Concurrency + Distribution = Availability + Scalability

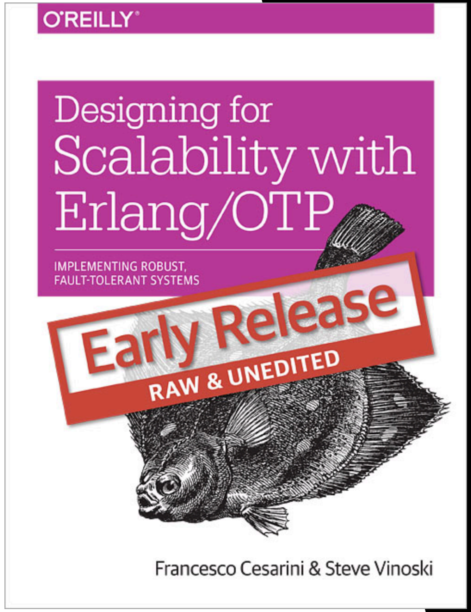
Francesco Cesarini

francesco@erlang-solutions.com
www.erlang-solutions.com
@francescoC

Erlang SOLUTIONS

Chapter 13

Ch 13: Node Architecture



O'REILLY

Designing for Scalability with Erlang/OTP

IMPLEMENTING ROBUST, FAULT-TOLERANT SYSTEMS

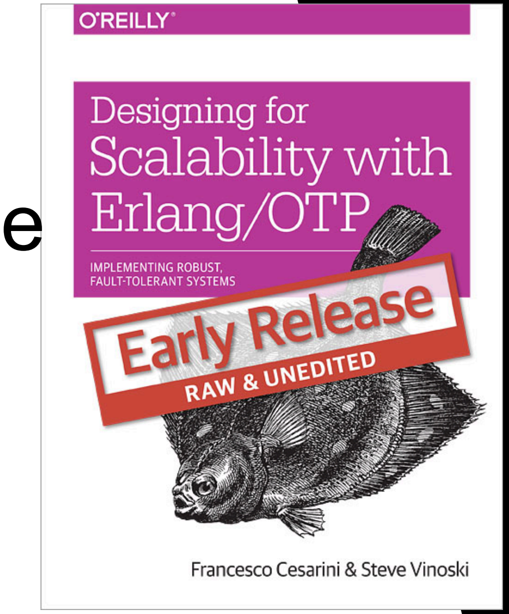
Early Release

RAW & UNEDITED

Francesco Cesarini & Steve Vinoski

Chapter 13

Ch 13: Distributed Architectures
Ch 14: Systems That Never Stop
Ch 15: Scaling Out
Ch 16: Monitoring and Preemptive
Support



O'REILLY

Designing for Scalability with Erlang/OTP

IMPLEMENTING ROBUST, FAULT-TOLERANT SYSTEMS

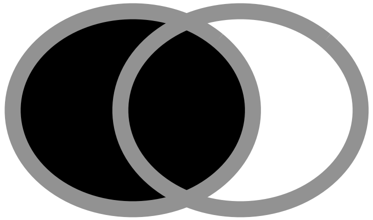
Early Release
RAW & UNEDITED

Francesco Cesarini & Steve Vinoski

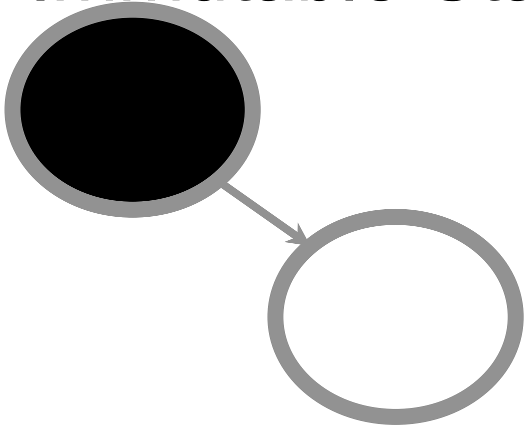
Concurrency

The world is concurrent. People speak to each other through message passing. Things fail.

Mutable State

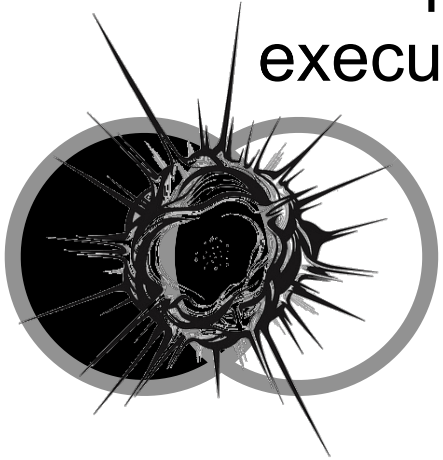


Immutable State



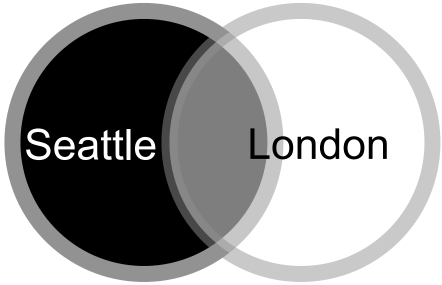
Problem 1 with mutable state:

Your program crashes whilst executing in the critical section…



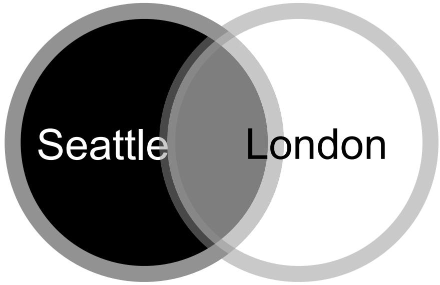
Problem 2 with mutable state:

Where do you locate your state…



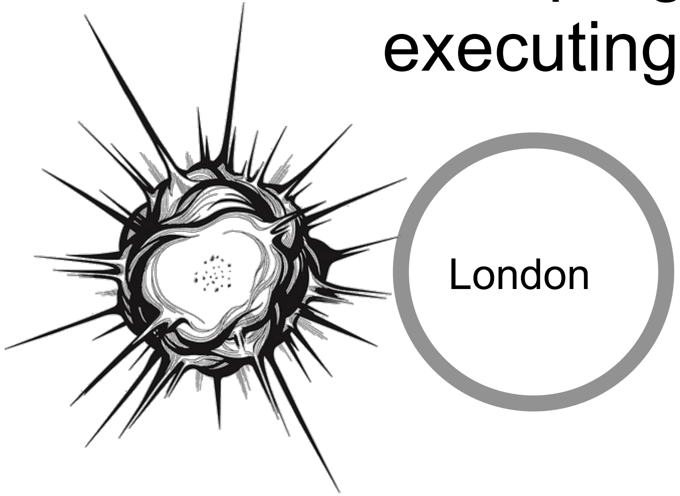
Problem 3 with mutable state:

What happens if your network connectivity fails…



Problem 1 with mutable state:

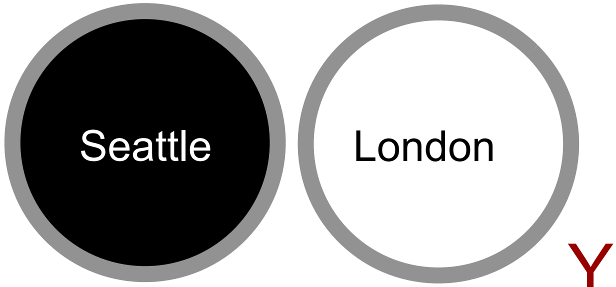
Your program crashes whilst executing in the critical section…



Your state **does not get corrupted**.

Problem 2 with mutable state:

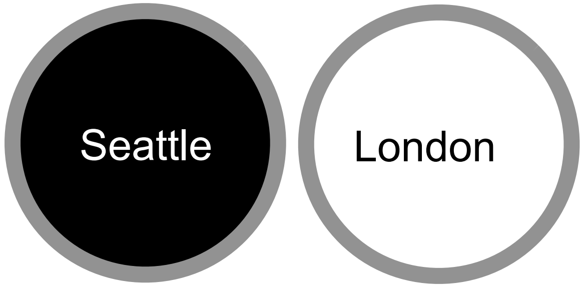
Where do you locate your state…



You **do not Locate state,**
you copy it.

Problem 3 with mutable state:

What happens if your network connectivity fails…



Seattle

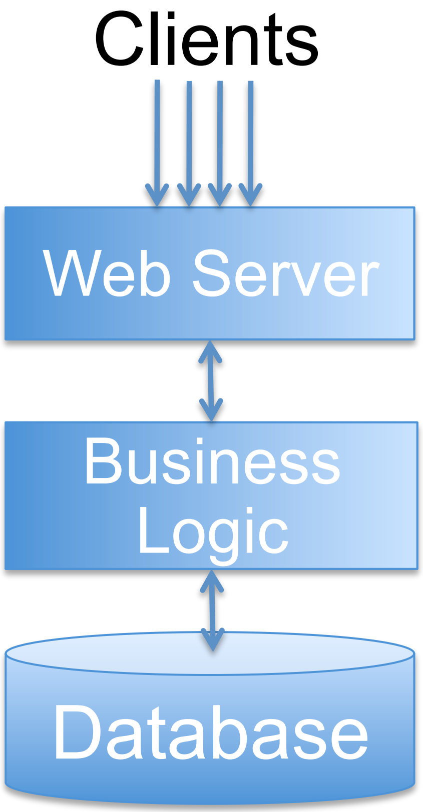
London

Make sure your business logic and databases handle network splits!

Distributed Architectures

A node is the smallest executable standalone unit consisting of a running instance of the Erlang runtime system.

Node Types

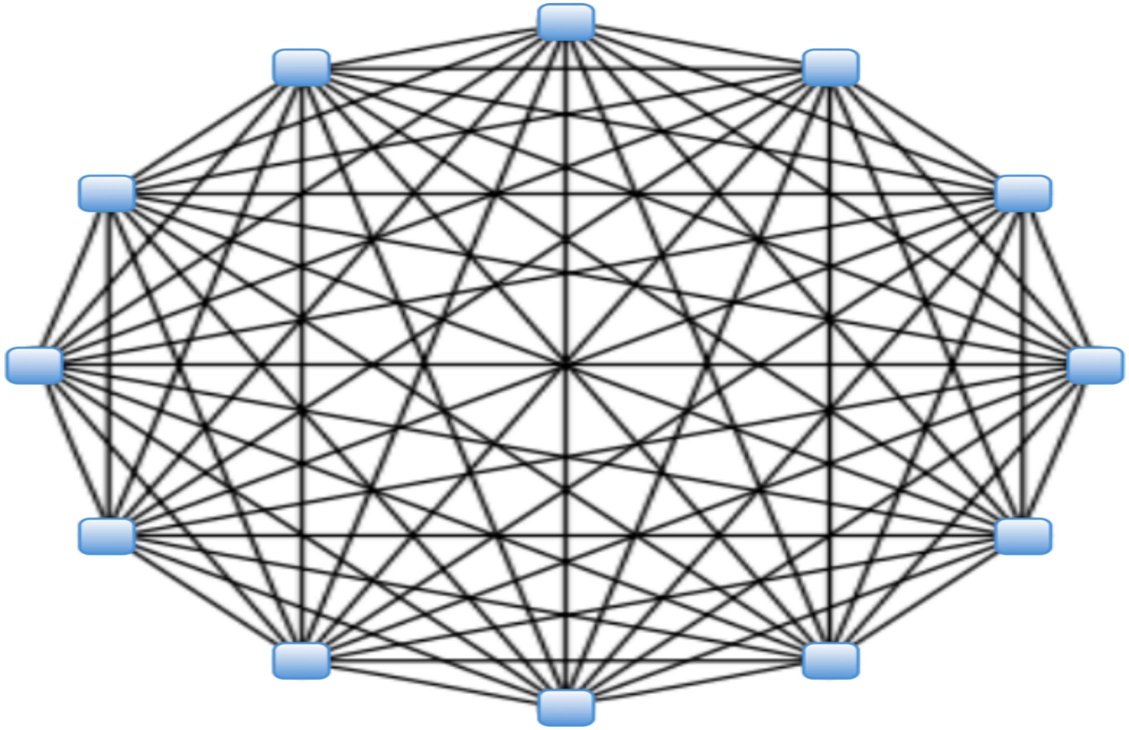


Front-end Node

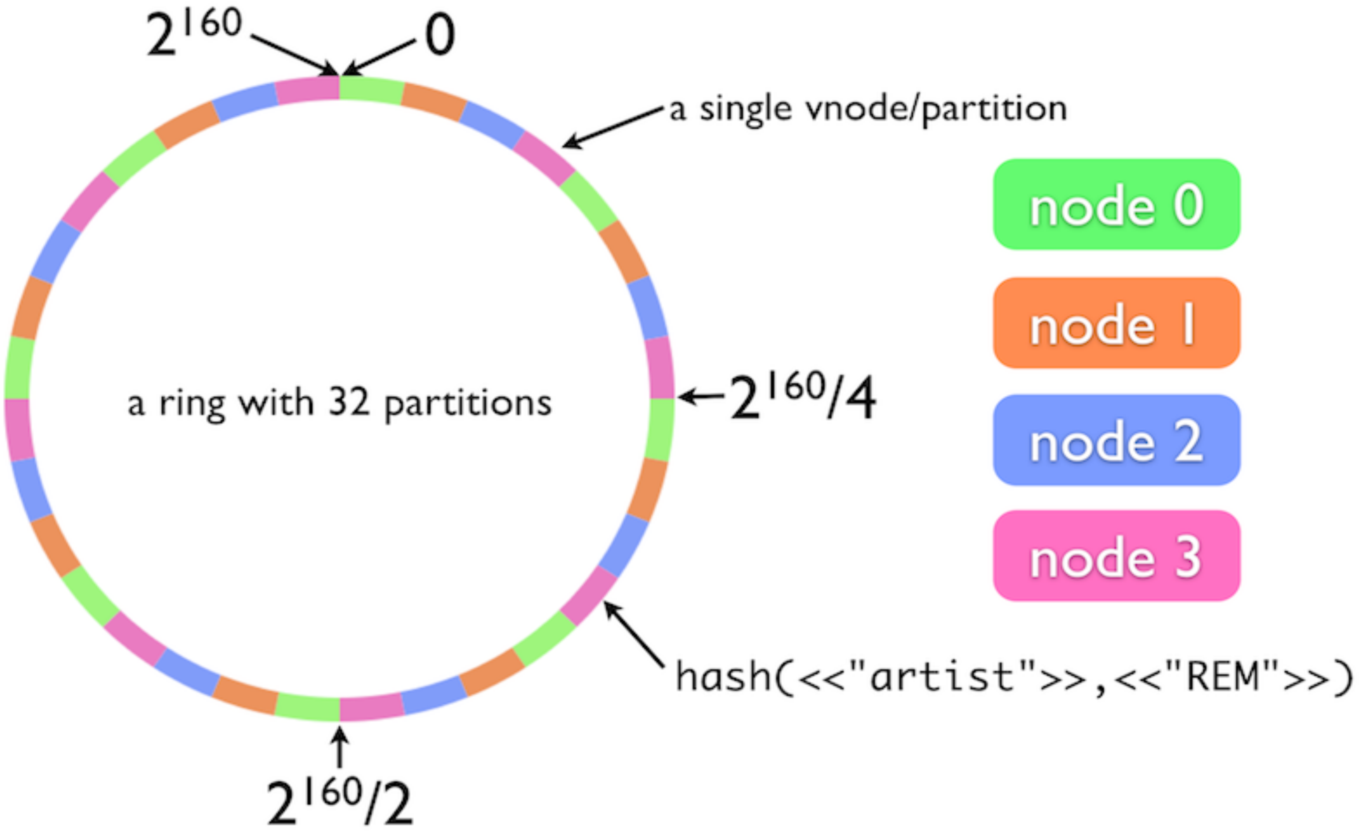
Logic Node

Service Node

Fully Meshed



Dynamo



2^{160}

0

a single vnode/partition

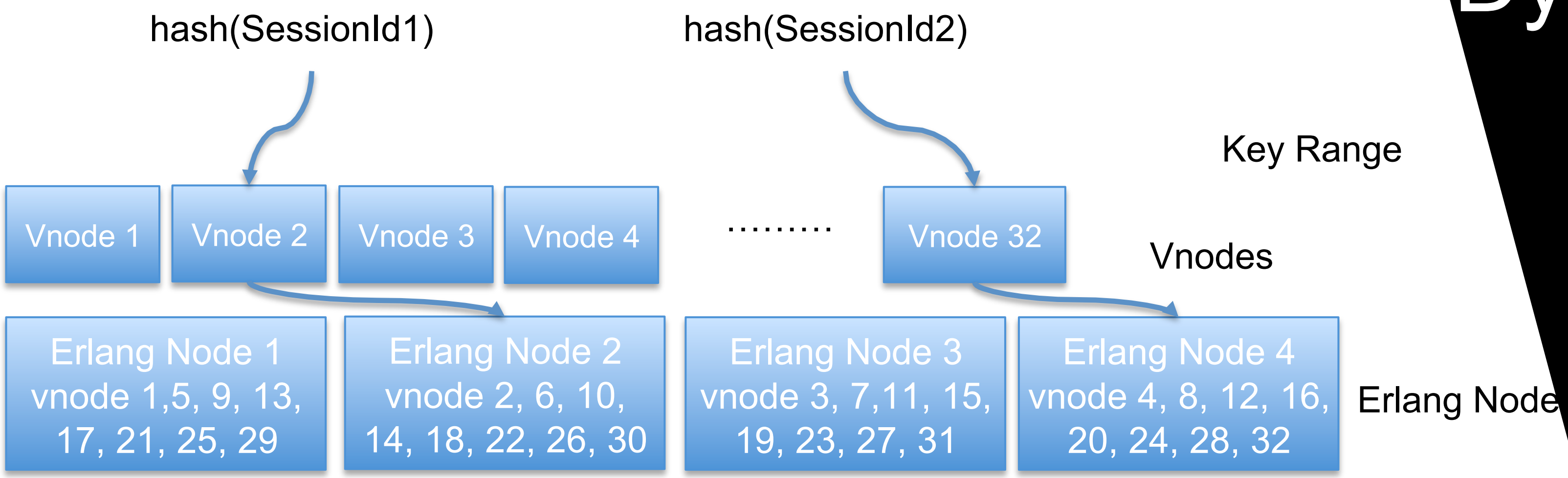
a ring with 32 partitions

$2^{160}/4$

$2^{160}/2$

`hash(<<"artist">>,<<"REM">>)`

Dynamo



hash(SessionId1)

hash(SessionId2)

Key Range

Vnode 1

Vnode 2

Vnode 3

Vnode 4

....

Vnode 32

Vnodes

Erlang Node 1
vnode 1,5, 9, 13,
17, 21, 25, 29

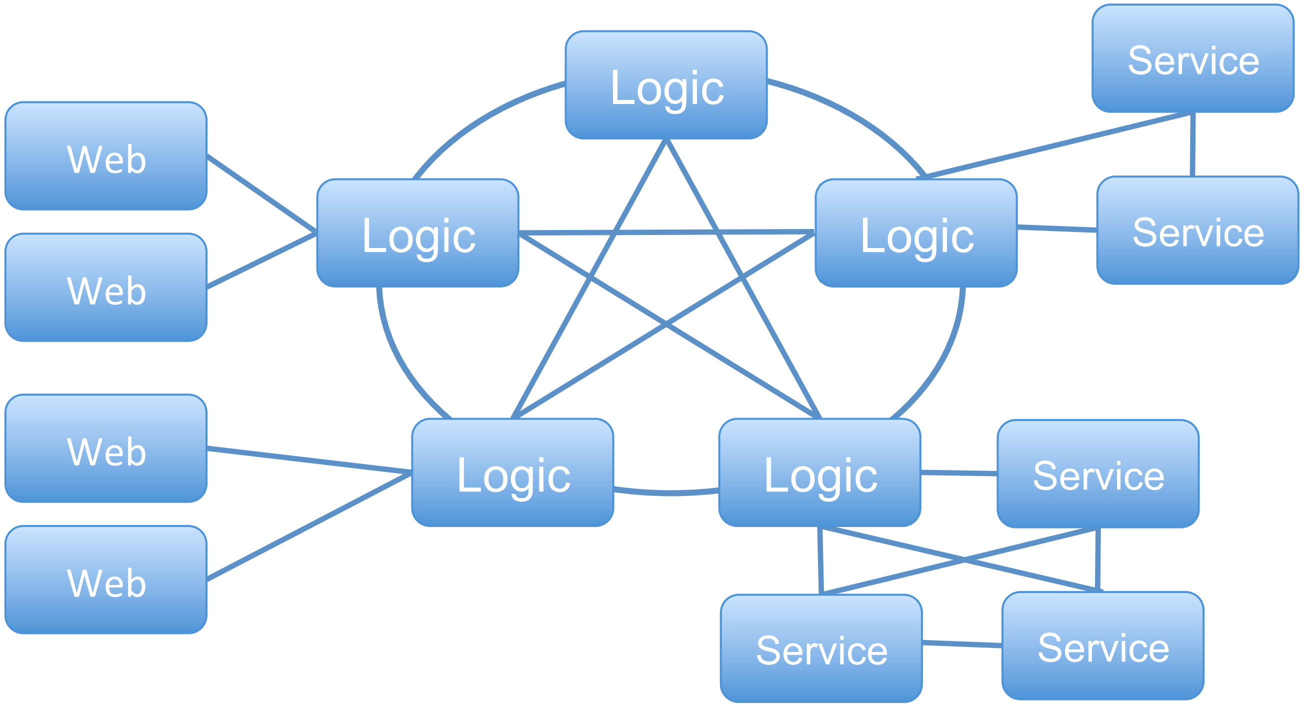
Erlang Node 2
vnode 2, 6, 10,
14, 18, 22, 26, 30

Erlang Node 3
vnode 3, 7,11, 15,
19, 23, 27, 31

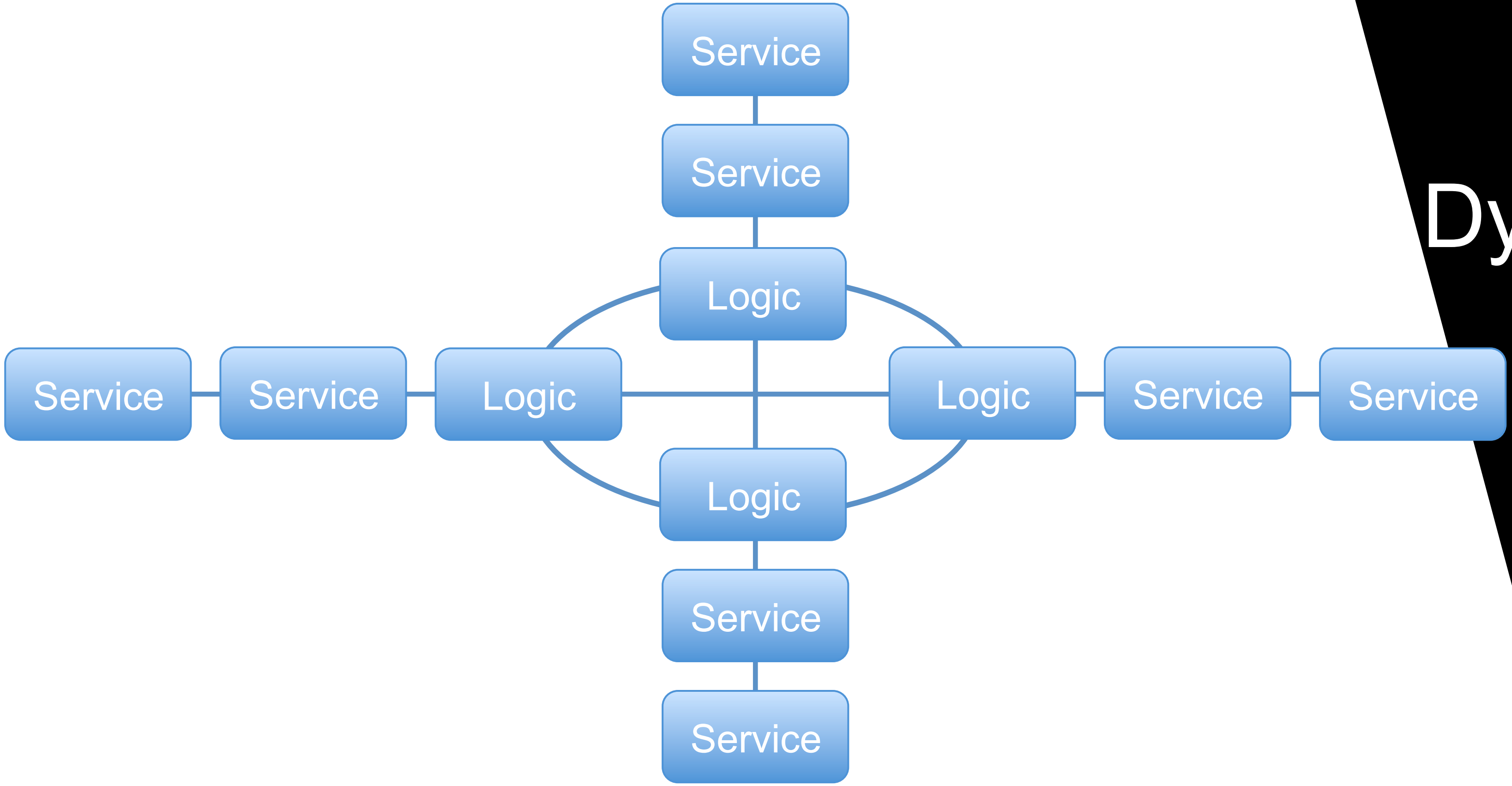
Erlang Node 4
vnode 4, 8, 12, 16,
20, 24, 28, 32

Erlang Node

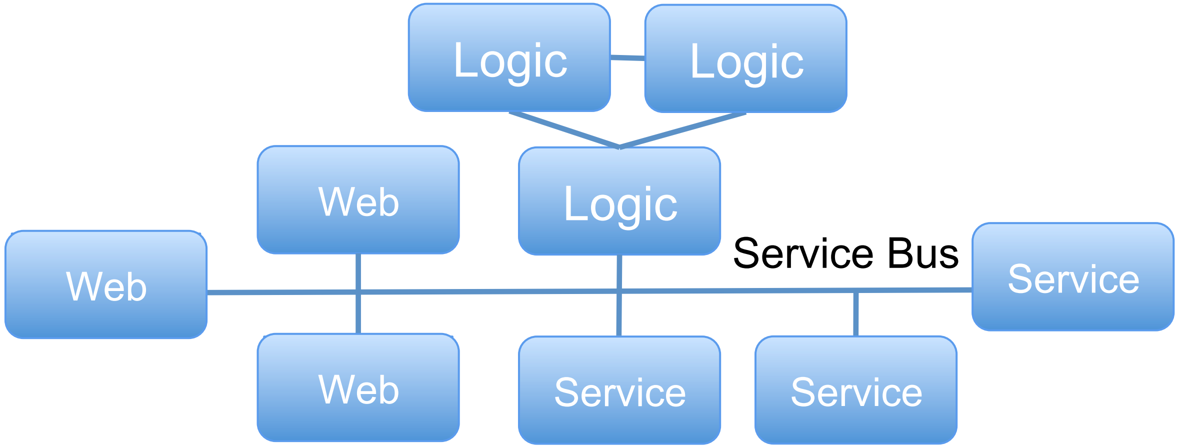
Dynamo


Web
Web
Web
Web
Logic
Logic
Logic
Logic
Logic
Service
Service
Service
Service
Service

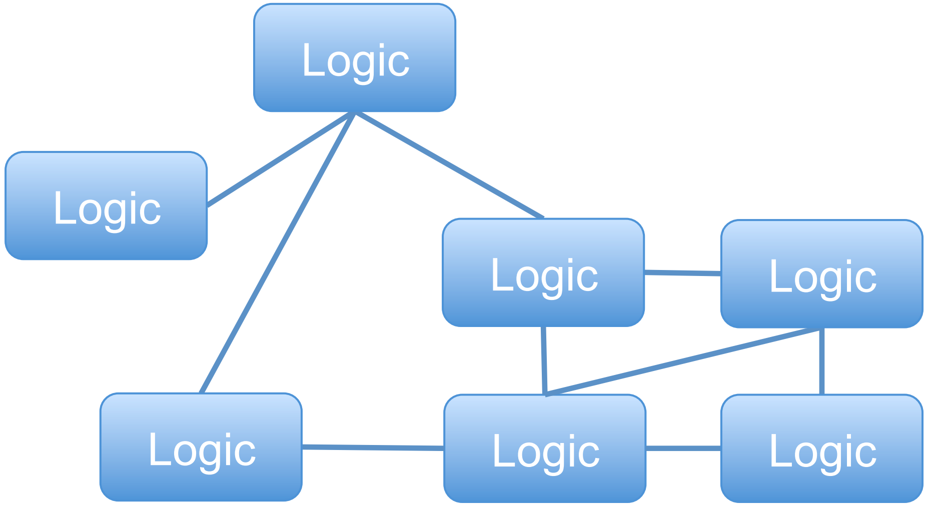
Dynamo


Service
Service
Logic
Service
Service
Logic
Logic
Service
Service
Logic
Service
Service

Service Bus

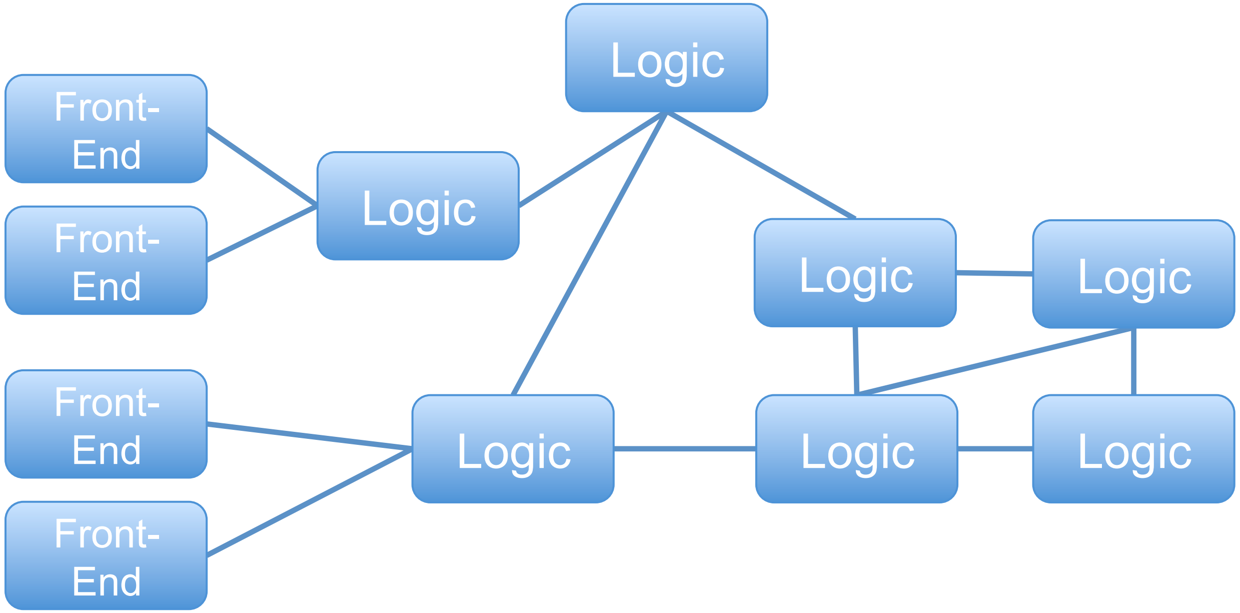

Logic
Logic
Web
Logic
Web
Service Bus
Service
Web
Service
Service

Peer to Peer

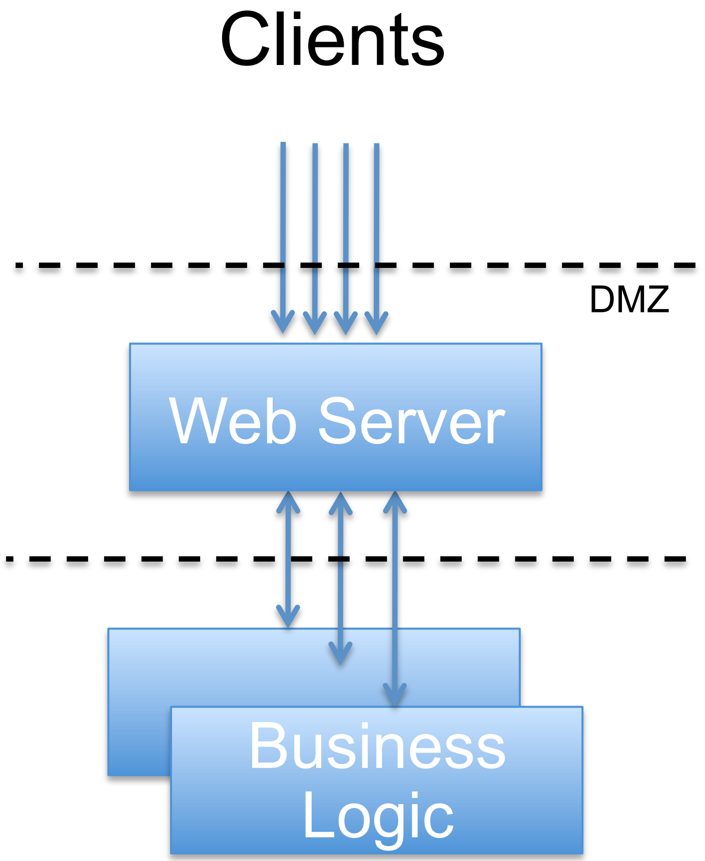


www.erlang-solutions.com

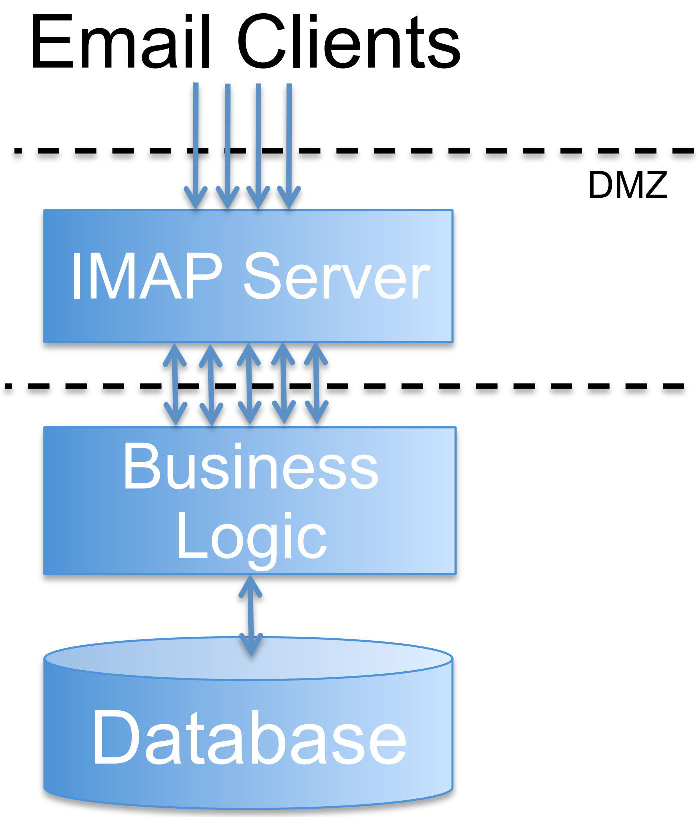
Peer to Peer



Networking

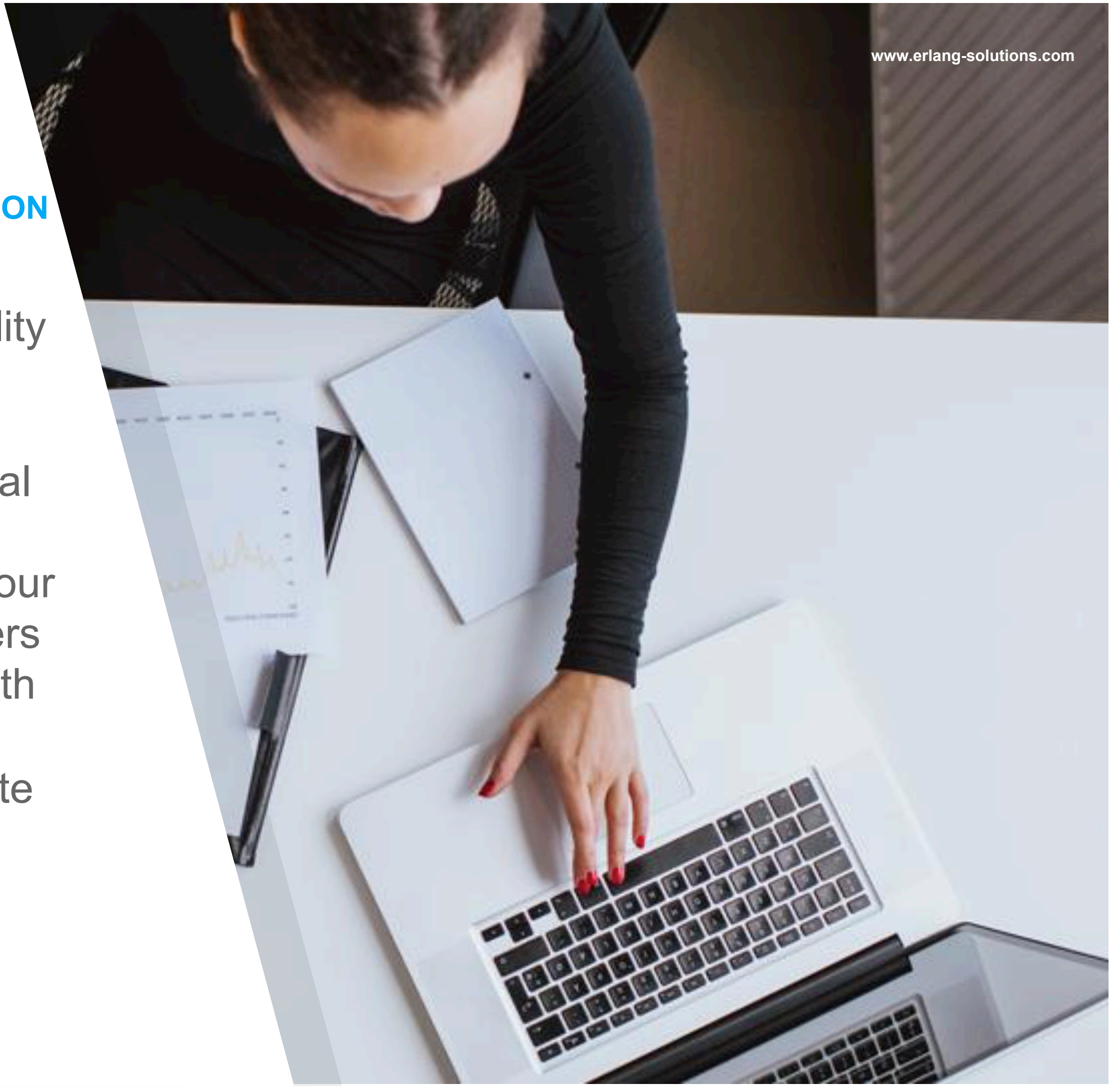


Networking



STEPS EVOLVING AROUND DISTRIBUTION

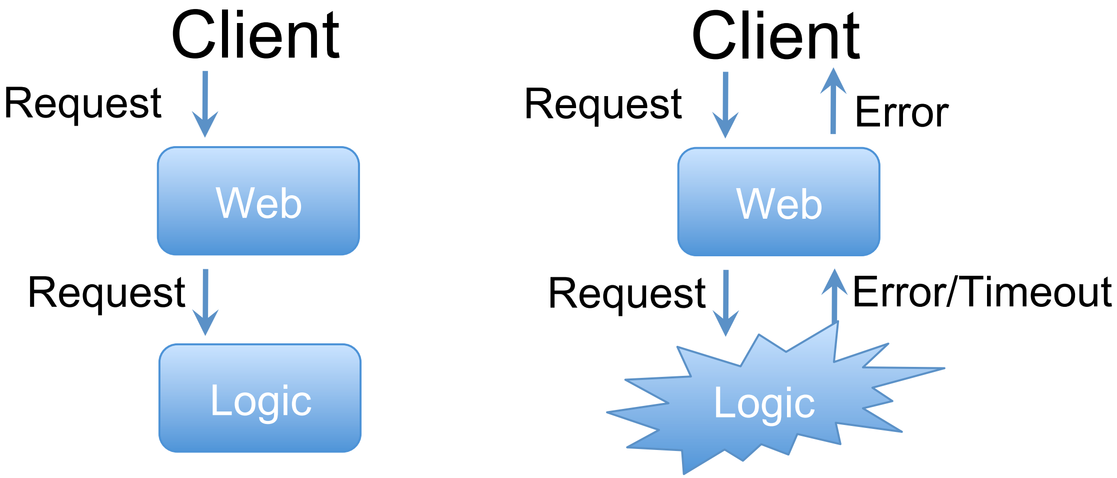
1. Split up your system's functionality into manageable, stand-alone nodes.
2. Choose a distributed architectural pattern.
3. Choose the network protocols your nodes, node families, and clusters will use when communicating with each other.
4. Define your node interfaces, state and data model.



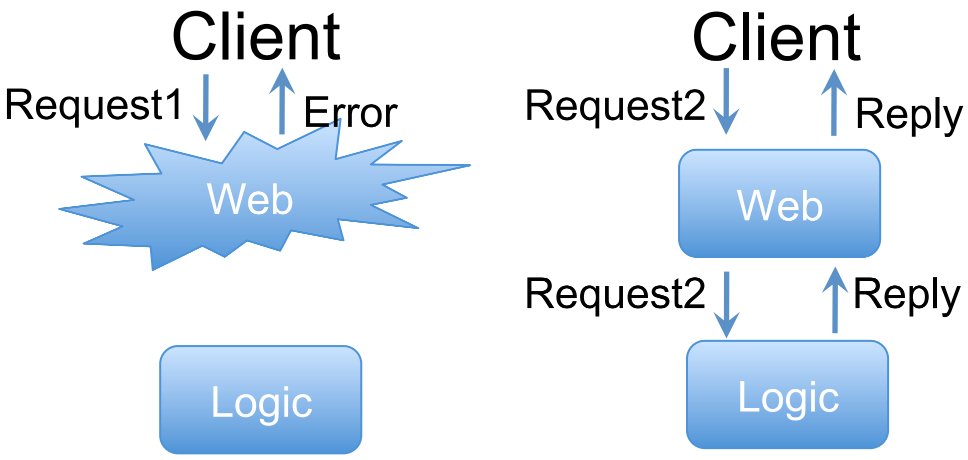
Systems That Never Stop

You need at least two computers to make a fault tolerant system.

Fault Tolerance

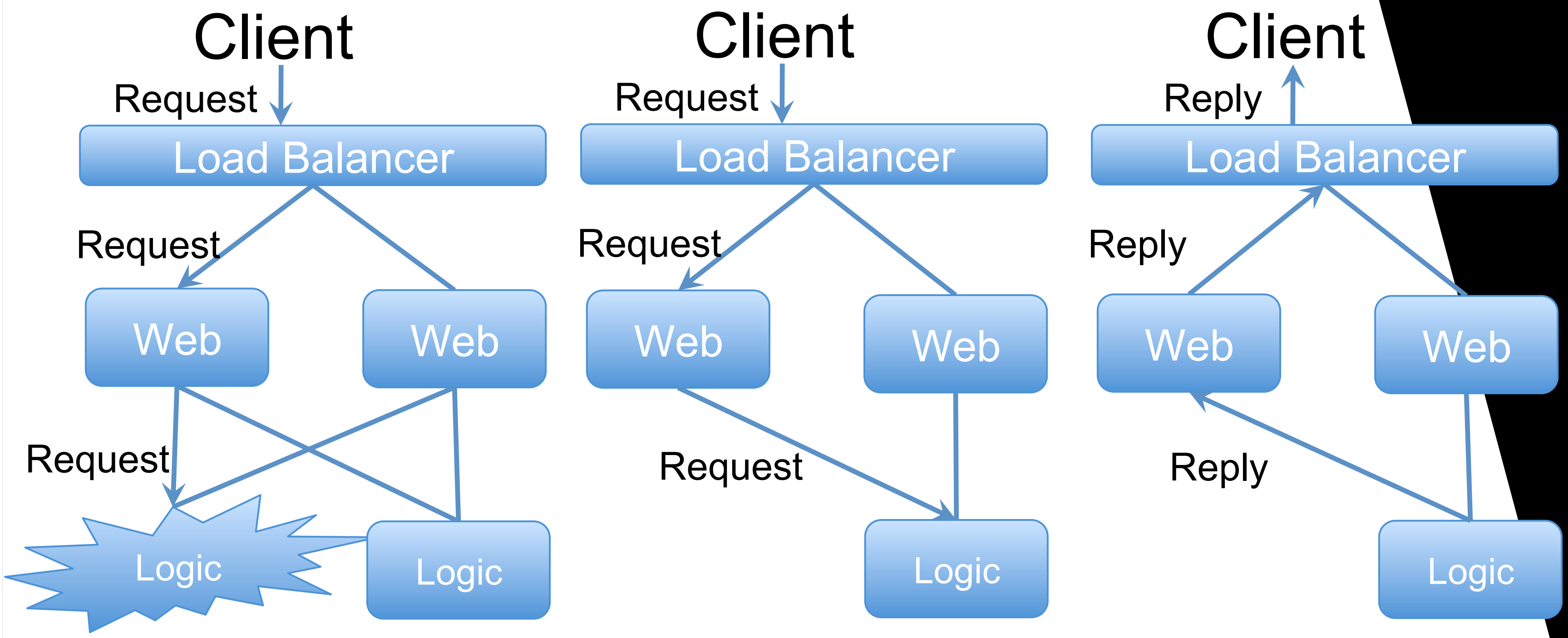


Resilience


Client
Request1
Error
Web
Logic
Client
Request2
Reply
Web
Request2
Reply
Logic

www.erlang-solutions.com

Reliability

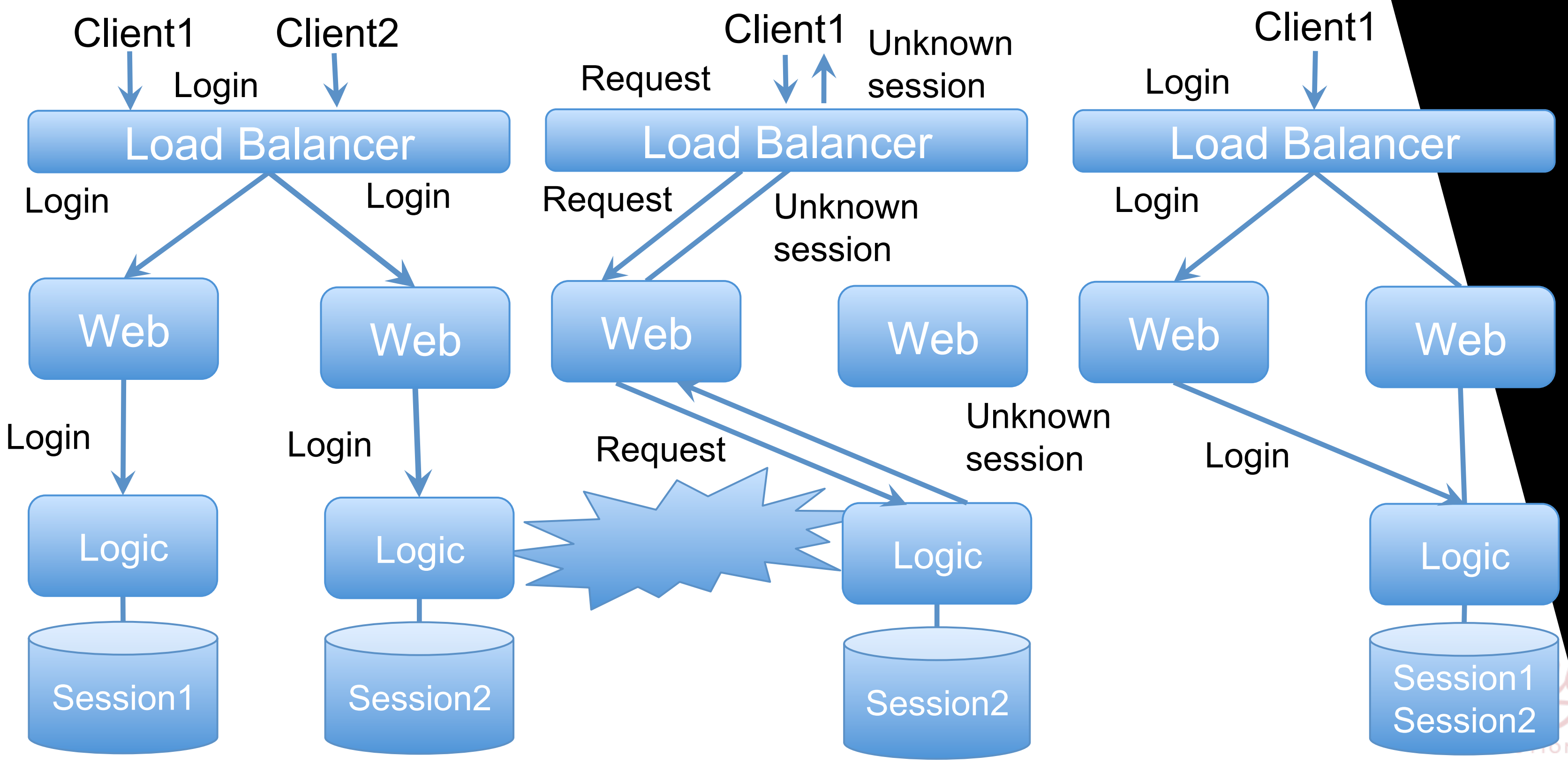


Sharing Data

You have at least two computers to make a fault tolerant system, you need to share state and data.

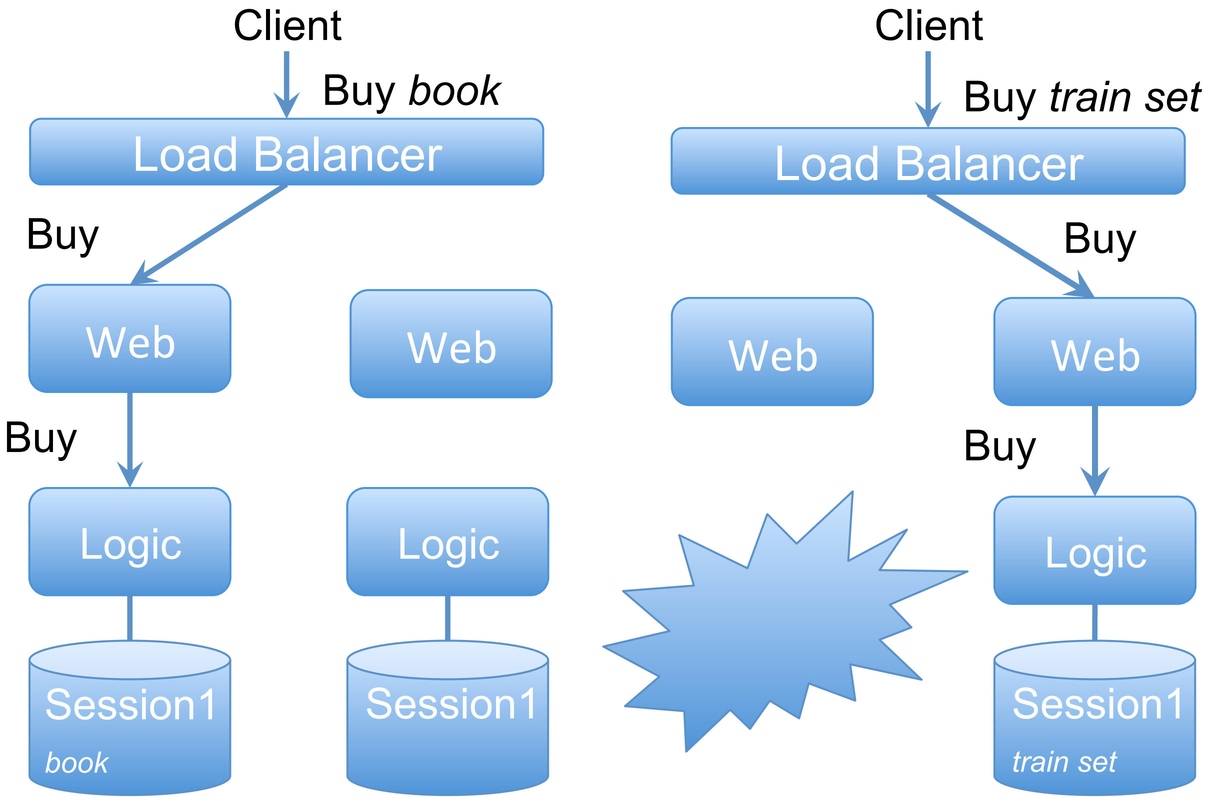
www.erlang-solutions.com

Share Nothing

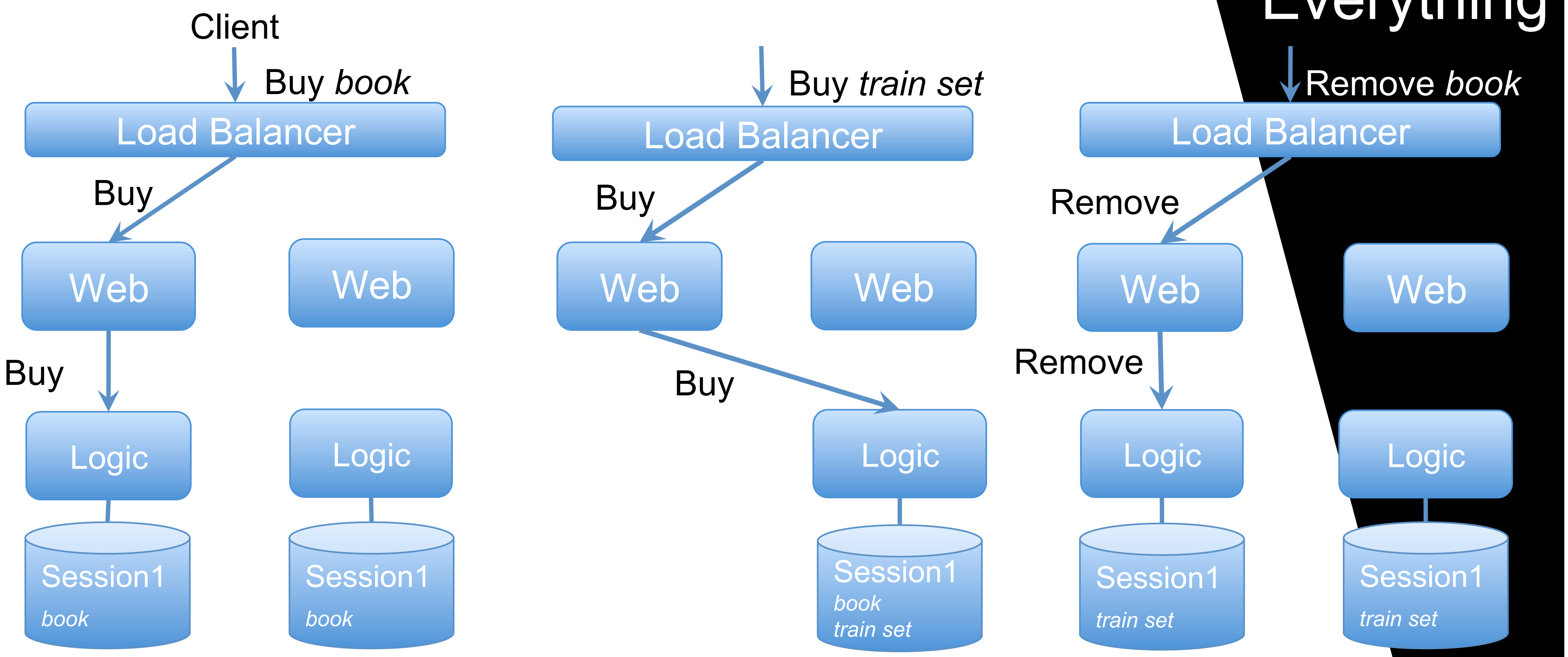


www.erlang-solutions.com

Share Something

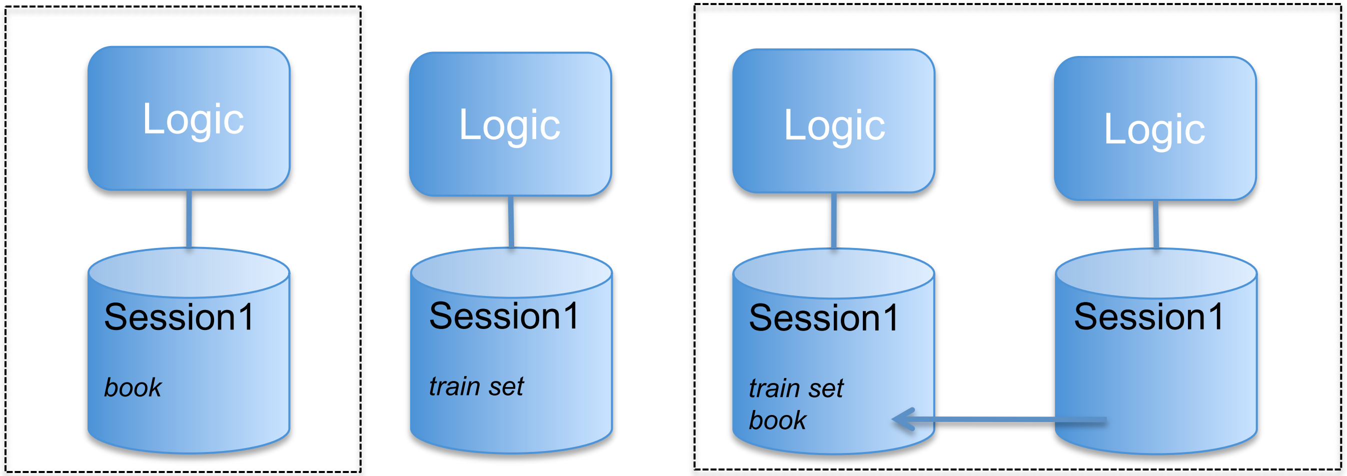


Share Everything



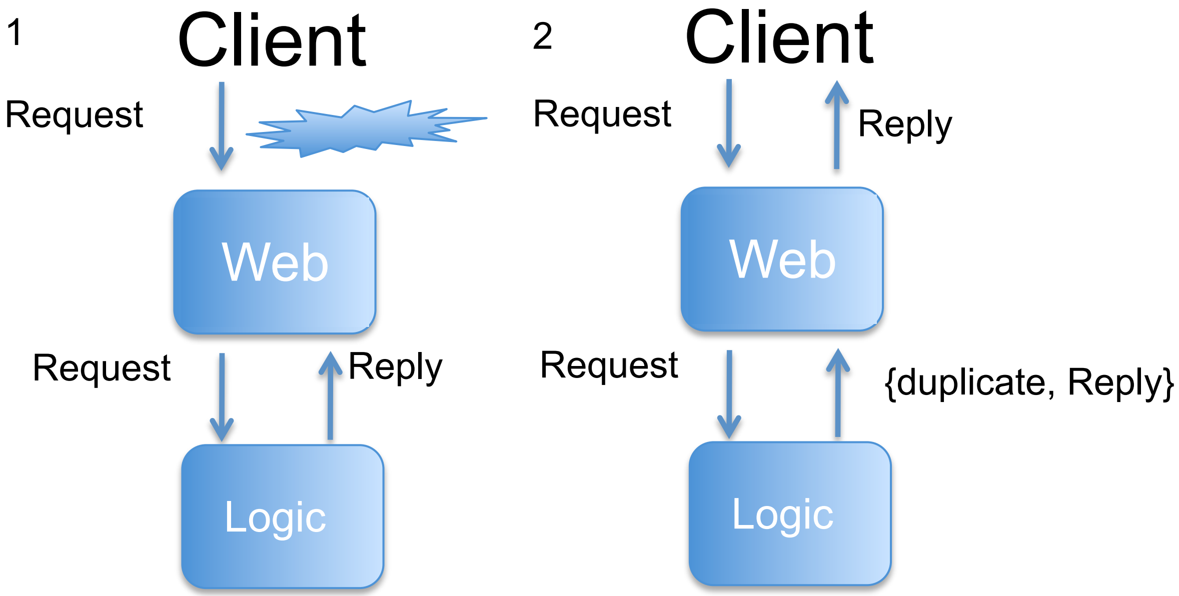
www.erlang-solutions.com

Network Partitions

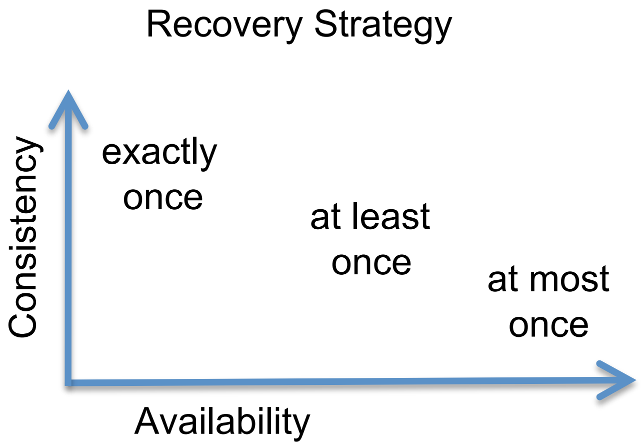


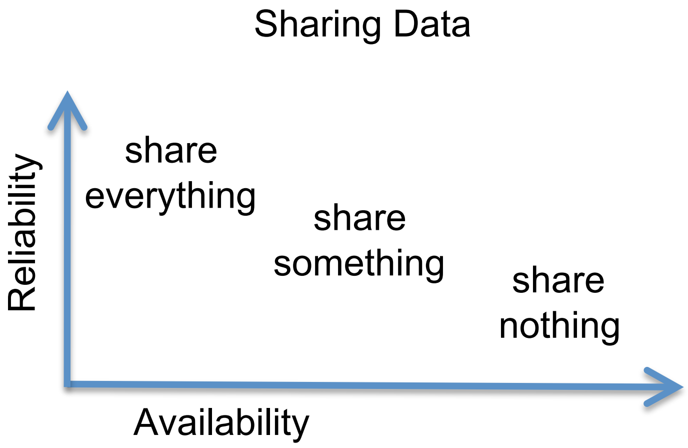
www.erlang-solutions.com

Retry Strategy



Trade-offs


Recovery Strategy
Consistency
exactly once
at least once
at most once
Availability


Sharing Data
Reliability
share everything
share something
share nothing
Availability

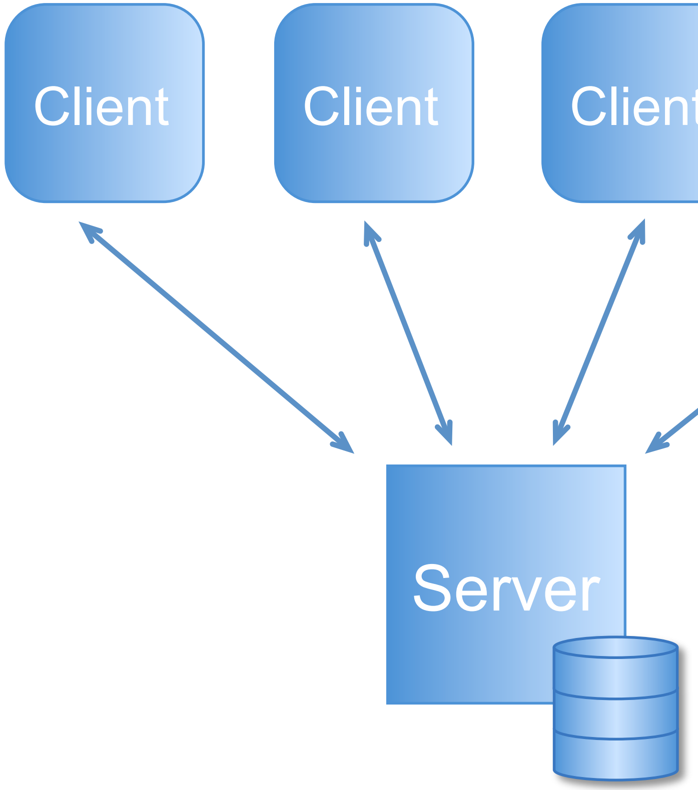
STEPS EVOLVING AROUND AVAILABILITY, CONSISTENCY & RELIABILITY

5. For every interface function in your nodes, you need to pick a retry strategy.
6. For all your data and state, pick your sharing strategy across node families, clusters and types, taking into consideration the needs of your retry strategy.

Scaling Out

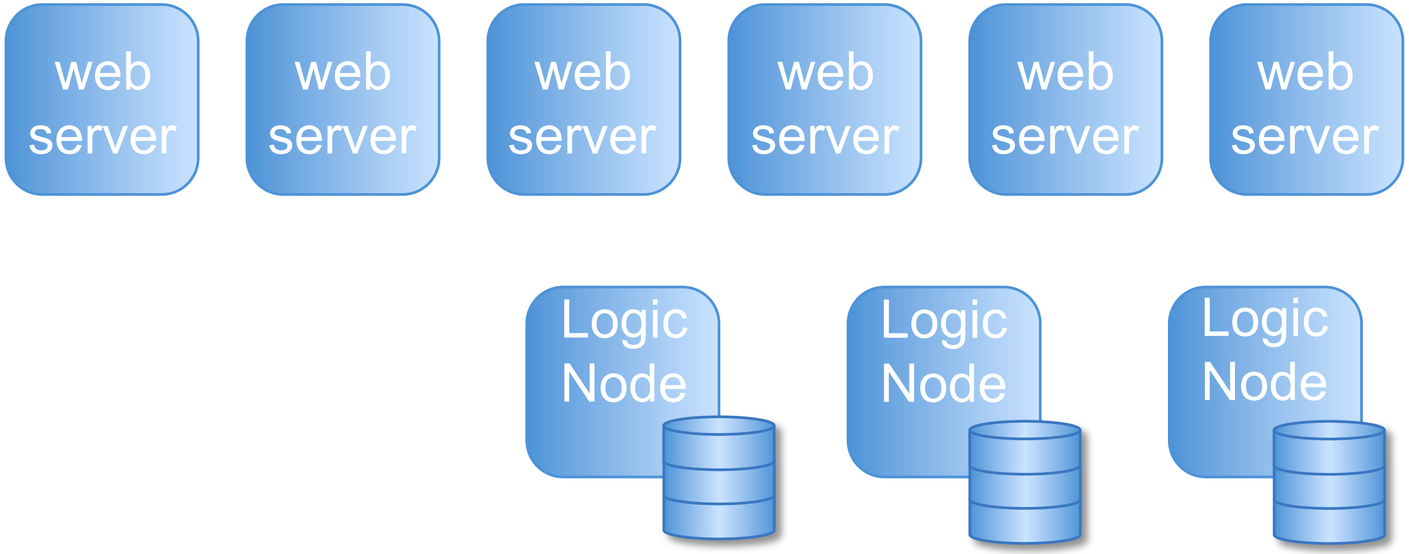
Distribute for scale and replicate for availability.

Scaling Vertically

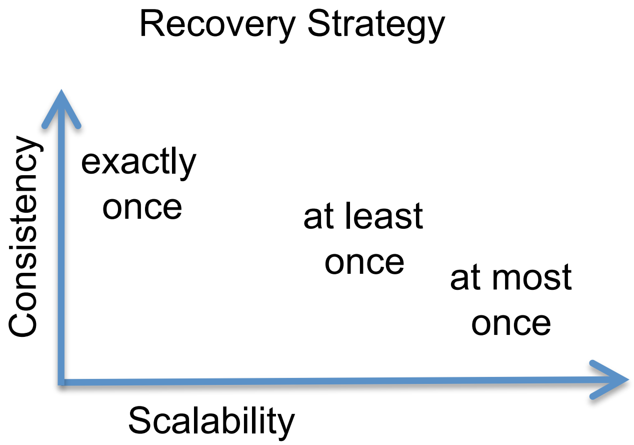


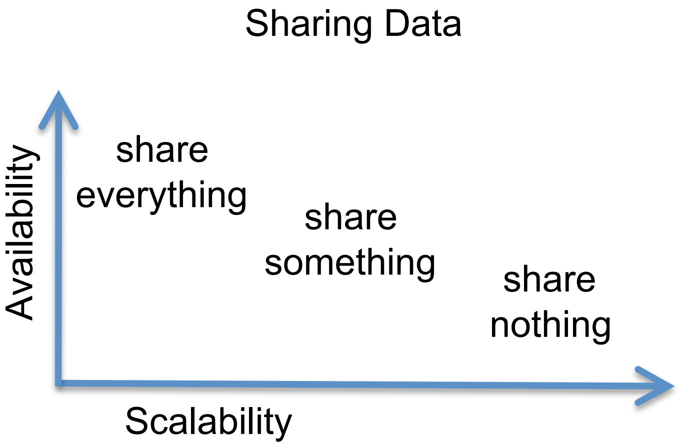
www.erlang-solutions.com

Scaling Horizontally



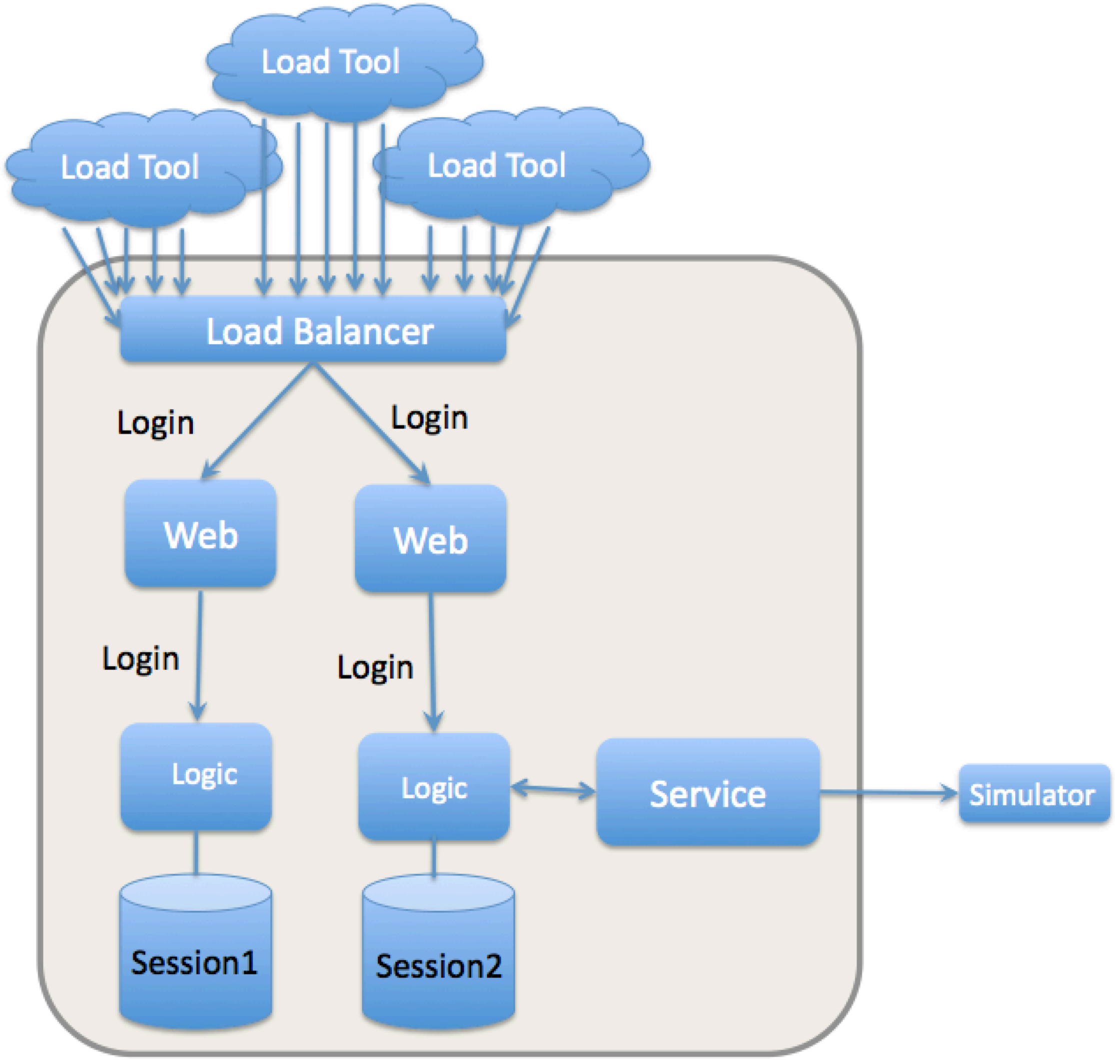
Trade-offs


Recovery Strategy
Consistency
exactly once
at least once
at most once
Scalability

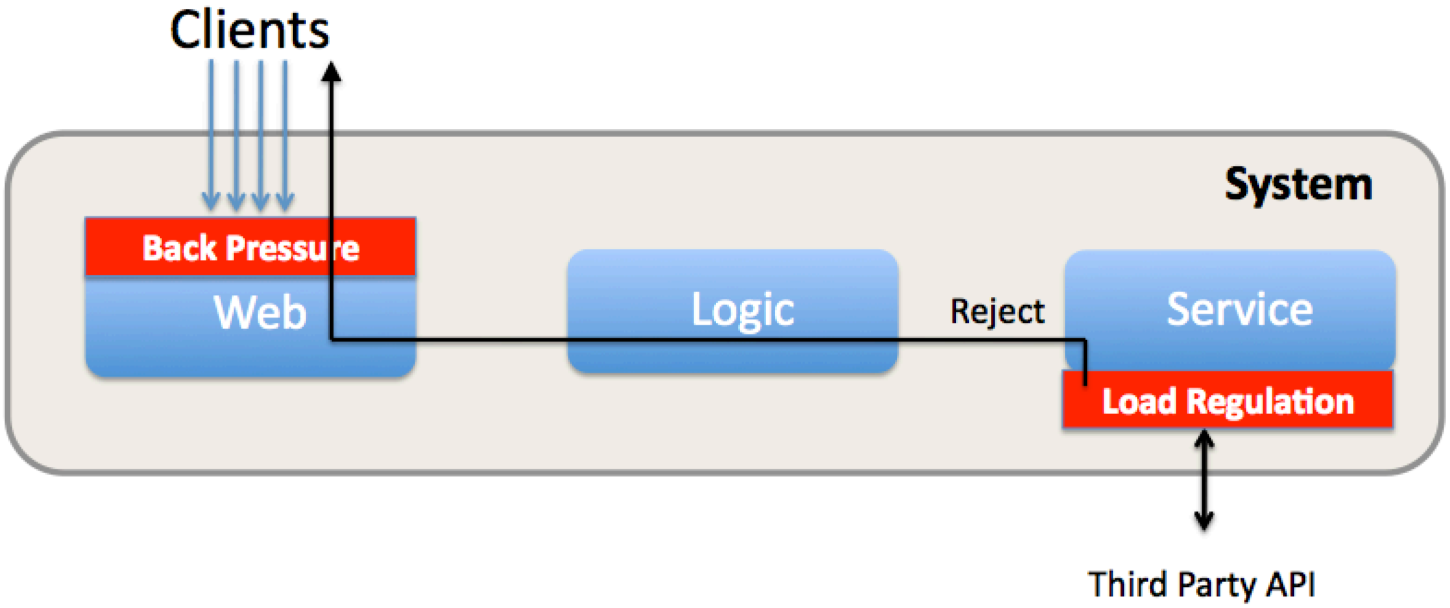

Sharing Data
Availability
share everything
share something
share nothing
Scalability

www.erlang-solutions.com

Capacity Planning



Capacity Planning



- CAPACITY PLANNING -

Capacity planning is the design phase which guarantees that your system can withstand the load it was built to handle, and with time, scaling to handle increased demand.

- No single point of failure
- Cluster blueprint for scalability
- Load Regulation
- Back Pressure

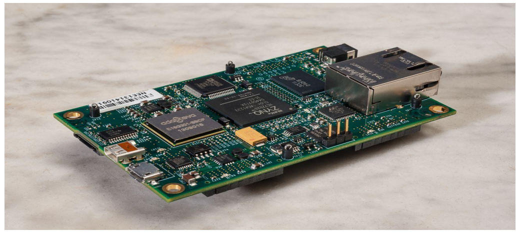
Monitoring and Preemptive Support

With the right tools and approach, the five nines once reserved for Telecom systems are now easily attainable in whatever other vertical for which you might be developing software.

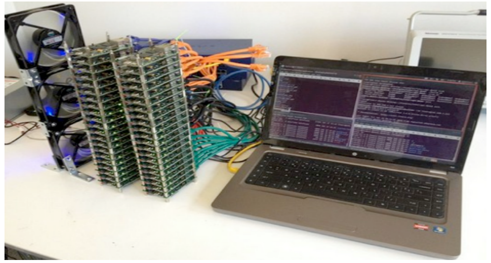
Heterogeneous multi-core hardware is here to stay

- Different cores doing different things
- CPUs, GPUs, FPGA

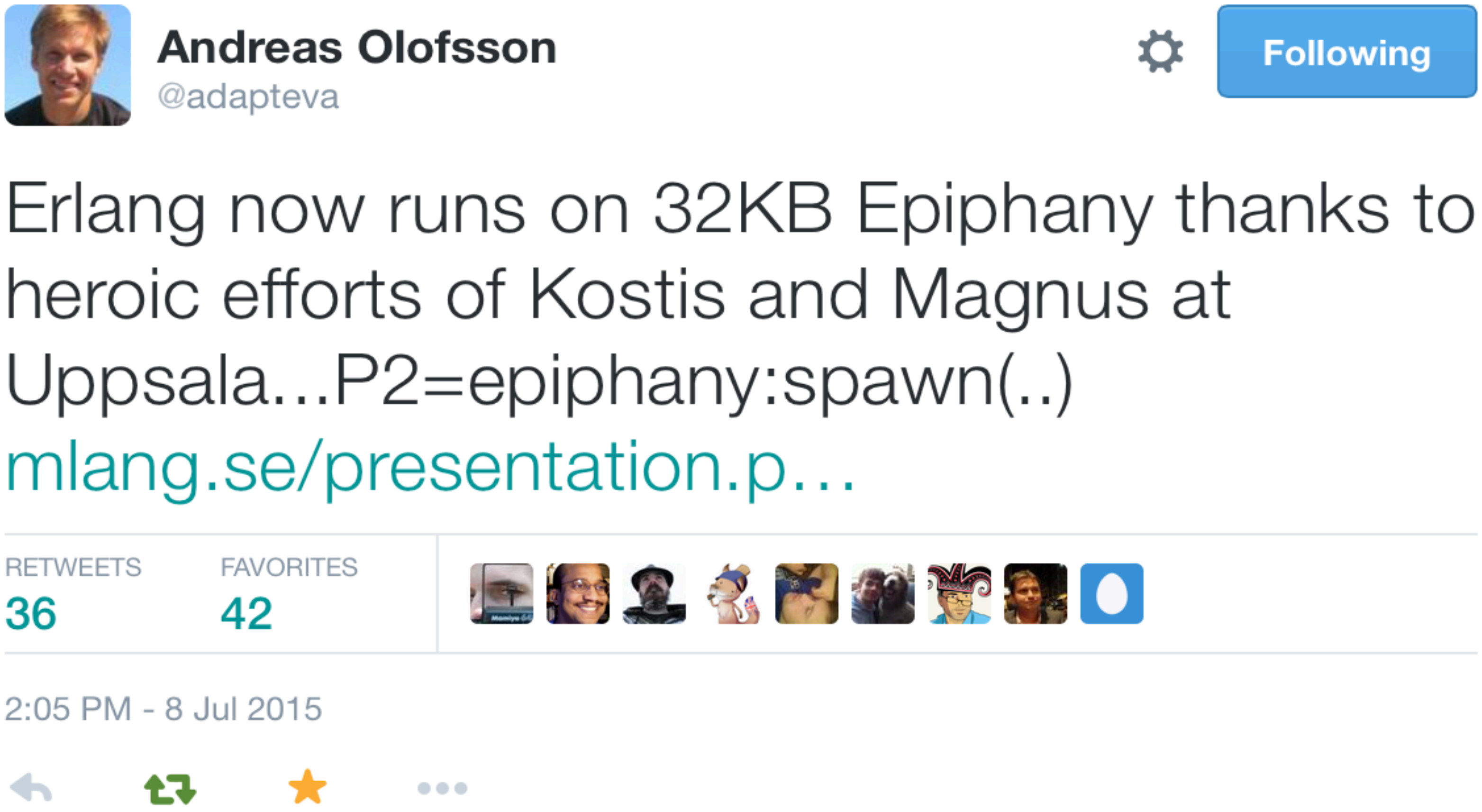
Parallella Board
Dual core ARM processor + FPGA
1GB RAM + MicroSD Card



16 or 64 core Epiphany co-processor
Gigabit Ethernet
2x USB ports + HDMI port



Heterogeneous multi-core hardware is



Andreas Olofsson
@adapteva

Erlang now runs on 32KB Epiphany thanks to heroic efforts of Kostis and Magnus at Uppsala...P2=epiphany:spawn(..) mlang.se/presentation.p…

RETWEETS 36
FAVORITES 42

2:05 PM - 8 Jul 2015

The Fastest Computer in the World!

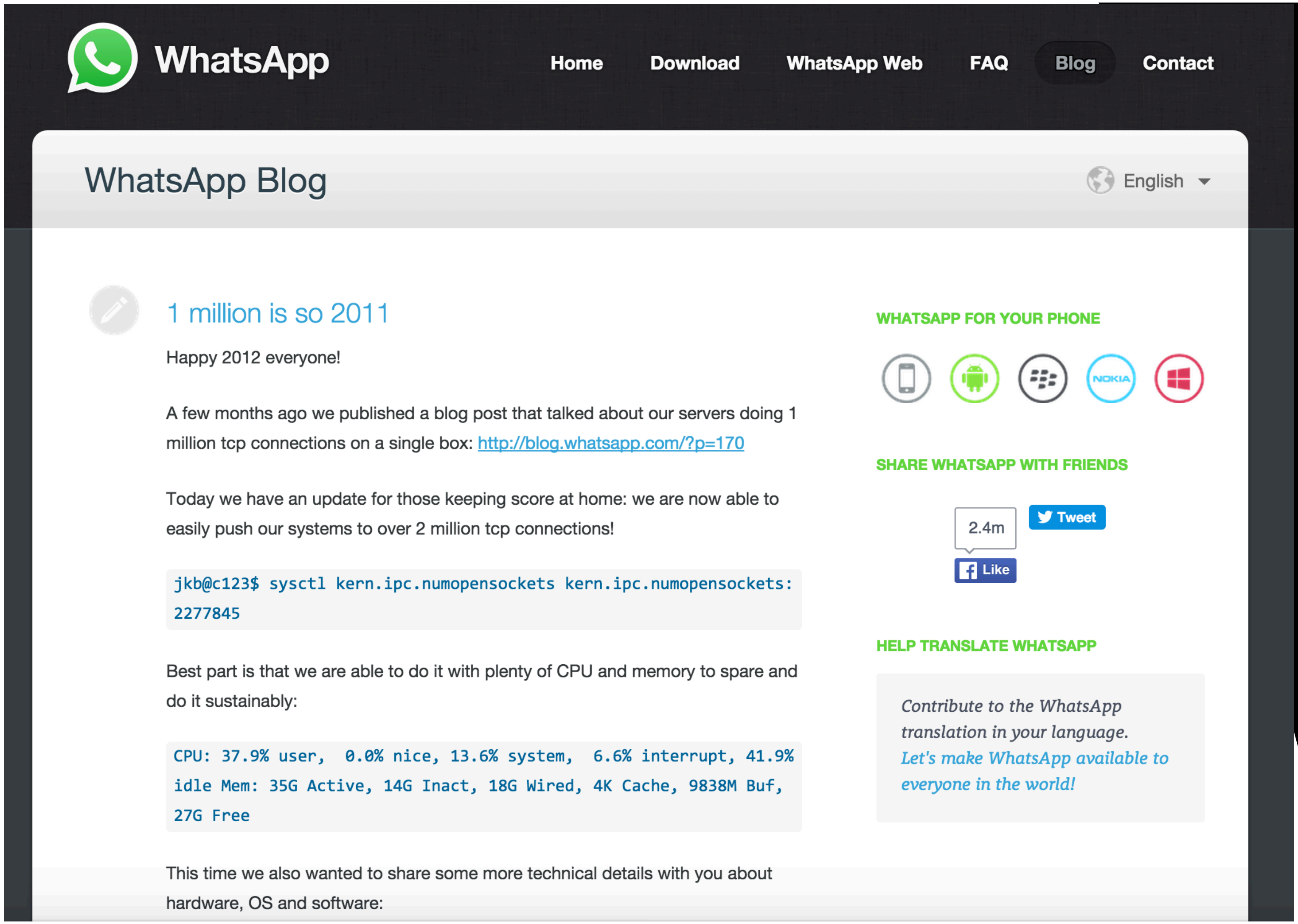
- Tianhe-2
- Chinese National University of Defence Technology

- 33.86 petaflops/s (November 20
- 16,000 Nodes, each with 2 Ivy Br multicores and 3 Xeon Phis
- 3,120,000 x86 cores in total

The Fastest Computer in the World!

- Tianhe-2
- Chinese National University of Defence Technology

- 33.86 petaflops/s (November 20
- 16,000 Nodes, each with 2 Ivy Br multicores and 3 Xeon Phis
- 3,120,000 x86 cores in total



WhatsApp Blog

1 million is so 2011

Happy 2012 everyone!

A few months ago we published a blog post that talked about our servers doing 1 million tcp connections on a single box: http://blog.whatsapp.com/?p=170

Today we have an update for those keeping score at home: we are now able to easily push our systems to over 2 million tcp connections!

```
jkb@c123$ sysctl kern.ipc.numopensockets kern.ipc.numopensockets:
2277845
```

Best part is that we are able to do it with plenty of CPU and memory to spare and do it sustainably:

```
CPU: 37.9% user,  0.0% nice, 13.6% system,  6.6% interrupt, 41.9%
idle Mem: 35G Active, 14G Inact, 18G Wired, 4K Cache, 9838M Buf,
27G Free
```

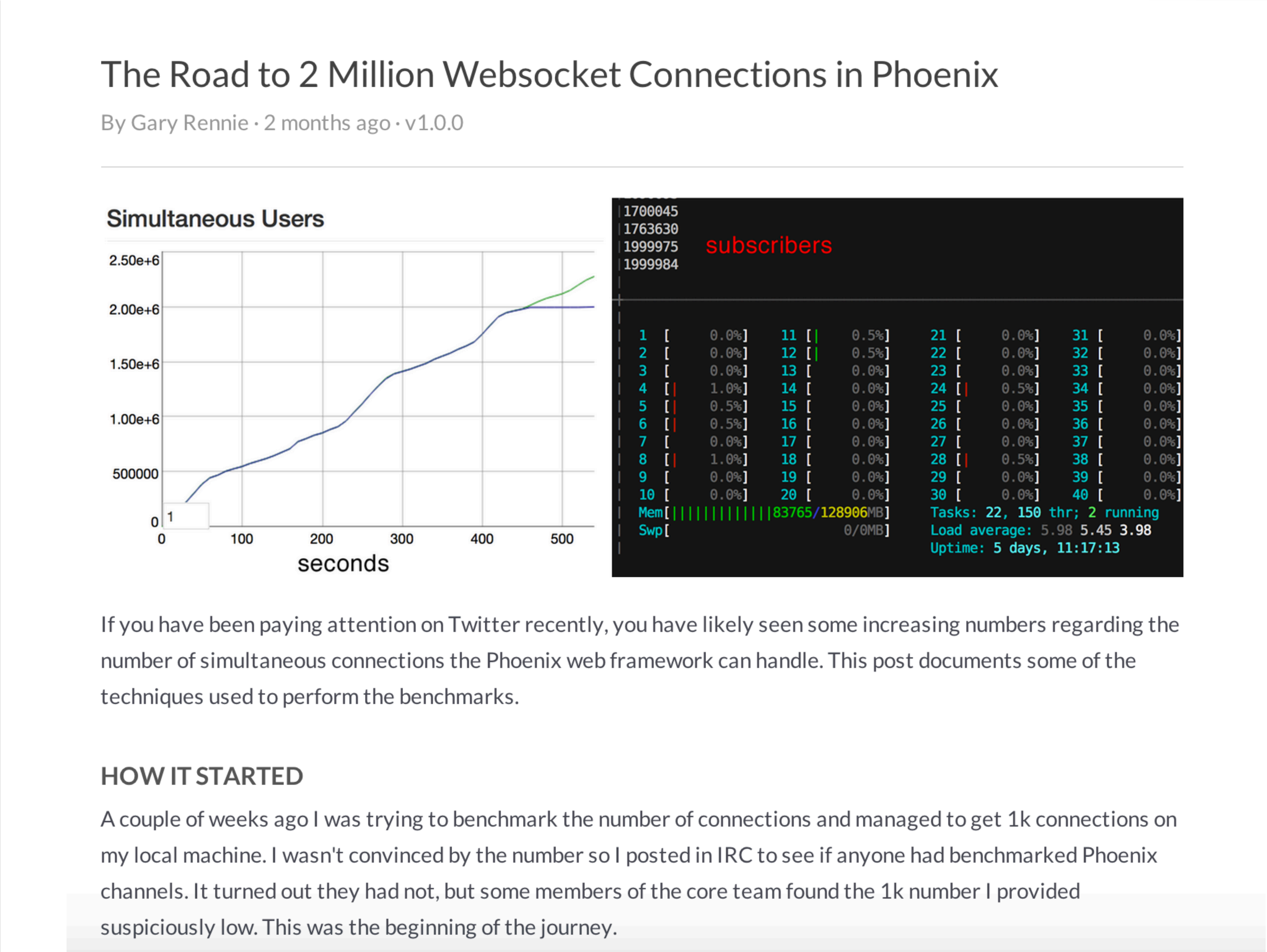
This time we also wanted to share some more technical details with you about hardware, OS and software:

WHATSAPP FOR YOUR PHONE

SHARE WHATSAPP WITH FRIENDS

HELP TRANSLATE WHATSAPP

Contribute to the WhatsApp translation in your language. Let's make WhatsApp available to everyone in the world!



The Road to 2 Million Websocket Connections in Phoenix

By Gary Rennie · 2 months ago · v1.0.0

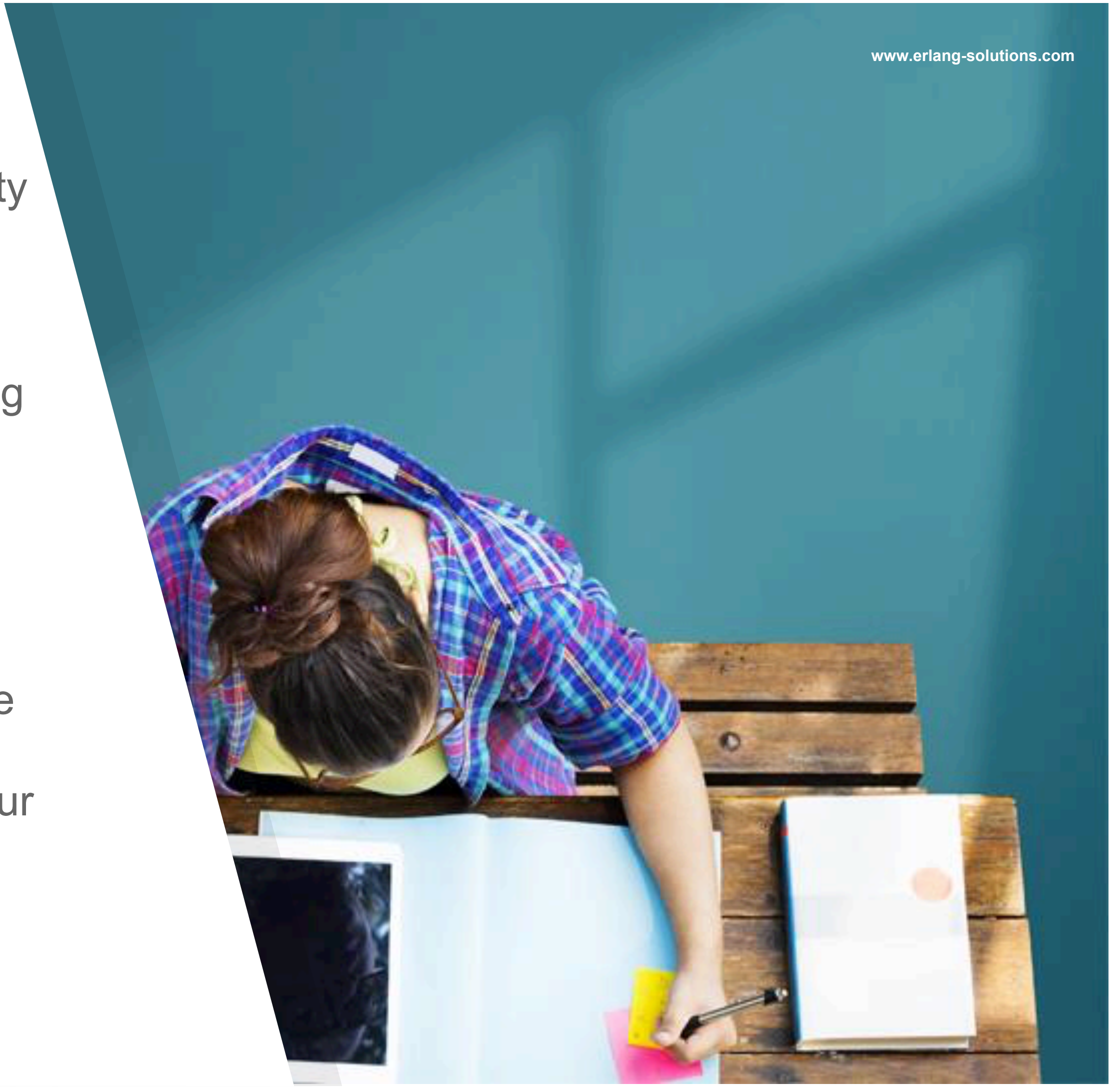
```
|1700045
|1763630
|1999975
|1999984
|
+
|
|  1  [        0.0%]    11 [|       0.5%]     21 [        0.0%]    31 [        0.0%]
|  2  [        0.0%]    12 [|       0.5%]     22 [        0.0%]    32 [        0.0%]
|  3  [        0.0%]    13 [        0.0%]     23 [        0.0%]    33 [        0.0%]
|  4  [|       1.0%]    14 [        0.0%]     24 [|       0.5%]    34 [        0.0%]
|  5  [|       0.5%]    15 [        0.0%]     25 [        0.0%]    35 [        0.0%]
|  6  [|       0.5%]    16 [        0.0%]     26 [        0.0%]    36 [        0.0%]
|  7  [        0.0%]    17 [        0.0%]     27 [        0.0%]    37 [        0.0%]
|  8  [|       1.0%]    18 [        0.0%]     28 [|       0.5%]    38 [        0.0%]
|  9  [        0.0%]    19 [        0.0%]     29 [        0.0%]    39 [        0.0%]
|  10 [        0.0%]    20 [        0.0%]     30 [        0.0%]    40 [        0.0%]
|  Mem[||||||||||||83765/128906MB]     Tasks: 22, 150 thr; 2 running
|  Swp[                    0/0MB]     Load average: 5.98 5.45 3.98
|                                      Uptime: 5 days, 11:17:13
```

If you have been paying attention on Twitter recently, you have likely seen some increasing numbers regarding the number of simultaneous connections the Phoenix web framework can handle. This post documents some of the techniques used to perform the benchmarks.

HOW IT STARTED

A couple of weeks ago I was trying to benchmark the number of connections and managed to get 1k connections on my local machine. I wasn't convinced by the number so I posted in IRC to see if anyone had benchmarked Phoenix channels. It turned out they had not, but some members of the core team found the 1k number I provided suspiciously low. This was the beginning of the journey.

1. Split up your system's functionality into manageable, stand-alone nodes.
2. Decide what distributed architectural pattern you are going to use.
3. Decide what network protocols your nodes, node families and clusters will use when. communicating with each other.
4. Define your node interfaces, state and data model.
5. For every interface function in your nodes, you need to pick a retry strategy.



1. For all your data and state, pick your sharing strategy across node families, clusters and types, taking

and data model.

5. Split up your system's functionality into manageable, stand-alone nodes.

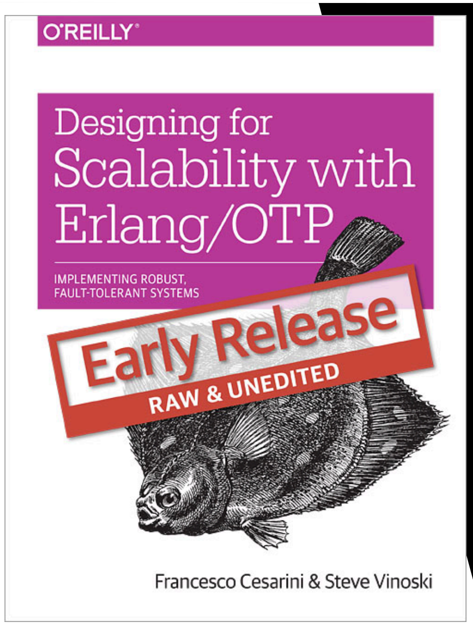
6. For all your data and state, pick your sharing strategy across node families, clusters and types, taking into consideration the needs of your retry strategy.
7. Reiterate through steps 1, 2, 3, 4, 5 & 6 until you have the trade-offs which suit your specification.
8. Design your cluster blueprint, looking at node ratios for scaling up and down.
9. Indentify where to apply back-pressure and load regulation.
10. Define your O&M approach, defining system and business alarms, logs and metrics.

www.erlang-solutions.com

THANK YOU!

Any questions?

francesco@erlang-solutions.com
www.erlang-solutions.com
@francescoC



O'REILLY

Designing for Scalability with Erlang/OTP

IMPLEMENTING ROBUST, FAULT-TOLERANT SYSTEMS

Early Release
RAW & UNEDITED

Francesco Cesarini & Steve Vinoski

Discount Code: **authd**
50% off the Early Release
40% off the printed copy

Erlang
SOLUTIONS