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 – Community F#

Agenda

- What's kind of World is out there?
- Reactive Manifesto
- Actor model - What & Why
- F# Agent
- Akka.Net Actor in Action
- Akka.NET F# API

Something about me – Riccardo Terrell

- Originally from Italy, I am currently living in USA ~8 years
  - Currently living 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, believer 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 Tech-World

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

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

Hottest technology trends

- Mobile
- Claud computing
- Real Time notificat
  SignalR
- Single Page Apps
- Big Data

- ble
- nable & Readable
- e & Testable
- sable & Reusable
- rent

The World is changed

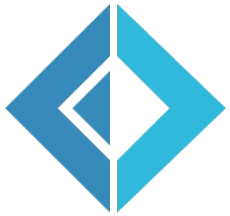
"In today's world, the demand for distributed systems has exploded. As customer expectations such as an immediate response, no failure, and access anywhere increase, companies have come to realize that distributed computing is the only viable solution."

- Reactive Application Development (Manning)

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

- Reactive Application Development (Manning)

The Problem..

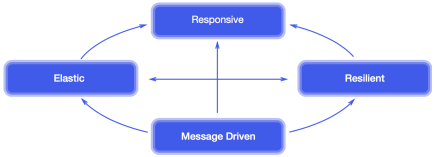


It is hard to build correct highly concurrent systems

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

It is hard to build resiliant & fault-tolerant System that self-heals

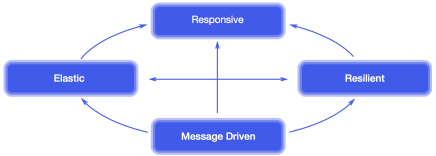
Reactive Manifesto



- Responsive
- Message-Driven
- Resilient
- Elastic

http://www.reactivemanifesto.org

Reactive Manifesto

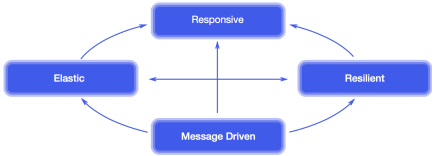


- Responsive
- Message-Driven
- Resilient
- Elastic

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

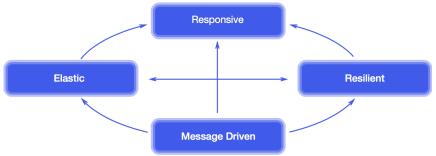


- Responsive
- Message-Driven
- Resilient
- Elastic

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

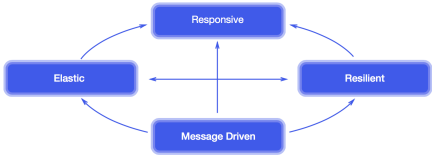


- Responsive
- Message-Driven
- Resilient
- Elastic

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



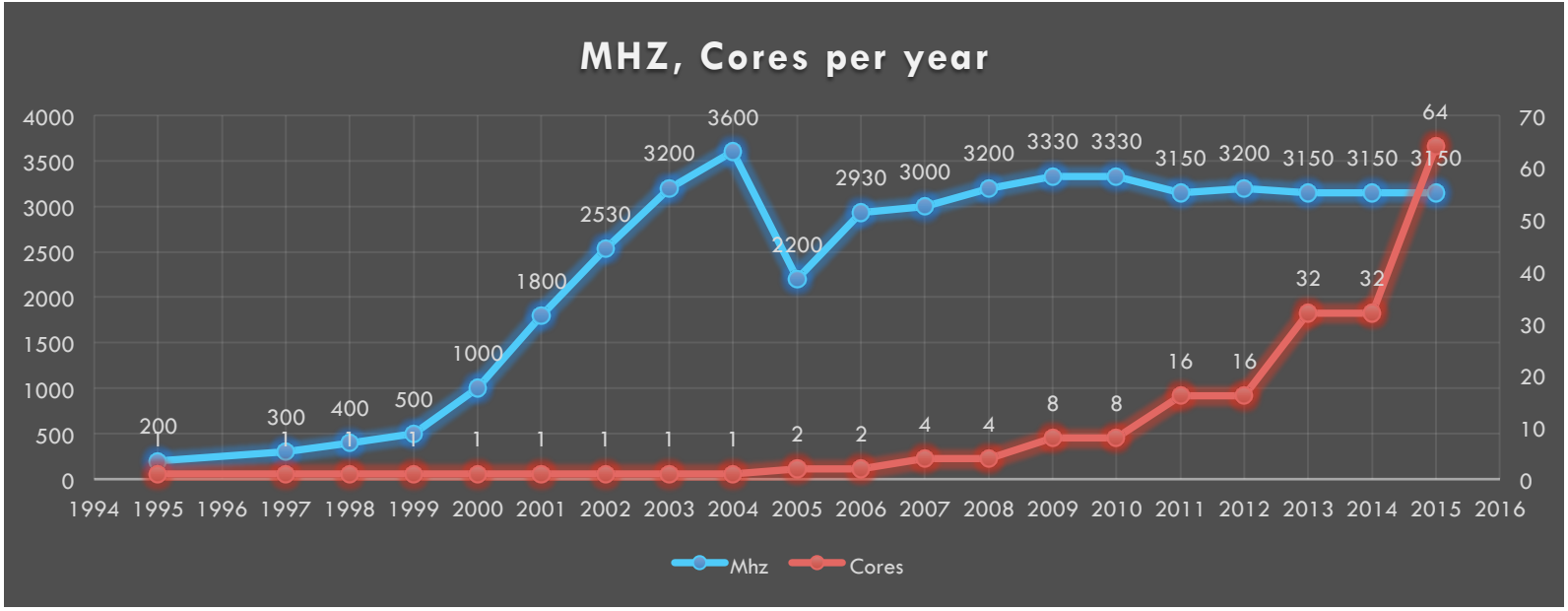
- Responsive
- Message-Driven
- Resilient
- Elastic

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 used to be!



Concurrency – The free lunch is over

There is a problem…

the free lunch is over

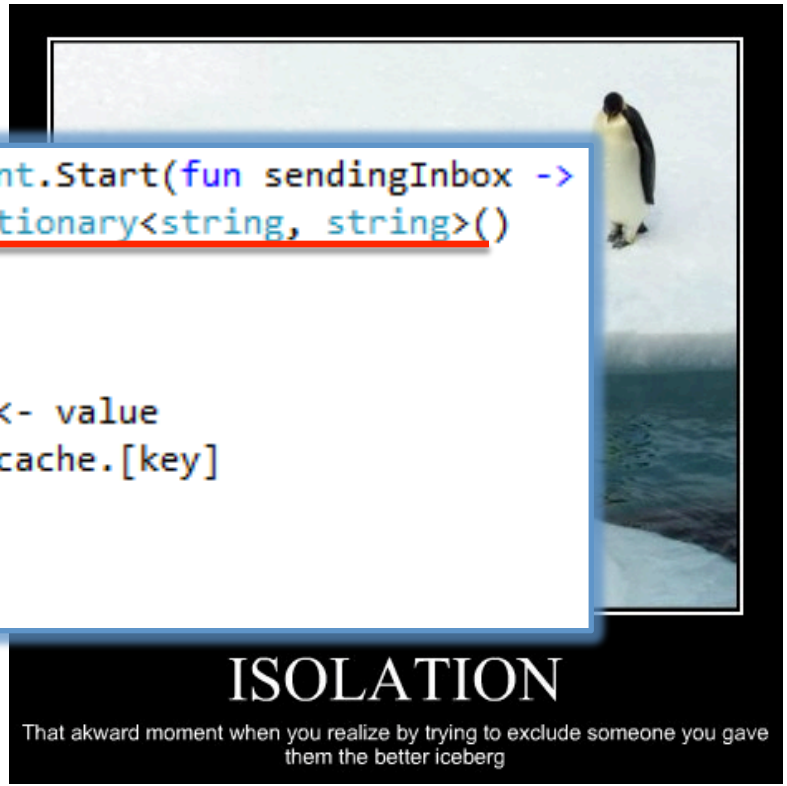
- Programs are not doubling in speed every couple of years for free anymore
- We need to start writing code to take advantage of many cores

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

Immutability & Isolation

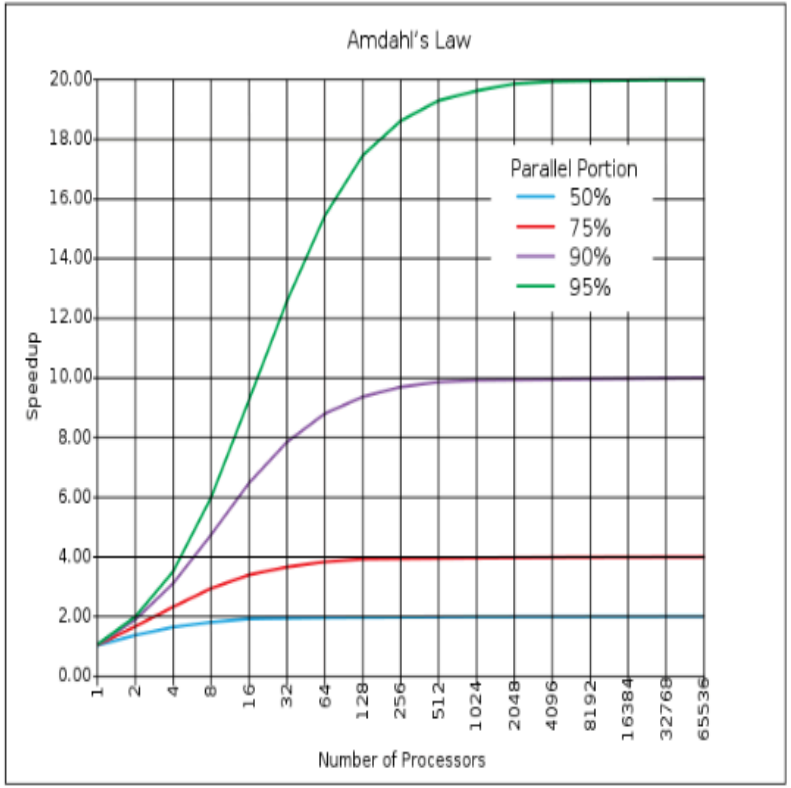
```
let (lockfree:Agent<Msg<string,string>>) = Agent.Start(fun sendingInbox ->
    let cache = System.Collections.Generic.Dictionary<string, string>()
    let rec loop () = async {
        let! message = sendingInbox.Receive()
        match message with
            | Push (key,value) -> cache.[key] 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



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



Concurrent Model Programming

An Actor is an independent computational entity which contains a queue, and receives and processes messages

It provides immutability and isolation

(it enforces coarse-grained isolation through message-passing)

What is

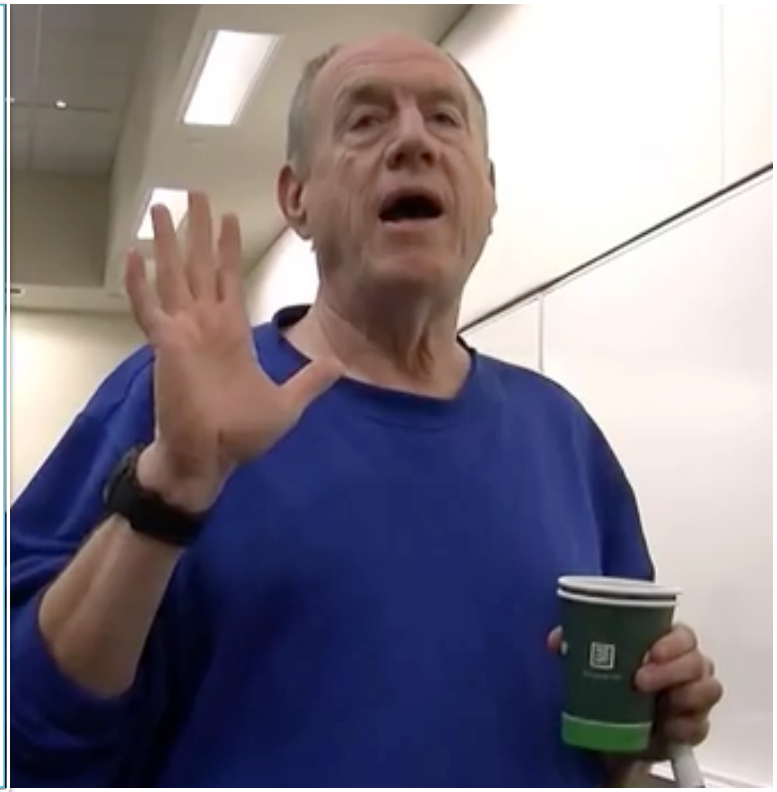
The Actor m ation
using actor eation
of actors, i s, and
interaction age
passing wit er.

kipedia]

What is an Actor?

- Processing
- Storage – State
- Communication only by messages
- Share Nothing
- Message are passed by value
- Lightweight object.
- Running own itself own thread.
- No shared state.
- Messages are kept in mailbox and processed in order.
- Massive scalable and lighting 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"

Tre axiom:

- Send messages to other Actors
  - One Actor is not Actor
- Create other Actor
- Decide how to hanlde next Message - Behavior

Real World Actor Model

- WhatsApp (Erlang)
- RabbitMQ
- CouchDB

- Twitter

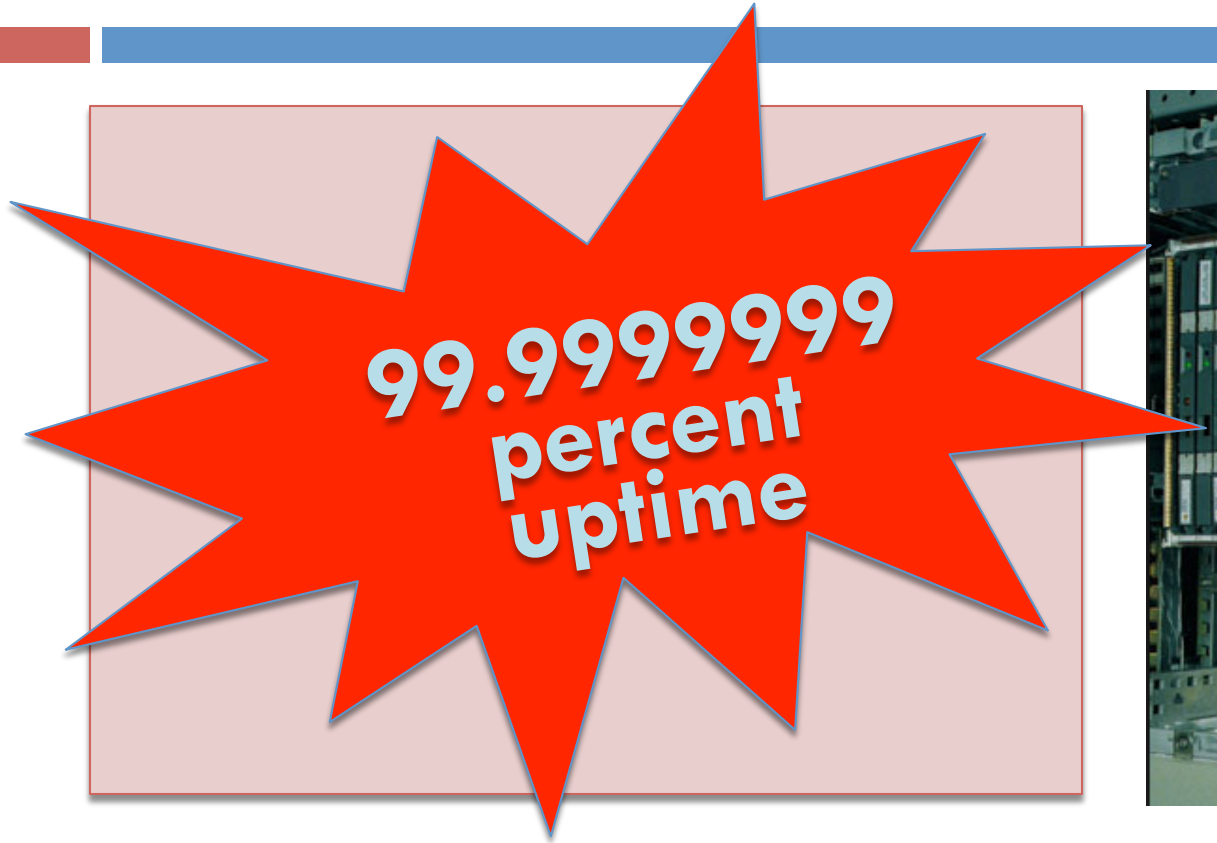
- Facebook

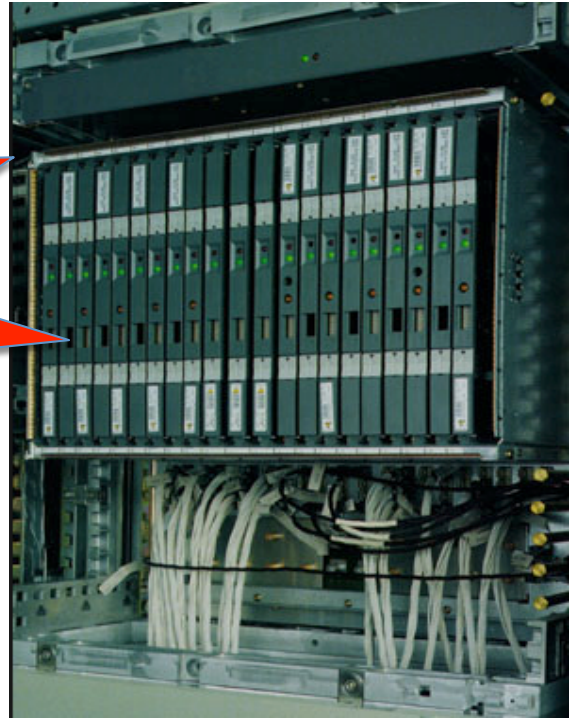
- Ejabberd XMPP
  - jabber.org

- LinkedIn.com (Akka)

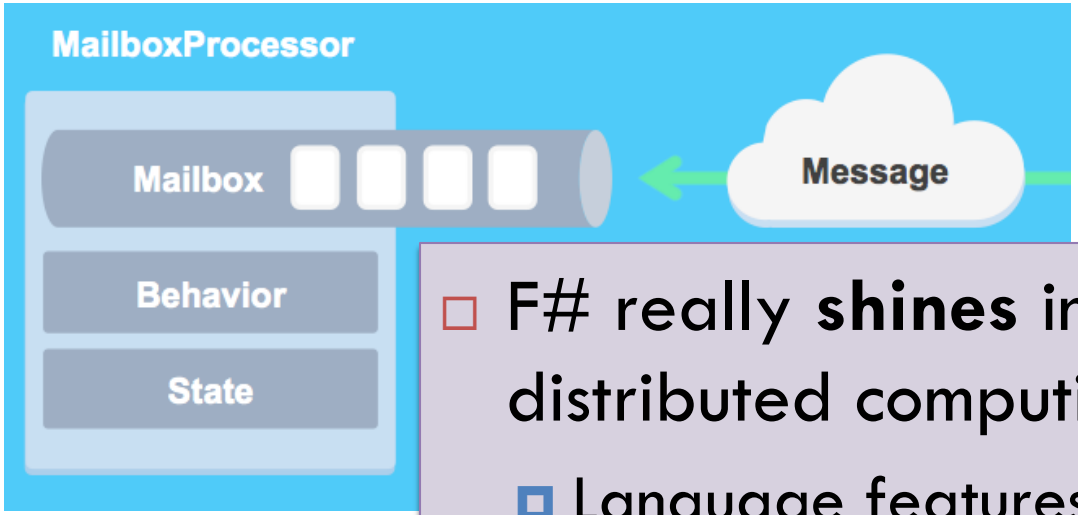
- Walmart.com (Akka)

Ericson AXD 301 Switch - 1996



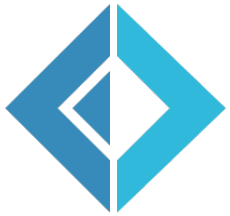


F# MailboxProcessor – aka Agent



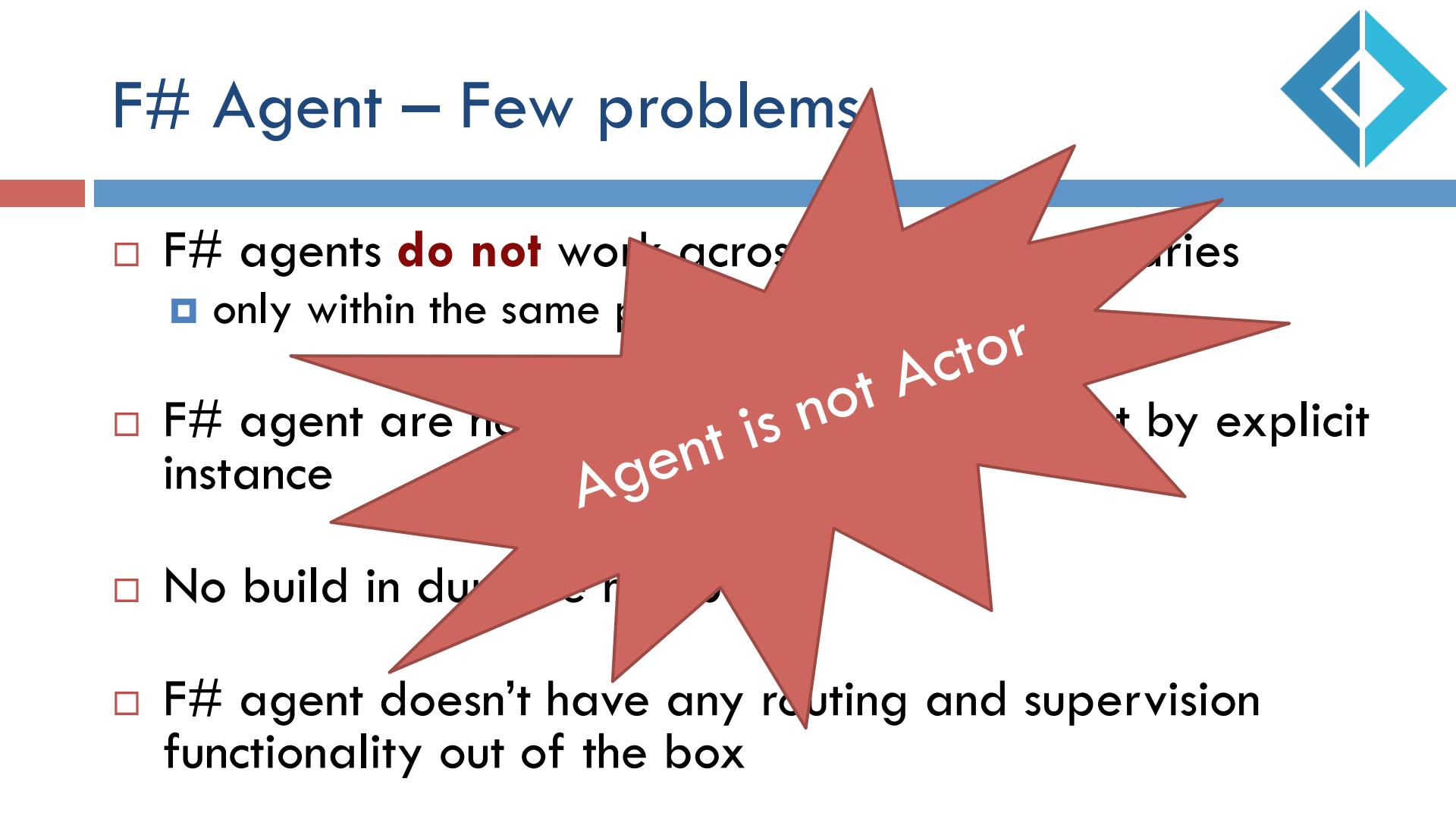
- F# really **shines** in the area of distributed computing
  - Language features such as Async Workflow and MailboxProcessor (a.k.a. agent) open the doors for computing that focuses on message passing concurrency

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)
```

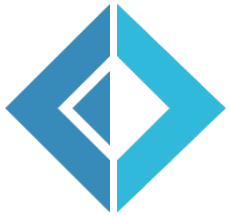
F# Agent – Few problems



- F# agents **do not** wo[illegible] acros[illegible] [illegible]aries
  - only within the same p[illegible]
- F# agent are n[illegible] [illegible]t by explicit instance
- No build in du[illegible] r[illegible]
- F# agent doesn't have any r[illegible]uting and supervision functionality out of the box

Agent is not Actor

F# 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)
  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.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**

Anatomy of 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

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

Akka.Net is a Scalable, distributed real-time transaction processing

- **Actors**
  - Simple and high-level abstractions for concurrency and parallelism
  - Asynchronous, non-blocking and highly performing event-driven programming model
- **Fault Tolerance**
  - Supervisor hierarchies with "let-it-crash" semantics
  - Supervisor hierarchies can span multiple VMs to provide truly fault-tolerant systems
  - Excellent for writing highly fault-tolerant systems that self-heal and never stop
- **Location Transparency**
  - Everything in Akka is designed to work in a distributed environment: all interactions of actors use pure message passing and everything is asynchronous.

Multi-Threads vs Multi Actors

1 MB per thread (4 Mb in 64 bit) vs 2.7 milion of Actor per Gigabyte

VS

What problems Akka can solve?

- Distributed System
- Asynchronous
- High Performance

Akka provides high-level abstractions for concurrency and parallelism.

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.

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 & Akka.NET

- Responsive — Event Driven
- Message-Driven — Communication by messages
- Resilient — Fault tolerant by Supervision
- Elastic — Clustering and Remoting across multiple machines

How Act lure?

akka.net

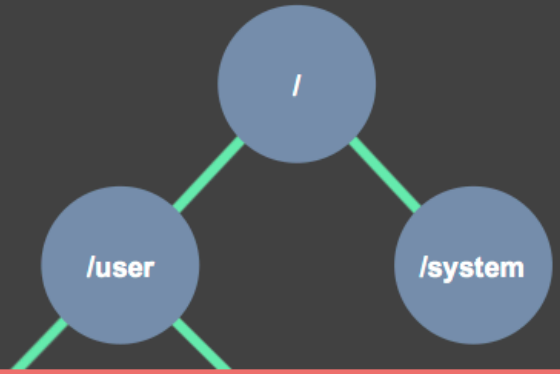
How do you keep y things go wrong?

How can the paren

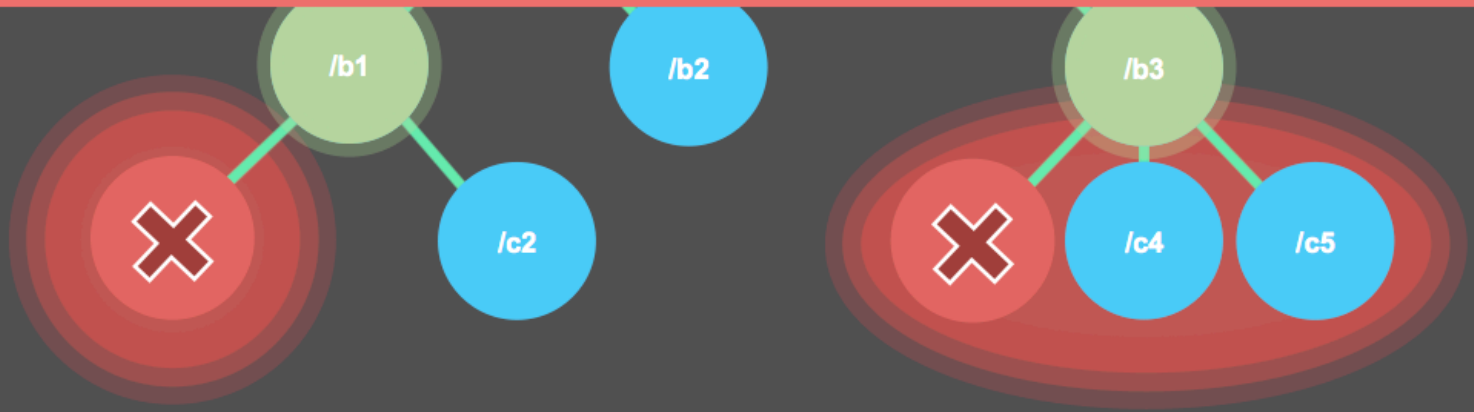
There are tw d:
- How the chi
- Parent's **Su**

KEEP CALM AND LET IT CRASH

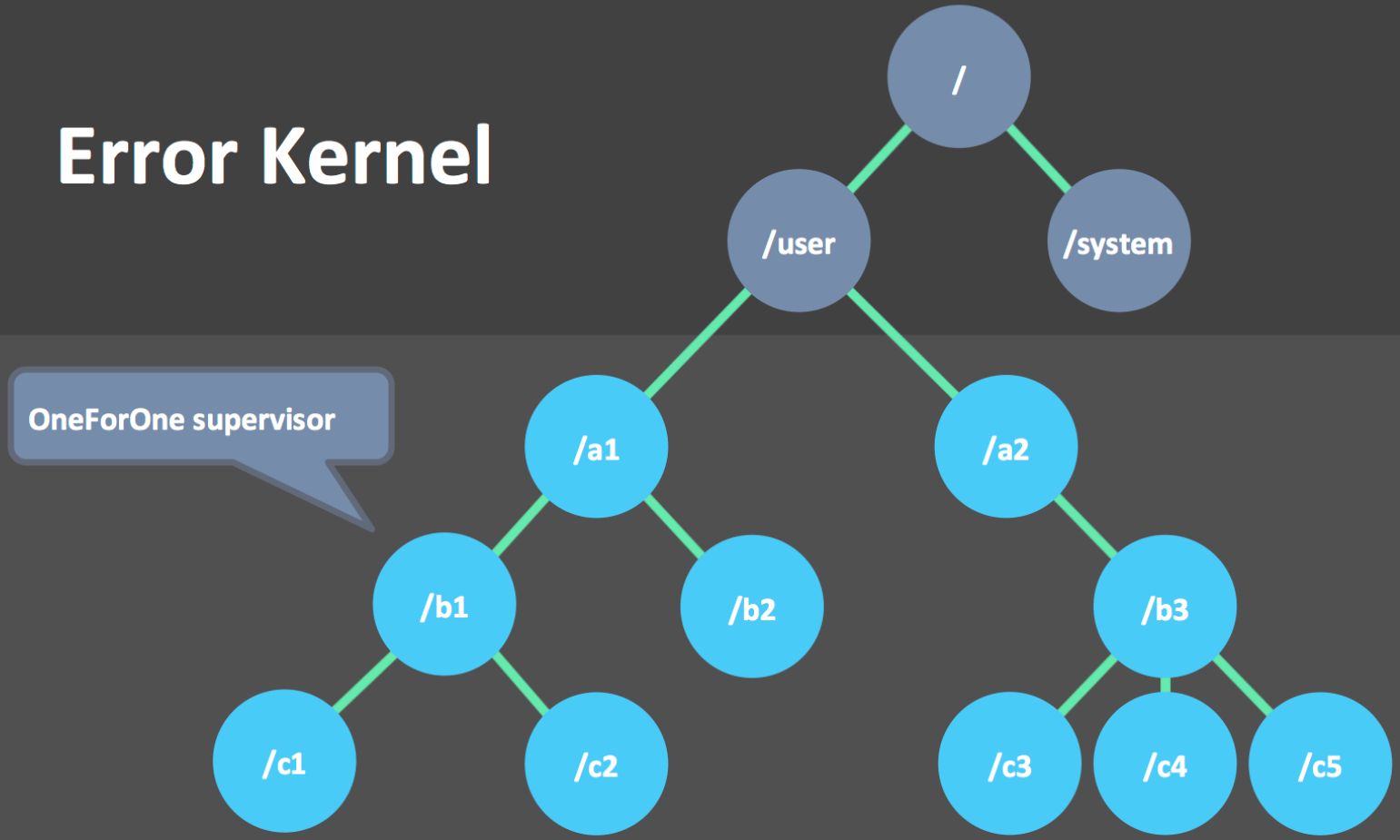
Error Kernel



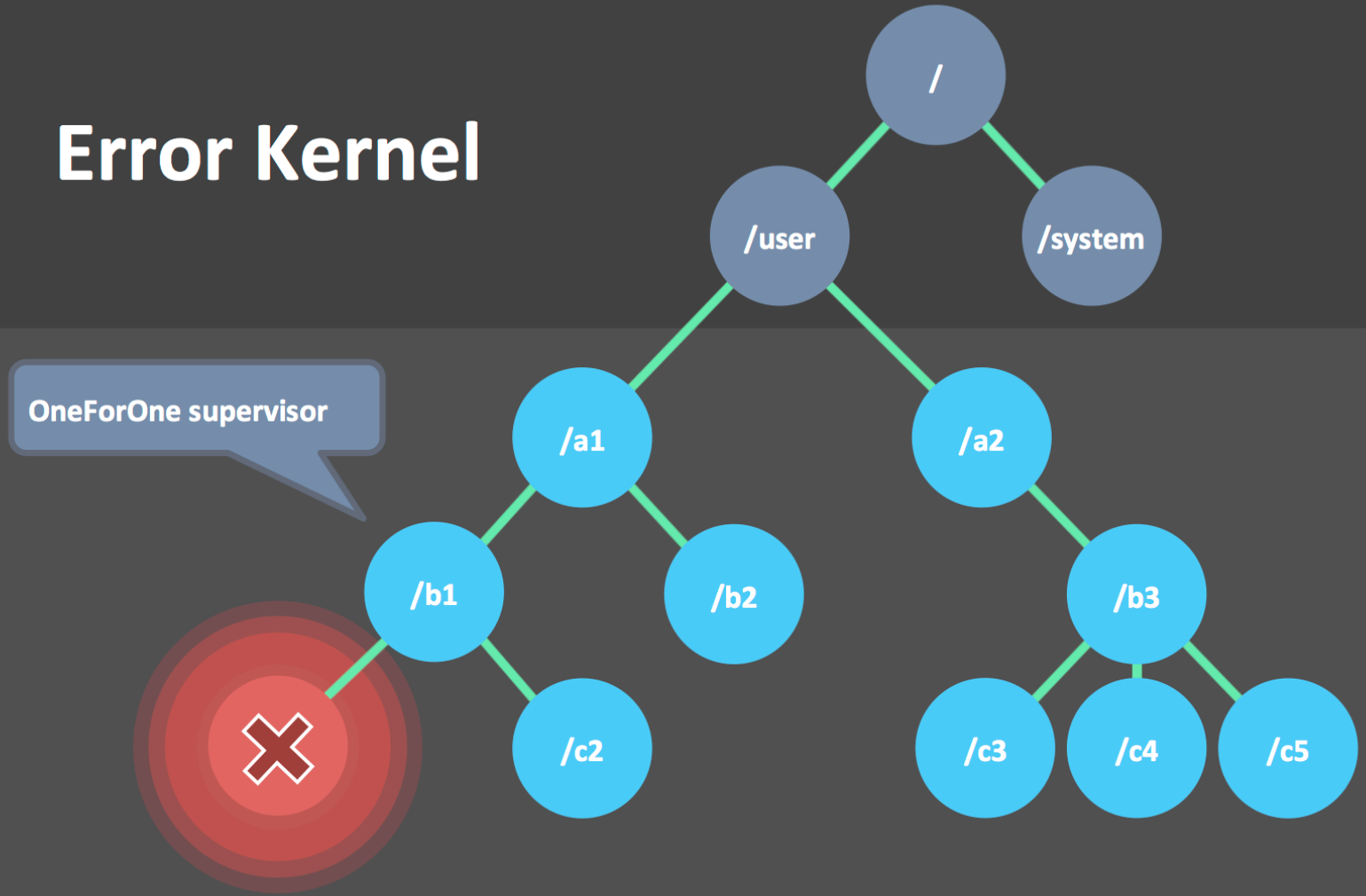
99.9999999% uptime
~0.03 seconds downtime per year



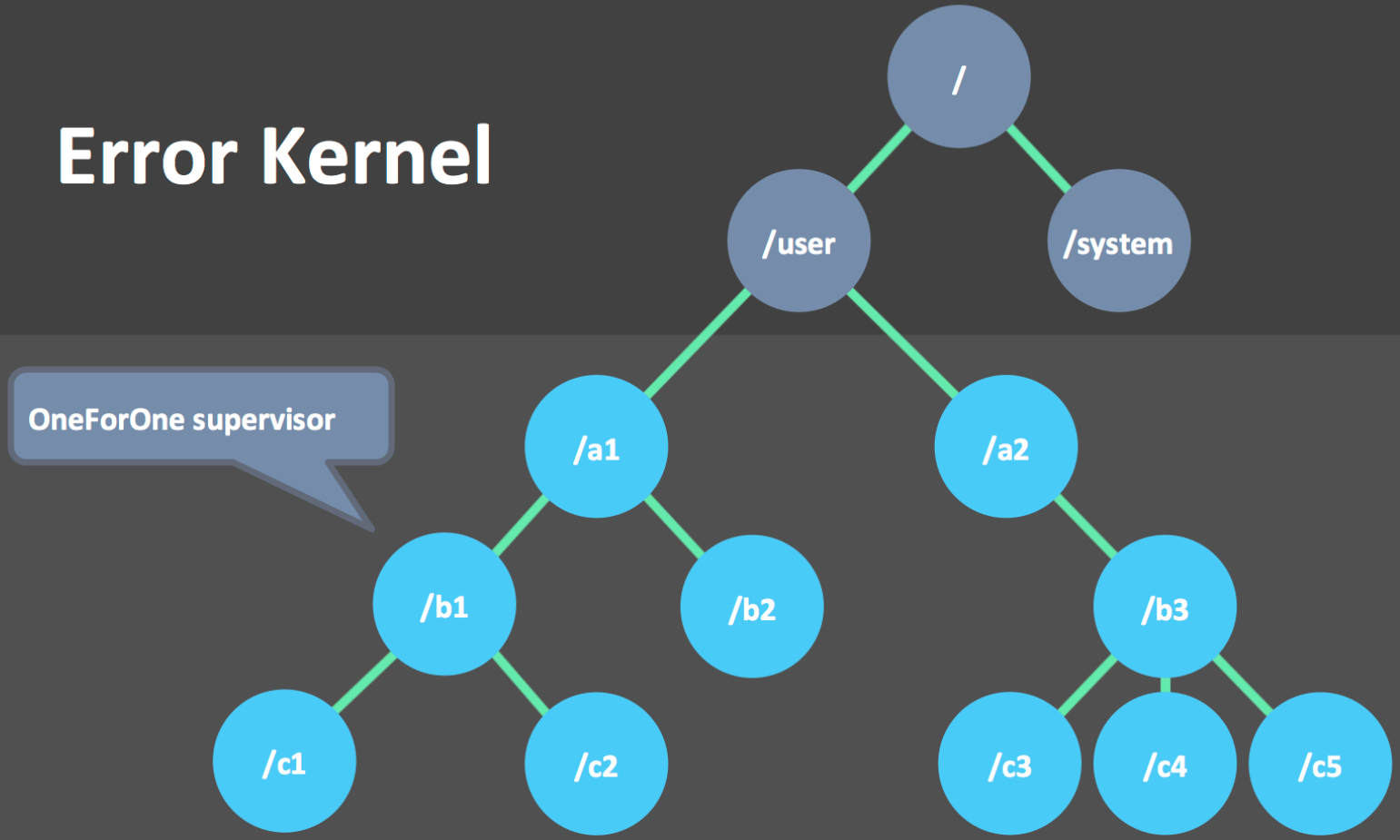
Error Kernel



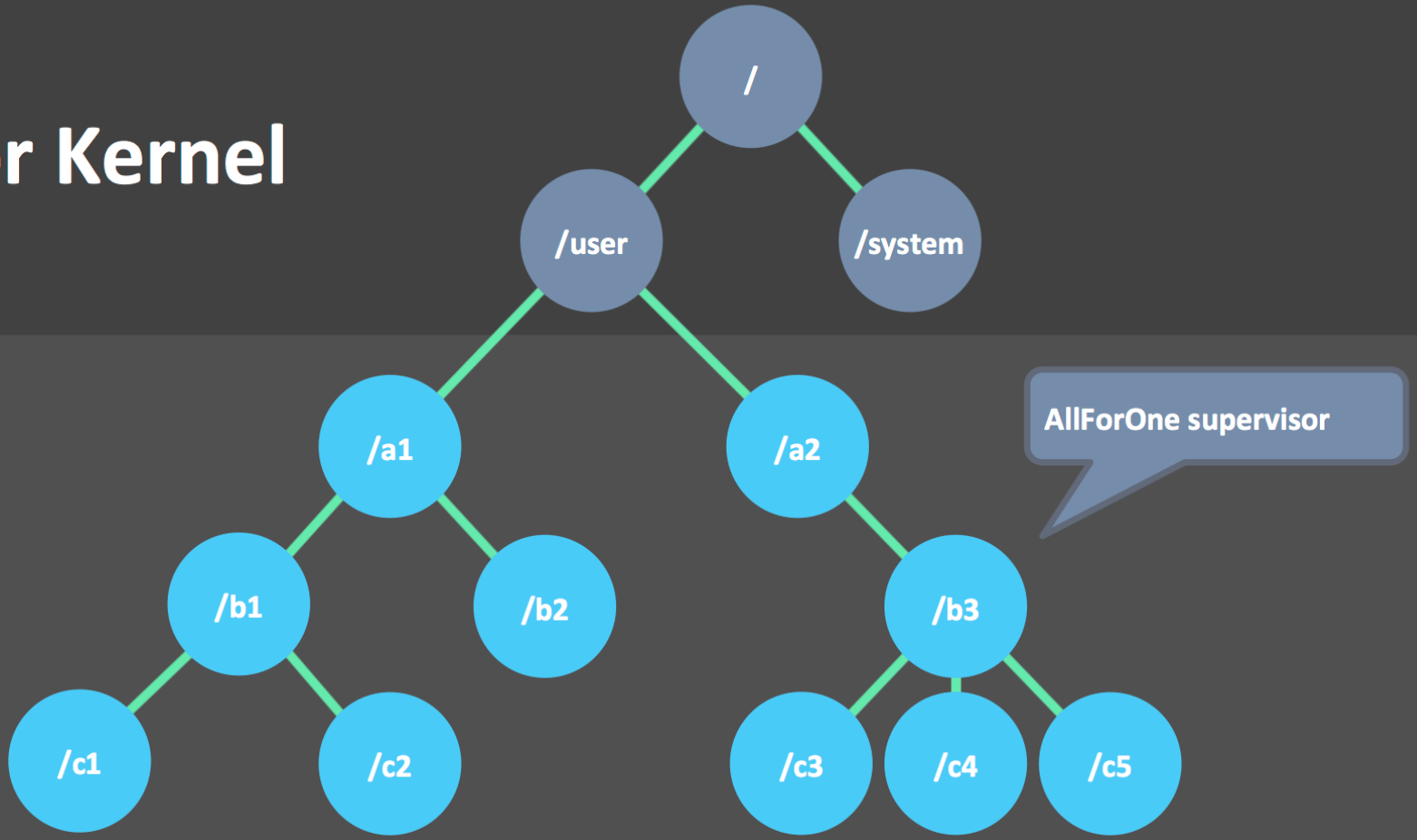
Error Kernel



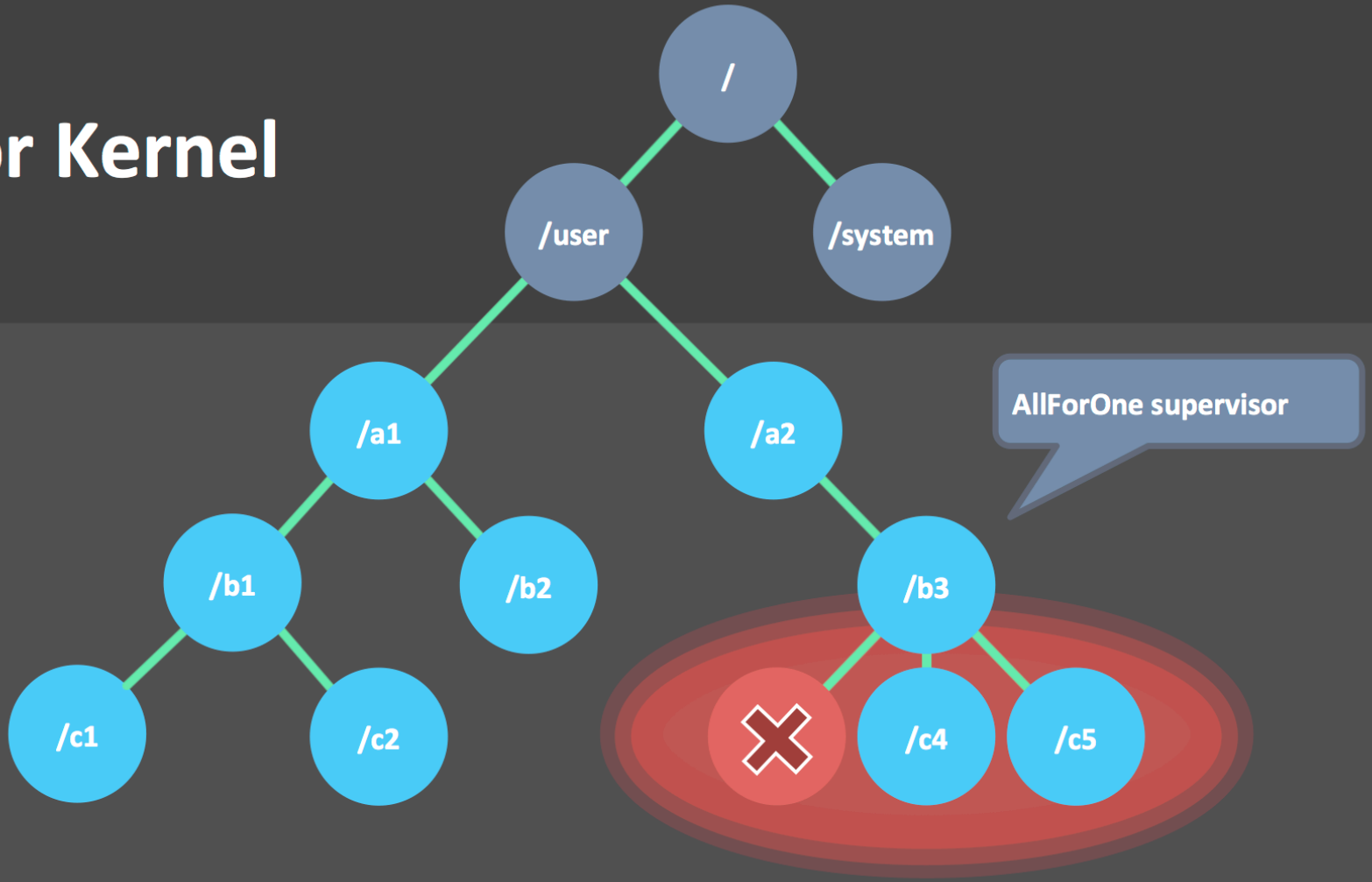
Error Kernel



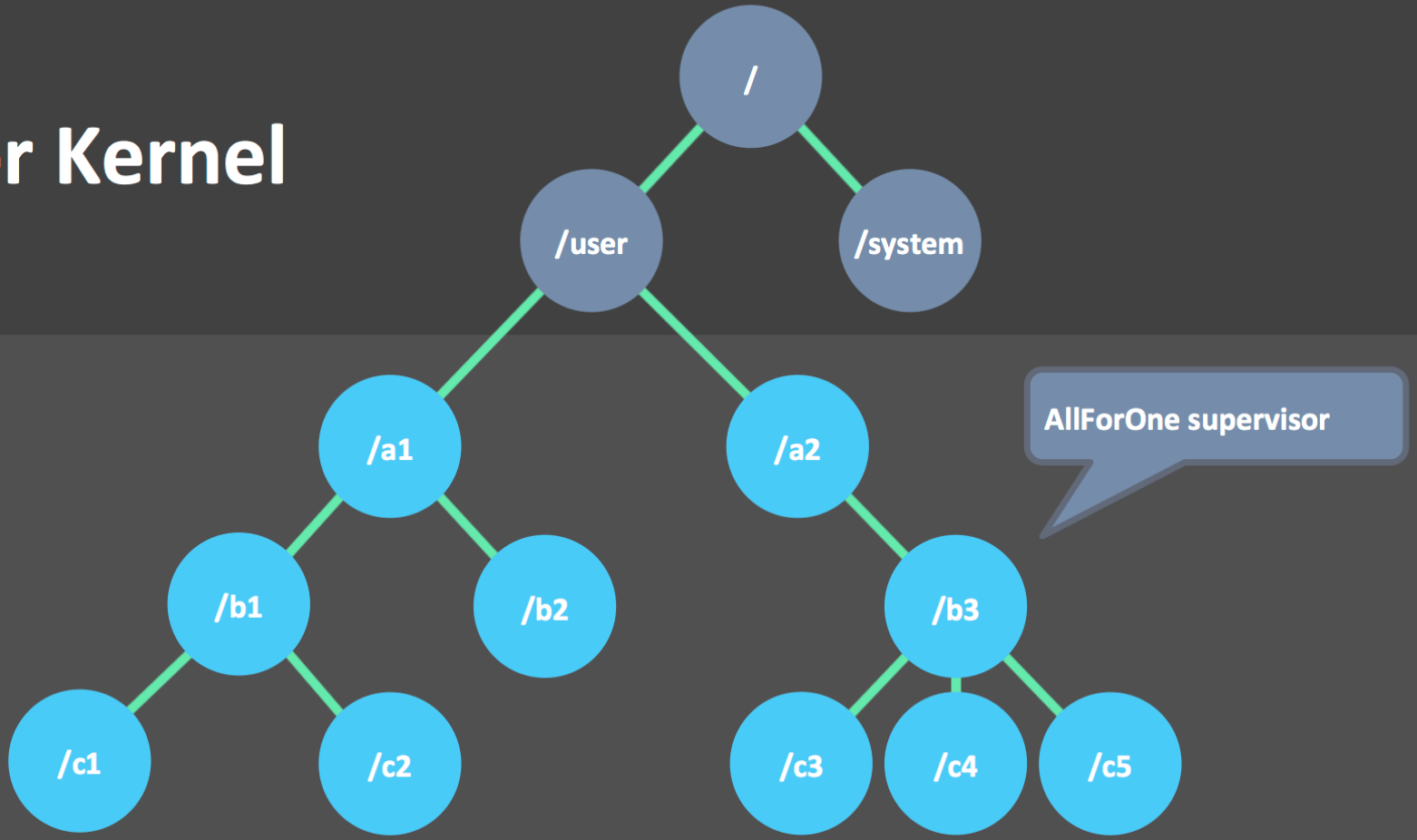
Error Kernel



Error Kernel



Error Kernel



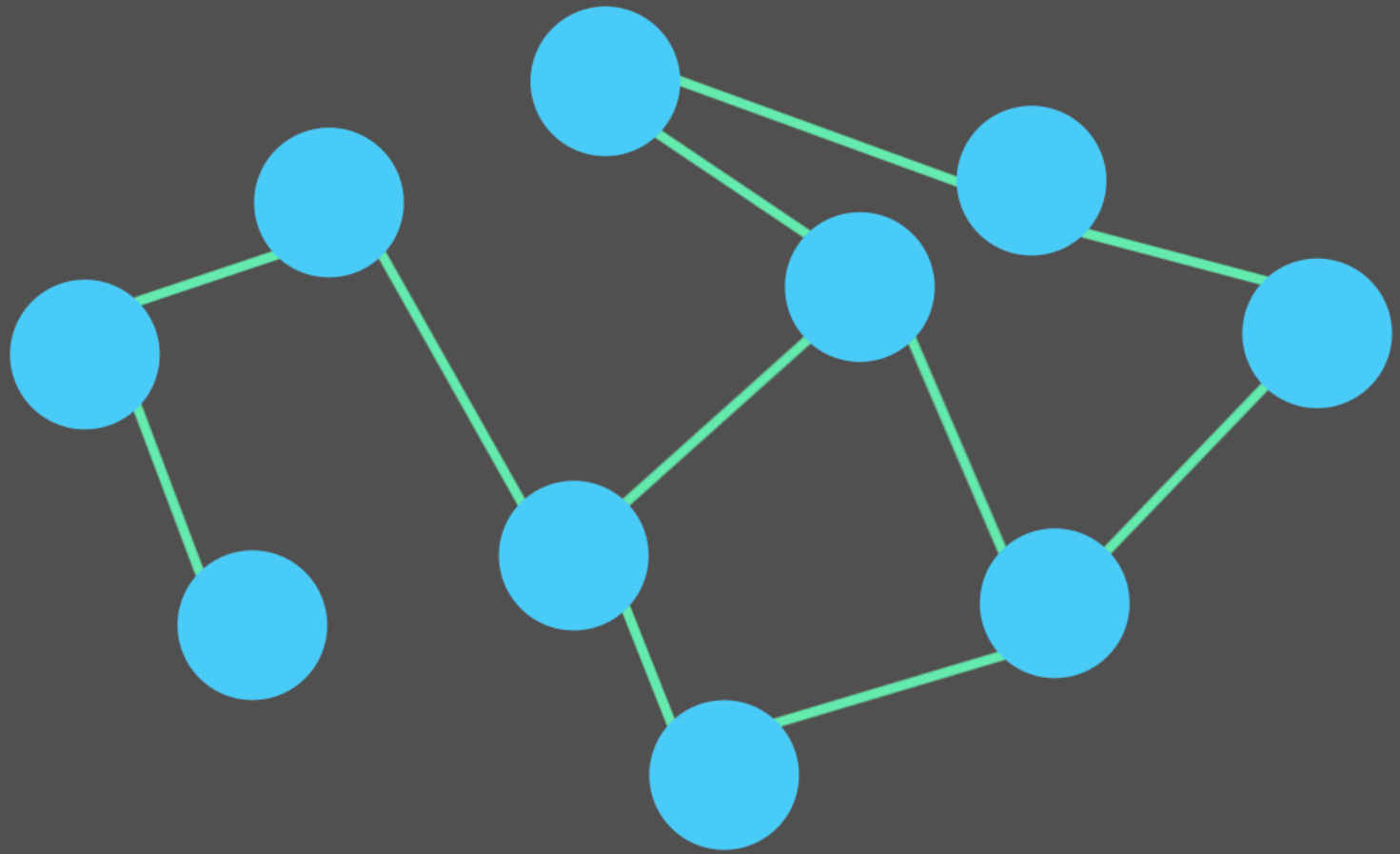
Akk.Net & F# Supervision

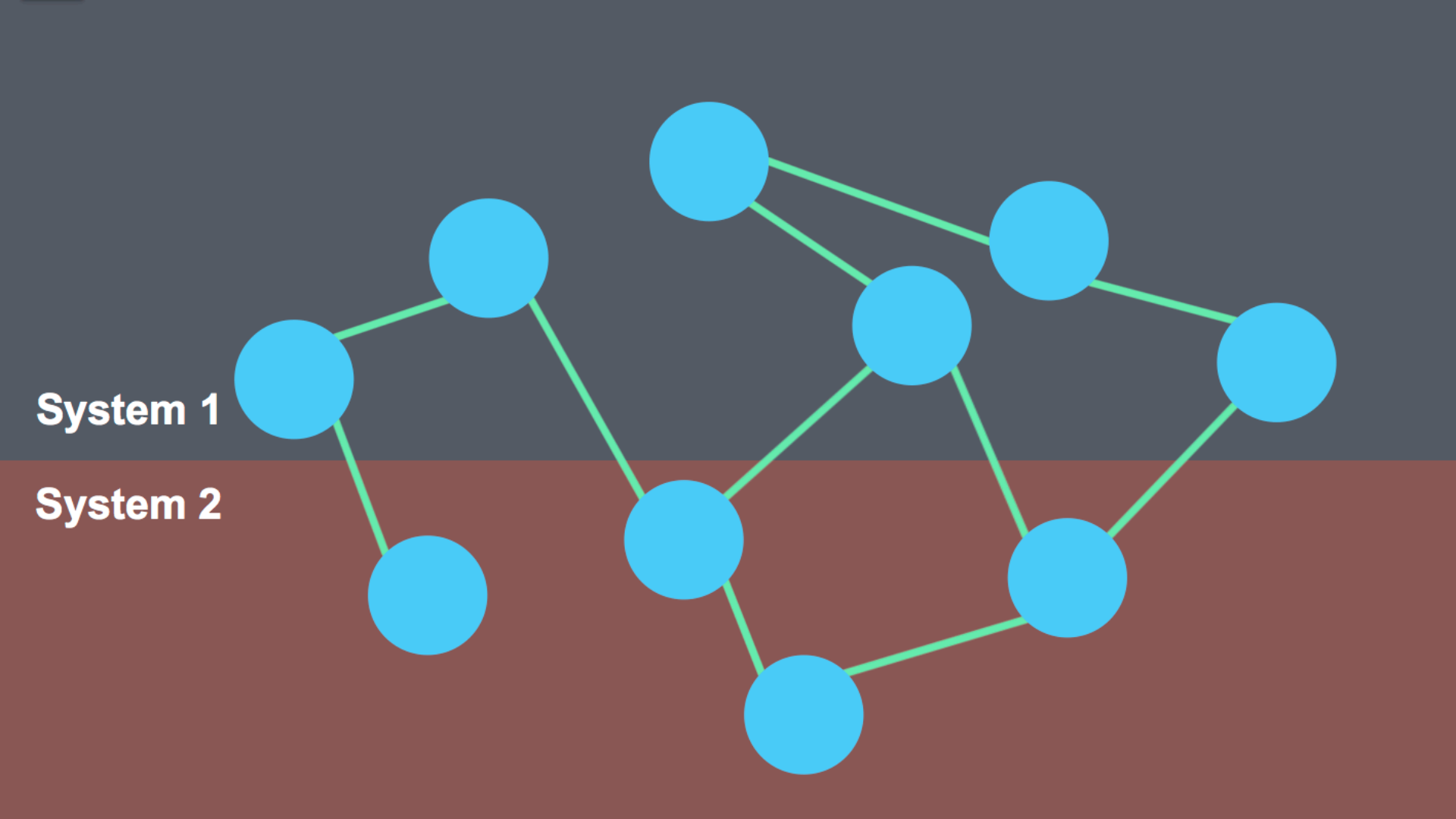

```
let options = [SpawnOption.SupervisorStrategy(Strategy.OneForOne (fun e -> Directive.Restart))]

let strategy =
    Strategy.OneForOne (fun e ->
        match e with
        | :? DivideByZeroException -> Directive.Resume
        | :? ArgumentException -> Directive.Stop
        | _ -> Directive.Escalate)

let actor = spawnOpt system "actor" <|
                        fun mailbox ->
                            let rec loop () =
                                actor {
                                    let! msg = mailbox.Receive ()
                                    match msg with
                                    | Boom -> raise <| Exception("Oops")
                                    | Print s -> printfn "%s" s
                                                 return! loop ()
                                }
                            loop ()
                        <| options
```

Akka.Net Remote Actors

- 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!




System 1
System 2

Location Transparency

What
messa
an ac
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#!!")
```

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)))]
```

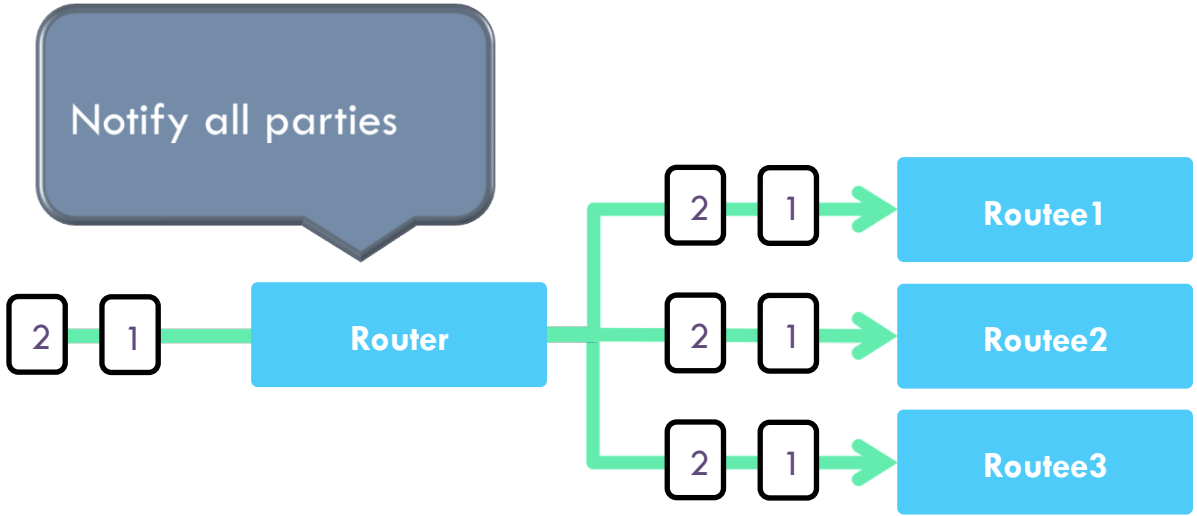
Akka.Net Routing

- Messages can be sent
  destination actors, knov
  or outside of an actor,
  or use a self contained
- On the surface routers
  implemented different
  efficient at receiving m
  routees.

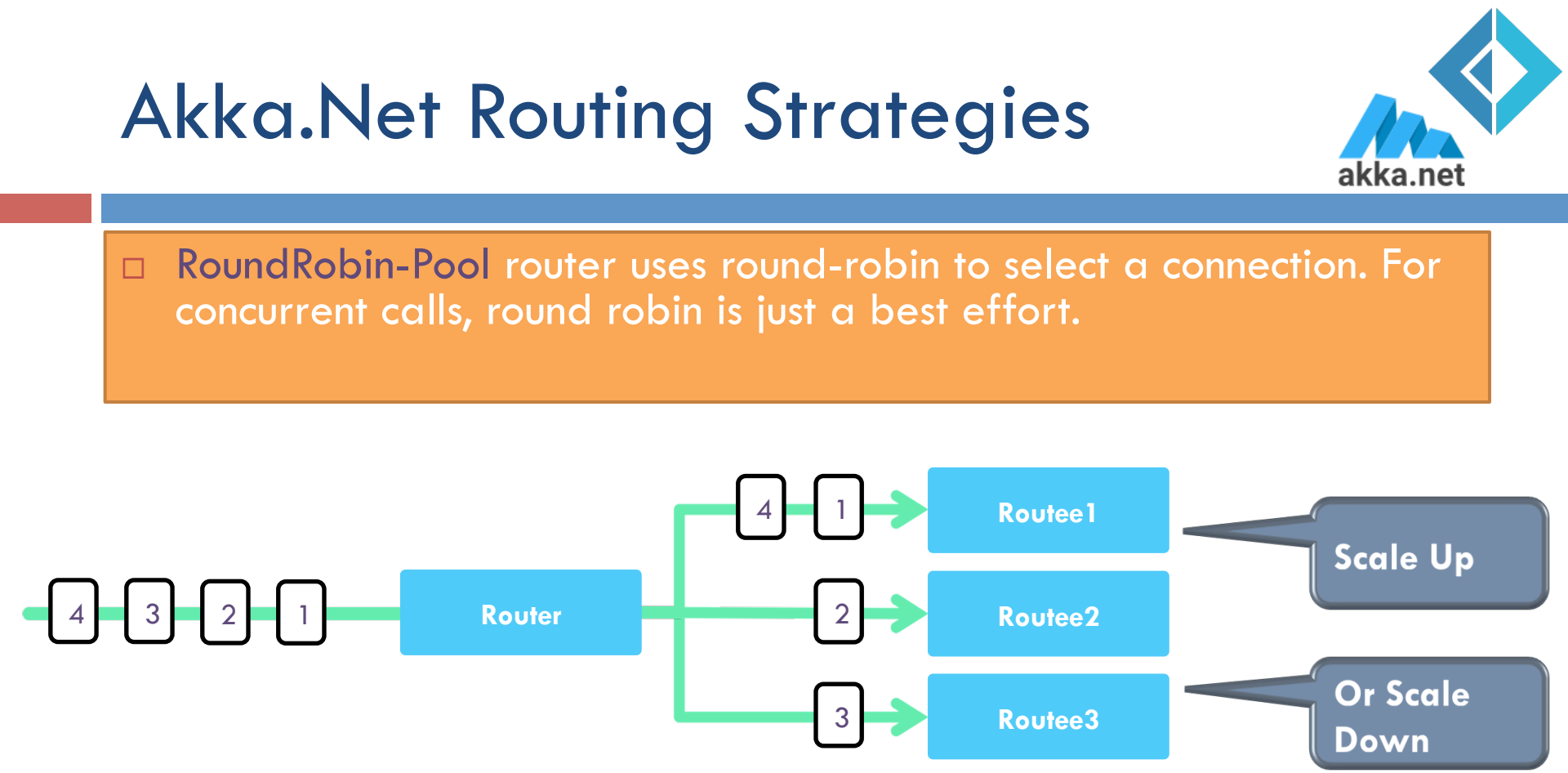
```
let config = ConfigurationFactory.ParseString(@"
                    akka {
                        actor {
                            deployment {
                                /localactor {
                                    router = round-robin-pool
                                    nr-of-instances = 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

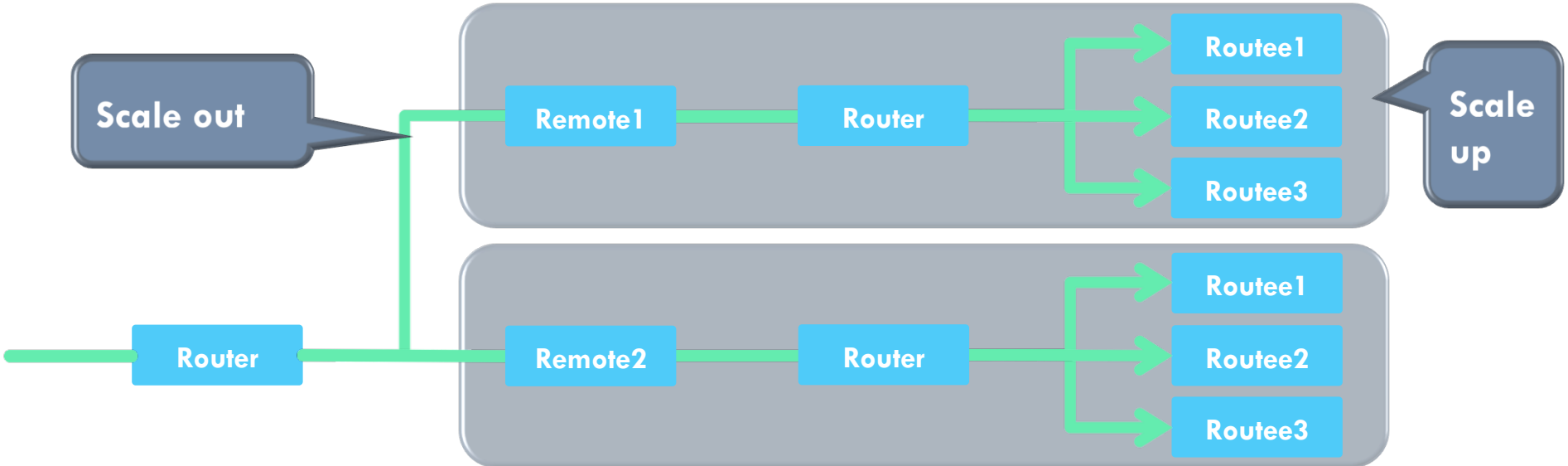


Akka.Net Routing Strategies



Akka.Net Routing Strategies

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



Actor as State Machine

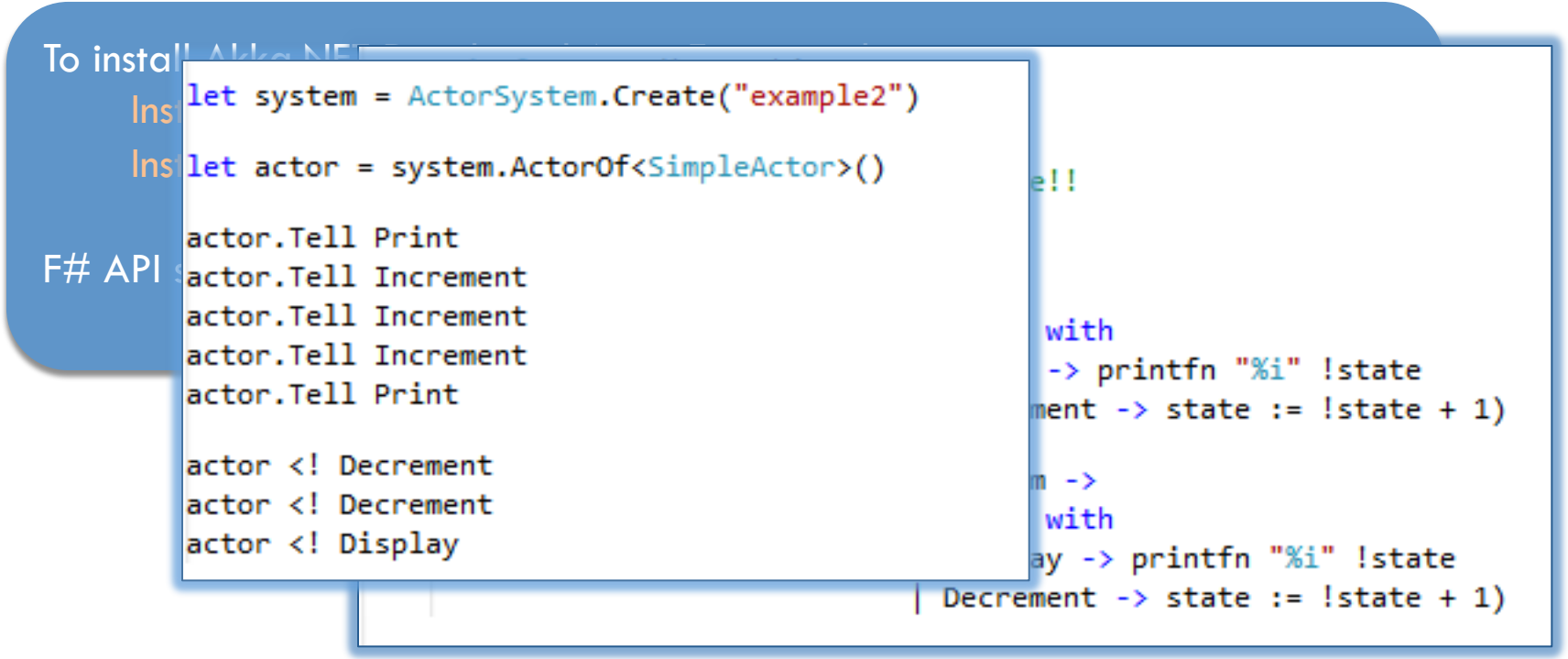
- Single
  - Accep
    to all
- **Exam**

```
// define the actor
let climateControl = MailboxProcessor.Start( fun inbox ->

    // the 'heating' state
    let rec heating() = async {
        printfn "Heating"
        let! msg = inbox.Receive()
        match msg with
        | CoolDown -> return! cooling()
        | _ -> return! heating()}

    // the 'cooling' state
    and cooling() = async {
        printfn "Cooling"
        let! msg = inbox.Receive()
        match msg with
        | HeatUp -> return! heating()
        | _ -> return! cooling()}

    // the initial state
    heating()
```

ts a

us

F# & Akka.NET


To instal
Inst
Inst
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
e!!
with
-> printfn "%i" !state
ment -> state := !state + 1)
m ->
with
ay -> printfn "%i" !state
| Decrement -> state := !state + 1)

Typed & Untyped Actor

```
let chatServerActor =
    spawn system "ChatServer" <| fun mailbox ->
        let rec loop (clients:ActorRef list) = actor {
            let! (msg:ChatMessage) = mailbox.Receive()
            match msg with
            | SayRequest(username, text) ->
                (* ... *)
                ()
            | ConnectRequest(username) ->
                (* ... *)
                ()
            | NickRequest(oldUsername, newUsername) ->
                (* ... *)
                ()
            }
        loop []
```


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 – Left routing for last and think Synchronous first	

Actor Pros & Cons

Pros	Cons
Easier to reason about Higher abstraction level Easier to avoid Race conditions - Deadlocks - Starvation Live locks Distributed computing	Actors don't work well when When shared of state is needed to achieve global consensus (Transaction) Synchronous behavior is required 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 Tech-World

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

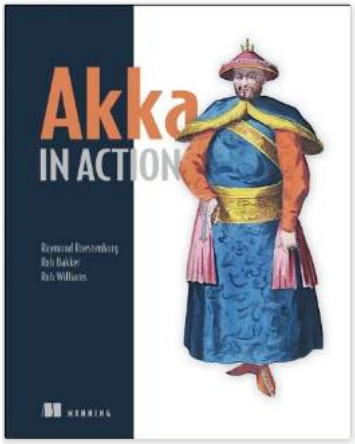
Actor is ~~the best~~ a great concurrent programming model that solves the problems for Scalling Up & Out

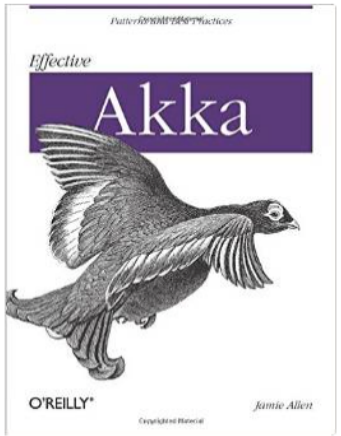
The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities.

-- Edsger Dijkstra

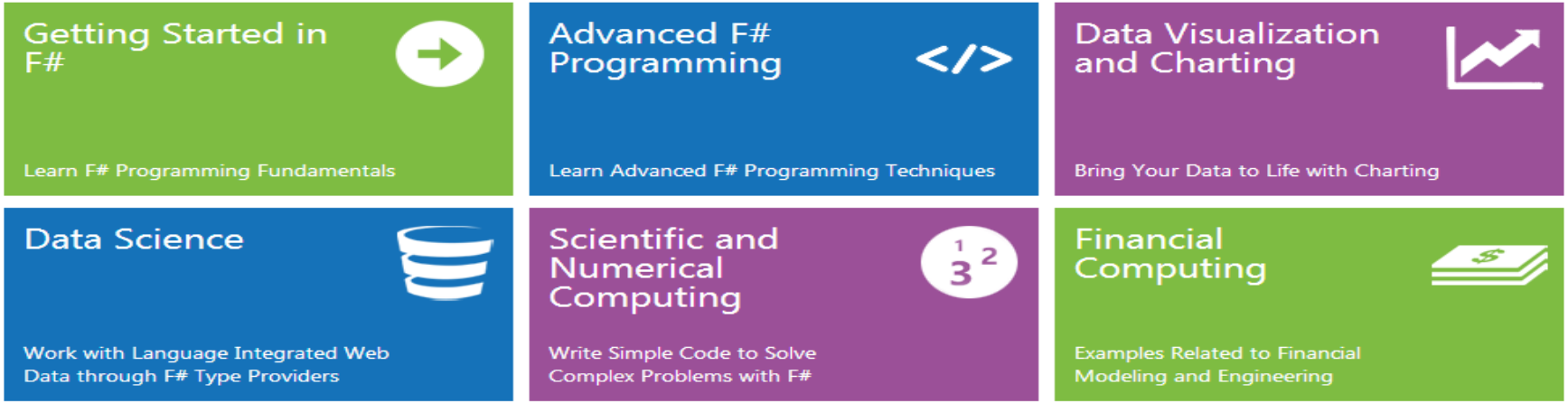
References

- https://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/AkkaActorModel

meetup.com/DC-fsharp

@DCFsharp @TRikace

rterrell@microsoft.com


That's all Folks!