional language:
    - higher-order functions,
    - pattern matching,
    - no update assignment (single assignment),
    - main data structures : tuples, lists and maps;
- Dynamically typed language:
    - functional “scripting language”,
    - good for rapid prototyping,
    - static verification tools; e.g. dialyzer;

Erlang Processes

- An Erlang process is managed by the Erlang virtual machine;
- Very little creation cost;
- Easy to have hundreds of thousand processes per host;
- Very little context switch cost (one or two orders of magnitude faster than OS threads);
- Some OS threads are used by the Erlang VM to exploit parallelism or for I/O.
- Each process maintains local state and communicates by message passing with other processes.

Message passing

- Erlang follows the actor model;
- Each process has a mailbox – message queue;
- Messages are sent specifying the destination process – no other concept, like channel, is used.
- Sending is asynchronous/non-blocking; it does not give an error even if the destination process has already terminated;
- Messages are kept in the mailbox by arrival order;
- Messages are removed from the mailbox by a blocking receive using pattern matching.

Memory management

- Functional language with garbage collection (GC);
- Processes do not share memory;
- In a functional language with immutable data structures, absence of sharing between processes could be only conceptual . . .
- . . . but in Erlang there is really no physical sharing:
  - each process has its own stack and heap;
  - messages involve physical copy of data structures;
- Why this choice:
  - messages are typically short;
  - heap per process allows GC per process;
  - avoids large pauses caused by the GC on a global heap;
  - results in greater locality;
  - a system may be tuned so that many processes are created without incurring in a single GC during their lifetime.
- There is a shared-heap option, normally not used.

Erlang processes as units of encapsulation

- State encapsulation in imperative languages or objects is one of the problems not yet solved;

  “The big lie of object-oriented programming is that objects provide encapsulation” *John Hogg*

- Mixing mutable state with functional style (e.g., in Python) brings curious surprises (bugs);
- In Erlang, a process truly encapsulates state;
- It can be seen as an object in the original sense (Alan Kay):
  - interacts by message passing;
  - encapsulates local state;

Previsibility and fast response times

- Erlang emphasizes fast response to external events;
- Used in soft-realtime systems;
- Reflected in the language /system:
  - does not use concepts that make performance estimatation difficult, like lazy evaluation;
  - ultra-lightweight processes with fast context switch;
  - per process GC to avoid long pauses;
  - in-memory database (Mnesia);

Systemic Reliability

- Large systems contain bugs;
- System should work, even having some incorrect part;
- Erlang philosophy – avoid defensive programming;
  - functions do not protect against invalid arguments;
  - do not mix error verifying code with normal code;
  - program for the normal case;
- Erlang philosophy – fail-crash:
  - if function cannot perform as supposed, do not try to return error but crash process;
  - example: if guards in an “if” or patterns in a “case” do not cover all cases, process crashes;
- Processes play an essencial role in error isolation and containment:
  - processes can be linked – propagating or containing failures;
  - supervisor processes can relaunch failed processes;
  - system organized as a supervision tree;

Non-stop operation and hot code swapping

- Erlang in systems that cannot be switched off or rebooted;
- Over time, new versions of some code need to be deployed;
    - bug crashes a process; error report is generated; bug is corrected;
    - new features are added to the application;
- Erlang can load new version of module in running application;
- Two ways to invoke a function:
    - intra-module; uses the same module version;
    - with module prefix; uses the more recent version;
- VM supports two versions of a module coexisting;
- Different processes can be running different versions.

Global state – ets and mnesia

- Last layer allows state that is global to all processes;
- Two modules: ets and mnesia;
- ets – Erlang Term Storage:
  - Allows tables with many entries; preferably small entries;
  - In memory, using hash tables or trees;
  - Belong to creating process; removed when it terminates;
  - May be created as private, protected, or public;
  - Kept in specific memory area, with no GC;
  - Terms are copied between processes and tables;

Global state – mnesia

“Mnesia is a distributed DataBase Management System (DBMS), appropriate for telecommunications applications and other Erlang applications which require continuous operation and exhibit soft real-time properties.”

- Written in Erlang, using ets;
- Allows storing generic Erlang terms, without impedance mismatch;
- Query Language based on list comprehensions;
- Persistent; but normally an in-memory DB;
- Extremely fast reading access;
- Nested transactions using plain Erlang functions;
- Replication and fragmentation of tables;
- Control of isolation level;
- Allows reconfiguration without stopping the system.

No impedance mismatch – back to simplicity

- Current applications tend to be written using many components that must be combined;
- Impedance mismatch (e.g., object-relational) hinders development;
- Erlang + Mnesia + Yaws allows writing an application in a simple and integrated way;

"Erlang + Yaws is so much easier to setup and install than Ruby + Gems + Rails + FastCGI + Lighttpd + MySQL + MySQL bindings + every-other-package-in-the-universe. I've been slaving away at these installation instructions for OS X for what feels like an eternity, and my patience is running very low." *Yariv Sadan*

Runtime and tools

Dialyzer:	static analysis tool to detect errors (types, unreachable code, unnecessary tests); performs data flow analysis, to determine the disjoint union of prime types;
Application Monitor:	graphic tool to monitor the supervision tree of an application;
Event Tracer:	to collect and visualize information using the Erlang tracing mechanism;
Process Manager:	to inspect processes (local or remote) and do tracing;
Table Visualizer:	to inspect or modify ets or mnesia tables;