# 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 – *LambdaConf 2015*

# Agenda

- Vector Check - 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 ~10 years
  - Living/working in Washington DC
- +/- 18 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

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

**Programmers must embrace concurrent programming**

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

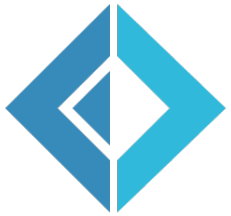
Akka provide a simple and unified programming model to reach high performance with simplicity

# Hottest technology trends

- Cloud Computing
- Reactive Application
- Real Time Notification
- Distributed Systems
- Big Data

- Scalable
- Maintainable & Readable
- Reliable & Testable
- Composable & Reusable
- Concurrent

# The World is changed



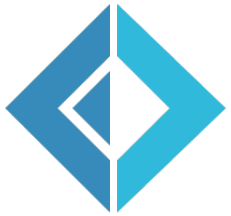
*"In today's world, the demand for distributed systems has exploded. As customer expectations such as an immediate response, no failure, and access anywhere increase, companies have come to realize that distributed computing is the only viable solution."*

*- Reactive Application Development (Manning)*

*"Modern applications must embrace these changes by incorporating this behavior into their DNA".*

*- Reactive Application Development (Manning)*

# The Problems...

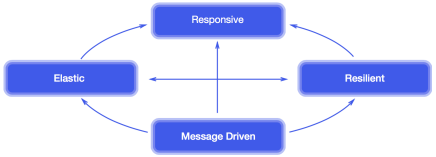


It is hard to build correct highly concurrent systems

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

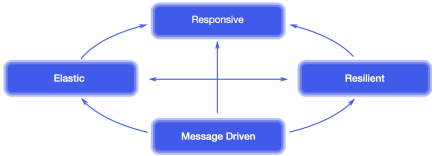
# Reactive Manifesto




- Responsive
- Message-Driven
- Resilient
- Elastic

http://www.reactivemanifesto.org

# Reactive Manifesto

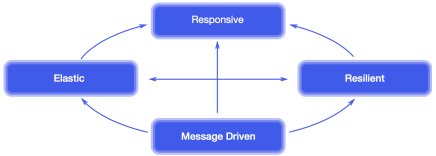



- Responsive
- Message-Driven
- Resilient
- Elastic

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.

http://www.reactivemanifesto.org

# Reactive Manifesto

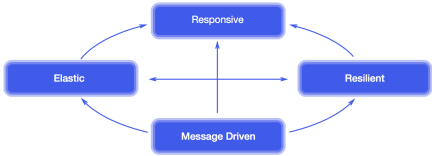



- Responsive
- Message-Driven
- Resilient
- Elastic

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.

http://www.reactivemanifesto.org

# Reactive Manifesto

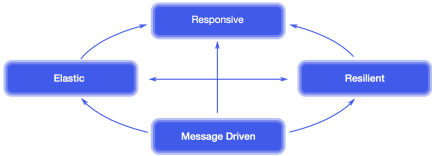



- Responsive
- Message-Driven
- Resilient
- Elastic

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.

http://www.reactivemanifesto.org

# Reactive Manifesto



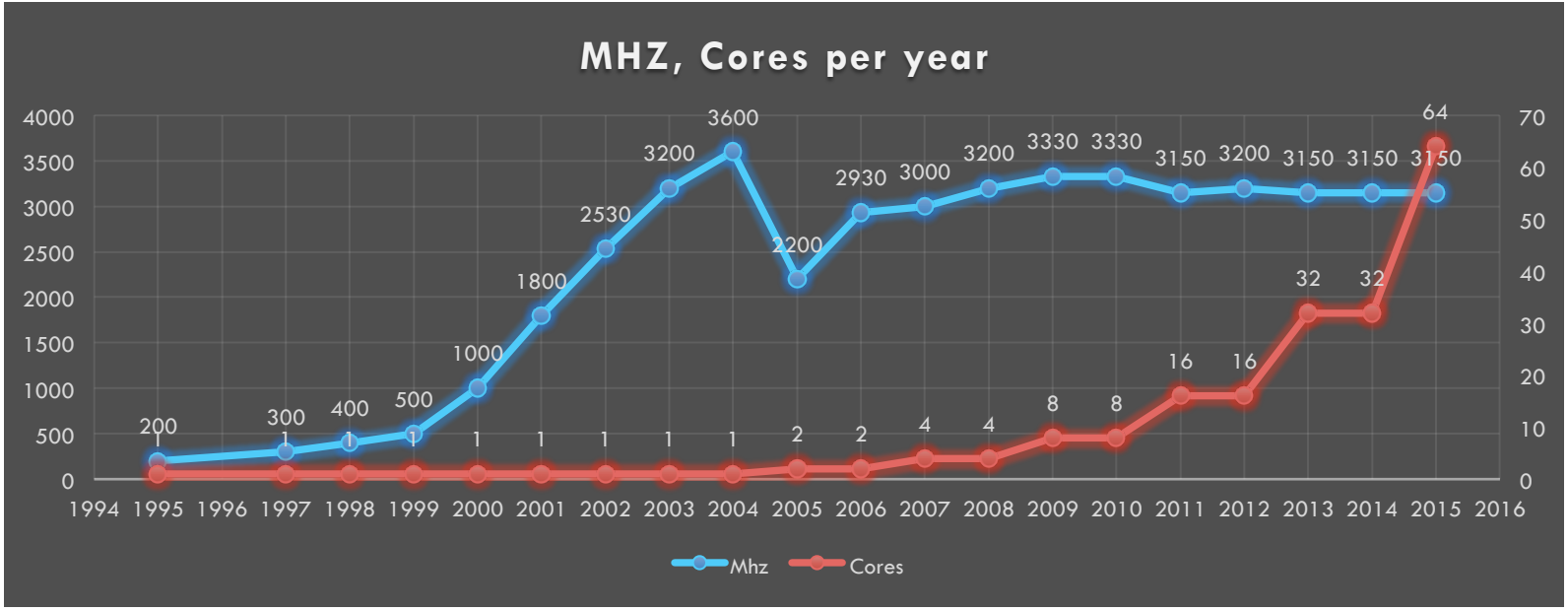

- Responsive
- Message-Driven
- Resilient
- Elastic

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.

http://www.reactivemanifesto.org

# 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 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

# Java: Imperative Style (JavaWorld 2012)

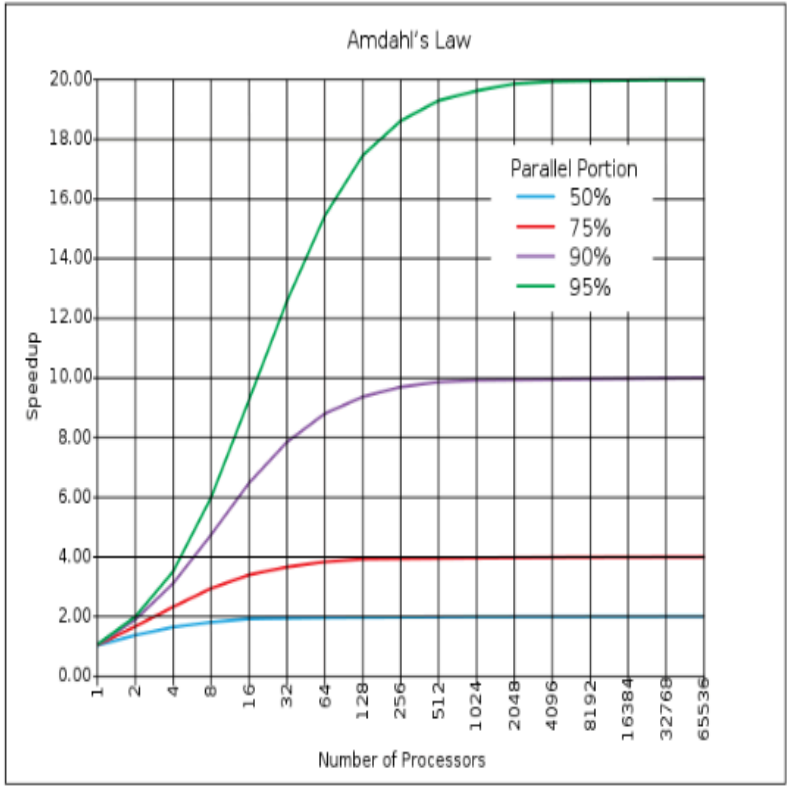
The better argument for functional programming is that, in modern applications involving highly concurrent computing on multicore machines, state is the problem. All imperative languages, including OOP languages involve multiple threads changing the shared state of objects. This is where deadlocks, stack traces, and low-level processor cache misses all take place. **If there is no state, there is no problem.**

http://www.javaworld.com/article/2078610/java-concurrency/functional-programming--a-step-backward.html

# 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 Solution is Immutability and Isolation

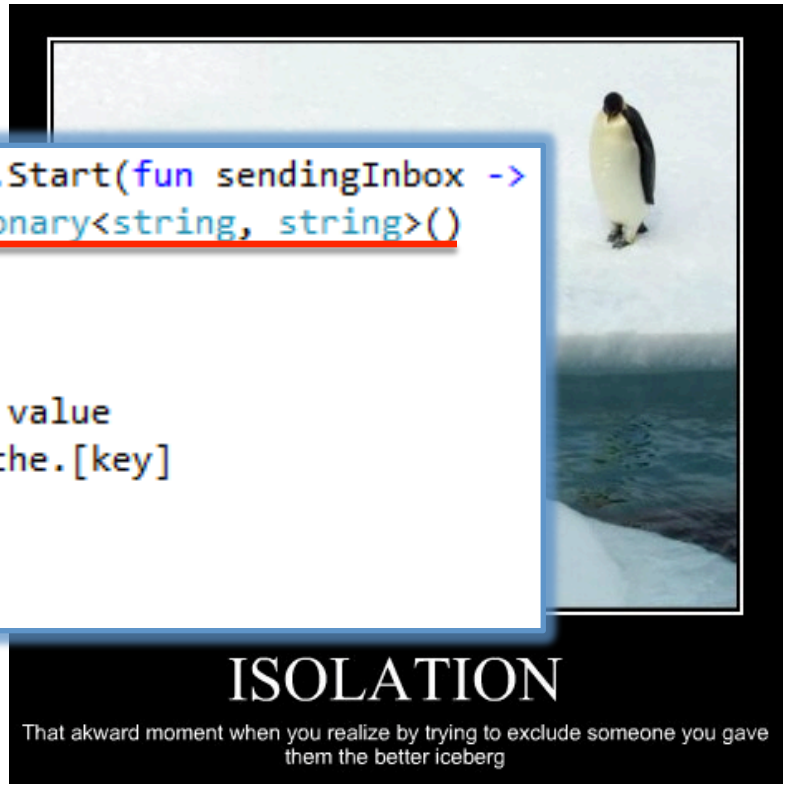
IMMUTABILITY +
ISOLATION =

---

**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 ())
```



# What is an Actor?

The Actor model is a model of concurrent computation using actors which is characterized by dynamic creation of actors, inclusion of actor addresses in messages, and interaction only through direct asynchronous message passing with no restriction on message arrival order.

[Wikipedia]

# 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


The fundamental unit of computations that embodies:

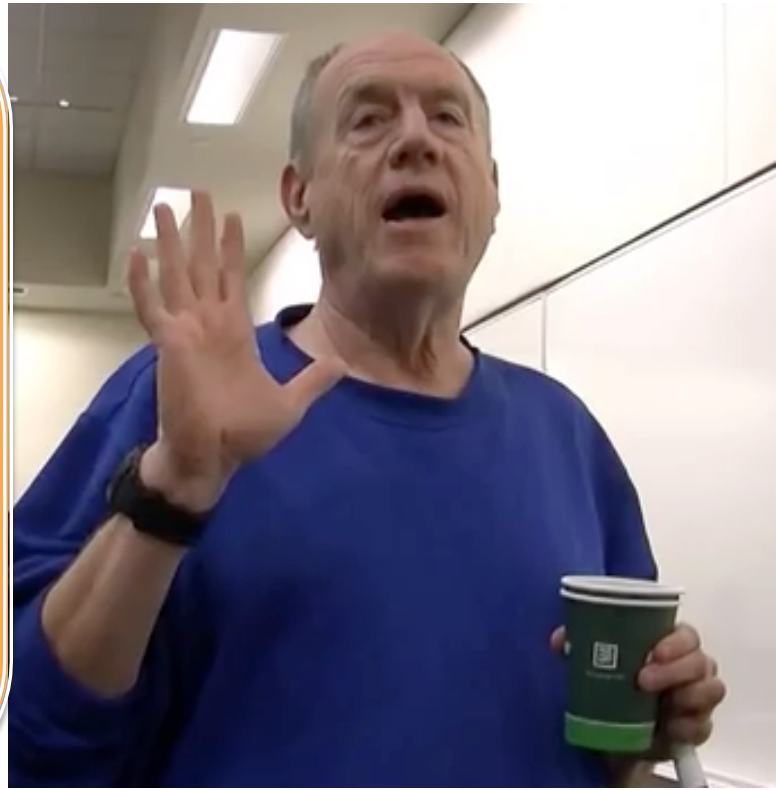
- Processing
- Storage
- Communication

**Actor Model Three axioms:**

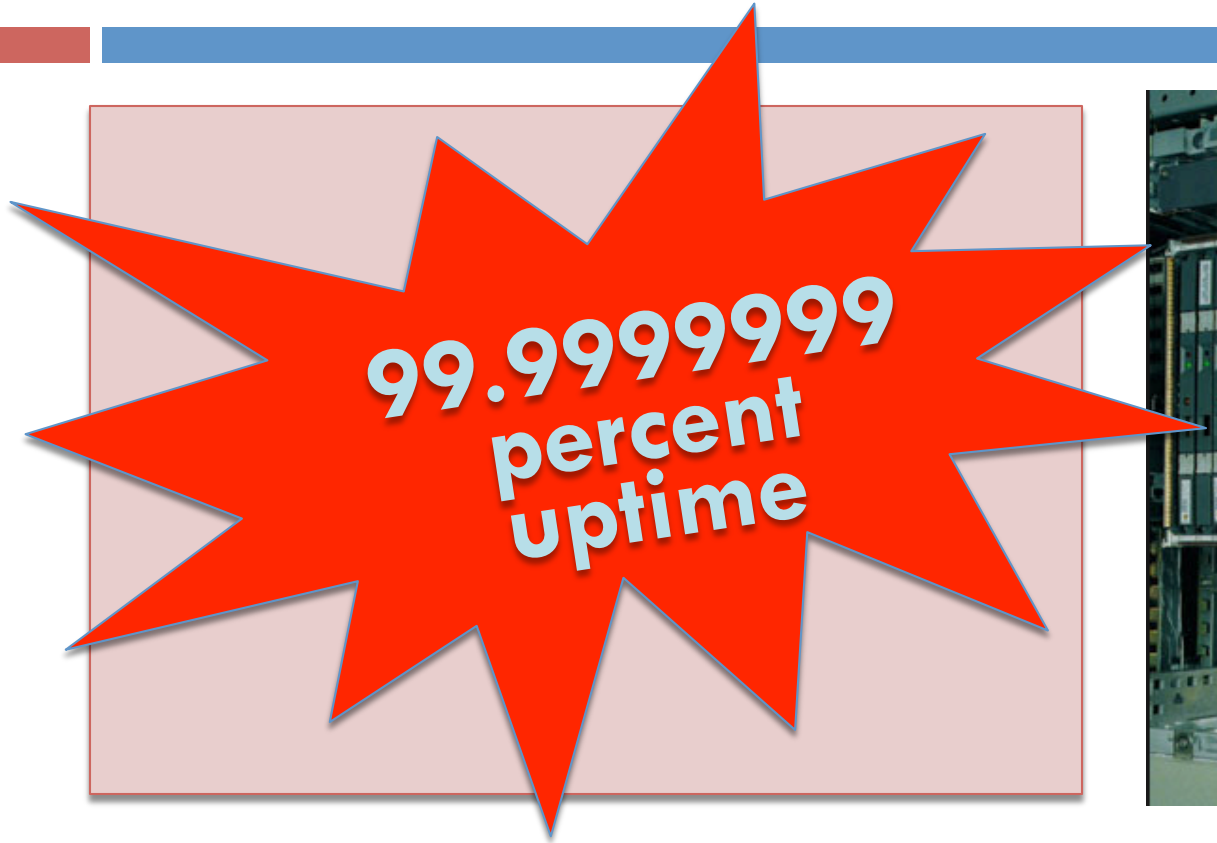
1. Send messages to other Actors

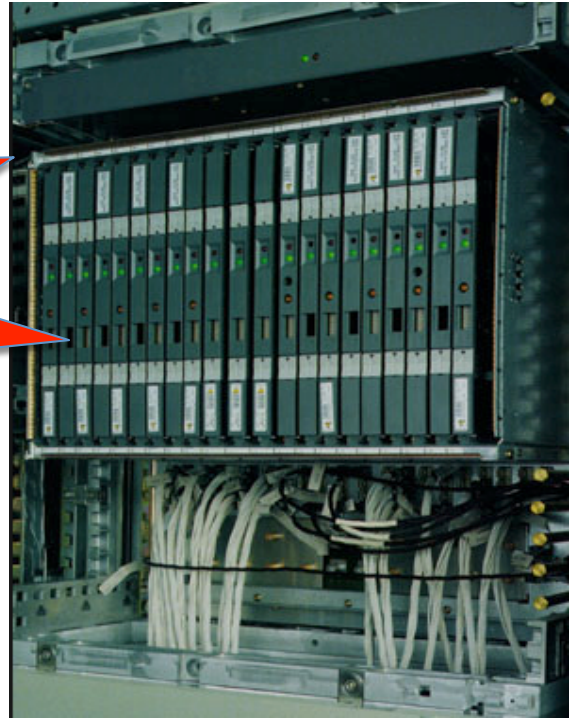
   - *One Actor is not Actor* -

2. Create other Actor
3. Decide how to handle next message



# Ericson AXD 301 Switch - 1996

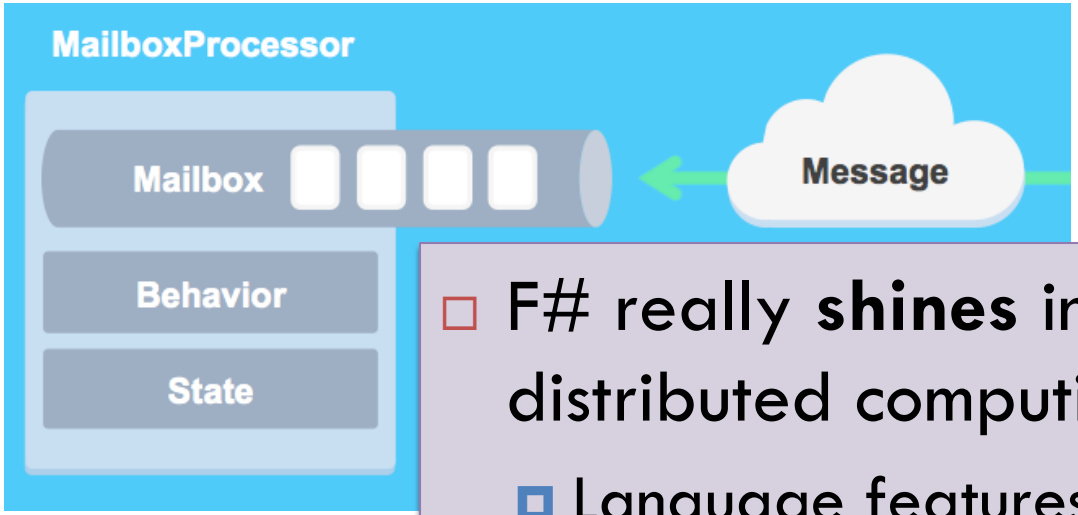


# Reactive Manifesto & Actor Model


| | |
|---|---|
| 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

# Asynchronous Workflows


- *Software is often I/O-bound, it provides notable performance benefits*
  - Connecting to the Database
  - Leveraging web services
  - Working with data on disk
- *Network and disk speeds increasing slower*
- *Not Easy to predict when the operation will complete* (no-deterministic)

# Anatomy of Async Workflows


```
let getLength url =
  let wc = new WebClient()
  let  data = wc.DownloadString(url)
  data.Length
```

- Easy transition from synchronous
  - Wrap in asynchronous workflow with the ***async*** keyword, use **let!** for async calls and add **return**
  - No need of explicit callback
  - Easy to debug
- Supports loops, recursion, exceptions, cancellation, resource management
- **Operation complete in the same scope**

# Anatomy of Async Workflows


```
let getLength url = async {
  let wc = new WebClient()
  let  data = wc.DownloadString(url)
  data.Length }
```

- Easy transition from synchronous
  - Wrap in asynchronous workflow with the ***async*** keyword, use **let!** for async calls and add **return**
  - No need of explicit callback
  - Easy to debug
- Supports loops, recursion, exceptions, cancellation, resource management
- **Operation complete in the same scope**

# Anatomy of Async Workflows


```
let getLength url = async {
  let wc = new WebClient()
  let  data = wc.DownloadString(url)
  return data.Length }
```

- Easy transition from synchronous
  - Wrap in asynchronous workflow with the ***async*** keyword, use **let!** for async calls and add **return**
  - No need of explicit callback
  - Easy to debug
- Supports loops, recursion, exceptions, cancellation, resource management
- **Operation complete in the same scope**

# Anatomy of Async Workflows


```
let getLength url = async {
  let wc = new WebClient()
  let! data = wc.AsyncDownloadString(url)
  return data.Length }
```

- Easy transition from synchronous
  - Wrap in asynchronous workflow with the ***async*** keyword, use **let!** for async calls and add **return**
  - No need of explicit callback
  - Easy to debug
- Supports loops, recursion, exceptions, cancellation, resource management
- **Operation complete in the same scope**

# Asynchronous Workflows


```
let openFileAsynchronous : Async<unit>
  async { use  fs = new  FileStream(@"C:\Program Files\..., …)
          let  data = Array.create (int fs.Length) 0uy
          let!  bytesRead = fs.AsyncRead(data, 0, data.Length)
          do  printfn "Read Bytes: %i, First bytes were:
          %i %i %i ..." bytesRead data.[1] data.[2] data.[3] }
```

- Async defines a block of code we would like to run asynchronously
- We use let! instead of let
  - let! binds asynchronously, the computation in the async block waits until the let! completes
  - While it is waiting it does not block
  - No program or OS thread is blocked

# 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 process boundaries
  - only within the same process.
- F# agent are not referenced by address but by explicit instance
- No build in durable mailboxes
- F# agent doesn't have any routing and supervision functionality out of the box

# 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*

VS

# How Actors Recover from Failure?

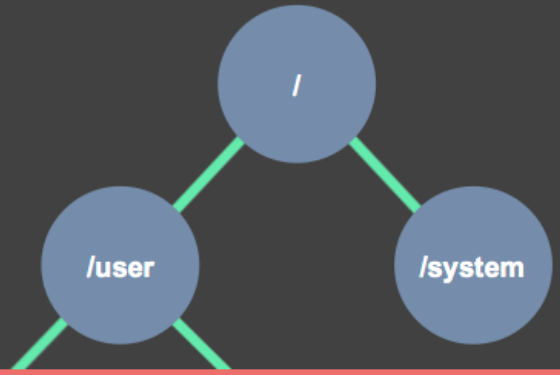
**How do you keep your actor system from falling apart when things go wrong?**

The short answer is **Supervision** *(parental supervision)*

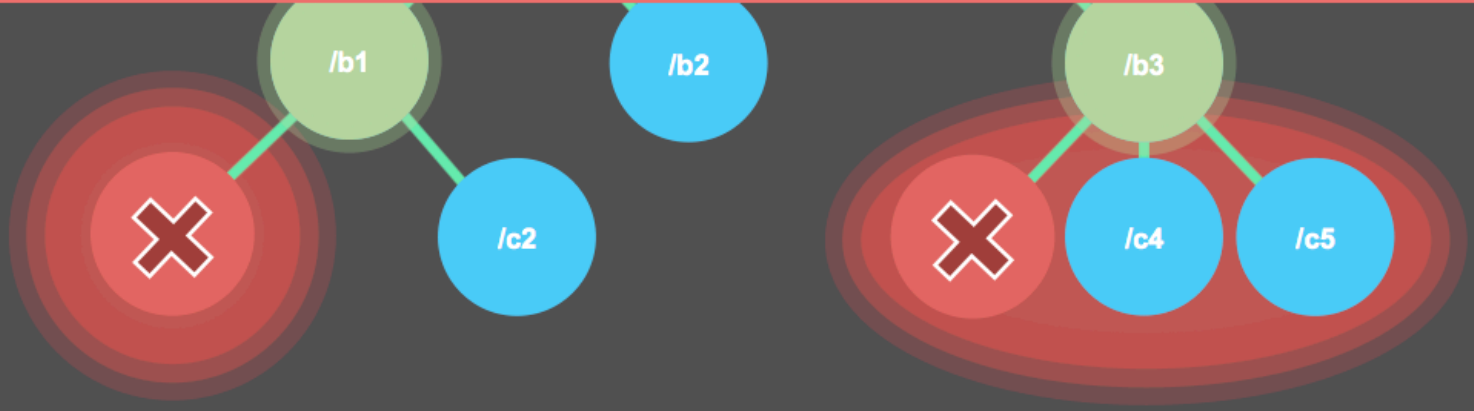
**How can the Supervision resolve the error?**

There are two factors that determine how a failure is resolved:
- How the child failed
- Parent's **SupervisionStrategy**

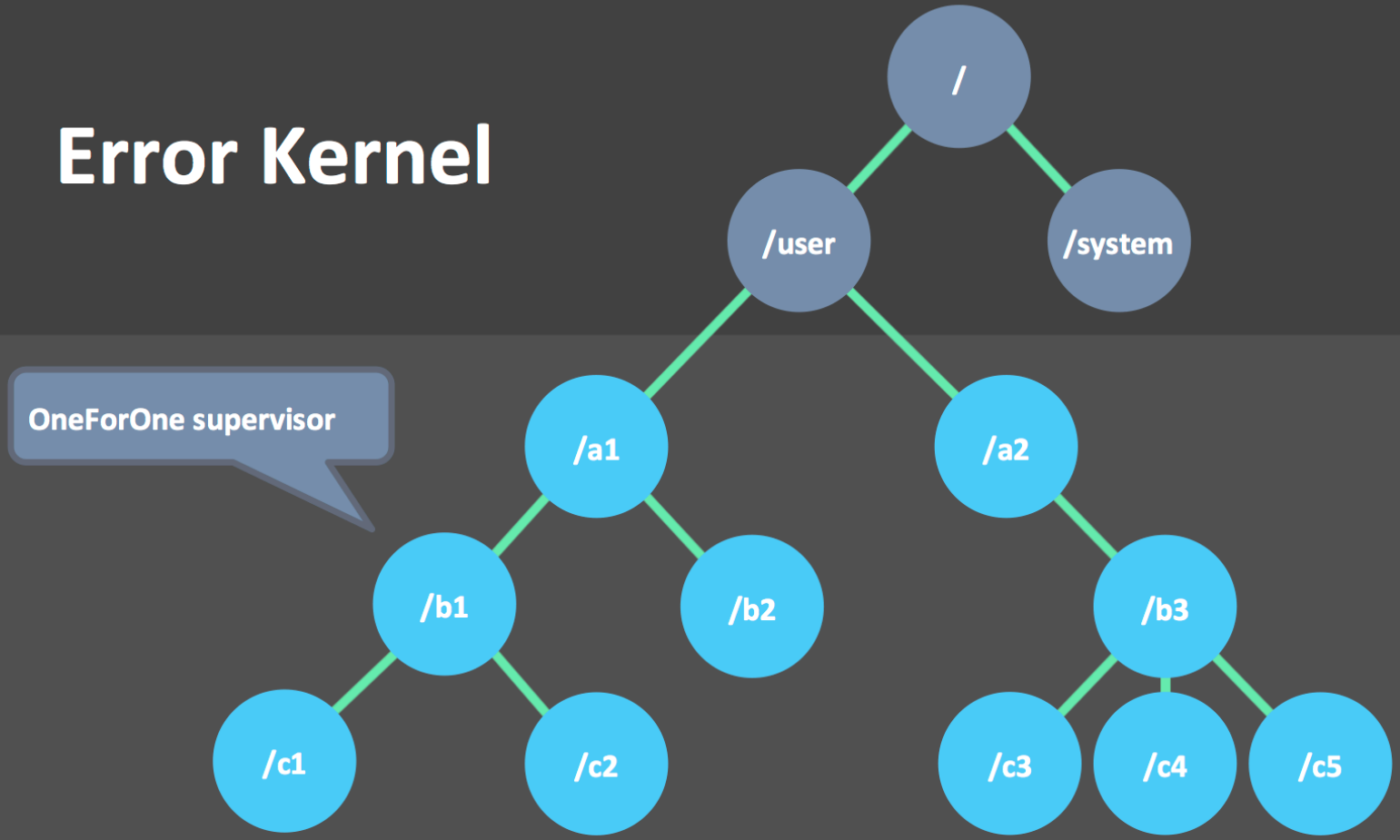
# Error Kernel



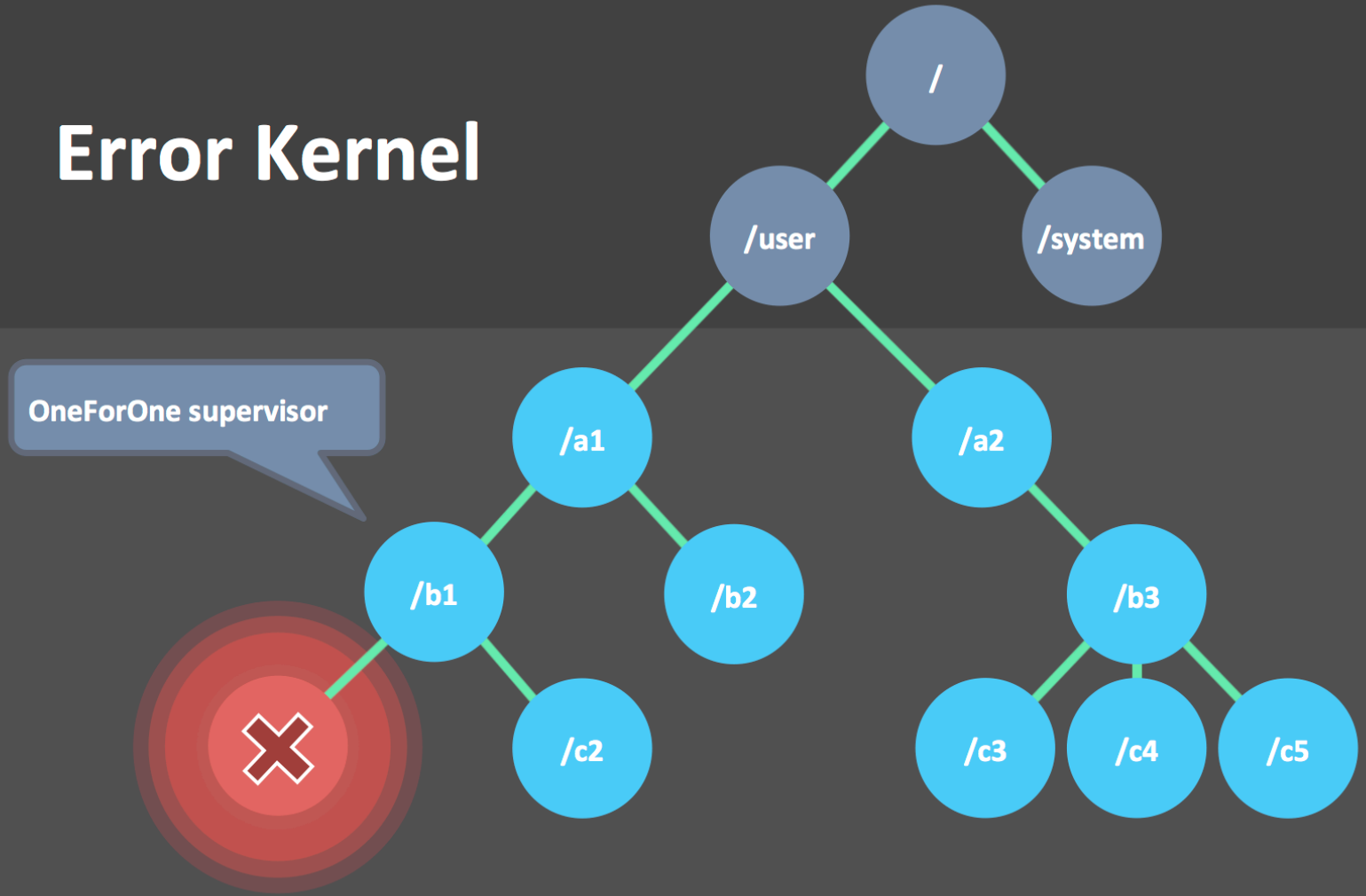

**99.9999999% uptime**
**~0.03 seconds downtime per year**



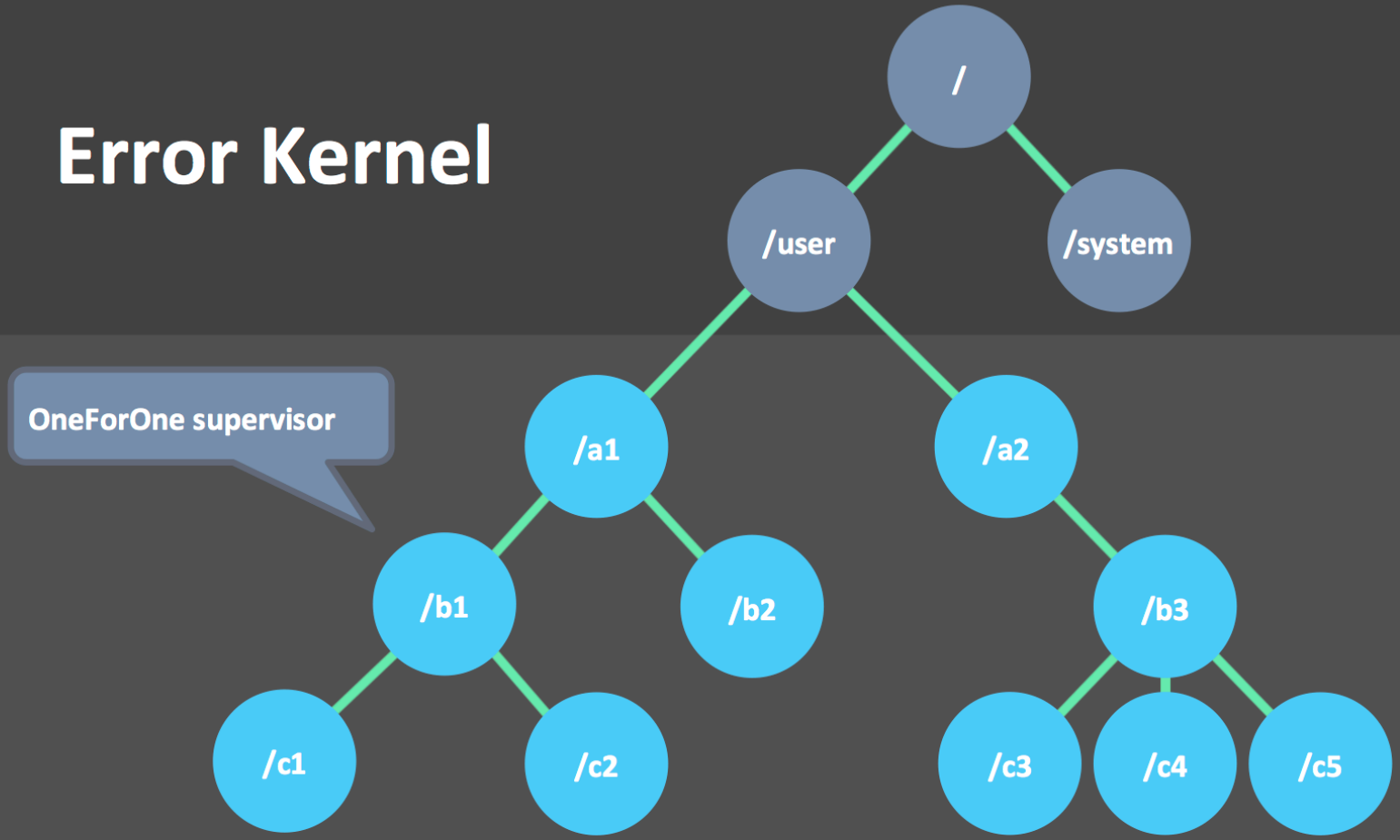
# Error Kernel



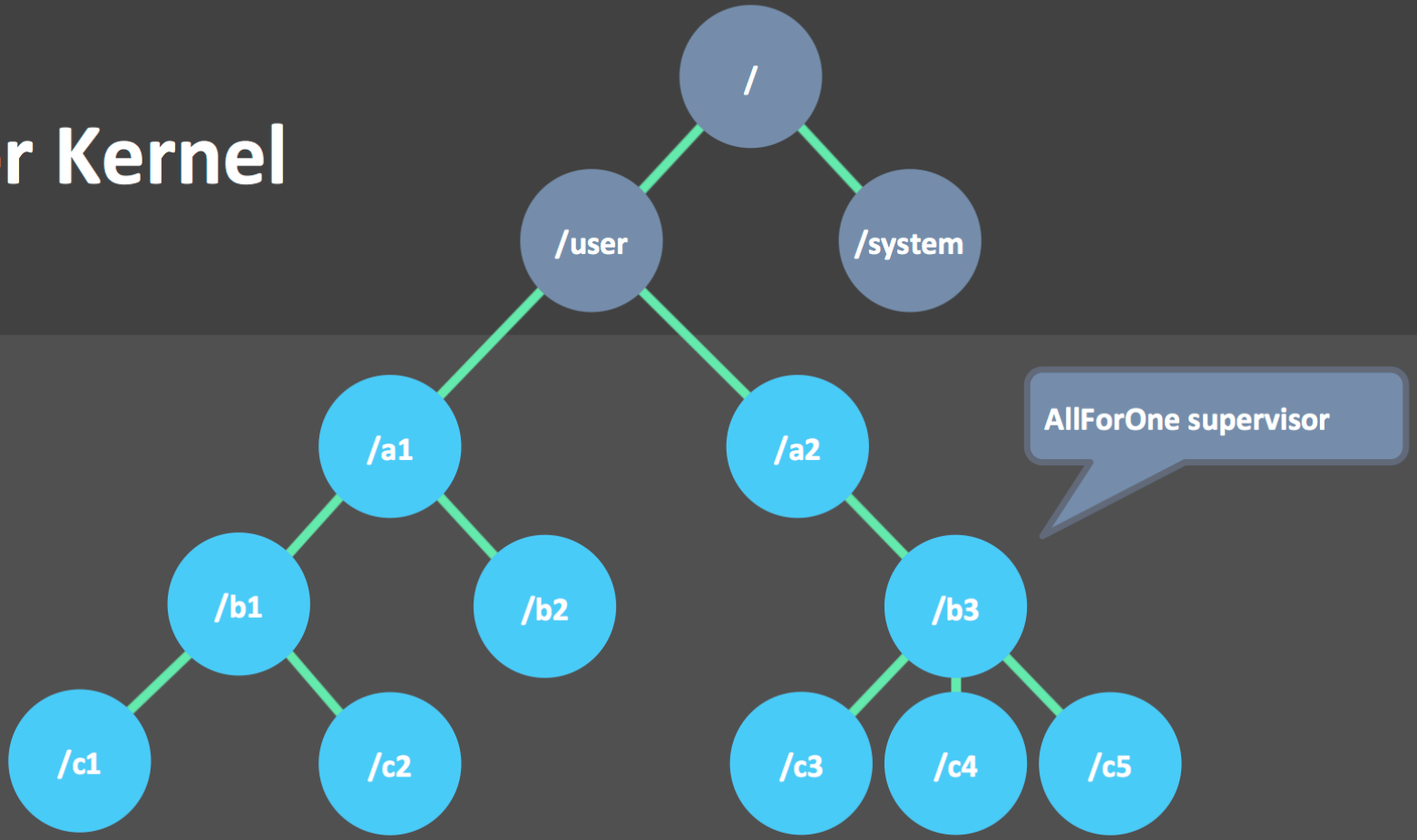
# Error Kernel



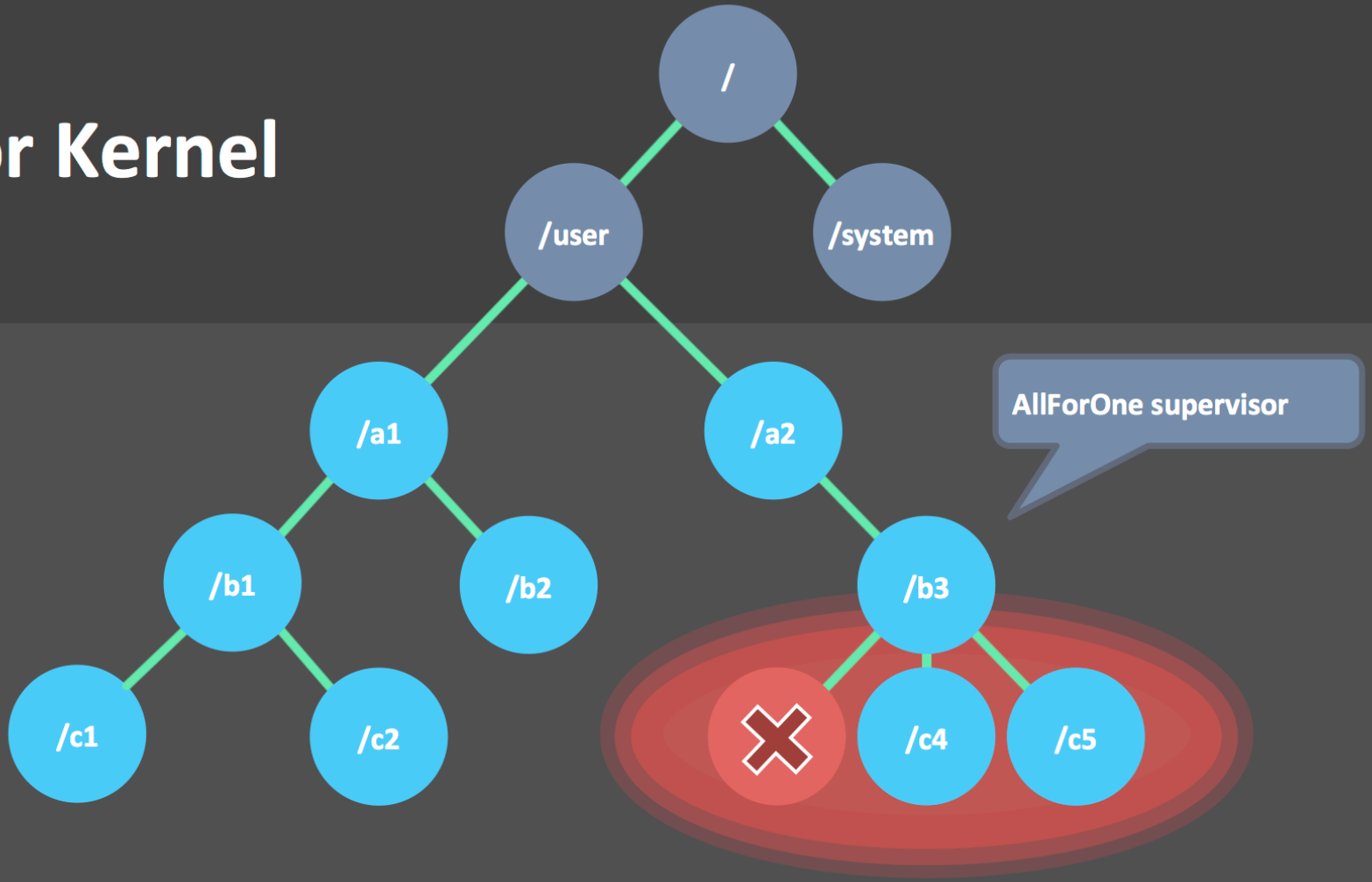
# Error Kernel



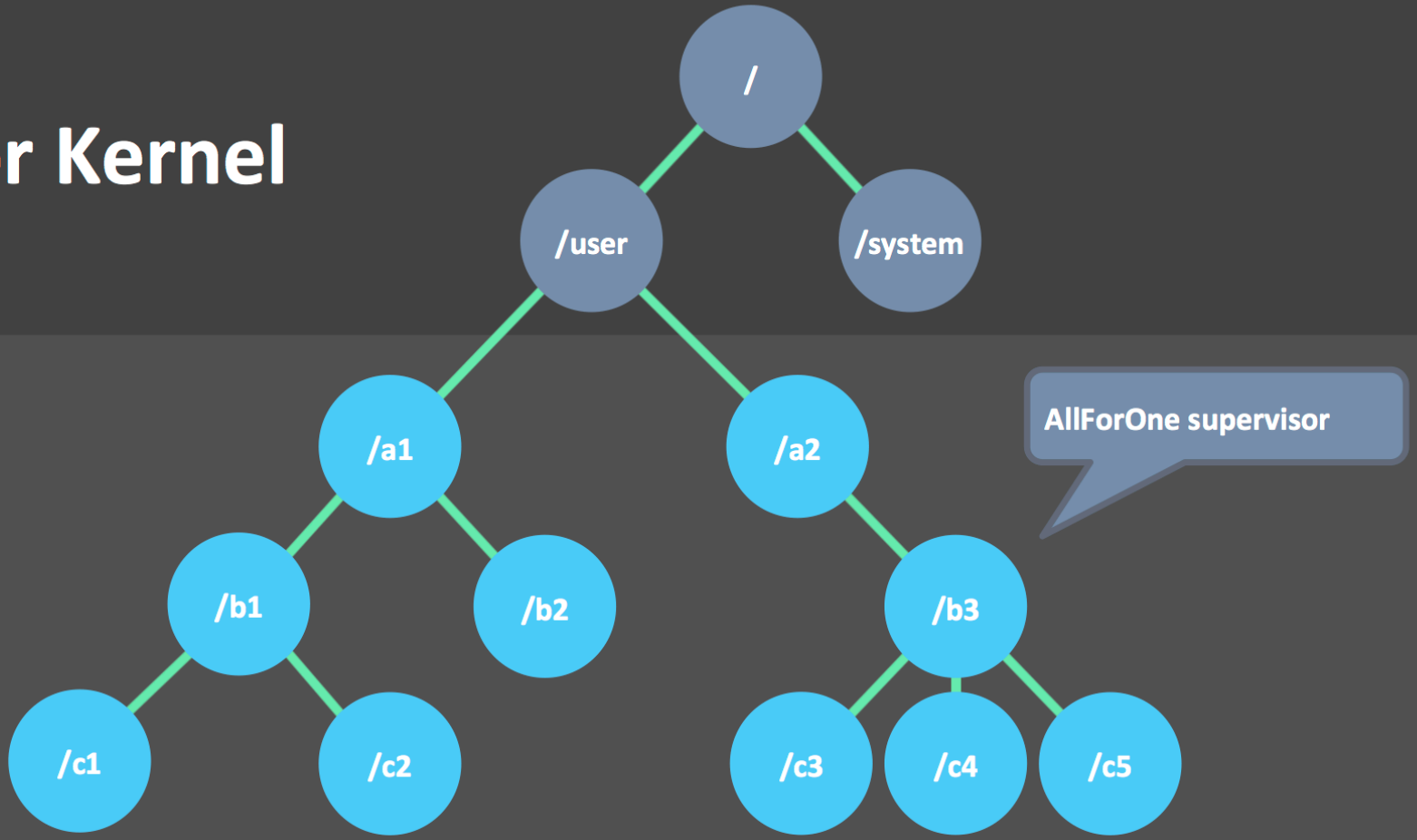
# Error Kernel



# Error Kernel



# Error Kernel



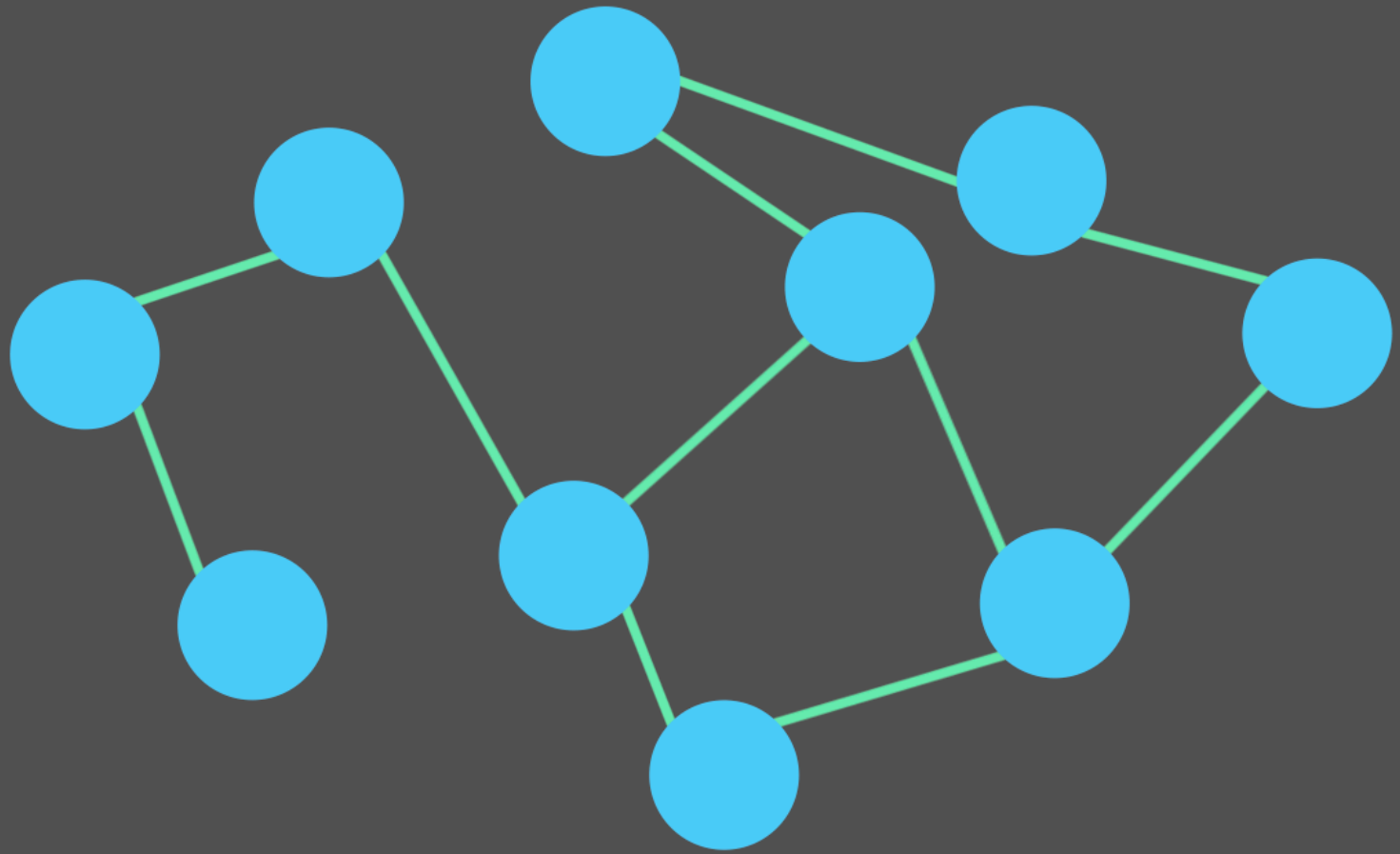
# Akka.Net Remoting

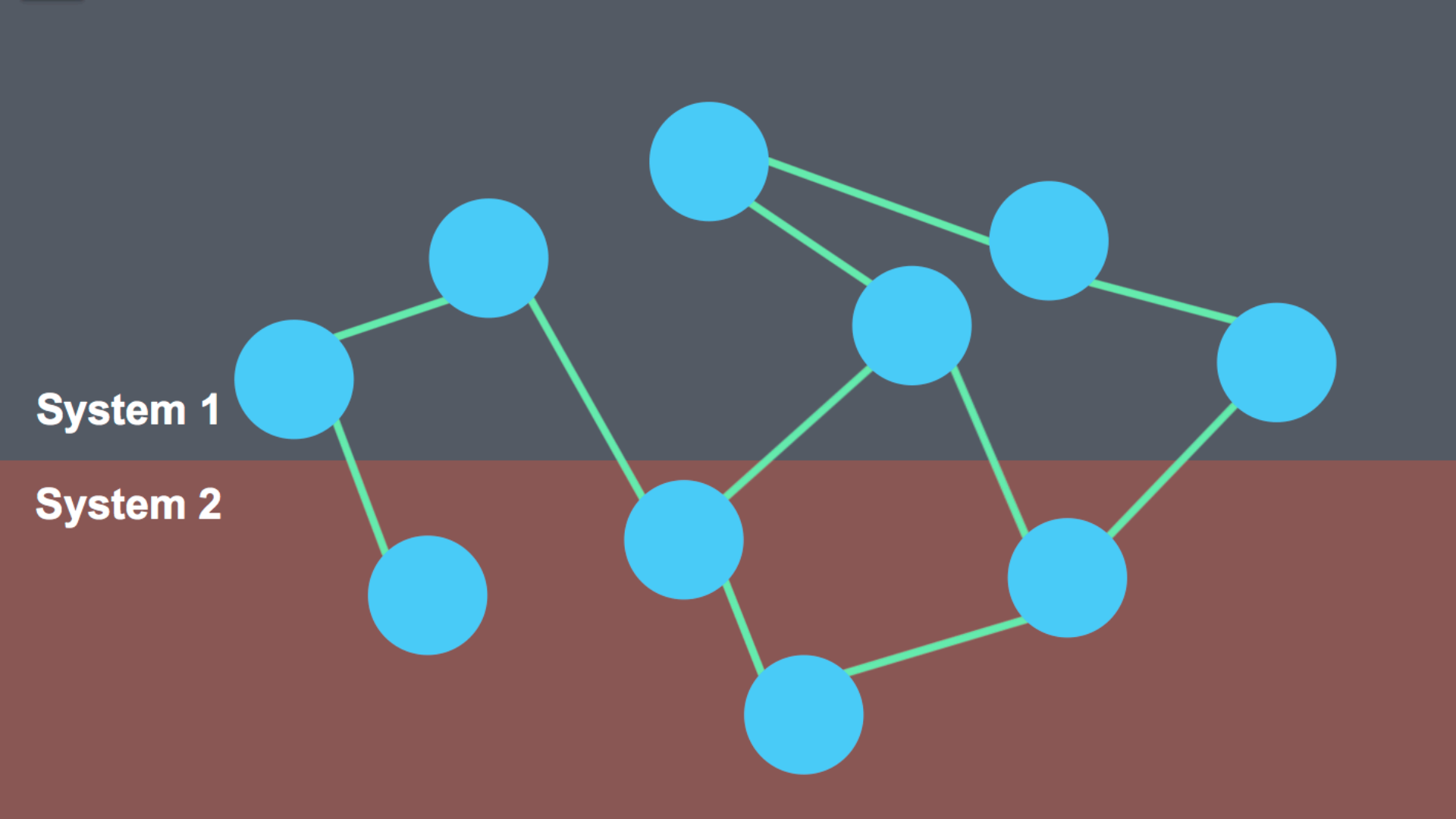

- Distributed by Default Mentality
- Provides a **unified programming model**
- Employees referential or location transparency
- Local and remote use the **same API**
- Distinguished by configuration, allowing a succinct way to code message-driven applications

***Going Scalable!***

# Location Transparency


What location transparency means is that whenever you send a message to an actor, you don't need to know where they are within an actor system, which might span hundreds of computers. You just have to know that actors' address.




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

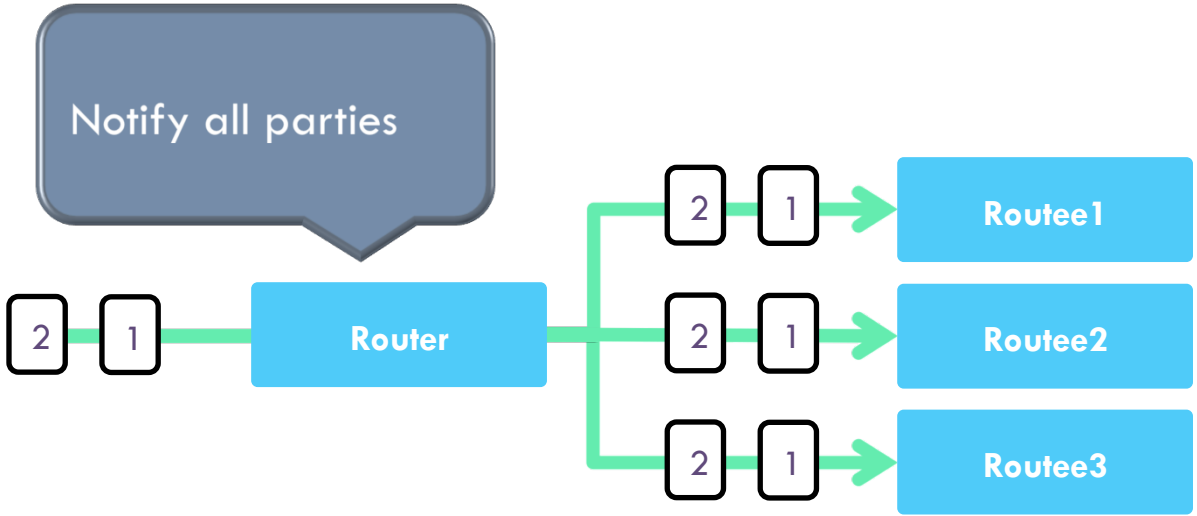
```
let aref =
    spawne localSystem "akka.tcp://remote-system@localhost:9234/" "hello"
      // actorOf wraps custom handling function with message receiver logic
      <@ actorOf (fun msg -> printfn "received '%s'" msg) @>

                [SpawnOption.Deploy (Deploy(RemoteScope (Address.Parse remoteSystemAddress)))]
```

# Akka.Net Routing


- Messages can be sent via a router to efficiently route them to destination actors, known as its routees. A Router can be used inside or outside of an actor, and you can manage the routees yourselves or use a self contained router actor with configuration capabilities.
- On the surface routers look like normal actors, but they are actually implemented differently. Routers are designed to be extremely efficient at receiving messages and passing them quickly on to routees.

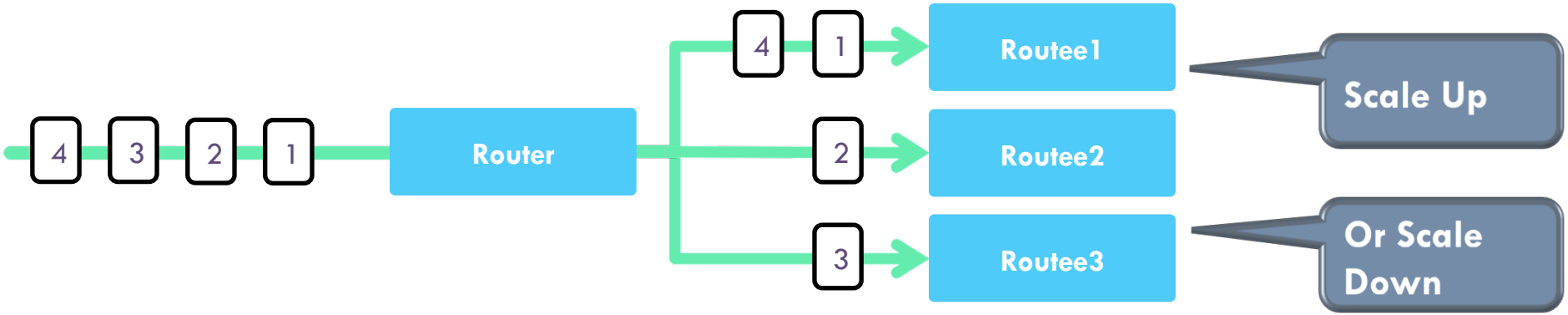
# Akka.Net Routing Strategies

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



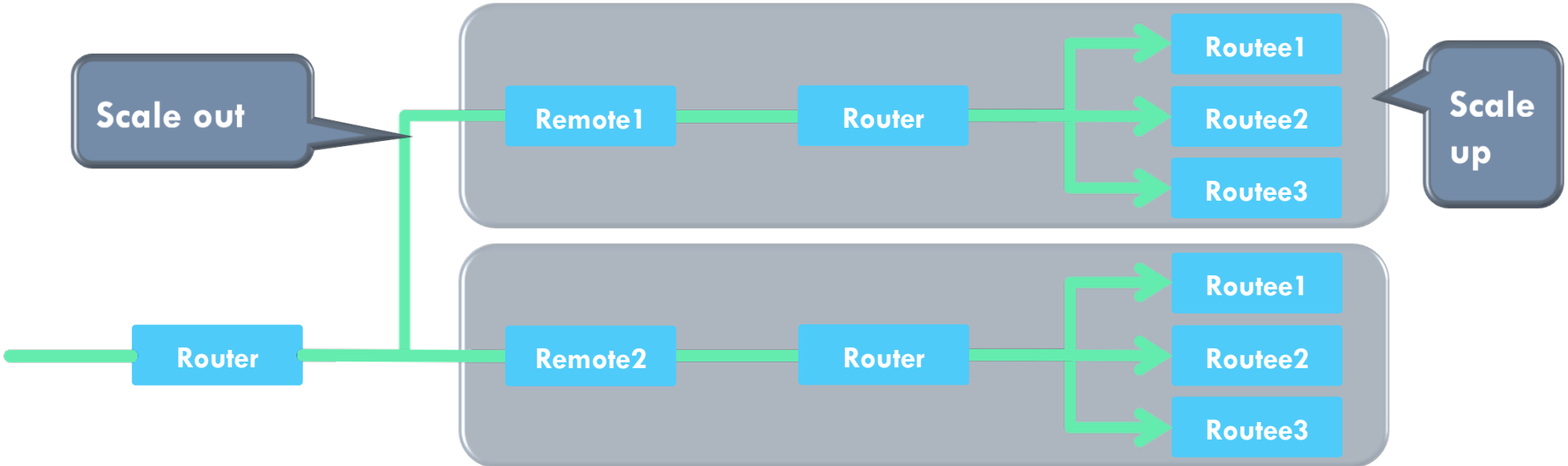
# Akka.Net Routing Strategies


- 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


TypedActors have a static interface, and the invocations of the methods on that interface is transformed into message sends

UntypedActors can receive any message and defines only one abstract method

| TypedActors vs. UntypedActors | |
|---|---|
| Advantage | Downside |
| TypedActors you have a static contract, and don't need to define your own messages | TypedActors places some limitations on what you can do and what you can't |

# F# & Akka.NET


To install Akka.NET Distributed Actor Framework

```
Install-Package Akka
Install-Package Akka.Remote
```

F# API support

```
Install-Package Akka.FSharp
```


DEMO

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

# Summary

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 Scalling Up & Out

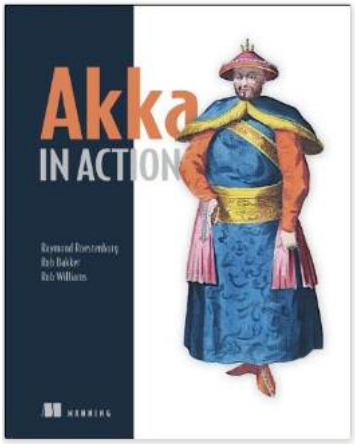
Akka provide a simple and unified programming model to reach high performance

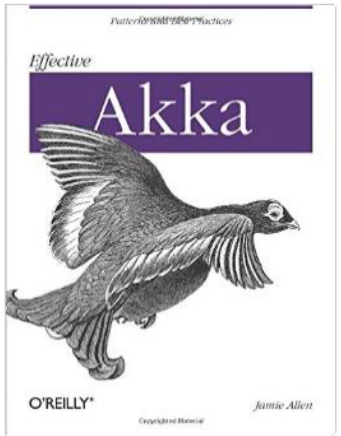

*The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities.*

*-- Edsger Dijkstra*

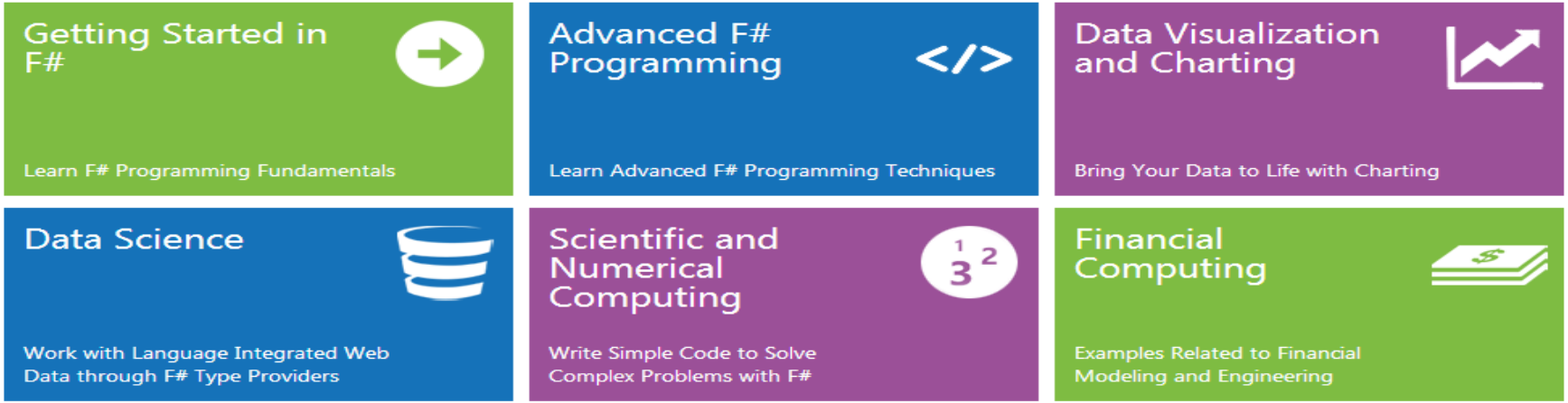
# References

- https://reactivemanifetso.com
- http://www.gettakka.net
- http://petabridge.com




# Online resources




- www.fsharp.org — Information & community
  www.tryfsharp.org — Interactive F# tutorials

# How to reach me


github.com/rikace/Presentations/AkkaActorModel

meetup.com/DC-fsharp

@DCFsharp @TRikace

rterrell@microsoft.com


That's all Folks!