

# Actor Model in F# and Akka.Net



“Although threads seem to be a small step from sequential computation, in fact, they represent a huge step. They discard the most essential and appealing properties of sequential computation: understandability, predictability, and determinism. Threads, as a model of computation, are wildly nondeterministic, and the job of the programmer becomes one of pruning that non-determinism.”

— Edward A. Lee

(*The Problem with Threads, Berkeley 2006*)

Riccardo Terrell – LambdaCon 2015

# Agenda

What kind of World is out there?

Reactive Manifesto

Actor model - What & Why

F# Agent

Akka.Net in Action

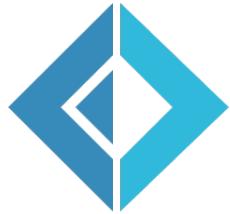
Akka.NET F# API

Code & Code

# Something about me – Riccardo Terrell

- Originally from Italy, currently living in USA ~8 years
  - Living/working in Washington DC
- +/- 16 years in professional programming
  - C++/VB → Java → .Net C# → C# & F# → ??
- Organizer of the DC F# User Group
- Software Architect @ Microsoft
- Passionate in Technology, believe in polyglot programming as a mechanism in finding the right tool for the job

# Goals - Plan of the talk

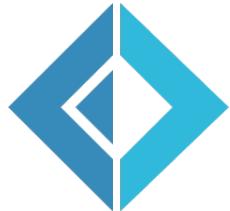


The free lunch is over,  
we are facing new  
challenges in today's  
Programming World

Today, applications must  
be built with concurrency  
in mind  
(possibly from the beginning)

Actor is ~~the best~~ a great  
concurrent programming  
model that solves  
problems for Scaling Up  
& Out

# Hottest technology trends



Cloud Computing

Reactive Application

Real Time Notification

NoSql

Big Data

Scalable

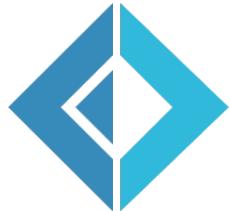
Maintainable & Readable

Reliable & Testable

Composable & Reusable

Concurrent

# The World is changed



*"In today's world, the demand for such as*

*come from*

*"Modern applica*

*- Real-time App*

*NA".  
n Development (Manning)*

**We are in a multicore  
cloud distributed era**

*higher expectations*

*companies have*

*on.*

*porating this behavior into their*



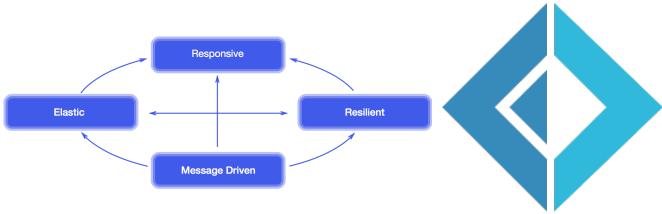
# The Problem..

It is hard to build correct highly concurrent systems

It is hard to build truly scalable systems that scale up and out

It is hard to build resiliant & fault-tolerant systems that self-heal

# Reactive Manifesto



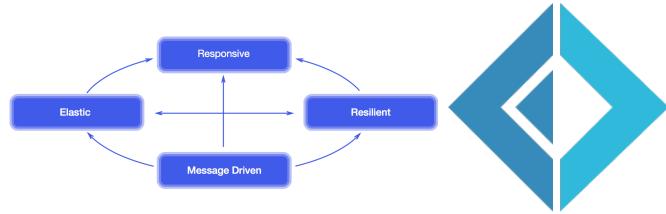
Responsive

Message-Driven

Resilient

Elastic

# Reactive Manifesto



Responsive

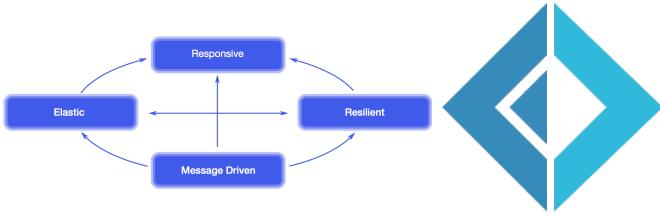
Message-Driven

Resilient

Elastic

The **system responds in a timely manner** if at all possible. Responsiveness is the cornerstone of usability and utility, but more than that, responsiveness means that **problems may be detected quickly** and dealt with effectively. Responsive systems focus on providing **rapid and consistent response times**, establishing reliable upper bounds so they deliver a consistent quality of service. This consistent behavior in turn simplifies error handling, builds end user confidence, and encourages further interaction.

# Reactive Manifesto



Responsive

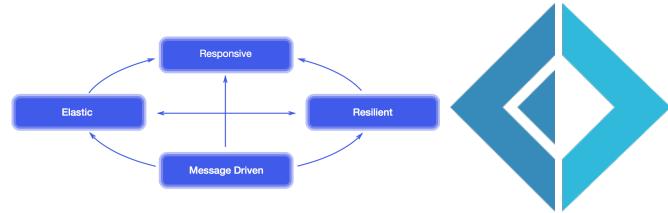
Message-Driven

Resilient

Elastic

Reactive Systems rely on **asynchronous message-passing** to establish a boundary between components that ensures loose coupling, isolation, location transparency, and provides the means to delegate errors as messages. Employing explicit message-passing **enables load management, elasticity, and flow control** by shaping and monitoring the message queues in the system and applying back-pressure when necessary. Location transparent messaging as a means of communication makes it possible for the management of failure to work with the same constructs and semantics across a cluster or within a single host.

# Reactive Manifesto



Responsive

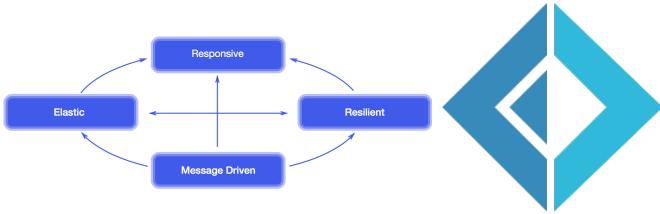
Message-Driven

Resilient

Elastic

The system **stays responsive in the face of failure**. This applies not only to highly-available, mission critical systems — any system that is not resilient will be unresponsive after a failure. Resilience is achieved by replication, containment, isolation and delegation. Failures are contained within each component, isolating components from each other and thereby ensuring that parts of the system can fail and recover without compromising the system as a whole. **Recovery of each component is delegated to another (external) component** and high-availability is ensured by replication where necessary. The client of a component is not burdened with handling its failures.

# Reactive Manifesto



Responsive

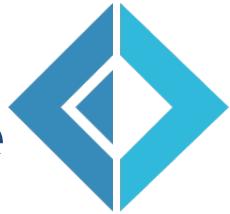
Message-Driven

Resilient

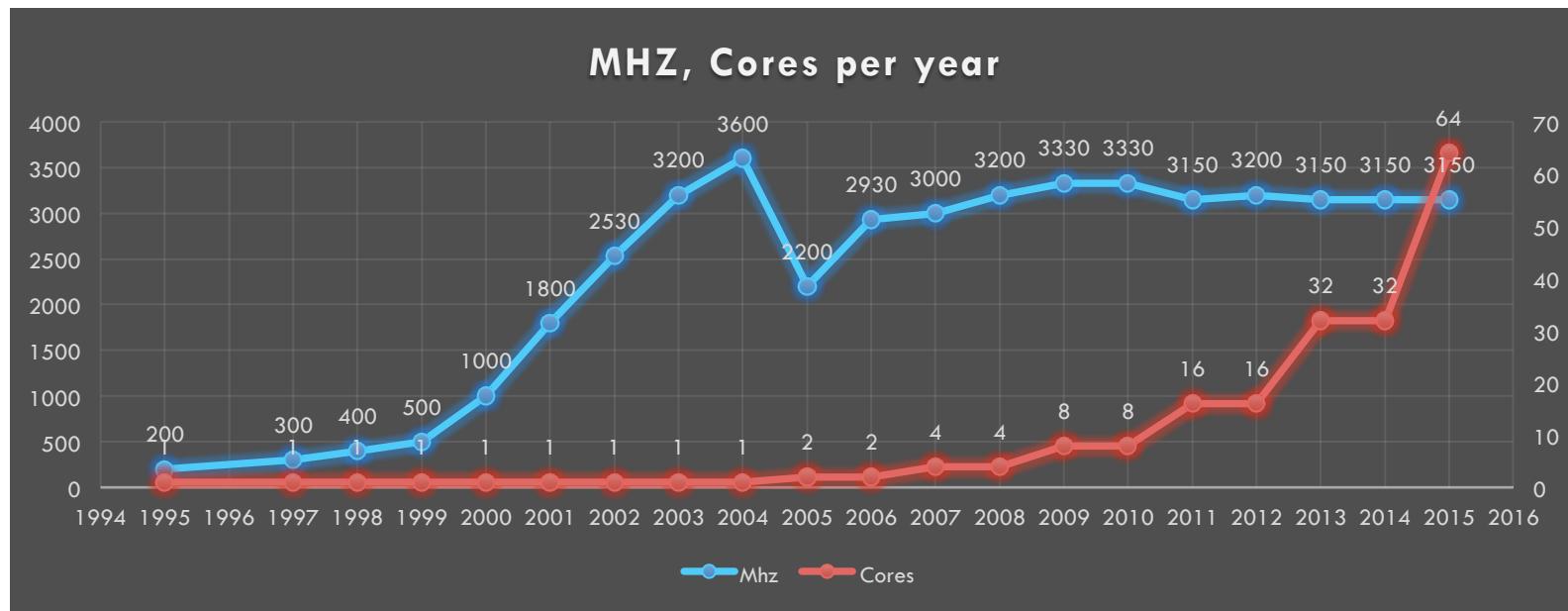
Elastic

The system stays responsive under varying workload. Reactive Systems can react to changes in the input rate by **increasing or decreasing the resources allocated** to service these inputs. This implies designs that have no contention points or central bottlenecks, resulting in the ability to shard or replicate components and distribute inputs among them. Reactive Systems support predictive, as well as Reactive, **scaling algorithms** by providing relevant live performance measures. They achieve elasticity in a cost-effective way on commodity hardware and software platforms.

# Moore's law - The Concurrency challenge



*Modern computers come with multiple processors, each equipped with multiple cores, but the single processor is slower than it used to be!*

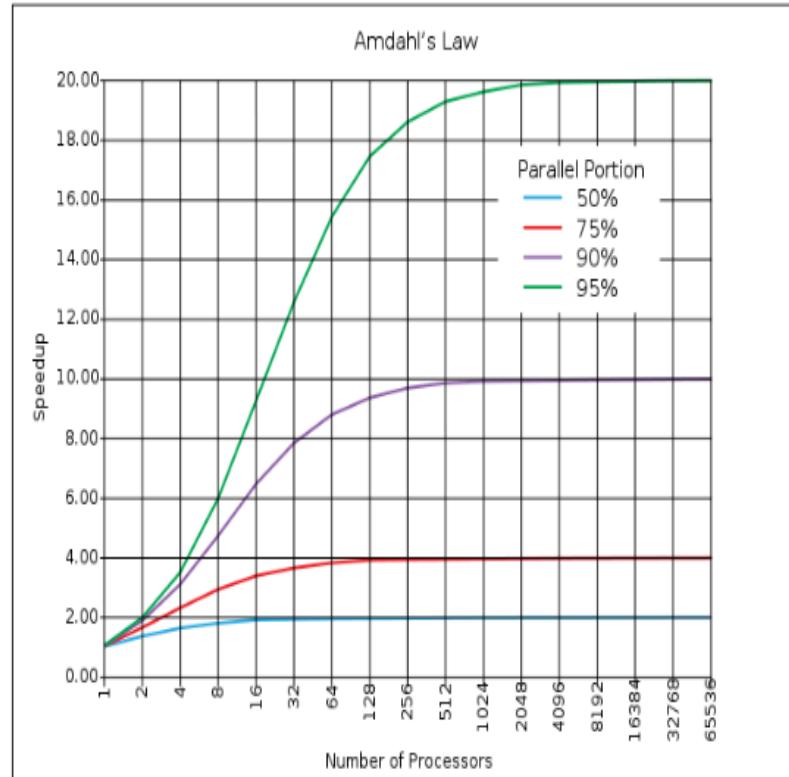


# The new reality : Amdahl's law



The speed-up of a program using multiple processors in parallel computing is limited by the sequential fraction of the program.

For example if 95% of the program can be parallelized, the theoretical maximum speedup using parallel computing would be 20 times, no matter how many processors are used



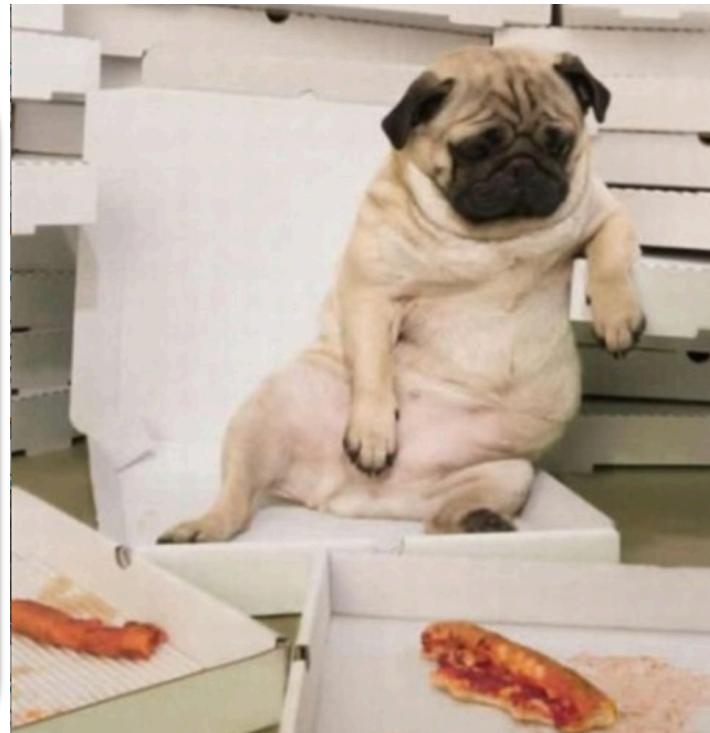


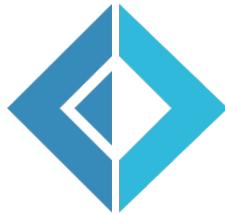
# The free lunch is over

*There is a problem...*

**the free lunch is over**

- Programs are not doubling in speed every couple of years for free anymore*
  
- We need to start writing code to take advantage of many cores*

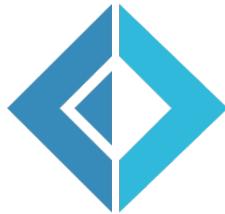




# The issue is Shared of Memory



- Shared Memory Concurrency
- Data Race / Race Condition
- Works in sequential single threaded environment
- Not fun in a multi-threaded environment
- Not fun trying to parallelize
- Locking, blocking, call-back hell



# The Solution is Immutability and Isolation

IMMUTABILITY +  
ISOLATION =

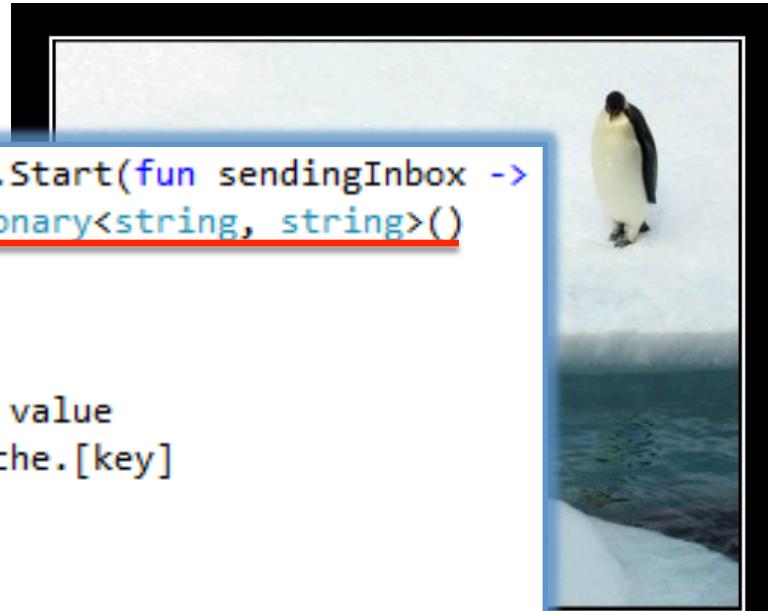
---

**BEST CONCURRENT MODEL PROGRAMMING**



# Immutability OR Isolation

```
let (lockfree:Agent<Msg<string,string>>) = Agent.Start(fun sendingInbox ->
    let cache = System.Collections.Generic.Dictionary<string, string>()
    let rec loop () = async {
        let! message = sendingInbox.Receive()
        match message with
            | Push (key,value) -> cache.[key] <- value
            | Pull (key,fetch) -> fetch.Reply cache.[key]
        return! loop ()
    }
    loop ())
```

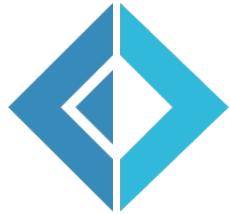


## ISOLATION

That awkward moment when you realize by trying to exclude someone you gave them the better iceberg

# What is

The Actor model  
using actor objects  
of actors, interacting  
interaction  
**passing** with



creation  
creation  
s, and  
age  
er.  
kipedia]

## Cars

It will be ok it's a rental.



# What is an Actor?



- Processing
- Storage – State
- Communication only by messages
- Share Nothing
- Message are passed by value
- Lightweight object.
- Running on it's own thread.
- No shared state.
- Messages are kept in mailbox and processed in order.
- Massive scalable and lightening fast because of the small call stack.

# Carl Hewitt's Actor Model

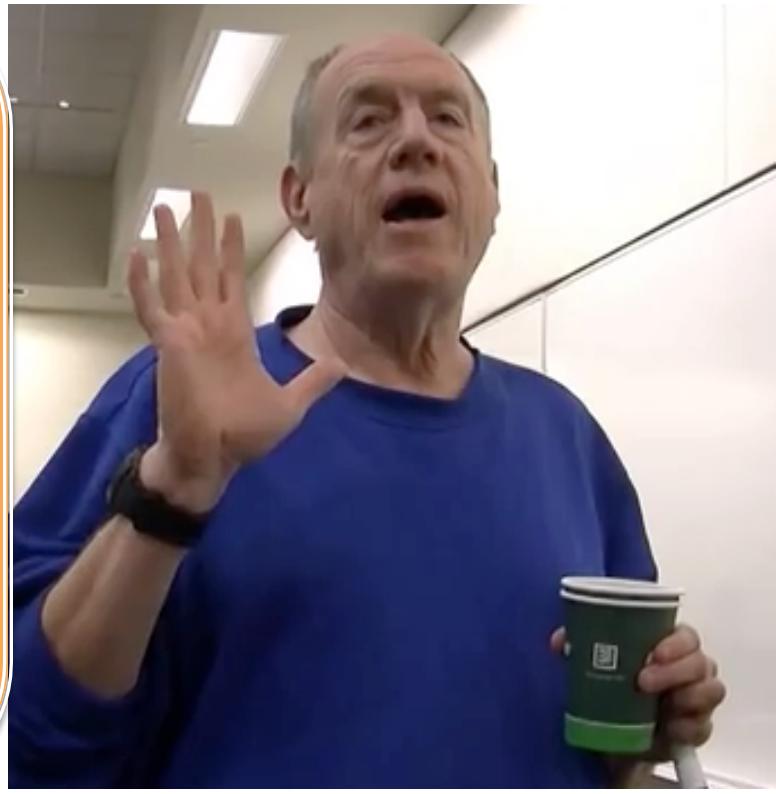
"An island of sanity in a sea of concurrency"

"Shared nothing", "Black box"

"Location transparent", "Distributable by design"

## **Actor Model Three axioms:**

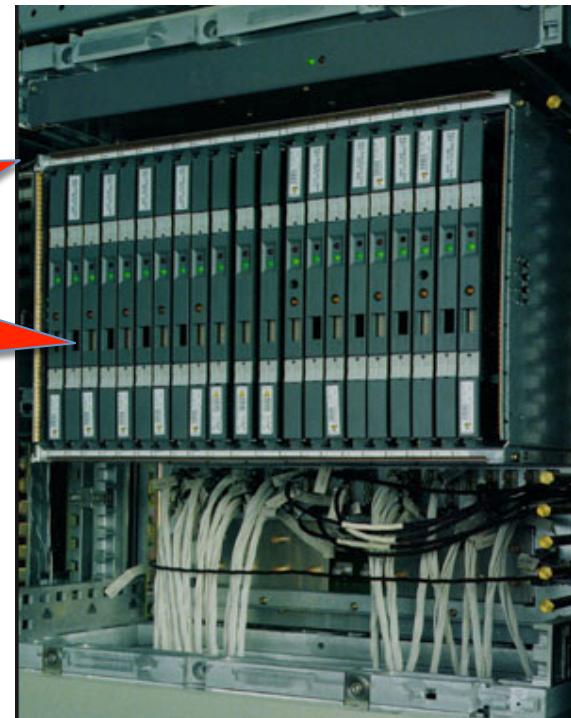
1. Send messages to other Actors
  - One Actor is not Actor -
2. Create other Actor
3. Decide how to handle next message



# Ericson AXD 301 Switch - 1996



99.999999  
percent  
uptime



# What problems Akka can solve?



Distributed System

Akka provides high-level abstractions for concurrency and parallelism.

Asynchronous

Multiple servers are capable of handling requests from clients in case any one of them is unavailable for any reason. The code throughout the application must NOT focus only on the details of **sending** and **receiving** remote messages. The code must be declarative and not full of details about how an operation is to be done, but explaining what is to be done.

High Performance

Akka gives us that ability by making the location of actors transparent across nodes.

# What problems Akka can solve?



Distributed System

Asynchronous

High Performance

Akka provides an Asynchronous, non-blocking and highly performant event-driven programming model. Asynchrony can have benefits both within a single machine and across a distributed architecture. In a single node, it is entirely possible to have tremendous throughput by organizing logic to be synchronous and pipelined. With asynchronous programming, we are attempting to solve the problem of not pinning threads of execution to a particular core, but instead allowing all threads access in a varying model of fairness. Asynchronicity provides a way for the hardware to be able to utilize cores to the fullest by staging work for execution.

# What problems Akka can solve?



Distributed System

Asynchronous

High Performance

Akka has the ability to handle tremendous loads very fast while at the same time being fault tolerant. Building a distributed system that is extremely fast but incapable of managing failure is virtually useless: failures happen, particularly in a distributed context (network partitions, node failures, etc.), and resilient systems are able deal with them. But no one wants to create a resilient system without being able to support reasonably fast execution.

Very lightweight event-driven processes

Scalable System - if my system is **fast** for a single user but **slow** when the system is under **heavy load**



# Reactive Manifesto & Actor Model

Responsive

Event Driven

Message-Driven

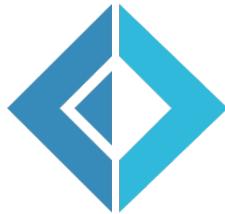
Communication by messages

Resilient

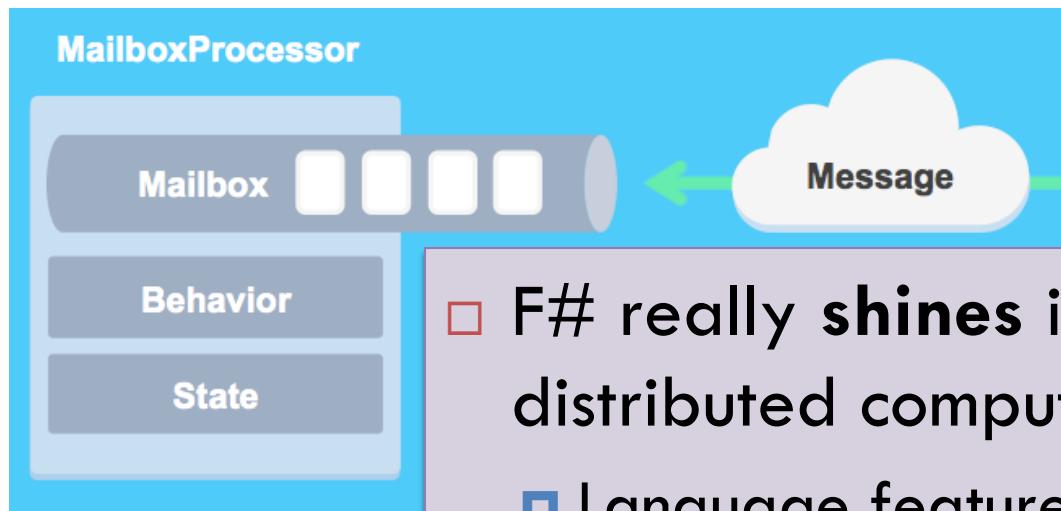
Fault tolerant by Supervision

Elastic

Clustering and Remoting across multiple machines



# F# MailboxProcessor – aka Agent



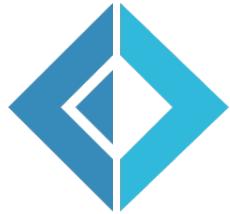
- F# really **shines** in the area of distributed computing
  - Language features such as **Async Workflow** and **MailboxProcessor** (a.k.a. agent) open the doors for computing that focuses on message passing concurrency



# F# MailboxProcessor – aka Agent

```
let agent = MailboxProcessor<Message>.Start(fun inbox ->
    let rec loop n = async {
        let! msg = inbox.Receive()
        match msg with
        | Add(i) -> return! loop (n + i)
        | Get(r) -> r.Reply(n)
                      return! loop n }
    loop 0)
```

# F# Agent – Few problems



- F# agents **do not** work across boundaries
  - only within the same process
- F# agent are not instances of Actor interface
  - can't be used in distributed systems without explicit serialization
- No build-in durability or persistence
- F# agent doesn't have any routing and supervision functionality out of the box

Agent is not Actor

# What is Akka.NET



Akka.NET is a port of the popular Java/Scala framework Akka to .NET.

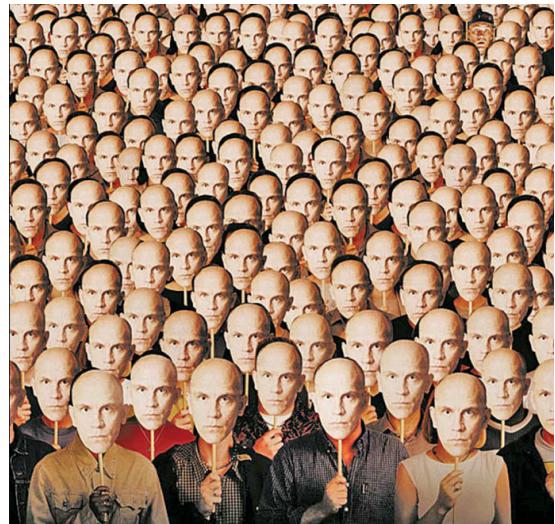
*“Akka is a toolkit and runtime for building highly concurrent, distributed, and resilient message-driven applications on the JVM.”*

- Typesafe

# Multi-Threads vs Multi Actors



*1 MB per thread (4 Mb in 64 bit) vs 2.7 million Actors per Gigabyte*



# How A

How do you keep  
things go wrong

The s

How can the Sup

There are  
- How the  
- Parent's



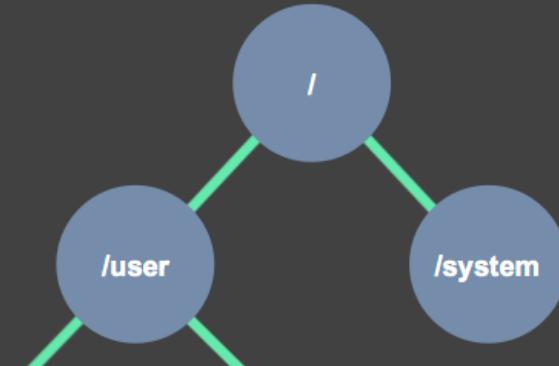
KEEP  
CALM  
AND  
LET IT  
CRASH

# ailure?

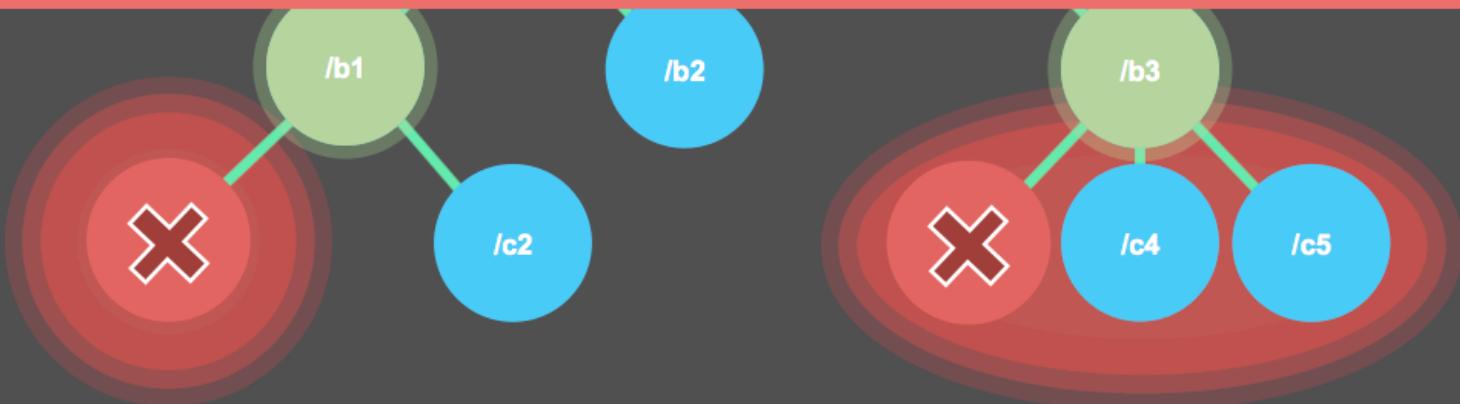


ved:

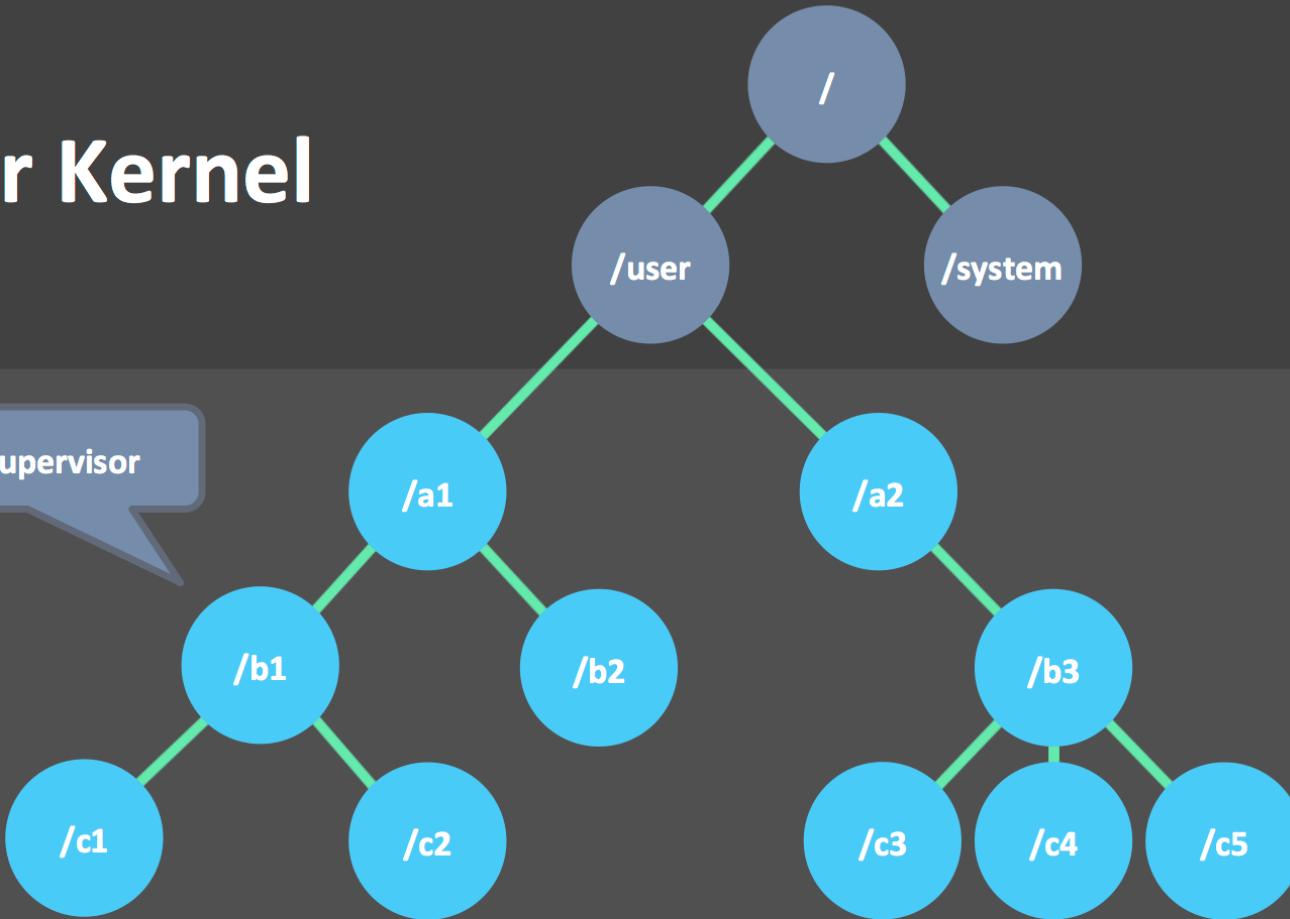
# Error Kernel



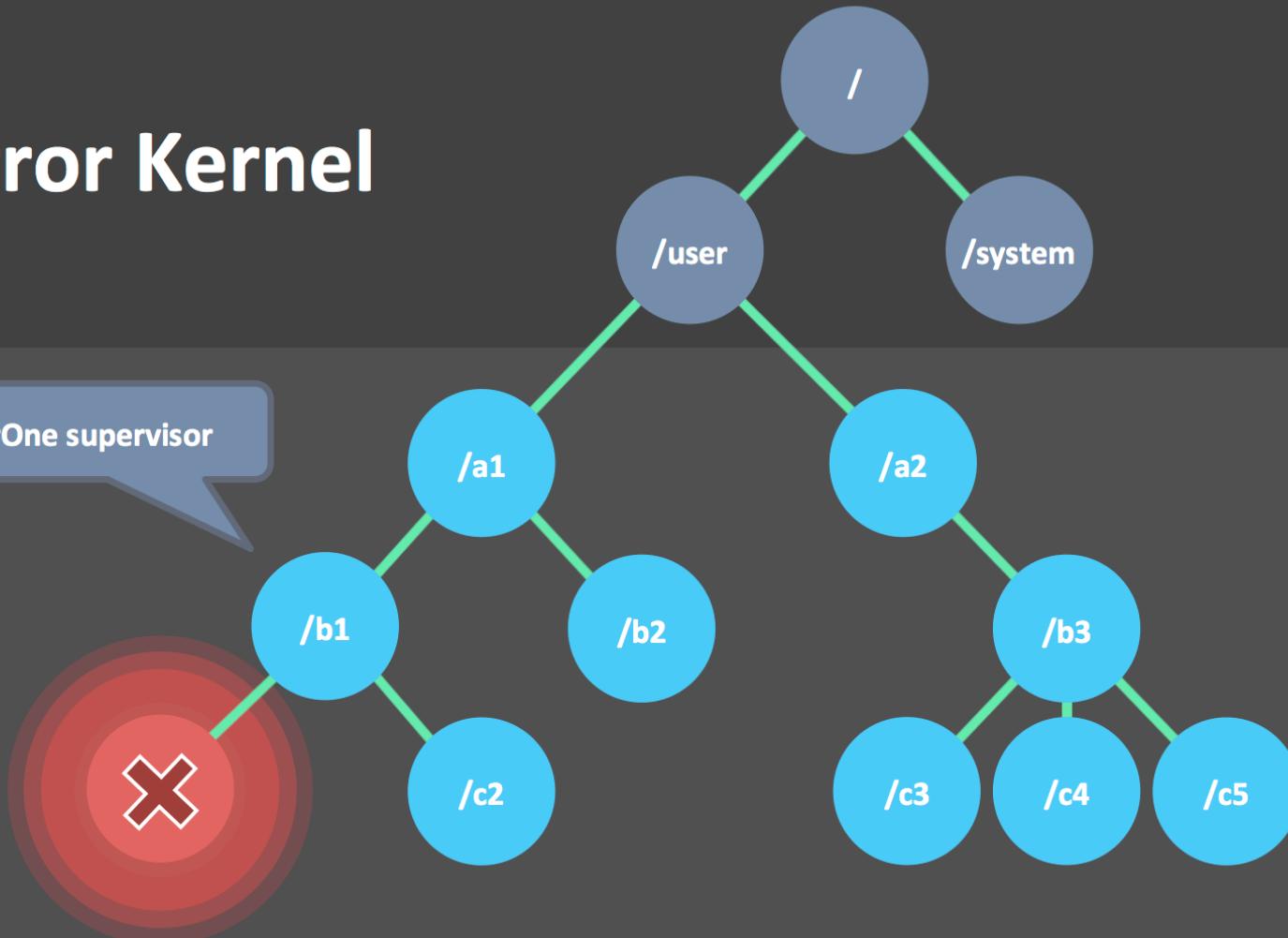
**99.9999999% uptime**  
~0.03 seconds downtime per year



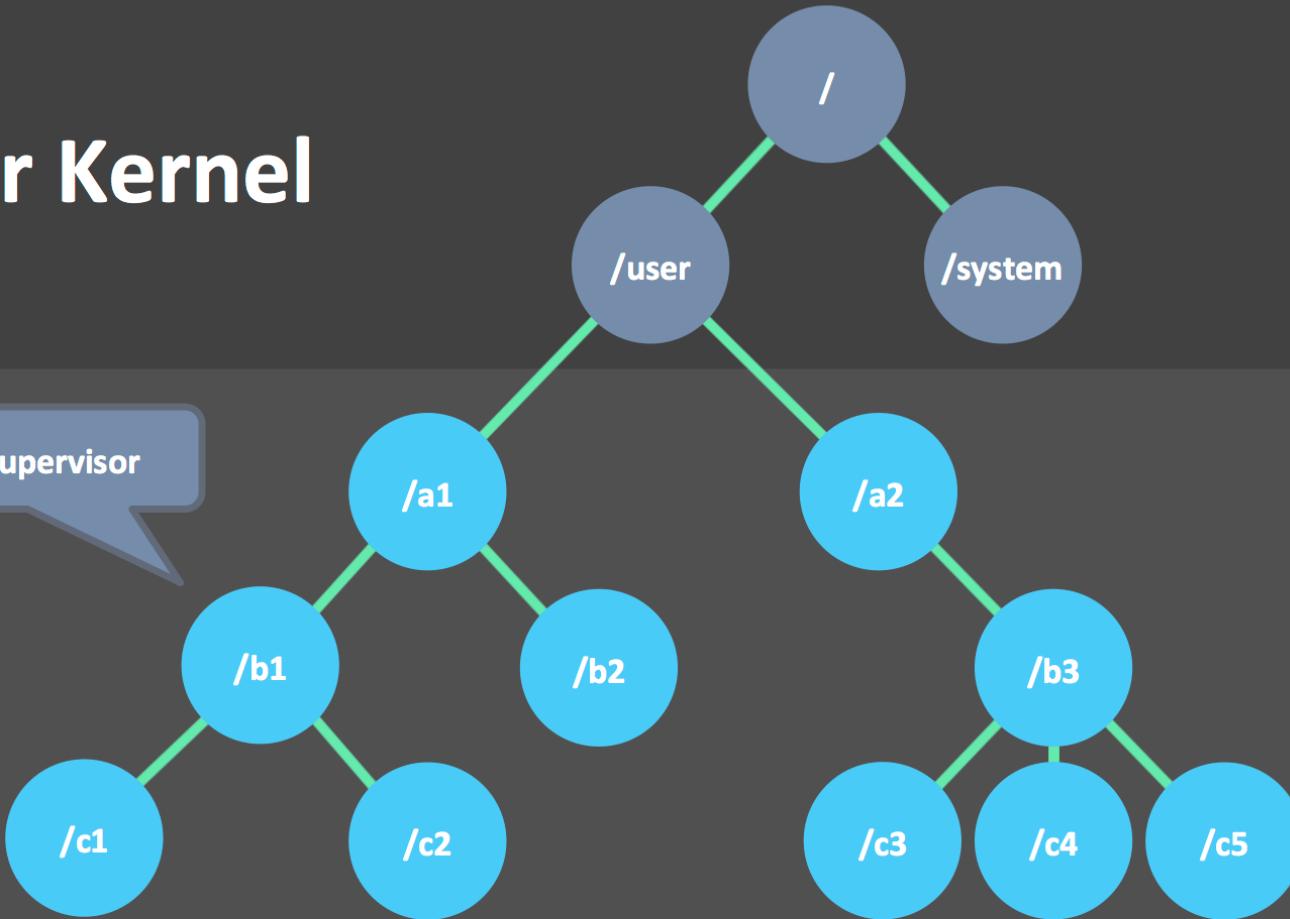
# Error Kernel



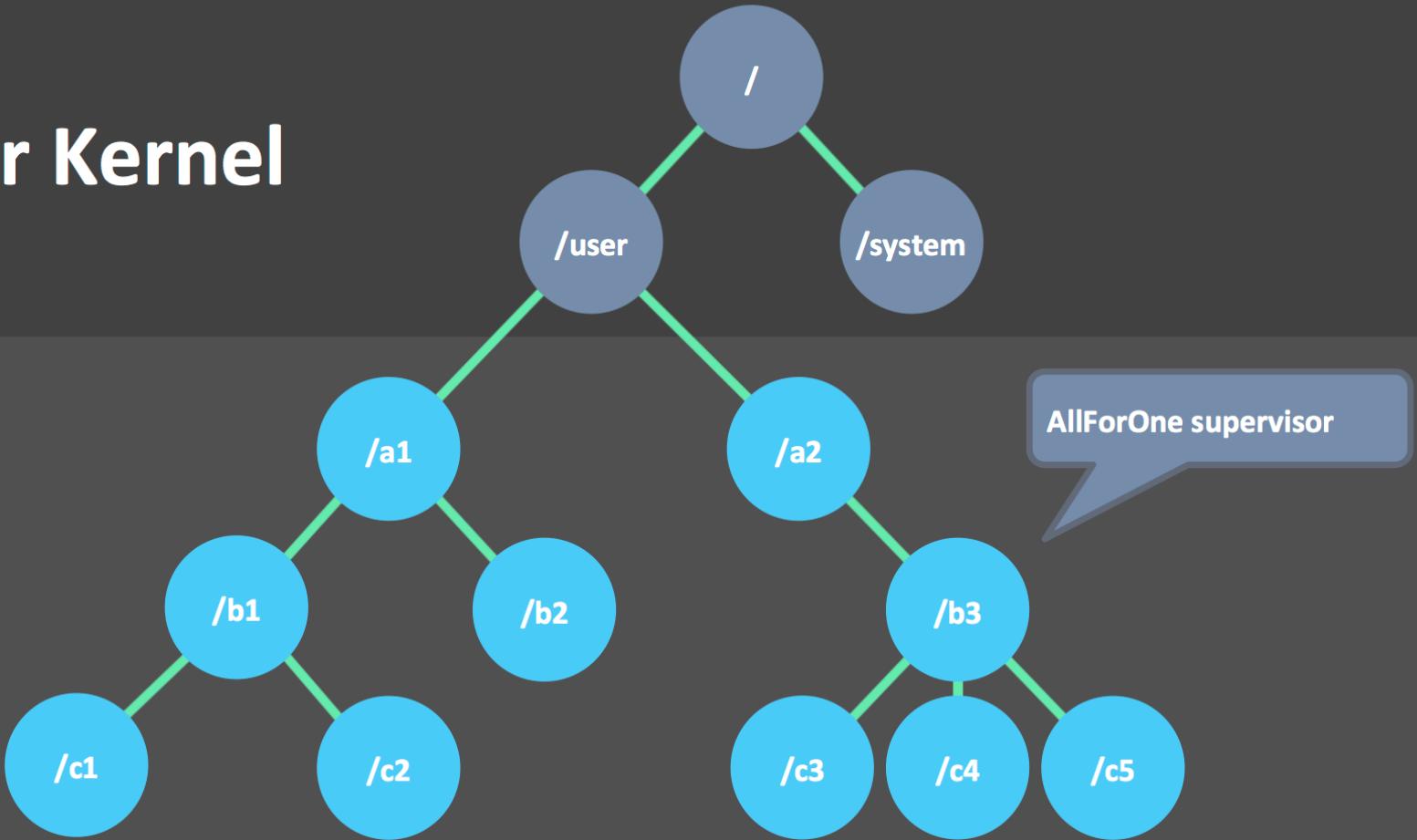
# Error Kernel



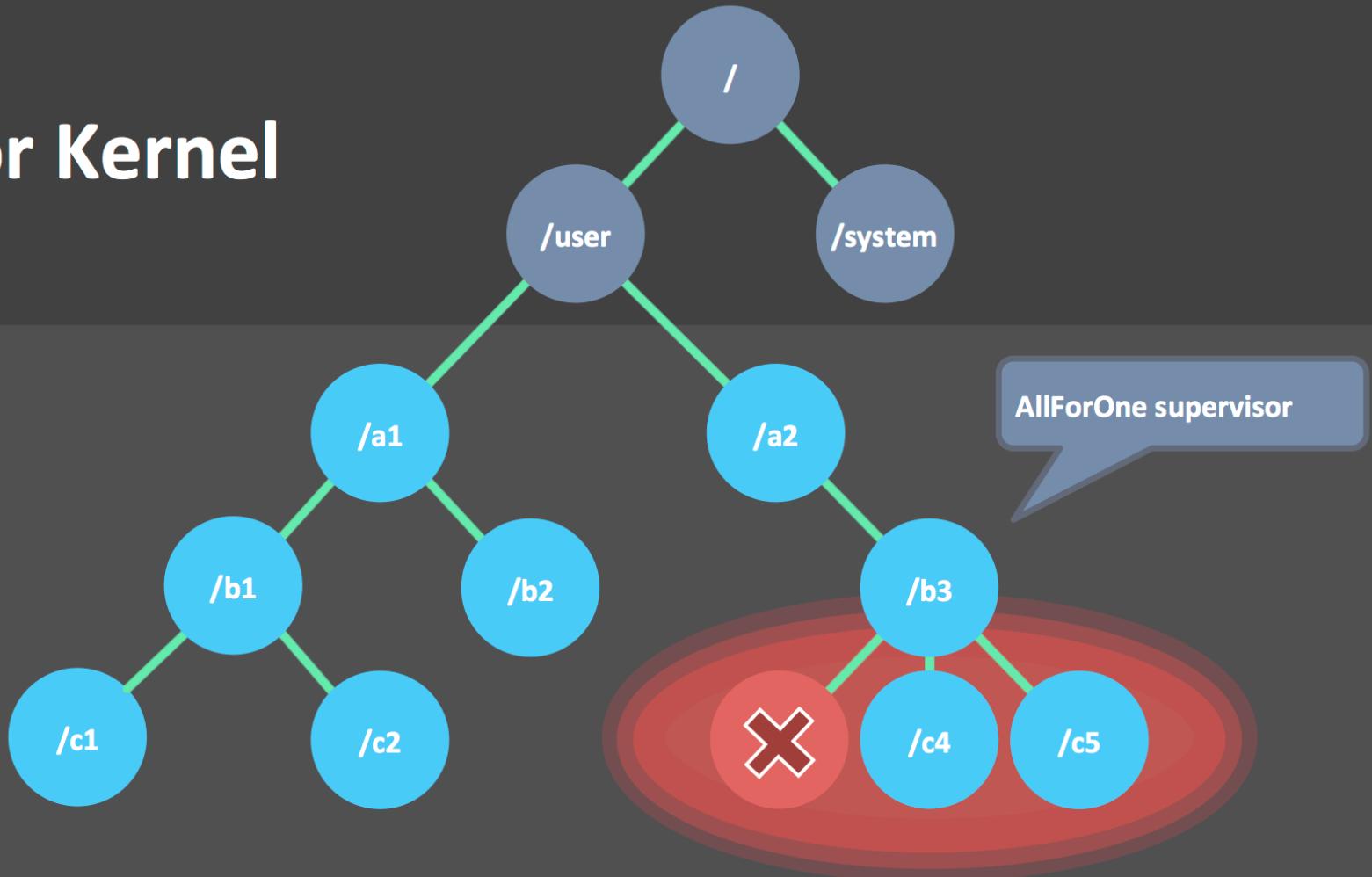
# Error Kernel



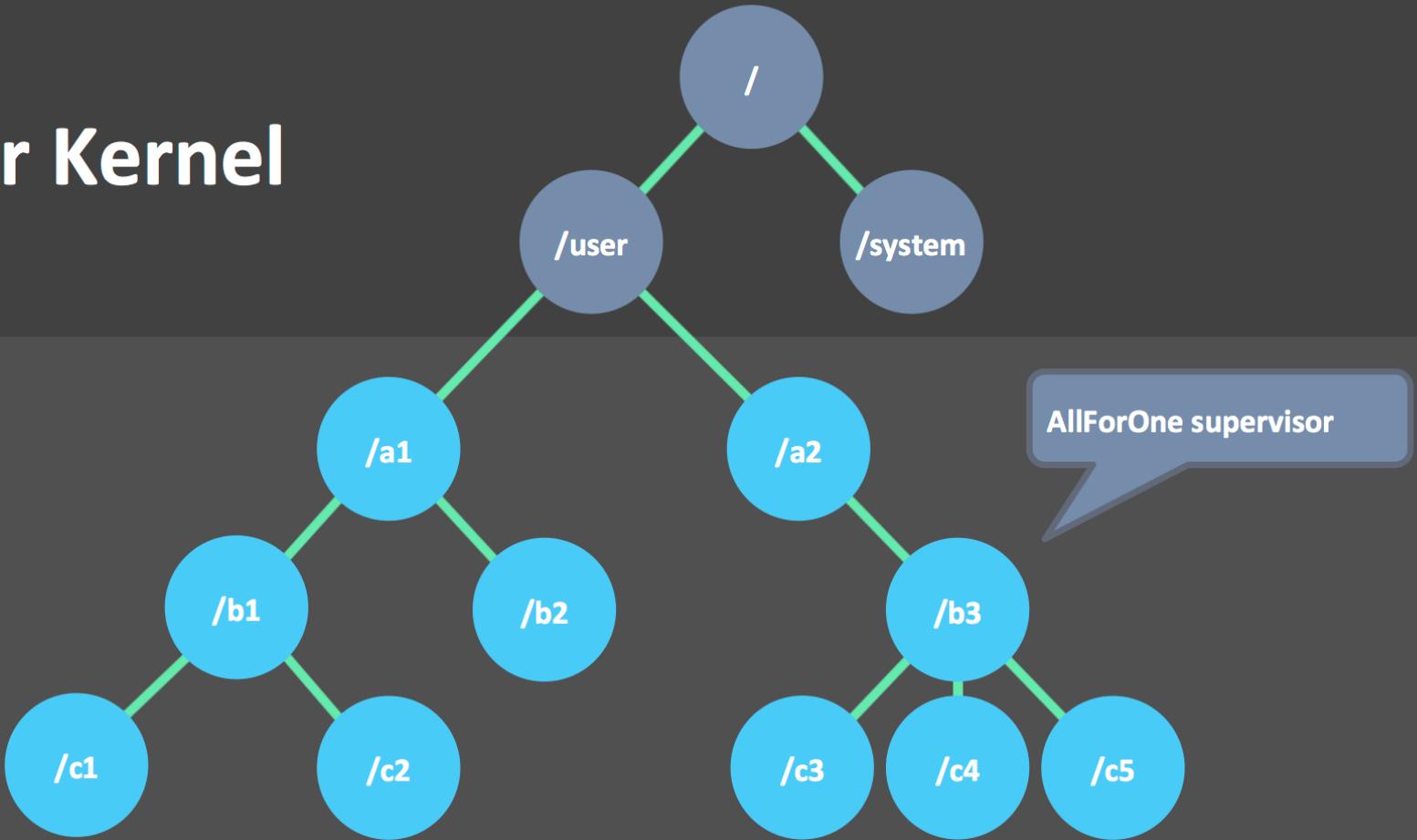
# Error Kernel



# Error Kernel



# Error Kernel



# Akka.Net Remoting

- Distributed by Default Mentality
- Provides a **unified programming model**
- Employees referential or location transparency
- Local and remote use the **same API**
- Distinguished by configuration, allowing a succinct way to code message-driven applications

*Going Scalable!*

# Location Transparency

What  
messag  
an act  
have

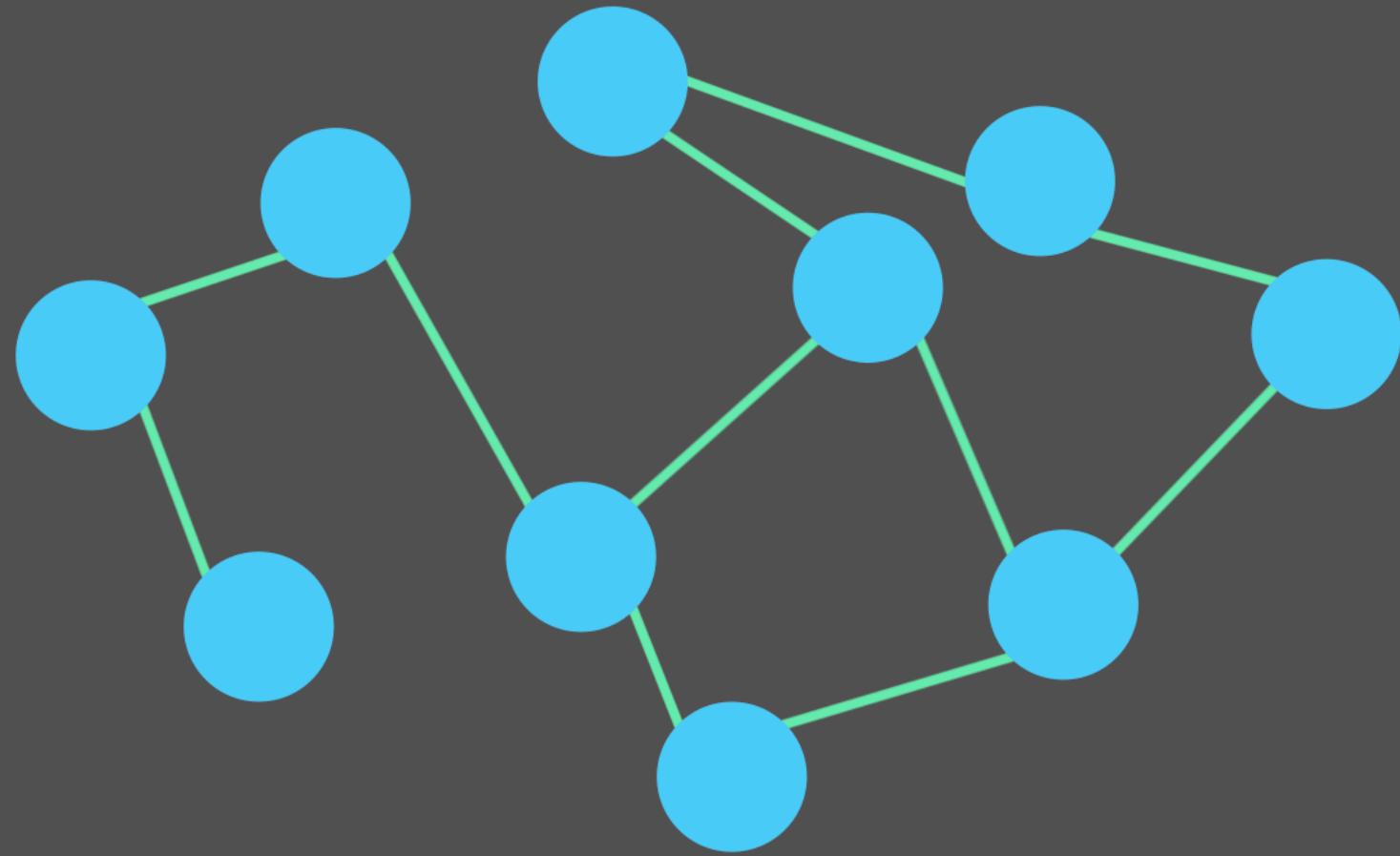
```
let config = ConfigurationFactory.ParseString(@"
    akka {
        actor {
            provider = ""Akka.Remote.RemoteActorRefProvider, Akka.Remote"""
        }
        remote {
            helios.tcp {
                port = 8090
                hostname = localhost
            }
        }
    }")
}

let system = ActorSystem.Create("MyClient", config)

//get a reference to the remote actor
let greeter = select "akka.tcp://MyServer@localhost:8080/user/greeter" system

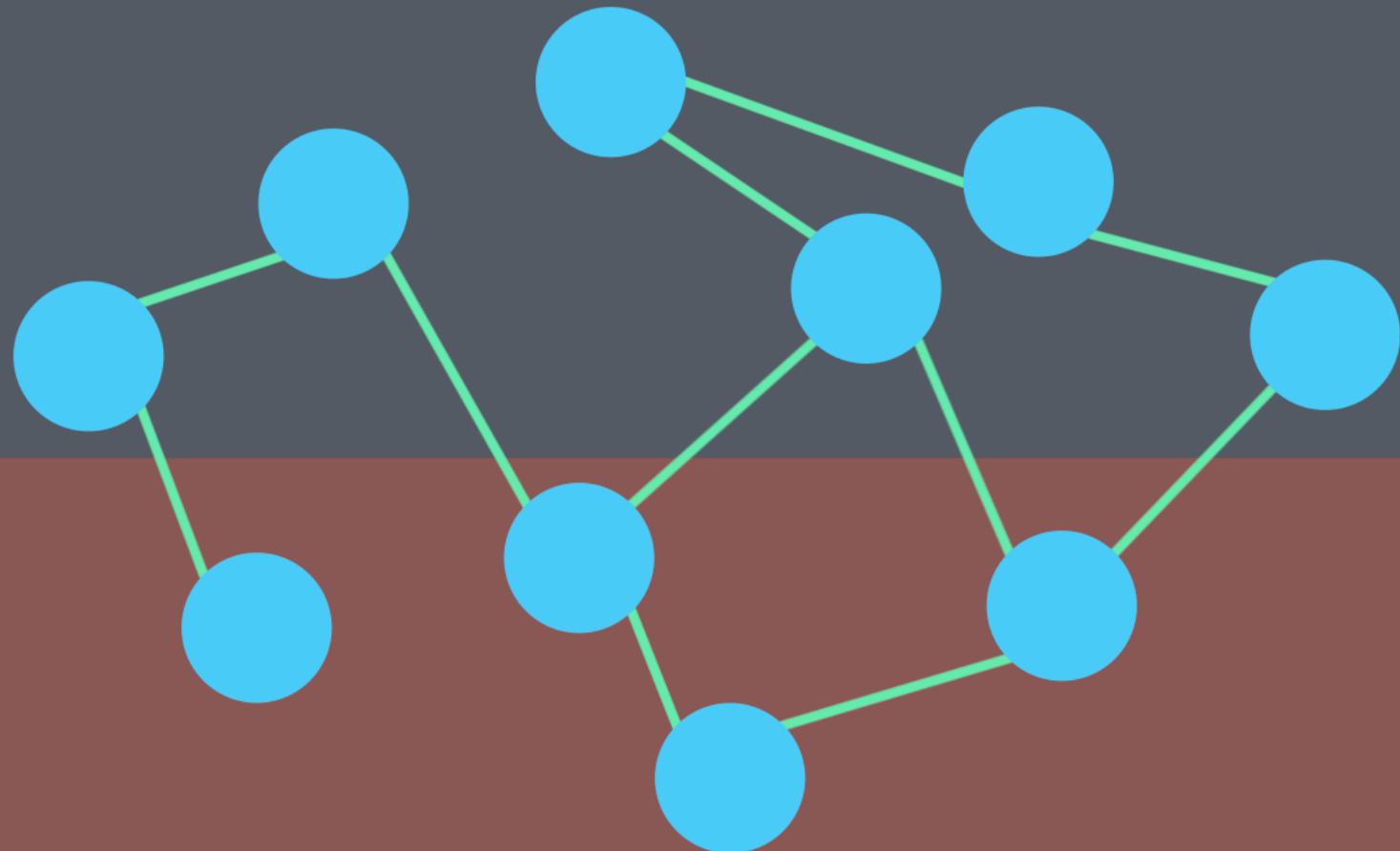
//let greeter = system
//    .ActorSelection("akka.tcp://MyServer@localhost:8080/user/greeter")

//send a message to the remote actor
greeter <! new Greet("Akka.NET & F#!!")
```



**System 1**

**System 2**



# Akka.Net + F# API = Hotswap



*Hotswap is the ability to change the behavior of an Actor at runtime without shutdown the system.*

F# uses <@ Code Quotations @> to deploy some behavior to a remote Actor

```
let aref =
    spawne localSystem "akka.tcp://remote-system@localhost:9234/" "hello"
        // actorOf wraps custom handling function with message receiver logic
    <@ actorOf (fun msg -> printfn "received '%s'" msg) @>
        [SpawnOption.Deploy (Deploy(RemoteScope (Address.Parse remoteSystemAddress)))]
```

# Akka.Net Routing



- Messages can be sent to destination actors, known or outside of an actor, or use a self contained router.
- On the surface routers implemented different strategies efficient at receiving messages from routees.

```
let config = ConfigurationFactory.ParseString(@"
    akka {
        actor {
            deployment {
                /localactor {
                    router = roundRobinPool
                    nr-of-instances = 2
                }
            }
        }
    }
")

let system = System.create "system1" <| config

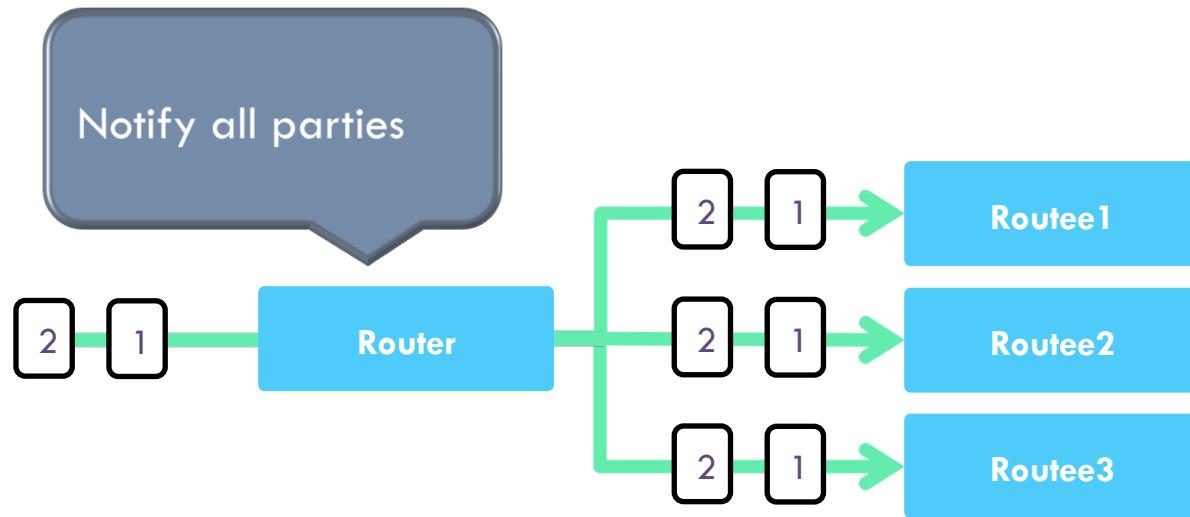
let router = spawnOpt system "localactor" (fun mailbox -> ...)

for i = 0 to 20 do
    router <! i
```

# Akka.Net Routing Strategies



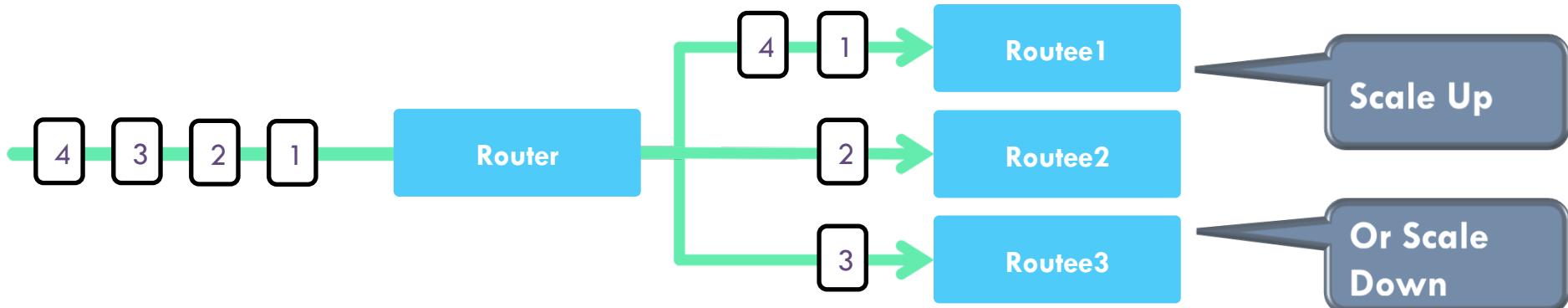
- ❑ **Broadcast** router will as the name implies, broadcast any message to all of its routees



# Akka.Net Routing Strategies

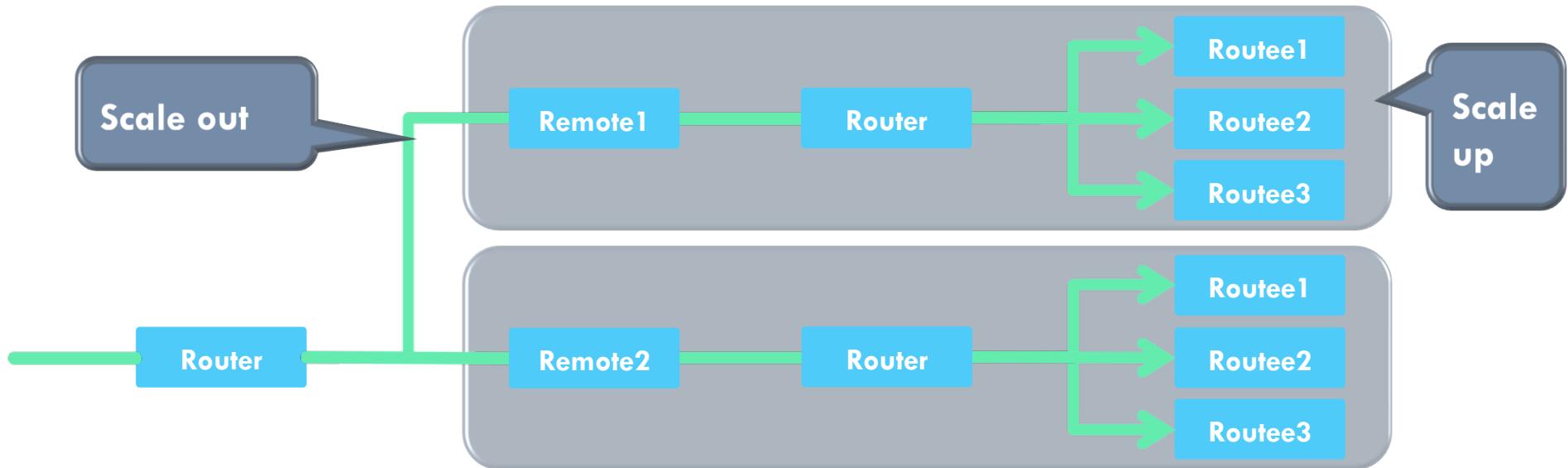


- RoundRobin-Pool router uses round-robin to select a connection. For concurrent calls, round robin is just a best effort.



# Akka.Net Routing Strategies

- ☐ RoundRobin-Group router uses round-robin to select a connection. For concurrent calls, round robin is just a best effort.



# Typed & Untyped Actor

```
type SimpleActor () as this =
    inherit ReceiveActor () // Untyped

    let state = ref 0 // mutable is safe!!

    do
        this.Receive<Message>(fun m ->
            match m with
            | Print -> printfn "%i" !state
            | Increment -> state := !state + 1)

        this.Receive<OtherMessage>(fun m ->
            match m with
            | Display -> printfn "%i" !state
            | Decrement -> state := !state + 1)

    override this.Unhandled(msg) =
        printfn "What should I do with this thing %A" (msg.GetType())
```

# F# & Akka.NET



To install Akka.NET

```
Install-Package Akka.NET  
Install-Package Akka.NET-Samples
```

F# API

```
let system = ActorSystem.Create("example2")  
  
let actor = system.ActorOf<SimpleActor>()  
  
actor.Tell Print  
actor.Tell Increment  
actor.Tell Increment  
actor.Tell Increment  
actor.Tell Print  
  
actor <! Decrement  
actor <! Decrement  
actor <! Display
```

with  
 -> printfn "%i" !state  
 increment -> state := !state + 1)  
  
| Decrement -> state := !state + 1)



A small white 3D-style figure stands behind a large, red, textured sign that reads "DEMO". The sign has a thick, rounded rectangular border and a central rectangular cutout. The background is plain white.



# Actor Best Practices

## Single Responsibility Principle

Keep the actors focused on a single kind of work, and in doing so, allow yourself to use them flexibly and to compose them

## Specific Supervisors

Akka only permits one strategy for each supervisor, there is no way to delineate between actors of one grouping for which OneForOne is the restart strategy you want, and another grouping for which AllForOne is the preferred restart strategy.

## Keep the Error Kernel Simple

build layers of failure handling that isolate failure deep in the tree so that as few actors as possible are affected by something going wrong. Never be afraid to introduce layers of actors if it makes your supervisor hierarchy more clear and explicit.

## Prefer the fire and forget when you can – Tell don't ask

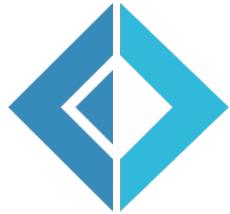
## Avoid Premature optimization – Leave routing for last and think Synchronous first



# Actor Pros & Cons

Pros	Cons
Easier to reason about	Actors don't work well when
Higher abstraction level	The shared of state is needed to achieve global consensus (Transaction)
Easier to avoid Race conditions Deadlocks Thread Starvation Live locks	Synchronous behavior is required
Distributed computing	It is not always trivial to break the problem into smaller problems

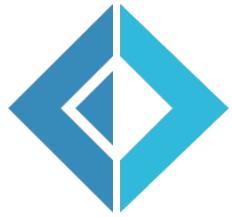
# Summary



The free lunch is over,  
we are facing new  
challenges in today's  
Programming World

Today Applications must  
be built for concurrency  
in mind  
(possibly from the begining)

Actor is ~~the best~~ a great  
concurrent programming  
model that solves  
problems for Scalling Up  
& Out



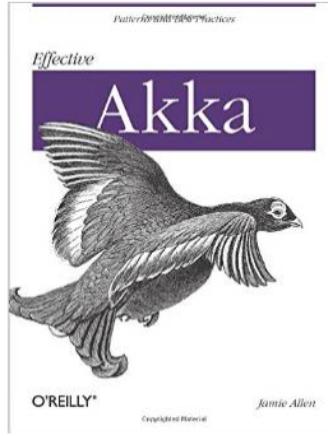
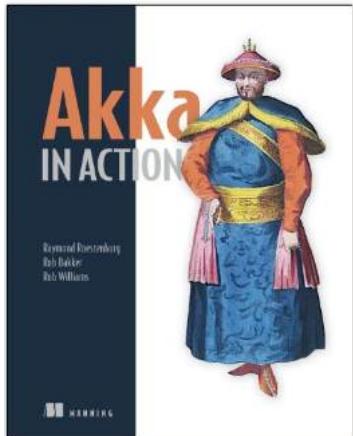
*The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities.*

-- Edsger Dijkstra

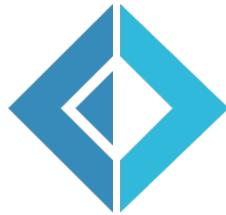


# References

- <https://reactivemanifesto.com>
- <http://www.getakka.net>
- <http://petabridge.com>



# Online resources



## Getting Started in F#



Learn F# Programming Fundamentals

## Advanced F# Programming



Learn Advanced F# Programming Techniques

## Data Visualization and Charting



Bring Your Data to Life with Charting

## Data Science



Work with Language Integrated Web Data through F# Type Providers

## Scientific and Numerical Computing



Write Simple Code to Solve Complex Problems with F#

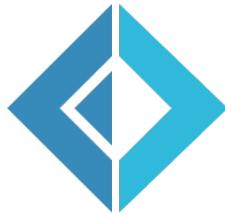
## Financial Computing



Examples Related to Financial Modeling and Engineering

- [www.fsharp.org](http://www.fsharp.org)  
[www.tryfsharp.org](http://www.tryfsharp.org)

Information & community  
Interactive F# tutorials



# How to reach me



[github.com/rikace/AkkaActorModel](https://github.com/rikace/AkkaActorModel)

[meetup.com/DC-fsharp](https://meetup.com/DC-fsharp)

@DCFsharp

@TRikace

[rterrell@microsoft.com](mailto:rterrell@microsoft.com)



*That's all Folks!*