Programming Distributed Systems

13 Troubleshooting Erlang

Annette Bieniusa

AG Softech
FB Informatik
TU Kaiserslautern

Summer Term 2019
Tricks and Tools for Software Development in Erlang
Erlang Software

- Composition of OTP applications
- Each application consists of top-level supervisor and dependent (child) processes
- Typical code organization

_build/
doc/
src/
test/
README.md
LICENSE
rebar.config
rebar.lock
Build tool: rebar3

- Generates templates for code repos
- Unifies different tools

help
- Display a list of tasks or help for a given task or subtask.

clean
- Remove compiled beam files from apps.
compile
- Compile apps .app.src and .erl files.
dialyzer
- Run the Dialyzer analyzer on the project.
do
- Higher order provider for running multiple tasks in a sequence.
edoc
- Generate documentation using edoc.
eunit
- Run EUnit Tests.
cover
- Perform coverage analysis.
shell
- Run shell with project apps and deps in path.
Extract from rebar.config for Minidote

```erlang
{deps, [  
% Replicated datatype library  
{antidote_crdt, {git, "https://github.com/AntidoteDB/antidote_crdt", {tag, "v0.1.2"}}},  
% Protocol buffer decoding/encoding  
{antidote_pb_codec, {git, "https://github.com/AntidoteDB/antidote_pb_codec", {tag, "v0.0.5"}}},  
% ranch socket acceptor pool for managing protocol buffer sockets  
{ranch, "1.5.0"},  
% lager for logging:  
{lager, "3.7.0"},  
{meck, "0.8.13"}  
]}. 

{profiles, [  
{test, [  
{deps, [  
% Antidote protocol buffer client for testing:  
{antidote_pb, {git, "https://github.com/AntidoteDB/antidote-erlang-client", {tag, "v0.2.4"}}},  
% meck mocking framework

```
Dependencies

- Open-source packages
  - Package manager **Hex**
  - Git repositories via URL (and optionally release version or commit hash for reproducability)
- `rebar3` pulls all dependencies recursively
- File `rebar.lock` contains information on exact version that is used
- Sometimes need to specify special build options, code transformations as compile time, etc.
How to Prevent Things Going Wrong . . .
Type checking: Dialyzer

- Dynamic checker based on *success typing*
- **Will not prove the absence of (type) errors, only best effort**
- Dialyzer will only report errors that will lead to a crash (when/if that code is executed)

```erlang
-module(dialyzer_example1).
-export([f/1]).

f(Y) ->
    X = case Y of
        1 -> ok;
        2 -> 3.5
    end,
    convert(X).

convert(X) when is_atom(X) -> atom_to_list(X).
```
Type specifications

- **Singleton types** (e.g. a given integer, empty list [], a given atom)
- **Built-in types** (e.g. any(), pid(), atom(), binary(), integer(), non_neg_integer(), pos_integer(), fun(),
  fun(Type1, Type2, ..., TypeN) -> Type, [Type()],
  {Type1, Type2, ..., TypeN})
- **Union types**, e.g.
  - `boolean()` is defined as `true` | `false`
  - `byte()` is `0` | `...` | `255`
  - `number()` is `integer()` | `float()`)
User-defined types

-type TypeName() :: TypeDefinition.

-type tree() :: 'leaf' | {'node', any(), tree(), tree()}.  
-type tree() :: 'leaf' | {'node', Val::any(), Left::tree(), Right ::tree()}.  

-record(student, {name = "" :: string(), matrikel :: non_neg_integer()}).  
-type student() :: #student{}.
General advice on Typing

- **Write type specifications** and use dialyzer
- For type checking and for documentation purposes
- For examples, take a look at the Antidote CRDT library
- Fix all the errors that Dialyzer finds
- Don’t despair - ask for help!
Let it crash fail

Erlang in Anger, p. 1 by Fred Hebert

Most other programming languages:

“Something going wrong at run-time is something that needs to be prevented, and if it cannot be prevented, then it’s out of scope for whatever solution people have been thinking about.”
Let it crash fail

Erlang in Anger, p. 1 by Fred Hebert

Most other programming languages:

“Something going wrong at run-time is something that needs to be prevented, and if it cannot be prevented, then it’s out of scope for whatever solution people have been thinking about.”

Erlang:

“[...] failures will happen no matter what.[...] It is rarely practical nor even possible to get rid of all errors in a program or a system.”
Supervisors

- Most faults and errors are transient (e.g. network problems, timing for concurrent start)
- Simple retrying is a surprisingly successful strategy
- Starting of supervisor tree is synchronous to establish a correct, stable initial state
When Things Go Wrong...
Connecting to nodes

- Erlang allows to connect to running virtual machines for live diagnosis
- Local and remote (requires typically shared cookie)
- Can also be used to re-load, re-compile and hot-swap code in production

Steps

1. Start an Erlang shell via `erl`
2. Press `^G` to enter the Job Control Mode
3. Press `h` for a list of options
4. `r` for starting remote shell, `c` to connect to that shell
5. Quit remote shell with `^G q`
Example

silverbird:annettebieniusa$ erl
Erlang/OTP 22 [erts-10.4.2] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:1] [hipe] [dtrace]

Eshell V10.4.2  (abort with ^G)
1>
User switch command
  --> h
    c [nn]   - connect to job
    i [nn]   - interrupt job
    k [nn]   - kill job
    j        - list all jobs
    s [shell] - start local shell
    r [node [shell]] - start remote shell
    q        - quit erlang
    ? | h    - this message
  -->
Observing the Behavior at Runtime

- **Useful library:** Recon
- **Information on a specific process:** `process_info/2` or `recon:info/1`
- `recon:get_state/1` yields internal state of OTP process for given `pid` (process identifier)
- For OTP Processes, check `sys` module for detailed statistics, logging of all messages and state transitions, etc.
Understanding Crash Dumps

- File `erl_crash.dump` generated after crashes
- Check for Slogan at the beginning to get hint on reason
- Contains a lot of information
Memory Leaks

- Common sources:
  - Don’t use dynamic atoms (i.e. atom names generated at runtime) because they are entered in a global table and cached forever! Check for `erlang:binary_to_term/1` and `erlang:list_to_atom/1`
  - ETS tables are never garbage collected, must be explicitly deleted
  - Process leaks by starting a dynamic number of processes that are never killed and keep looping
Problem: Overloading

When nodes are running out of memory, look for the following things:

1. Log messages with `io:format`
   - Replace with calls to `lager` (or `logger` since Erlang 22)
2. Blocking operations (e.g. waiting on TCP sockets, messaging patterns prone to deadlock)
   - Message queues might fill up during blocked waiting
   - Move the waiting out of the critical paths into an asynchronous call
   - But beware of “call-back hell”
3. Unexpected messages (e.g. typos in message type atom)
   - Check that generic handler is in place that matches any pattern

Example for OTP `gen_server`:

```erlang
handle_call(_Request, _From, _State) ->
  erlang:error(not_implemented).
```
What if there are more client requests than the server can handle?

Example
What if there are more client requests than the server can handle?

Example

Strategies for dealing with backpressure:

- Add more resources and scale out
- Drop requests (often not acceptable)
- Store requests temporarily (for dealing with short bursts)
- Control the producer / clients and restrict number of requests
Further reading

- Erlang in Anger by Fred Hebert
- Learn you some Erlang for Great Good! by Fred Hebert