Agner Krarup Erlang (1878–1929)

Csaba Hoch (Erlang Solutions)

Concurrency in Erlang/OTP

(Image source: Wikipedia)

Overview

1. Introduction

2. Processes – language constructs

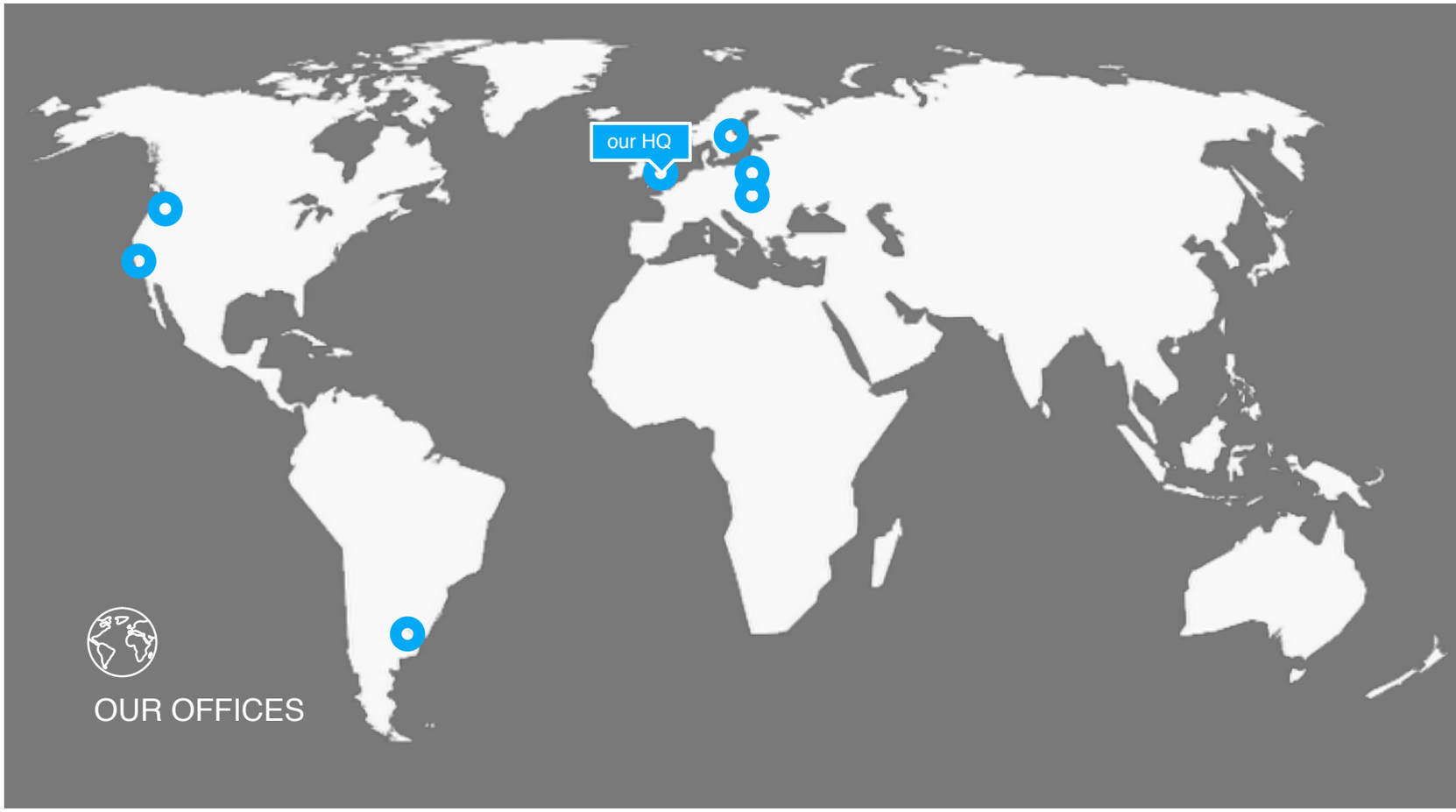
3. OTP (Open Telecom Platform) – libraries

About me

(Image source: Google Maps)

Erlang Solutions

- ~100 employees
- Consulting
- Development
- Research
- Support
- Training
- Organizing conferences



Erlang key features

- **Concurrent** (processes)
- **Multi-core** (an Erlang VM can use many cores)
- **Distributed** (Erlang VMs can easily communicate with each other)
- Makes it easier to build **scalable**, **fault-tolerant** systems

(Image source: Wikipedia)

Overview

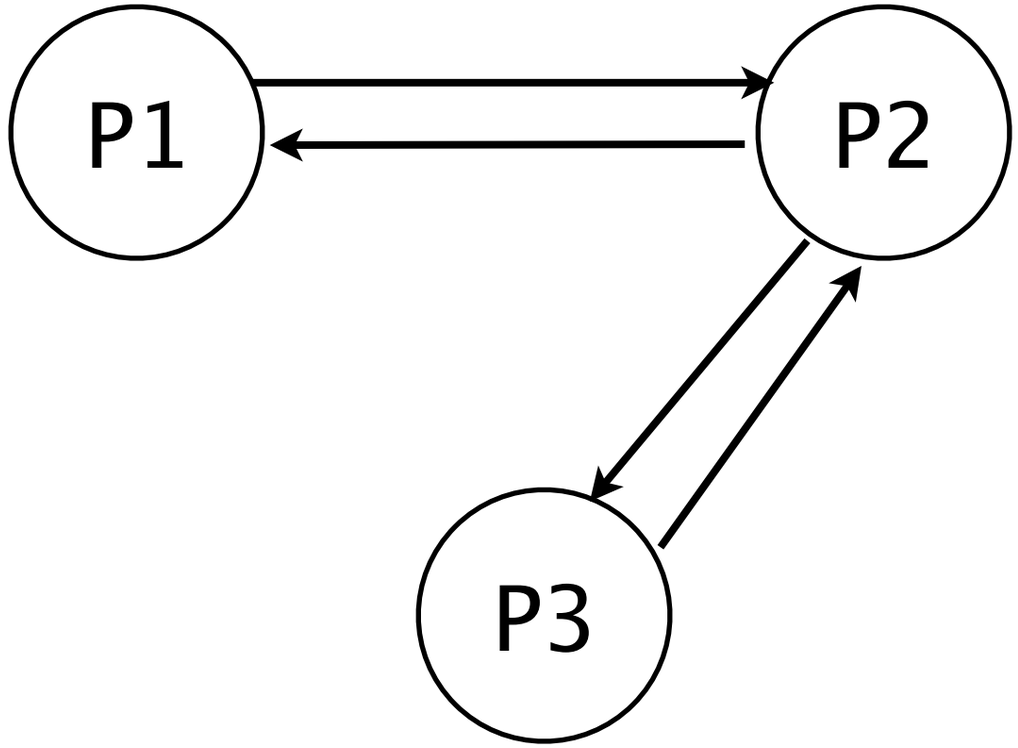
1. Introduction

2. Processes – language constructs

3. OTP (Open Telecom Platform) – libraries

The actor model

- Processes are **isolated**
- Processes communicate via **messages** (no shared memory!)



Creating and terminating processes

- **Start** a process:

```
spawn(fun loop/0) → Pid
```

- Each process has a **pid** ("process identifier"; e.g. <0.25.0>)

- A process **terminates** when...
  - there is no more code to execute ("normal")
  - an error happens and is not caught
  - exit(Pid, Reason) is called
    - exception: if trap_exit is 'true' and Reason is not 'kill' → only a message is sent

Messages

- Each process has a **mailbox** which stores the messages sent to the process
- **Sending** a message to a process:

```
Pid ! {get_id, MyPid}
```

- **Receiving** a message:

```
receive
    Message ->
        ...
end
```

Processes are cheap

RAM footprint per unit of concurrency (approx)

1.3KB	Haskell ThreadId + MVar (GHC 7.6.3, 64-bit)
2.6 KB	Erlang process (64-bit)
8.0 KB	Go goroutine
64.0 KB	Java thread stack (minimum)
64.0 KB	C pthread stack (minimum)
1 MB	Java thread stack (default)
8 MB	C pthread stack (default, 64-bit Mac OS X)

(Data&Diagram source: Bob Ippolito's presentation)

Implementation

- The Erlang VM has schedulers
  - Erlang VM = OS process
  - Scheduler = OS thread
  - Idea: one scheduler = one CPU core

Let's write a server process!

- Processes can live run forever using recursion
- Best practice: modules should provide interface functions
- Registered names:
  - Processes can register an atom as their name:

```
register(id_server, self())
```

  - Other processes can send messages to them using this name:

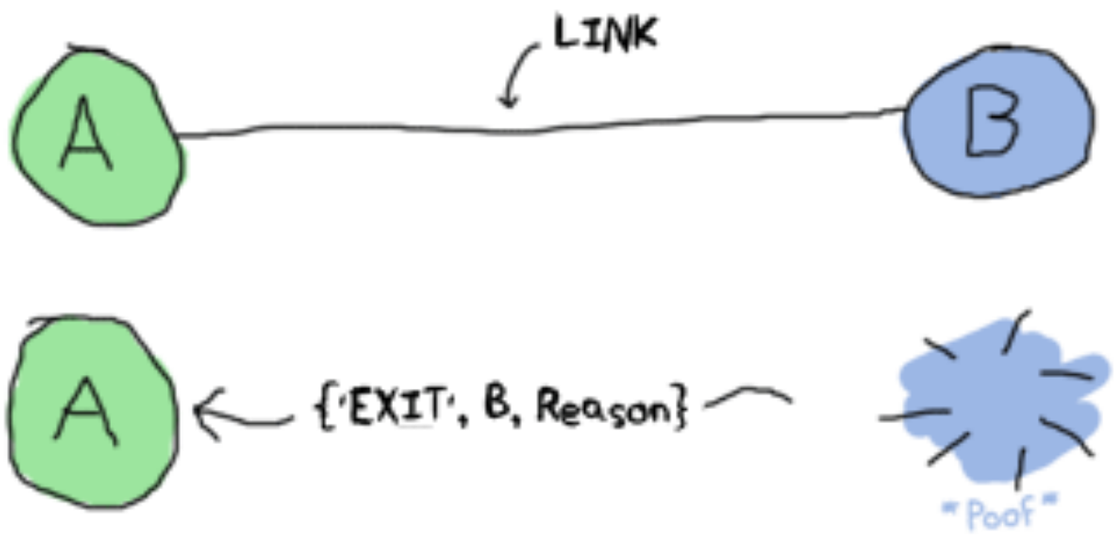
```
id_server ! {get_id, MyPid}
```

Process links

- Processes can be **linked**: if one crashes, the other one will be terminated too

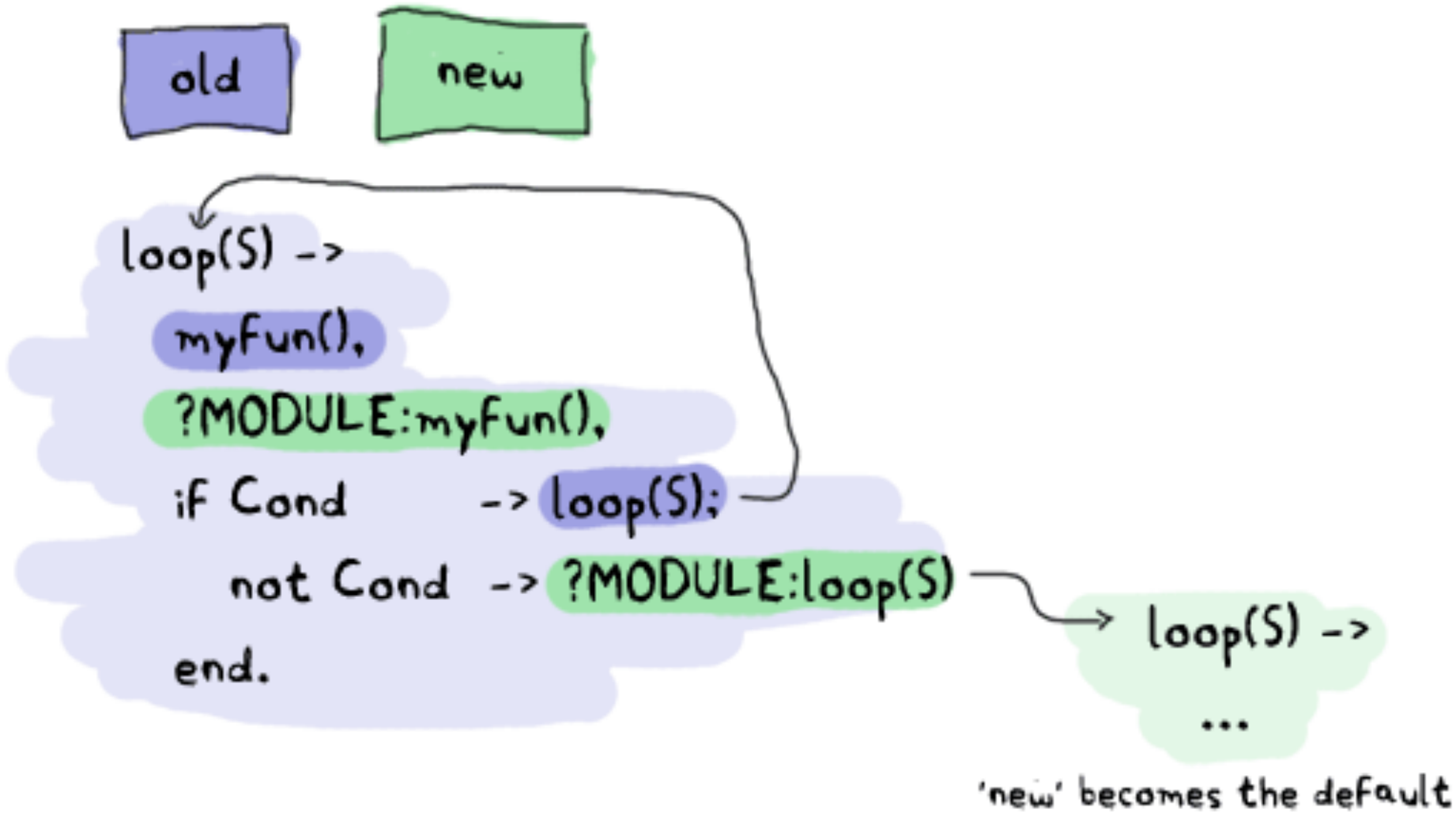
```
link(Pid)
spawn_link(fun loop/0) → Pid
```

- **trap_exit** is useful to stop the propagation



(Image source: Learn You Some Erlang)

Hot code loading



(Image source: Learn You Some Erlang)

Overview

1. Introduction

2. Processes – language constructs

3. OTP (Open Telecom Platform) – libraries

OTP – Open Telecom Platform

- Design principles
- Generic process templates (behaviours)
  - Server processes: gen_server, gen_fsm, gen_event
  - Supervisors
- Etc.

gen_server

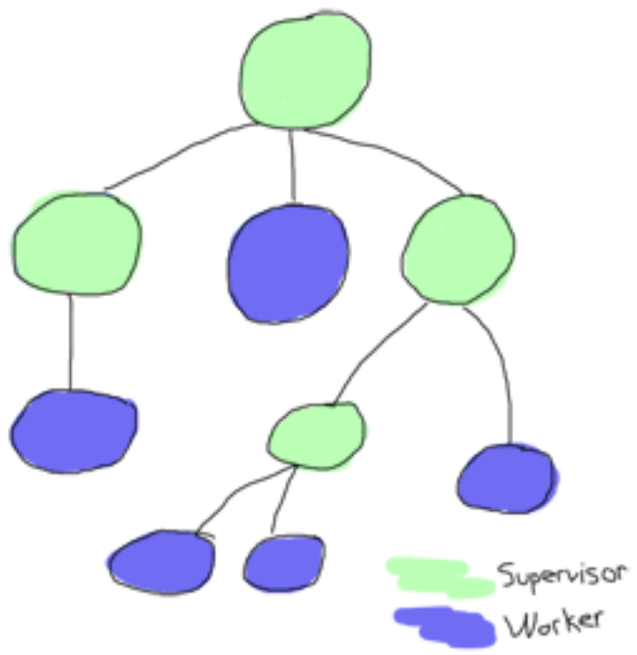
- Server processes have **common code**
- This common code was **extracted** into the gen_server module
- Using gen_server, the programmer implements only **callback functions**
  - These callbacks will be called by the gen_server module

Supervisors

- What should happen when a process crashes?
  - We should try to **restart** it (a few times)
  - Maybe we should also **restart related processes**
  - If it still crashes, let's try to **restart the component** that contains the process...

Supervisors

- Using supervisors, the programmer organizes the processes into a tree, and specifies the followings:
  - **In what order** should the processes be (re)started and stopped?
  - **How many times** should we try to **restart** the processes?
  - **Which processes** should we **restart** when restarting a process?



(Image source: Learn You Some Erlang)

Erlang/OTP – a few other things

- Distribution
- ETS: term storage
- Mnesia: distributed database
- OTP applications:
  - Components that can be started/stopped
  - They can be reused
- OTP releases:
  - Release = a system consisting of applications
  - One Erlang VM executes one release

Summary

- Processes
  - Actor model: messages
  - Start, termination
  - Registration, links
  - Hot code loading
- OTP
  - gen_server
  - Supervisors