Message passing for **actors** and **humans**

Hi

name - Peter Saxton

@internets - CrowdHailer

@works - paywithcurl.com

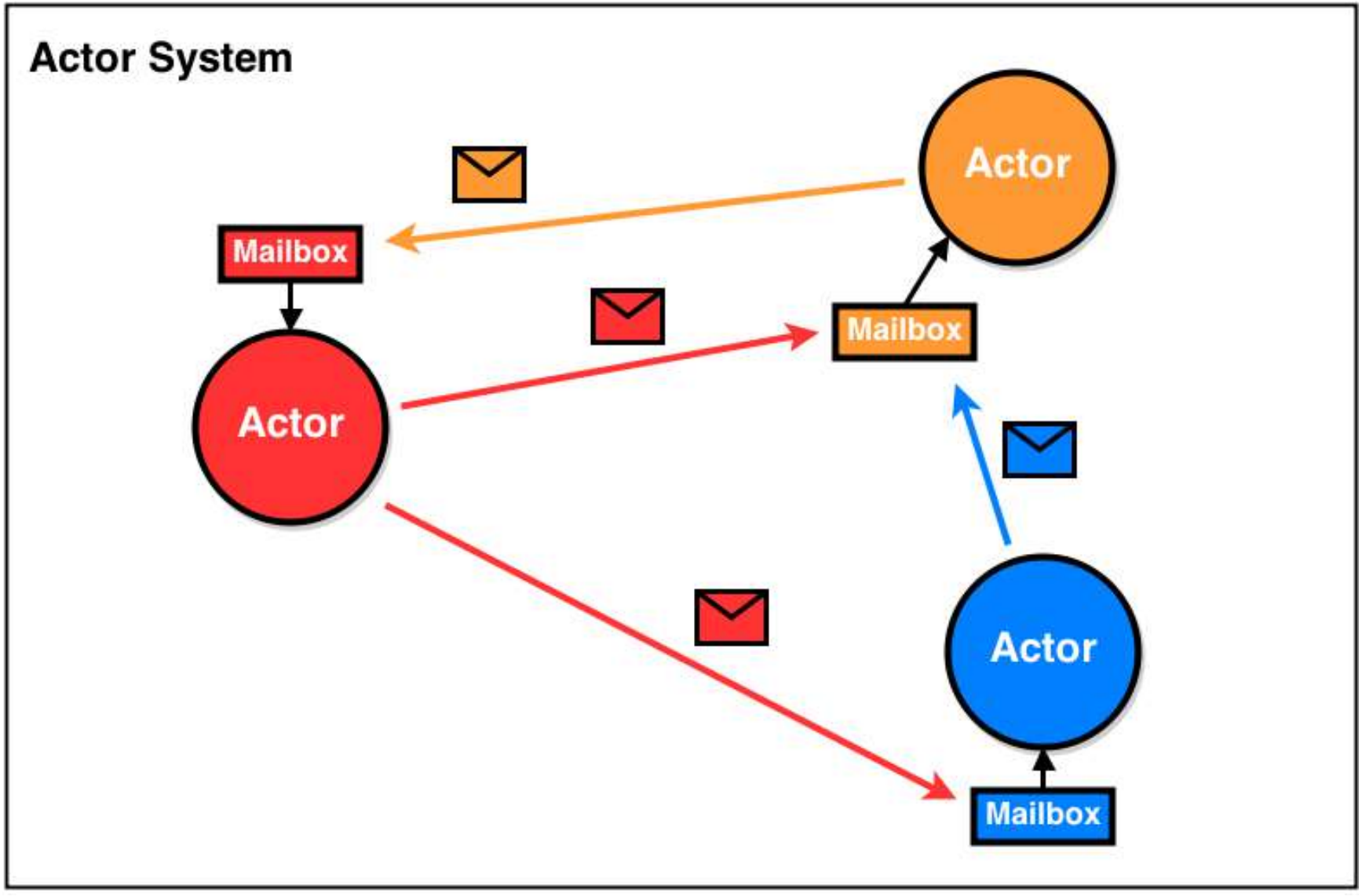
lets talk about,

The Actor model

Actors

- Universal primitive of concurrent computation
- Communicate via asynchronous message passing
- React to messages by making local decisions

First proposed by Carl Hewitt in 1973


Actor System
Actor
Mailbox
Mailbox
Actor
Actor
Mailbox

Universal primitive

- An actor has three essential elements:
  1. Processing
  2. Storage
  3. Communication
- Everything is an actor

Asynchronous message passing

- No delivery guarantees
- No order guarantees

Similar to original Object Oriented programming (OOP)

Local decisions

- No shared/global state
- In response to a message, an actor may:
    1. Create more actors
    2. Send messages to other Actors that it has addresses for
    3. Designate how the Actor is going to handle the next message it receives

In the wild

- Elixir
- erlang
- Akka (JVM)

Let's build our own

in JavaScript

The simplest actor

```
var state = 0

while (true) {
  const message = await mailbox.receive()
  state = state + 1
}
```

```
// somewhere else
mailbox.deliver(message)
```

Blocking mailbox

```
function Mailbox () {
  var messages = [], awaiting = undefined

  function receive () {
    return new Promise(function (resolve) {
      if (next = messages.shift()) {
        resolve(next)
      } else {
        awaiting = resolve
      }
    })
  }

  async function deliver (message) {
    messages.push(message)
    if (awaiting) {
      awaiting(messages.shift())
      awaiting = undefined
    }
  }

  return {receive: receive, deliver: deliver}
}
```

General purpose actor

```
function init () { return 0 }

function handle (message, count) {
  return count + 1
}

var state = init()
while (true) {
  const message = await mailbox.receive()
  state = handle(message, state)
}
```

Starting actors

```
function Actor (init, handle) {
  const mailbox = Mailbox()

  (async function run () {
    var state = init()

    while (true) {
      const message = await mailbox.receive()
      state = handle(message, state)
    }
  })()

  return {deliver: mailbox.deliver}
}
```

Guarantees only this actor is able to receive from the mailbox

Actor system - requirements

Actors specify a concurrent program. To run the program requires an Actor System that handles.

- Allocating addresses
- Delivering messages
- Scheduling actors

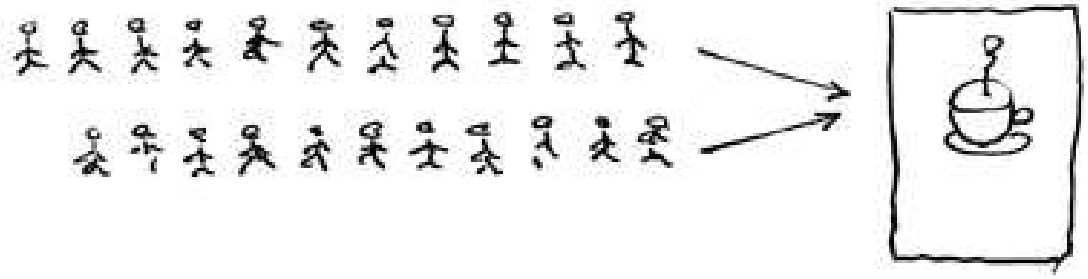
Actor System

```
const actors = []

function start (init, handle) {
  return actors.push(Actor(init, handle)) - 1
}

async function deliver (address, message) {
  actors[address].dispatch(message)
}

ActorSystem = {start, deliver}
```

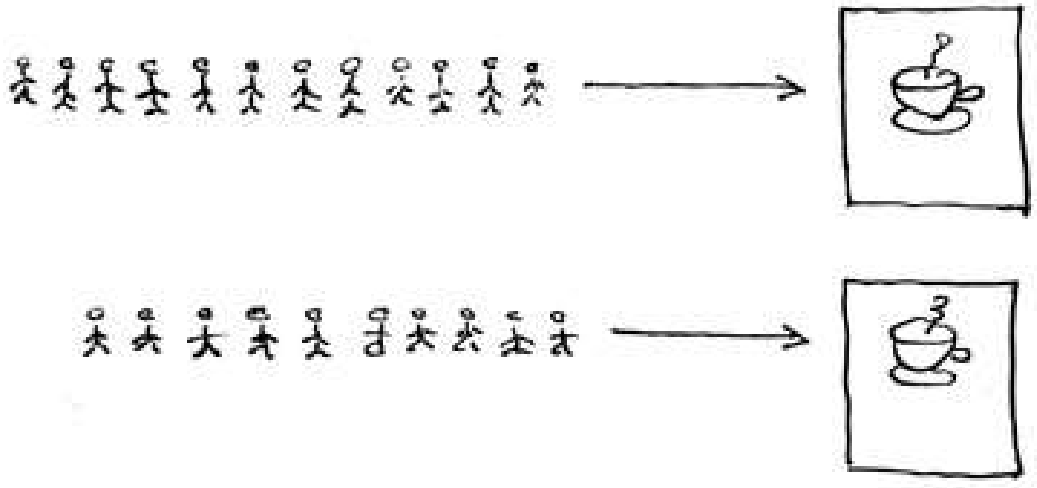
Actor system - trade offs

- Concurrent (not parallel)
- Cooperative (not preemtive)
- Voluntary (not obligatory)

Concurrent = Two Queues One Coffee Machine



Parallel = Two Queues Two Coffee Machines



Cooperative vs Preemtive

- Cooperative - processes must yield control.
- Preemtive - a process can be stoped at any time.

```
const message = mailbox.receive()

// greedy process
while (true) {
  // run forever
}
```

Voluntary vs Mandatory

```
// send a mutable message
const message = []
ActorSystem.dispatch(actor, message)

// later
message.push('surprise')
```

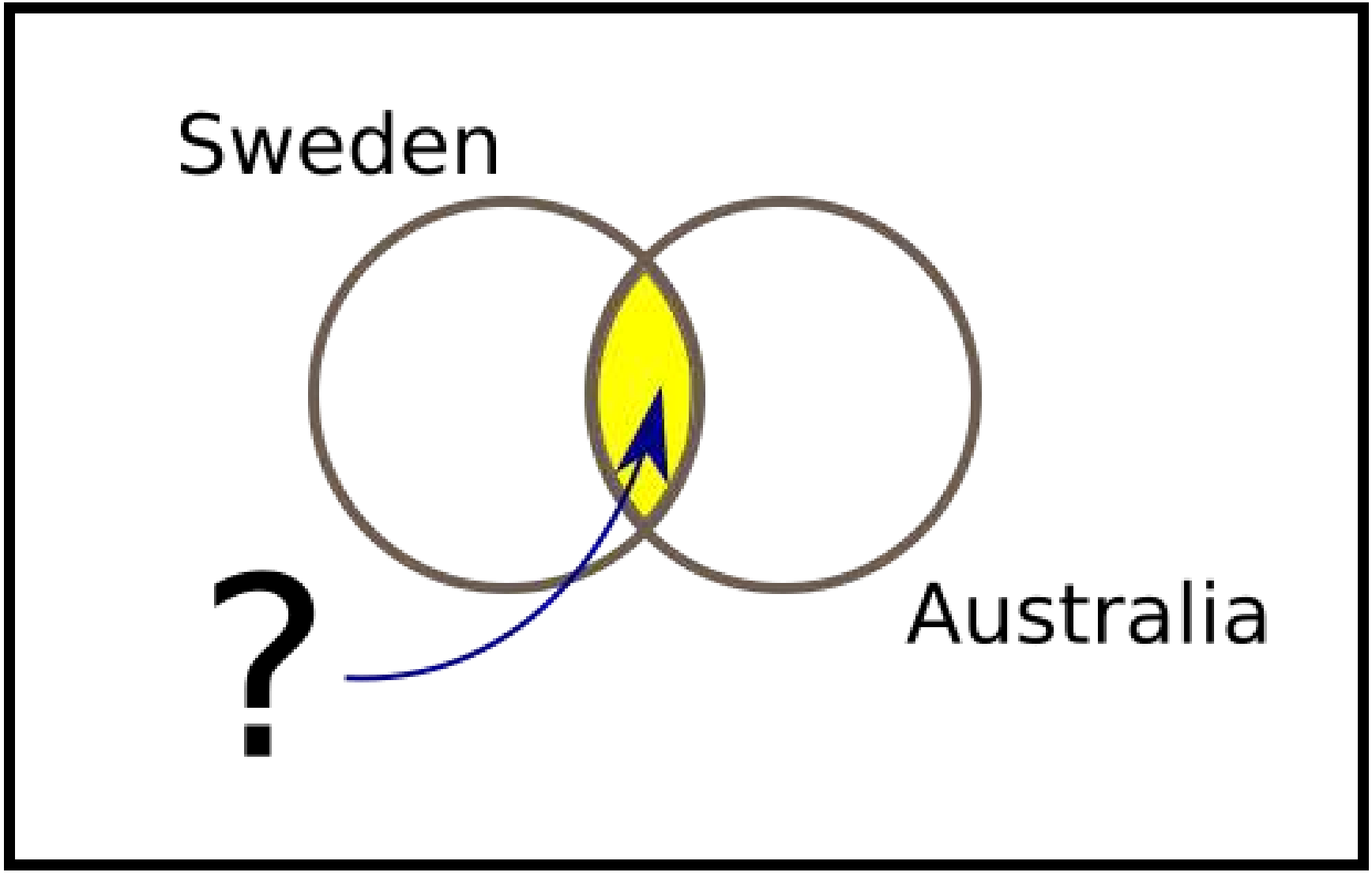
```
const message = mailbox.receive()

// use global state
window.message = message
```

Example: Ping pong

```
// actor behaviour
function init () { return null }

function handle ({type, address}, state) {
  if (type == 'ping') {
    ActorSystem.dispatch(address, {type: 'pong'})
  } else {
    console.log('Received Pong!')
  }
  return state
}

// run
const a1 = ActorSystem.start(init, handle)
const a2 = ActorSystem.start(init, handle)

ActorSystem.dispatch(a1, {type: 'ping', address: a2})
```

Why?

Where is shared memory

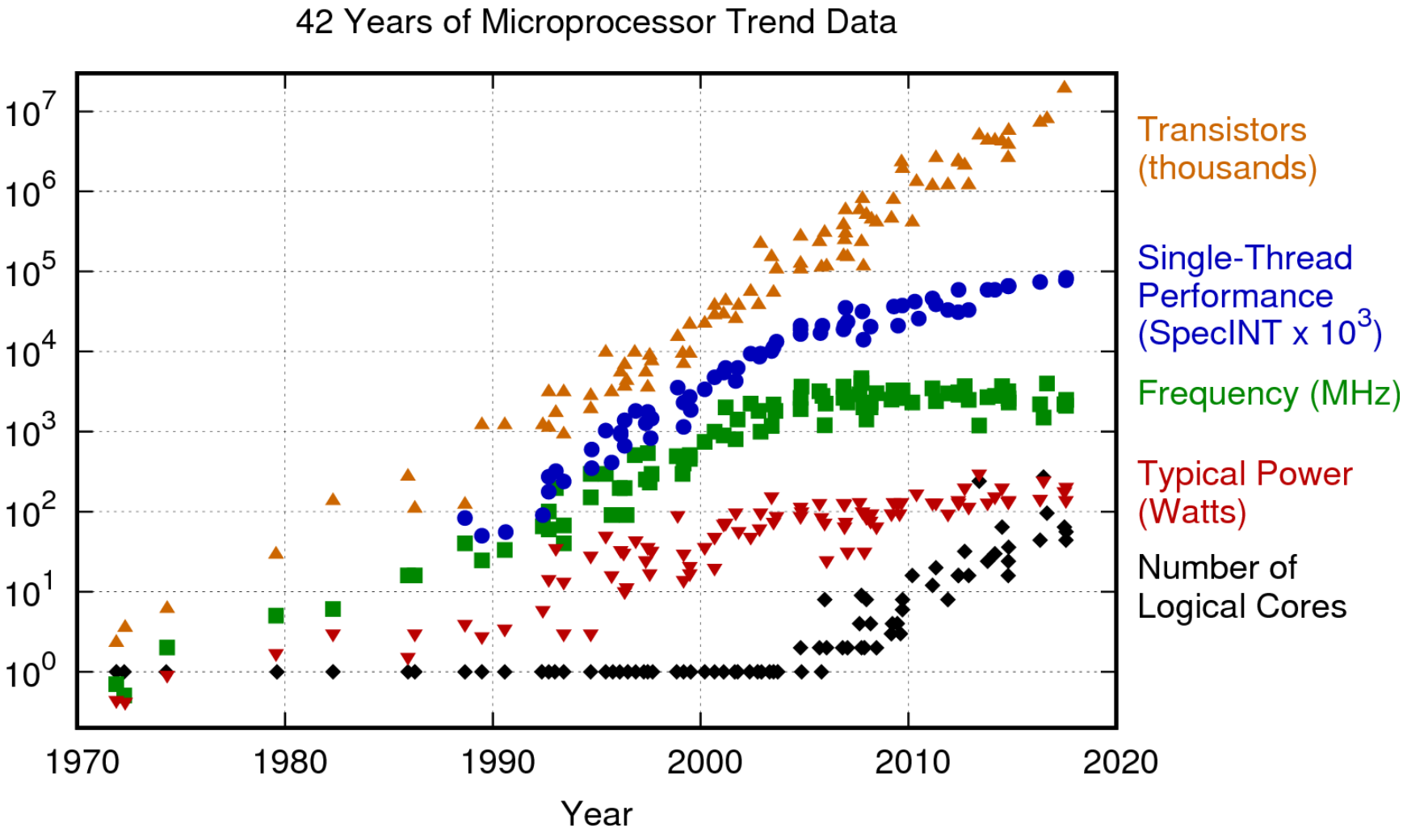


Your distributed system

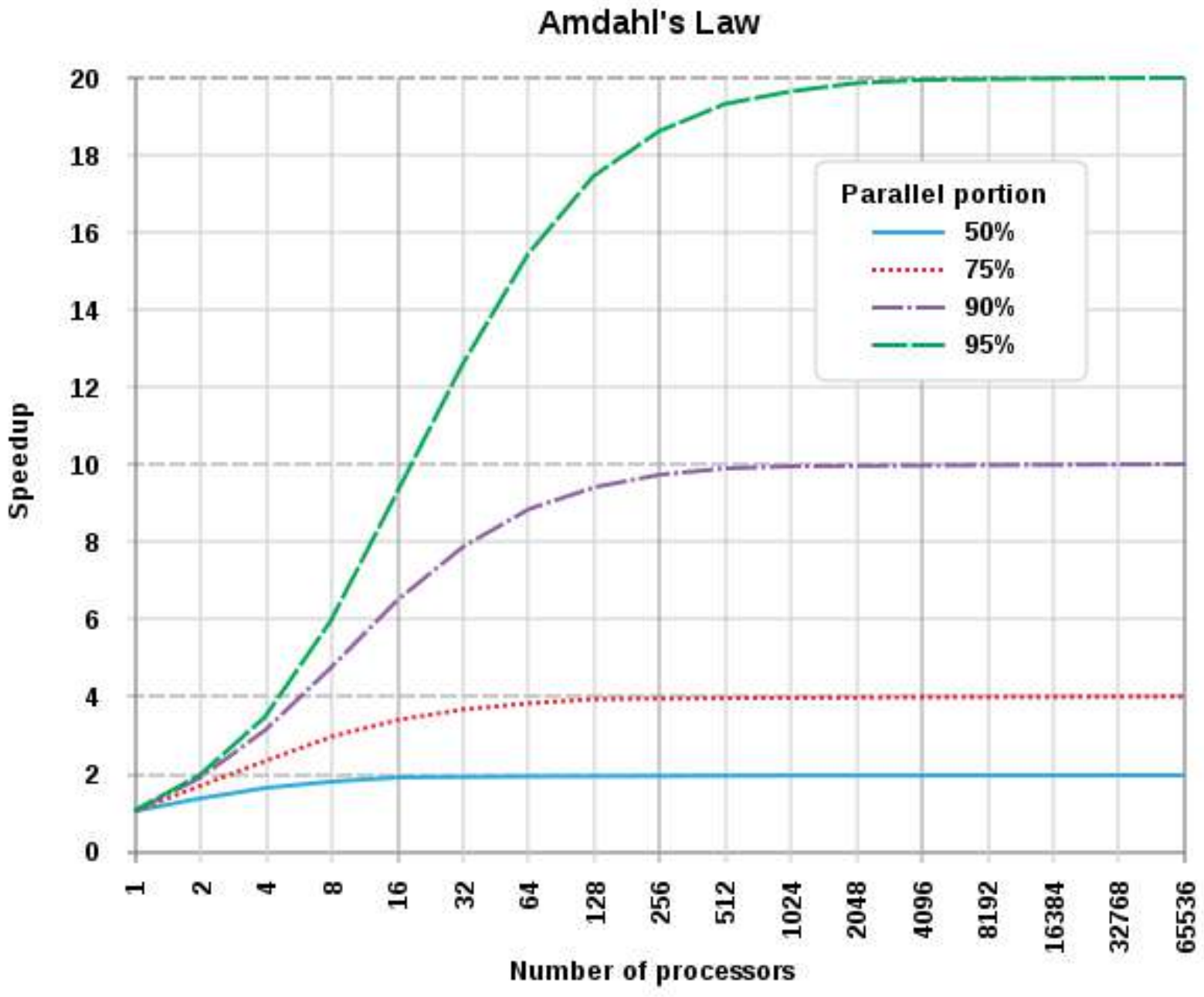
- High availability
- client and server
- backups failover
- multicore

Sending data ALWAYS has latency, and is unreliable.

Programming is parallel



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp


Amdahl's Law
Speedup
Number of processors
Parallel portion
50%
75%
90%
95%
0
2
4
6
8
10
12
14
16
18
20
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536

> “My first message is that concurrency is best regarded as a program structuring principle”

Tony Hoare

> Packing huge big rocks into containers is very very difficult, but pouring sand into containers is really easy. If you think of processes like little grains of sand and you think of schedulers like big barrels that you have to fill up, filling your barrels up with sand, you can pack them very nicely, you just pour them in and it will work.

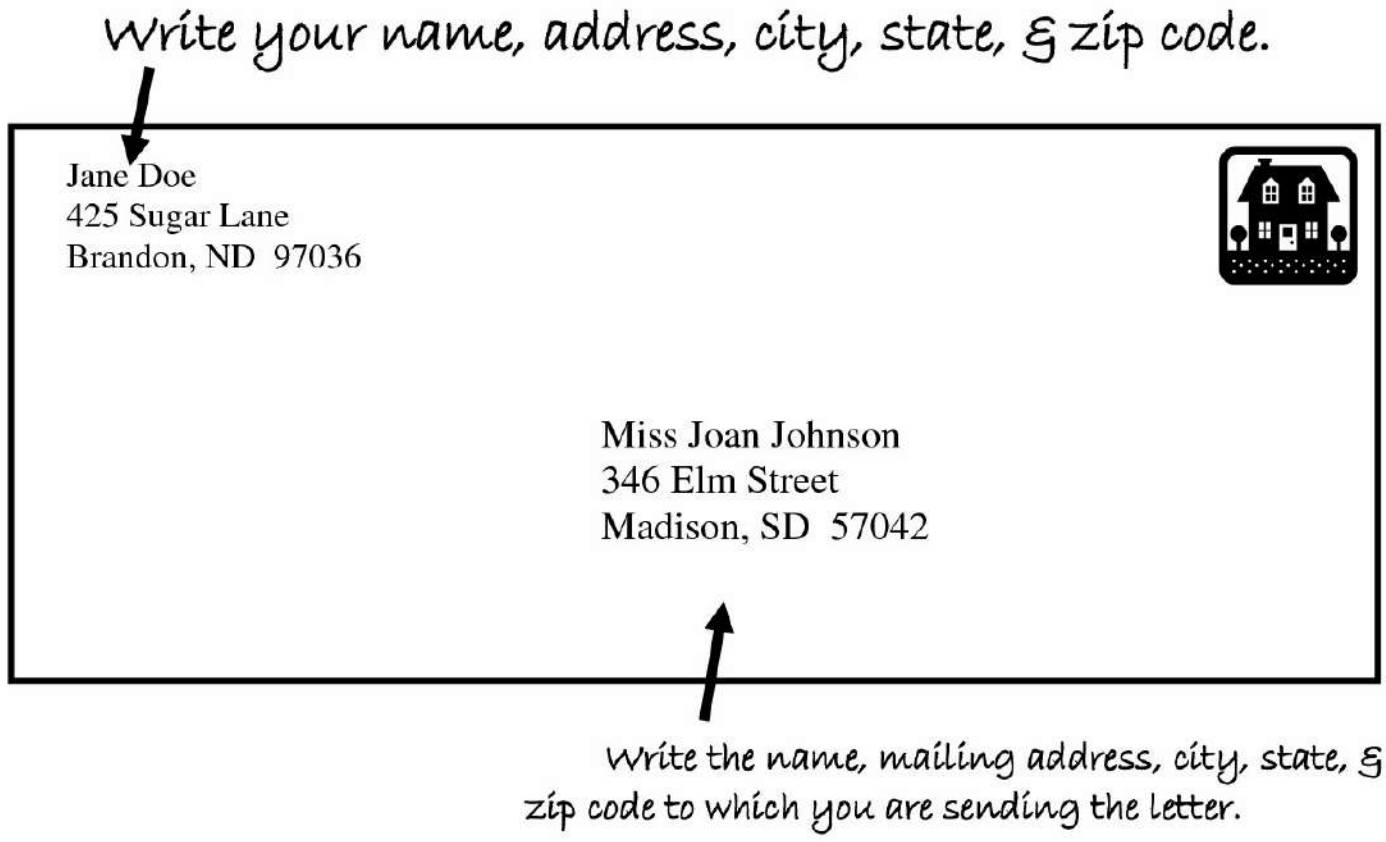
Joe Armstrong

ActorSystem2

```
navigator.hardwareConcurrency
// 4

new Worker('./actor-system.js')
```

Left as an exercise for the reader.

Abstracted communication



Many to Many relationship among Actors and Addresses.

```
$ dig +short google.com
216.58.204.14
```

Descriptive side effects

```
var state = init()
while (true) {
  const message = await mailbox.receive()
  {outbound, state} = handle(message, state)
  outbound.forEach(doSend)
}
```

handle can now be a totally pure function

Session types

http://www.di.unito.it/~dezani/papers/sto.pdf

HTTP is message passing

> The Hypertext Transfer Protocol (HTTP) is a stateless application- level request/response protocol that uses extensible semantics and self-descriptive message payloads for flexible interaction with network-based hypertext information systems.

Let's build a server

Mandatory actor system

> The Erlang view of the world is that everything is a process and that processes can interact only by exchanging messages.

Joe Armstrong

What is **Raxx**?

1. Elixir interface for HTTP servers, frameworks (and clients)
2. Toolkit for web development
3. Simple streaming support

and Ace?

A server to run Raxx applications

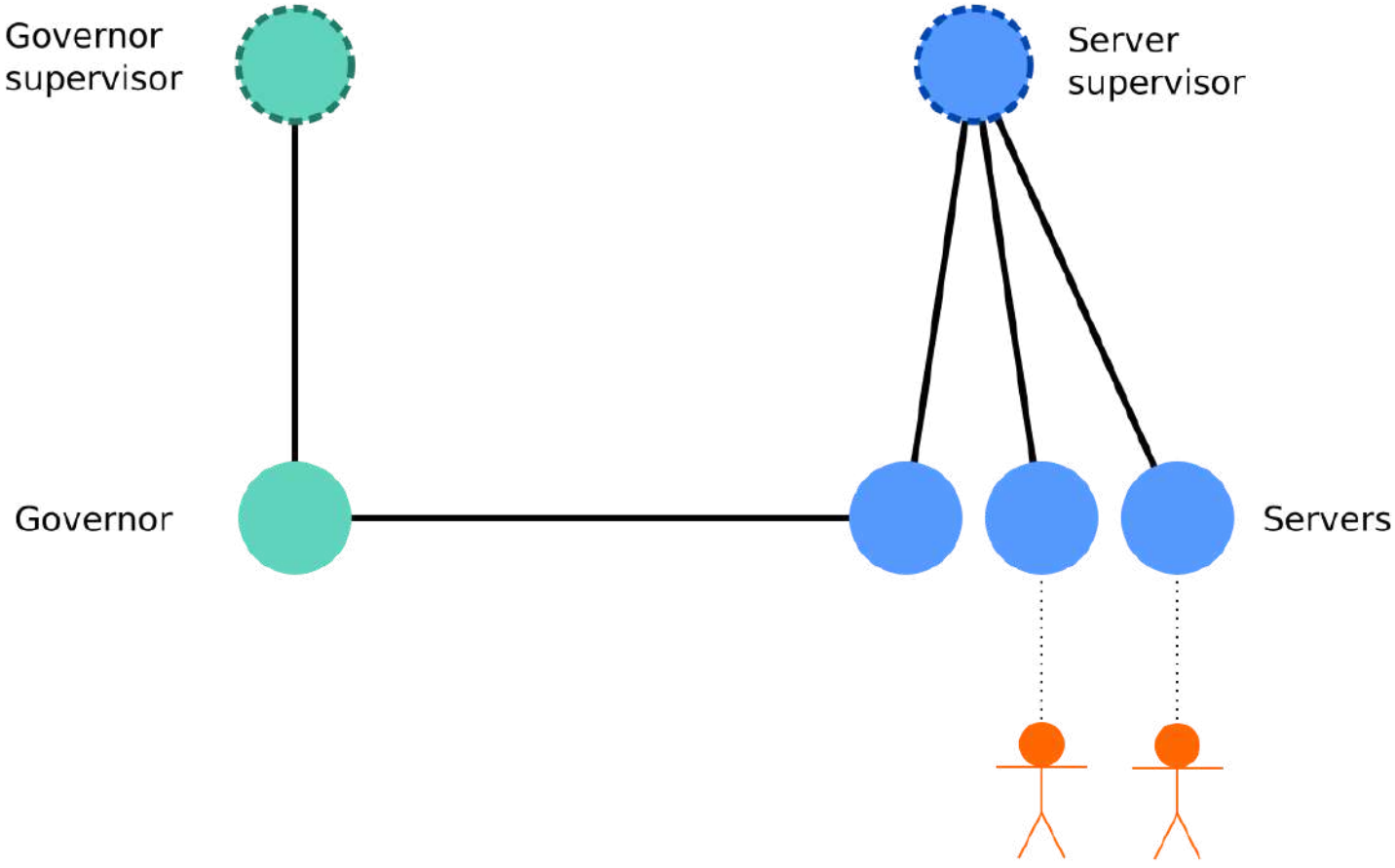
1. HTTP/2 + HTTPS, by default
2. Isolated message exchanges

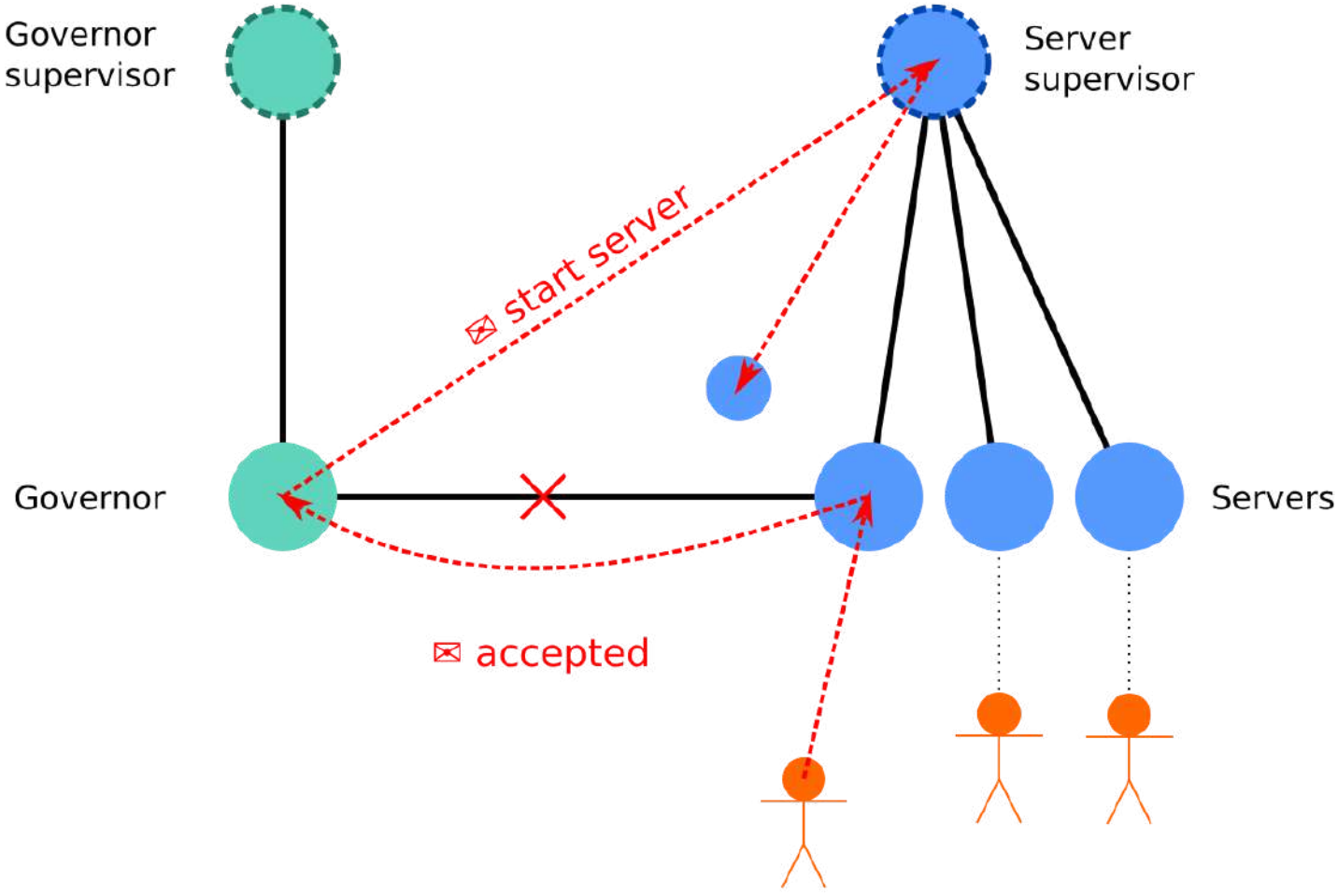
Walking tour of Ace

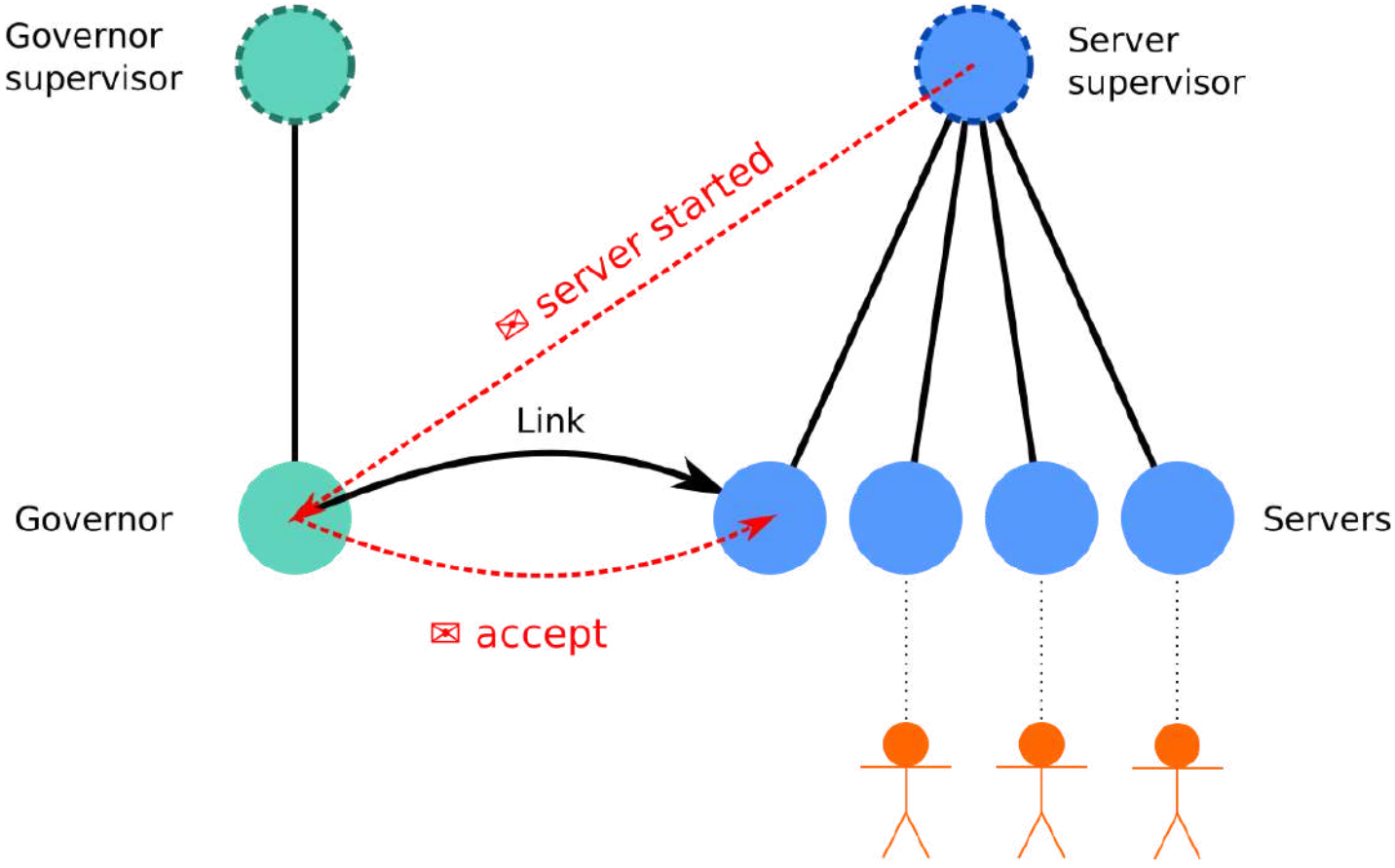
> We do not have ONE web-server handling 2 millions sessions. We have 2 million webservers handling one session each.

Managing Two Million Webservers

Joe Armstrong


Governor
supervisor
Server
supervisor
Governor
Servers


Governor
supervisor
Server
supervisor
start server
Governor
Servers
accepted


Governor
supervisor
Server
supervisor
server started
Link
Governor
Servers
accept

GenServer

```
defmodule MyServer do
  use GenServer

  def handle_call(:request, _from, state) do
    {:reply, :response, state}
  end
end
```

Raxx.SimpleServer

```
defmodule Greetings do
  use Raxx.Server

  def handle_request(
    _request,
    _state)
  do
    %Raxx.Response{status: 200,
      headers: ["content-type", "text/plain"]
      body: "Hello, World!"}
  end
end
```

Raxx.SimpleServer

```
defmodule Greetings do
  use Raxx.Server

  def handle_request(
    _request,
    _state)
  do
    response(:ok)
    |> set_header("content-type", "text/plain")
    |> set_body("Hello, World!")
  end
end
```

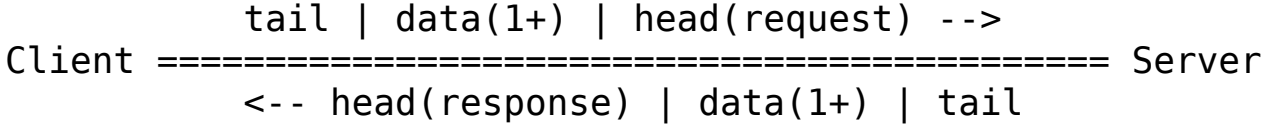
Raxx.SimpleServer

```
defmodule Greetings do
  use Raxx.Server

  def handle_request(
    %{path: ["name", name]},
    _state)
  do
    response(:ok)
    |> set_header("content-type", "text/plain")
    |> set_body("Hello, #{name}!")
  end
end
```

Raxx.SimpleServer

```
defmodule Greetings do
  use Raxx.Server

  def handle_request(
    %{path: ["name", name]},
    %{greeting: greeting})
  do
    response(:ok)
    |> set_header("content-type", "text/plain")
    |> set_body("#{greeting}, #{name}!")
  end
end
```

What about **streaming**?

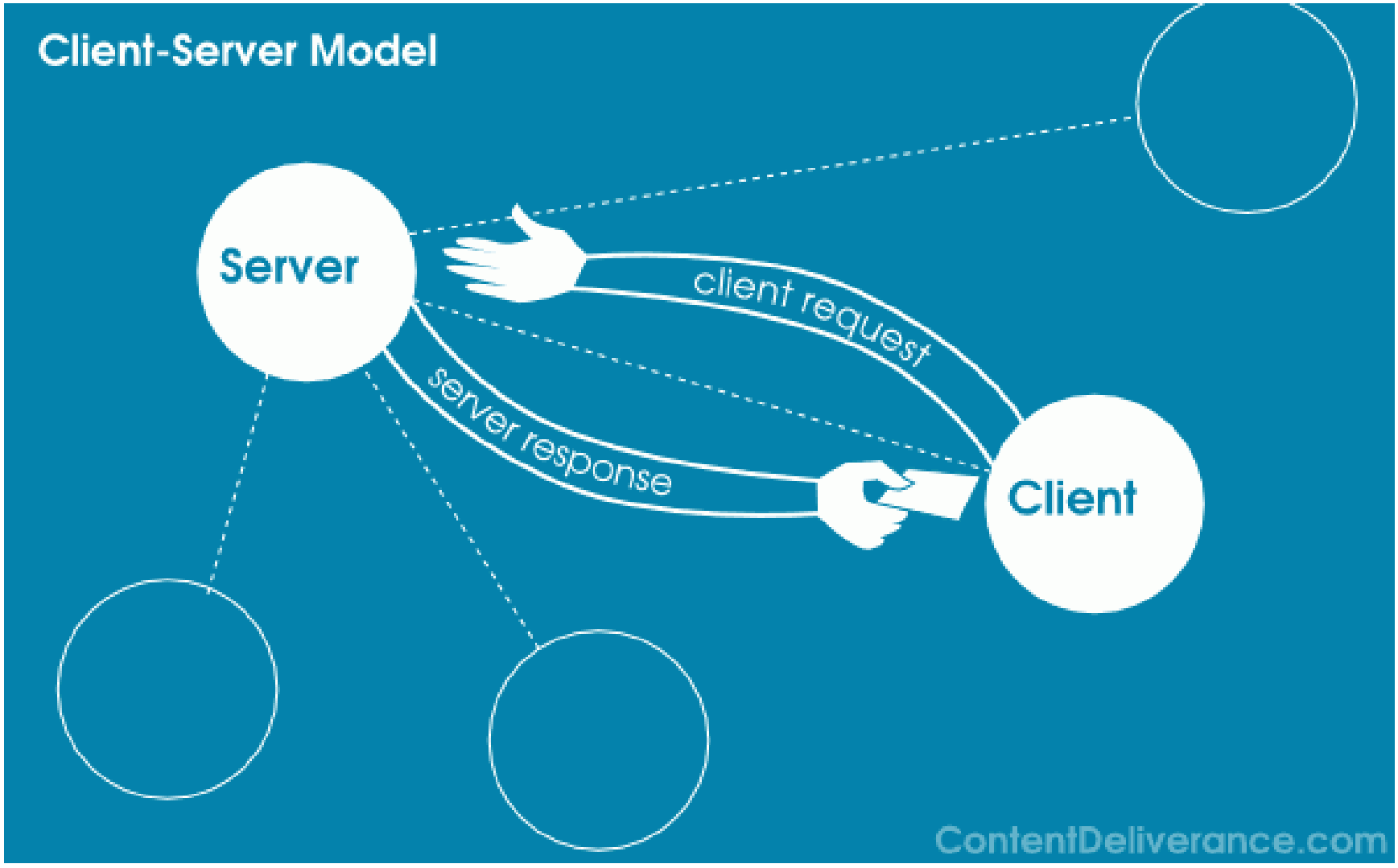

tail | data(1+) | head(request) -->
Client
Server
<-- head(response) | data(1+) | tail

```elixir
defmodule Upload do
  use Raxx.Server

  def handle_head(%{path: ["upload"] body: true}, _) do
    {:ok, io_device} = File.open("my/path")
    {[], {:file, device}}
  end

  def handle_data(data, state = {:file, device}) do
    IO.write(device, data)
    {[], state}
  end

  def handle_tail(_trailers, state) do
    response(:see_other)
    |> set_header("location", "/")
  end
end
```

The Raxx toolkit

- Routing ✔
- Middleware ✔
- Templates ✔ (EExHTML)
- Code reloading ✔ (Raxx.Kit)
- Project generators ✔ (Raxx.Kit)

Next?


Client-Server Model
Server
client request
server response
Client
ContentDeliverance.com

GenBrowser

GenBrowser treats clients as just another process in one continuous, if widely distributed, system. Every client gets an address to which messages can be dispatched; and a mailbox where messages are delivered.

Actor lifecycle

1. Client joins, it is not started.

```
const client = await GenBrowser.start('http://localhost:8080')

const {address, mailbox, send, communal} = client
console.log(address)
// "g2gCZA ..."
```

2. Disconnected clients are not dead.

Messages from the server

```
message = %{
  "type" => "ping",
  "from" => GenBrowser.Address.new(self())
}
GenBrowser.send("g2gCZA ...", message)

receive do
  message ->
    IO.inspect(message)
end
# => %{"type" => "pong"}
```

Messages from a client

```
client.send("g2gCZA ...", {type: 'ping', from: client.address})

const reply = await client.mailbox.receive({timeout: 5000})
console.log("Pong received")
```

Guarantees

- There should be only one actor per mailbox
  - Reconnection requires a cursor signed by the server
- Addresses are unforgable
  - Addresses are all signed by the server
  - Object capability model

Try it out

```
# Plug/Phoenix integration
communal = %{myProcess: GenBrowser.Address.new(MyNamedProcess)}

plug GenBrowser.Plug, communal: communal
```

```
# Docker playground
docker run -it -e SECRET=s3cr3t -p 8080:8080 gen-browser
```

What's behind an address?

```
{:via, GenBrowser, "abc123"}
```

```
{:via, IOTSensor, "lightbulb"}
```

```
{:via, PersistantActor, "4ever"}
```

```
{:via, EmailService, "bob@example.com"}
```

Humans

Humans

- Universal primitive of concurrent computation
- Communicate via asynchronous message passing
- React to messages by making local decisions

Thank you

See the code

- github.com/crowdhailer/raxx
  Interface for HTTP webservers, frameworks and clients.
- github.com/crowdhailer/gen_browser
  Actors for the client and server

Comments and questions

- twitter.com/CrowdHailer