



# **VIRTUAL ACTORS**

## **The What, Why, and How**

**Mark Tkachenko**  
**AtlasDelivery.io**  
Moscow, 2019-12-14



# We use Akka: why?

---

- CQRS + ES
- Scalable parallelism
- Easier state management

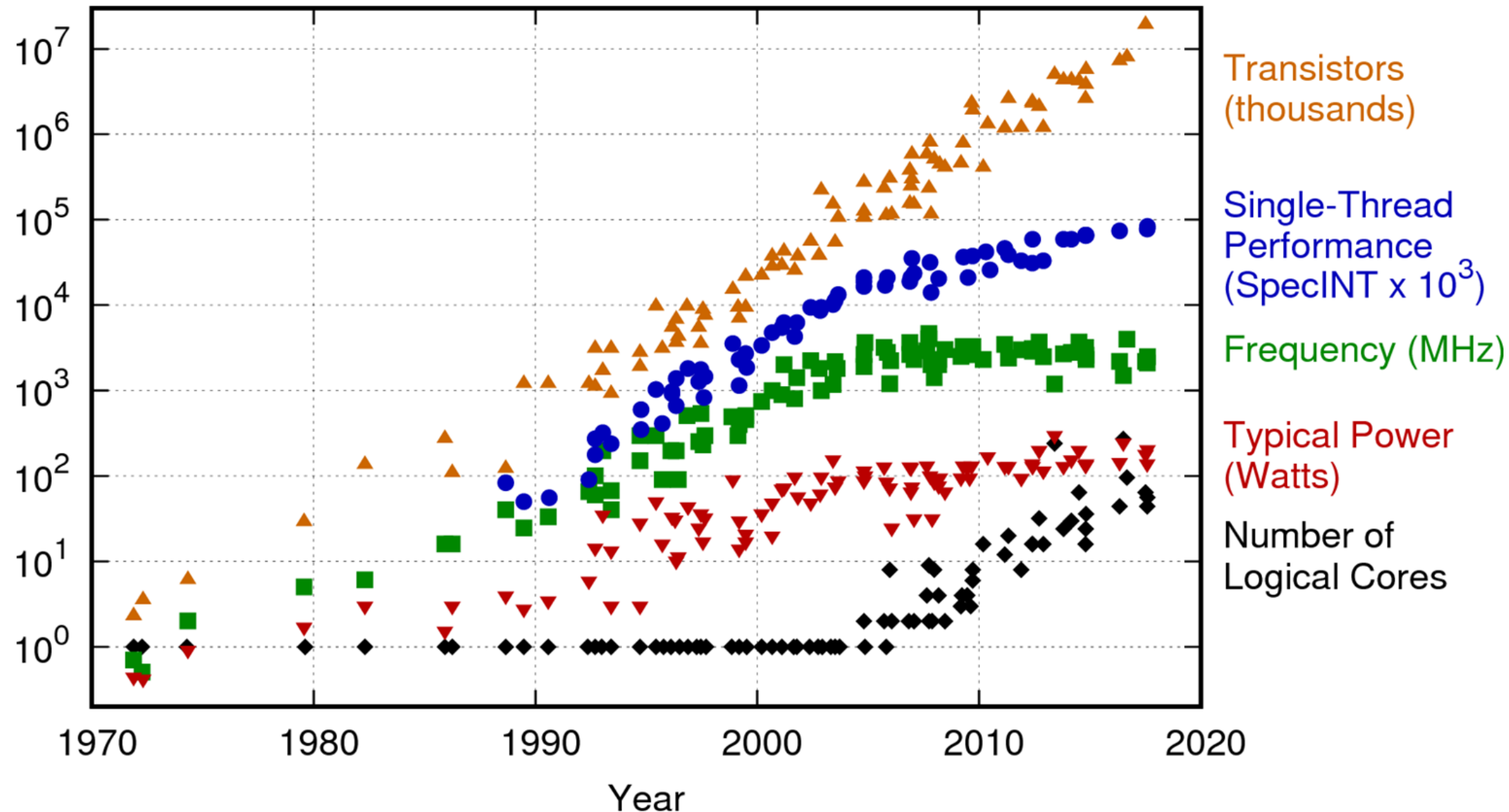
# We use Akka: problems

---

- Learning curve
- Boilerplate
- Language support



# 42 Years of Microprocessor Trend Data



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

New plot and data collected for 2010-2017 by K. Rupp

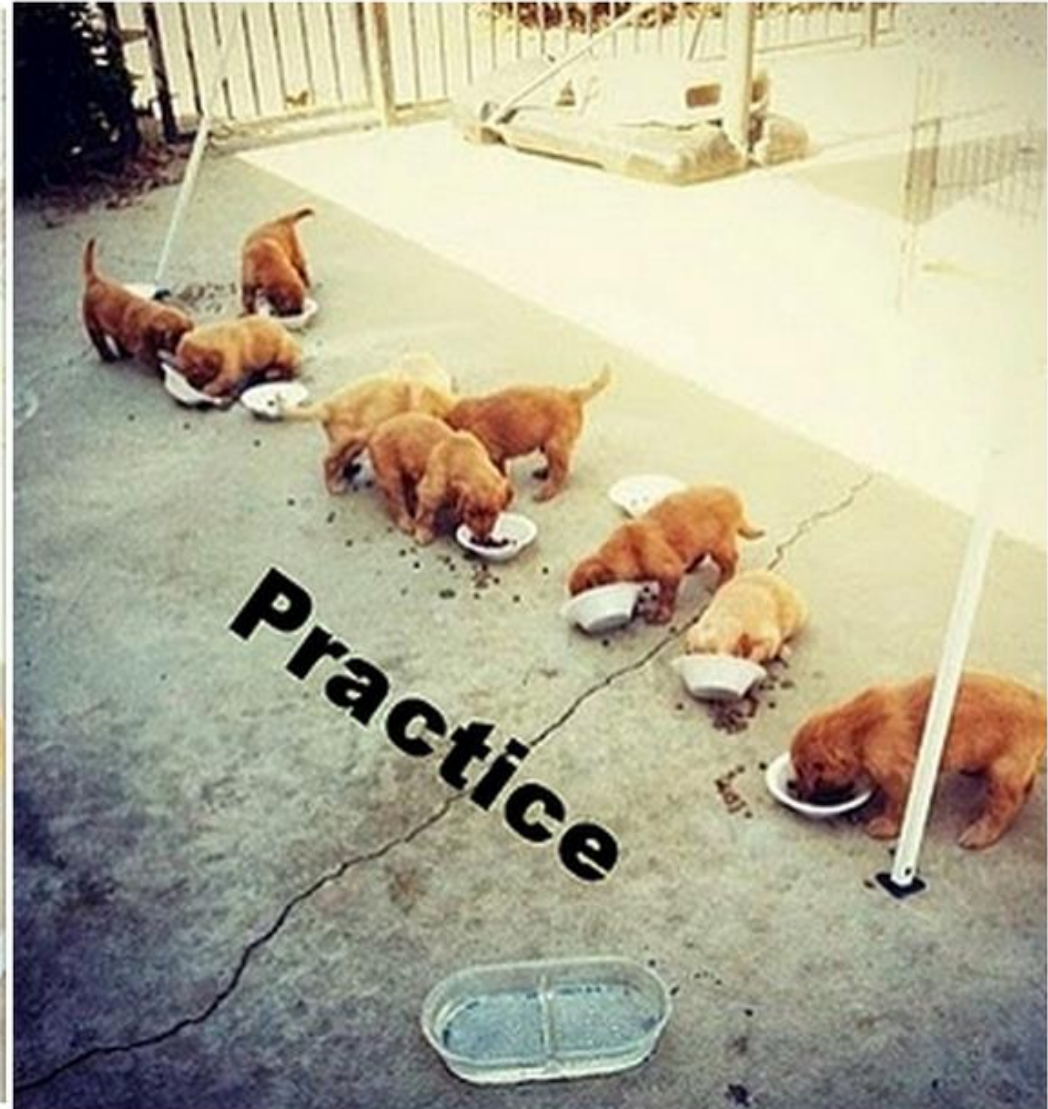
<https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/>

# Problem: Concurrency

---

- Shared state
- Resource management
- Communications
- Performance

# Multithreading

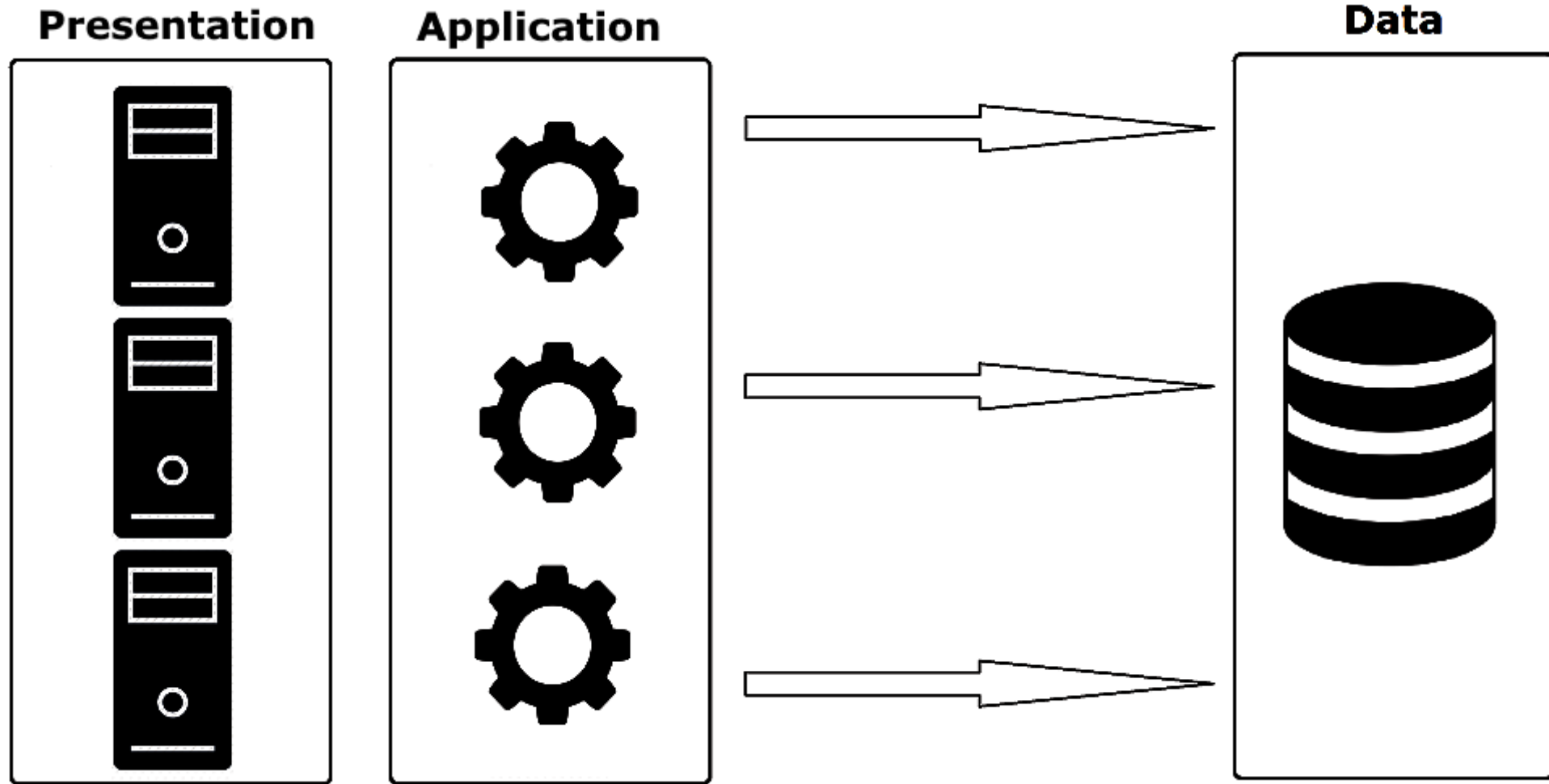


# Problem: Scalability

---

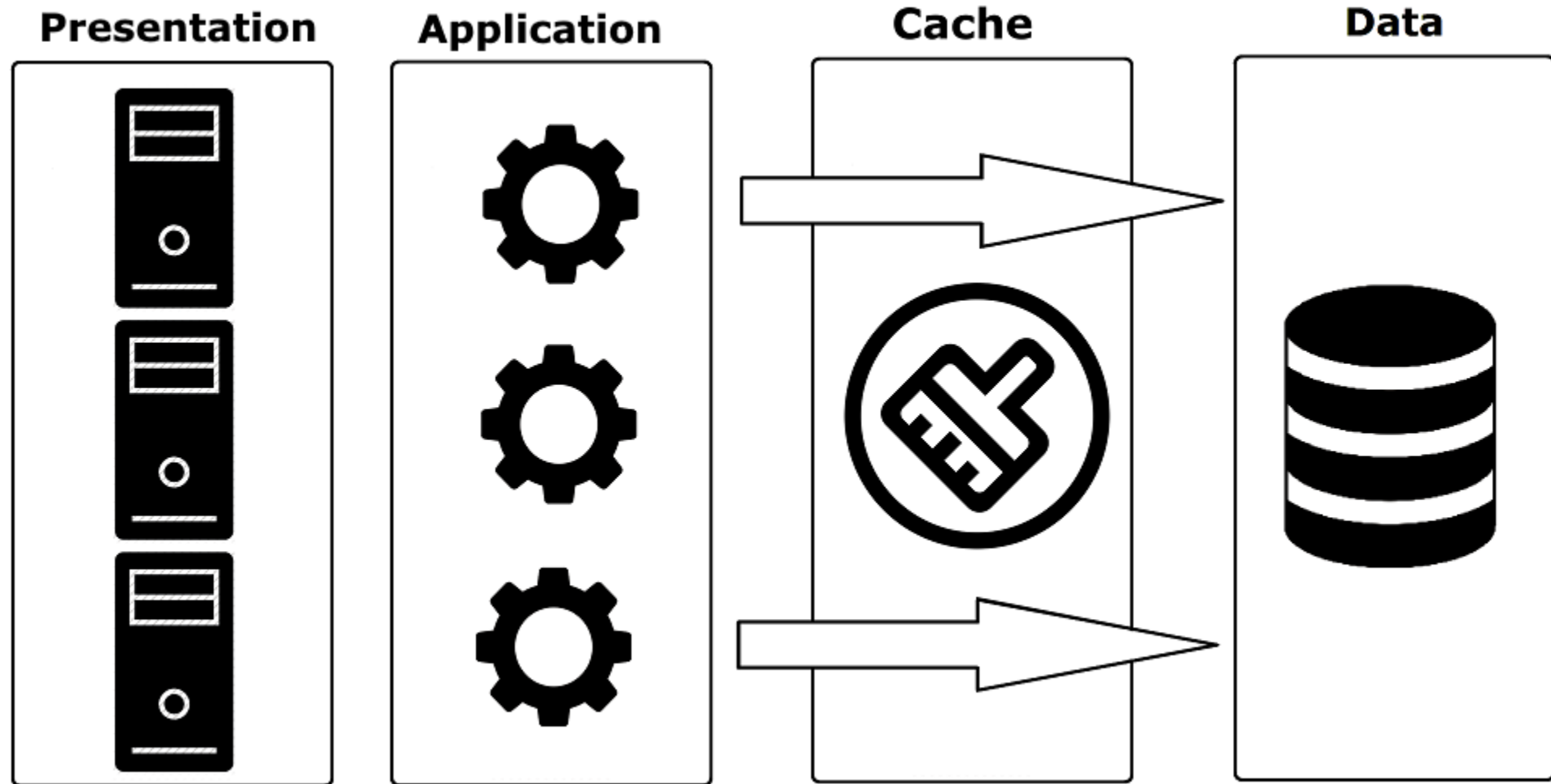
- State storage scalability
- Resilience
- Error handling

# 3-tier

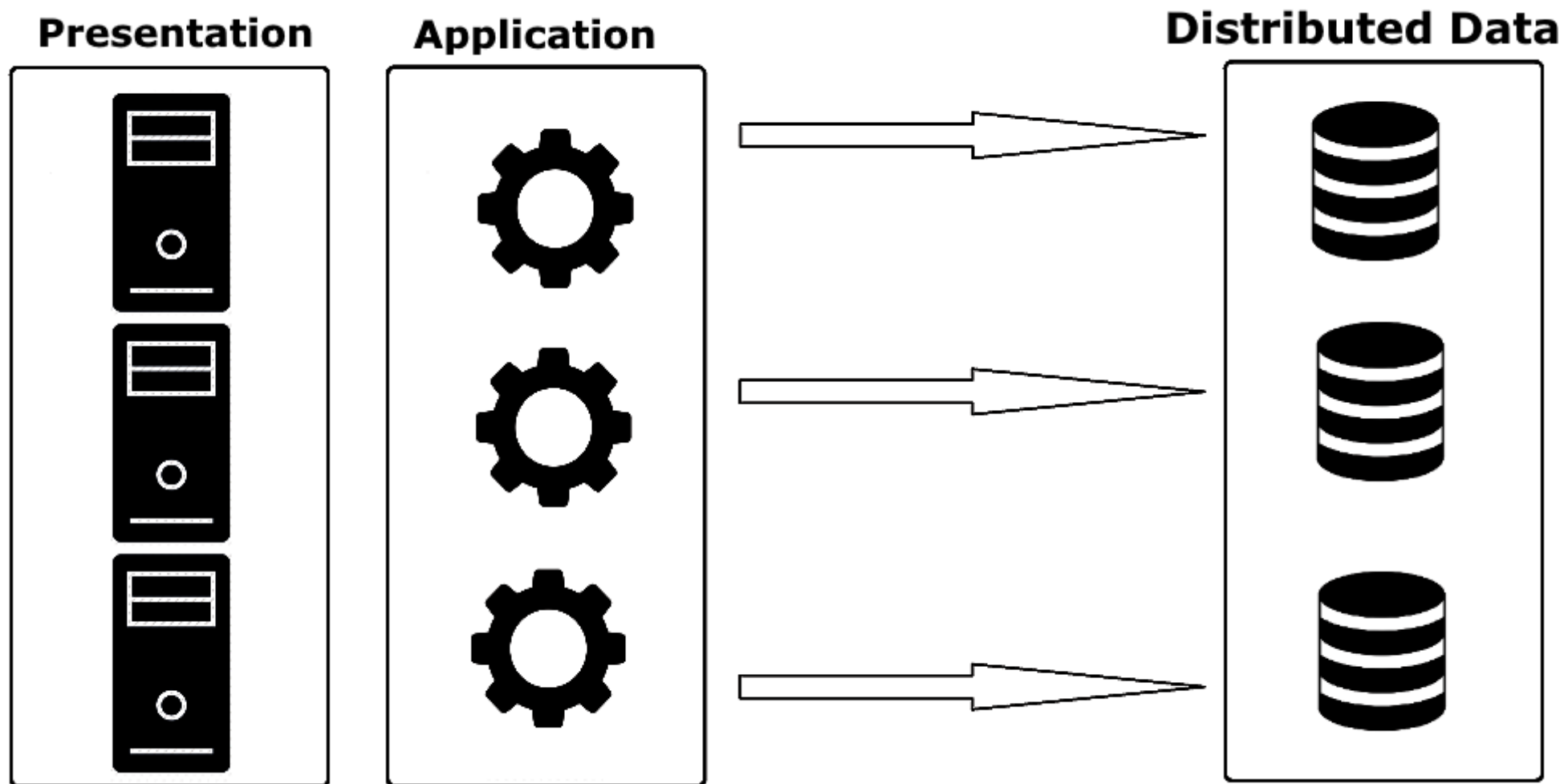




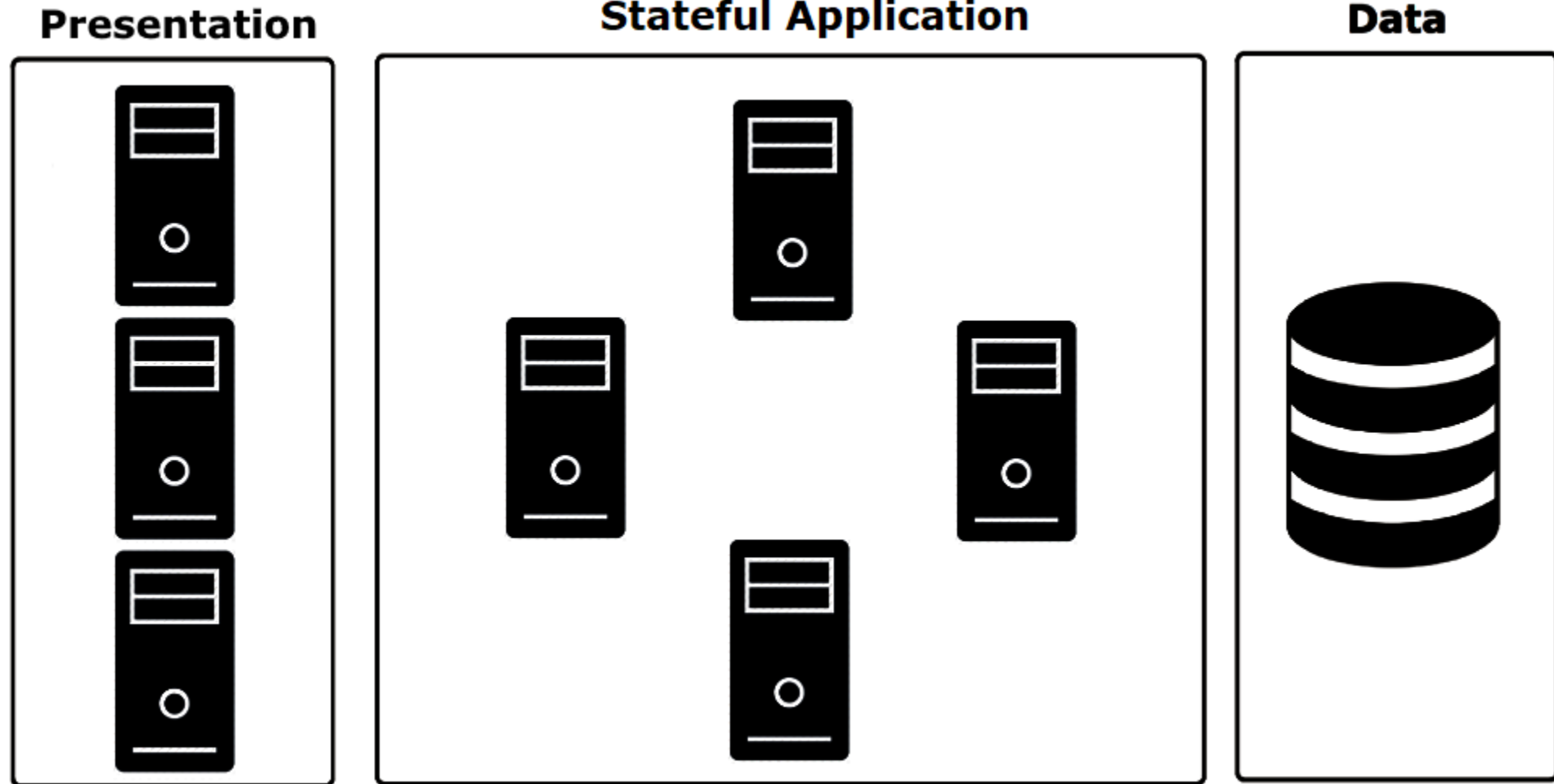
# 3-tier with cache



# Distributed DB



# Stateful Application



# Actor model: 1973

A Universal Modular ACTOR Formalism  
for Artificial Intelligence

Carl Hewitt

Peter Bishop

Richard Steiger

Abstract

This paper proposes a modular ACTOR architecture and definitional method for artificial intelligence that is conceptually based on a single kind of object: actors [or, if you will, virtual processors, activation frames, or streams]. The formalism makes no presuppositions about the representation of primitive data structures and control structures. Such structures can be programmed, micro-coded, or hard wired in a uniform modular fashion. In fact it is impossible to determine whether a given object is "really" represented as a list, a vector, a hash table, a function, or a process. The architecture will efficiently run the coming generation of PLANNER-like artificial intelligence languages including those requiring a high degree of parallelism. The efficiency is gained without loss of programming generality because it only makes certain actors more efficient; it does not change their behavioral characteristics. The architecture is general with respect to control structure and does not have or need goto, interrupt, or semaphore primitives. The formalism achieves the goals that the disallowed constructs are intended to achieve by other more structured methods.

PLANNER Progress

"Programs should not only work,  
but they should appear to work as well."

PDP-1X Dogma



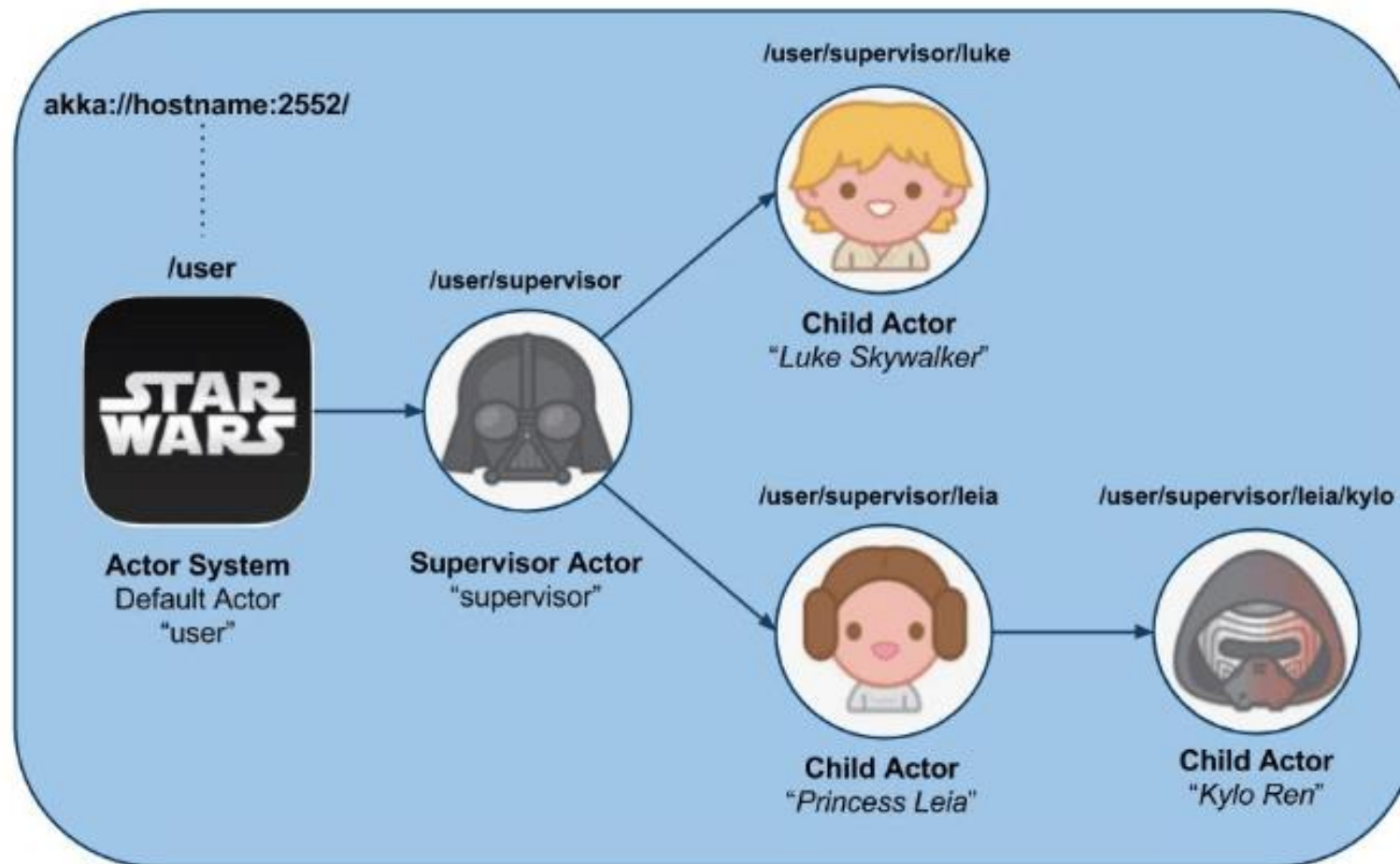
# Actor

---

Universal primitive of concurrent computation

- Maintain its own private state
- Send messages
- Receive messages
- Create more actors

# Actor hierarchy



# Advantages

---

- In-memory
- Sequential message processing
- No shared state
- Asynchronous messaging
- Automatic thread management

# Usages

---

- Erlang
- Facebook WhatsApp (Erlang)
- RabbitMQ (Erlang)
- CouchDB (Erlang)
- LinkedIn (JVM Akka)
- Walmart (JVM Akka)
- Blizzard (JVM Akka)



# Classic model problems

---

- Explicit actor lifetime management
- Boilerplate
- Learning curve

# C# and Akka.NET

---

```
public class MyActor: ReceiveActor
{
    public MyActor()
    {
        Receive<string>( handler: x :string => Sender.Tell( message: 42));
    }
}
```

```
var system = ActorSystem.Create( name: "example-system");
var actor = system.ActorOf( props: Props.Create<MyActor>());

var result :string = await actor.Ask<string>( message: "Hello!");
```

# C# and Akka.NET

```
[DoesNotReturn]
[StackTraceHidden]
public void Throw()
{
    // Restore the exception dispatch details before throwing the exception.
    _exception.RestoreDispatchState(_dispatchState);
    throw _exception;
}
```

```
// InvalidCastException Stack Trace Explorer
// System.InvalidCastException: Unable to cast object of?
// type 'System.Int32' to type 'System.String'.
[DoesNotReturn]
public void Throw()
{
    at Akka.Actor.Futures.Ask[T](ICanTell self, Func`2 messageFactory, Nullable`1 timeout, Cancellation.Token cancellationToken)
    at ConsoleApp3.Program.Main(String[] args) in
    C:\Projects\ConsoleApp3\ConsoleApp3\Program.cs:13
}
```

# C# and Akka.NET

---

```
public class MyActor: ReceiveActor
{
    public MyActor()
    {
    }
}
```

```
var system = ActorSystem.Create( name: "example-system");
var actor = system.ActorOf( props: Props.Create<MyActor>());

var result:string = await actor.Ask<string>( message: "Hello!");
```



# F# and Akka.NET (with Akkling)

```
type ActorMessage = ActorMessage

let myAct (mailbox : Actor<ActorMessage>) = actor {
    let! message = mailbox.Receive()
    match message with
    | ActorMessage -> ()
}

[<EntryPoint>]
let main argv =
    let home = spawn system "system" <| props myAct
    home <! ActorMessage
    0
```

# F# and Akka.NET (with Akkling)

```
type ActorMessage = ActorMessage
type NotActorMessage = NotActorMessage

let myAct (mailbox : Actor<ActorMessage>) = actor {
    let! message = mailbox.Receive()
    match message with
    | ActorMessage -> ()
}

[<EntryPoint>]
let main argv =
    let home = spawn system "system" <| props myAct
    home <! NotActorMessage
    0
```

This expression was expected to have type  
'ActorMessage'  
but here has type  
'NotActorMessage'

# Virtual actors: 2009

## Orleans: Distributed Virtual Actors for Programmability and Scalability

Philip A. Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, Jorgen Thelin  
*Microsoft Research*

### Abstract

High-scale interactive services demand high throughput with low latency and high availability, difficult goals to meet with the traditional stateless 3-tier architecture. The actor model makes it natural to build a stateful middle tier and achieve the required performance. However, the popular actor model platforms still pass many distributed systems problems to the developers.

The Orleans programming model introduces the novel abstraction of virtual actors that solves a number of the complex distributed systems problems, such as reliability and distributed resource management, liberating the developers from dealing with those concerns. At the same time, the Orleans runtime enables applications

required application-level semantics and consistency on a cache with fast response for interactive access.

The actor model offers an appealing solution to these challenges by relying on the *function shipping paradigm*. Actors allow building a stateful middle tier that has the performance benefits of a cache with data locality and the semantic and consistency benefits of encapsulated entities via application-specific operations. In addition, actors make it easy to implement horizontal, “social”, relations between entities in the middle tier.

Another view of distributed systems programmability is through the lens of the object-oriented programming (OOP) paradigm. While OOP is an intuitive way to model complex systems, it has been marginalized by the popular service-oriented architecture (SOA). One can

# Key concepts

---

- Perpetual existence
- Automatic instantiation
- Location transparency
- Automatic scale out



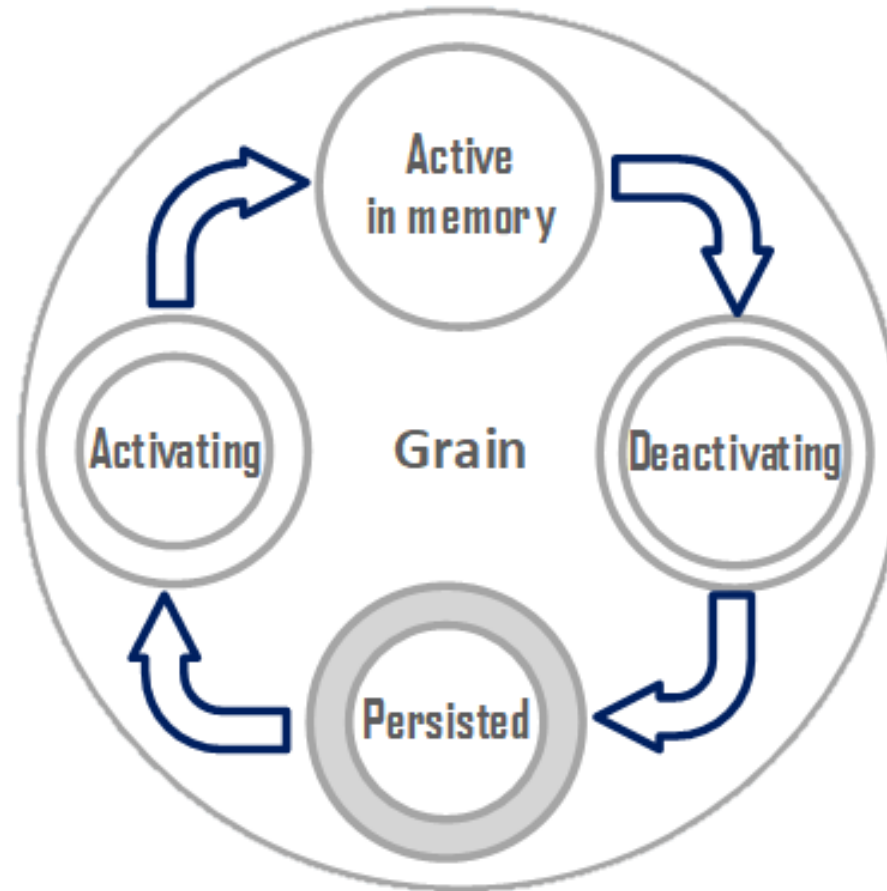
# Virtual vs Classic Actor

Implicit lifetime management	Manual create and shutdown
Implicit messaging (async method calls)	Explicit messaging
Automatic Actor placement	Manual Actor placement
No hierarchy	Hierarchy
At least once message delivery	At most once message delivery
You don't know physical location	Address reflects location

# ORLEANS



# Orleans Grain Lifecycle



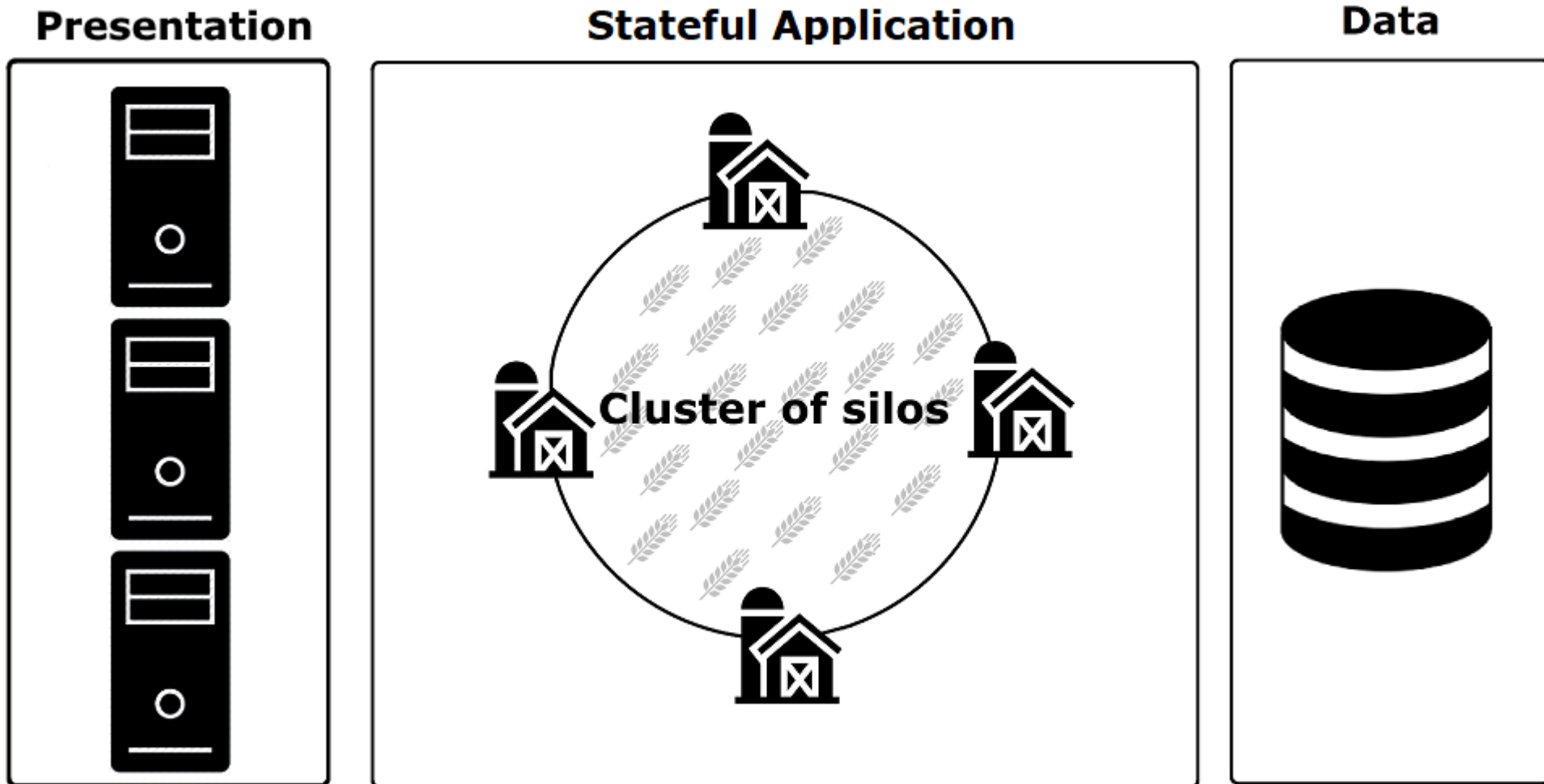
# Orleans Grain into the wild

```
public interface IMyGrain : IGrainWithStringKey
{
    Task<string> Hello();
}
public class MyGrain : Grain, IMyGrain
{
    public Task<string> Hello() => Task.FromResult(result: "Hello");
}

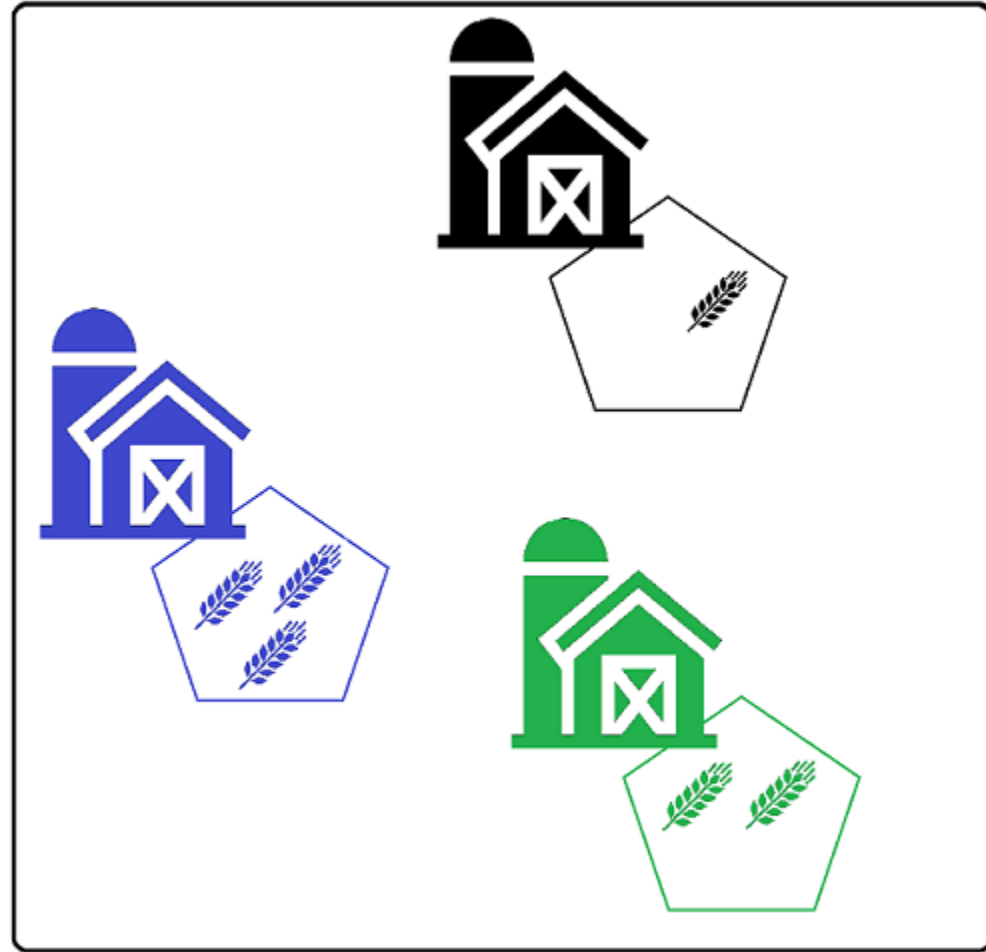
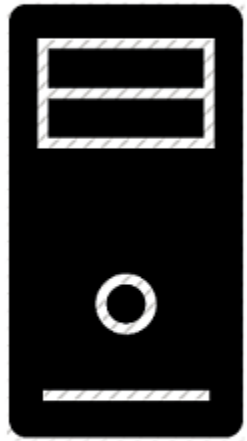
var grain = clusterClient.GetGrain<IMyGrain>(primaryKey: "Hello");
var result:string = await grain.Hello();
```



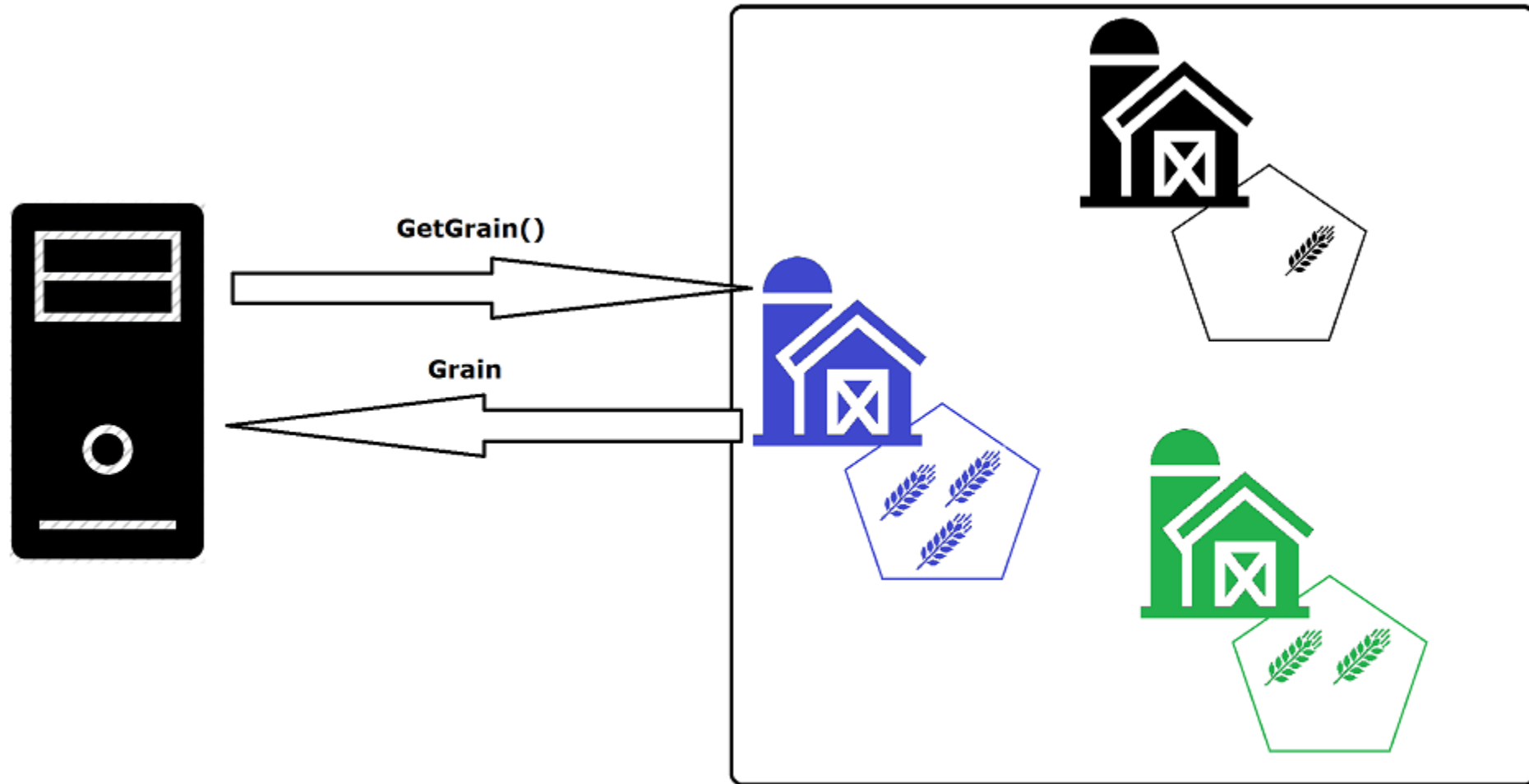
# Actor-based Middle Tier



# Silo Failure

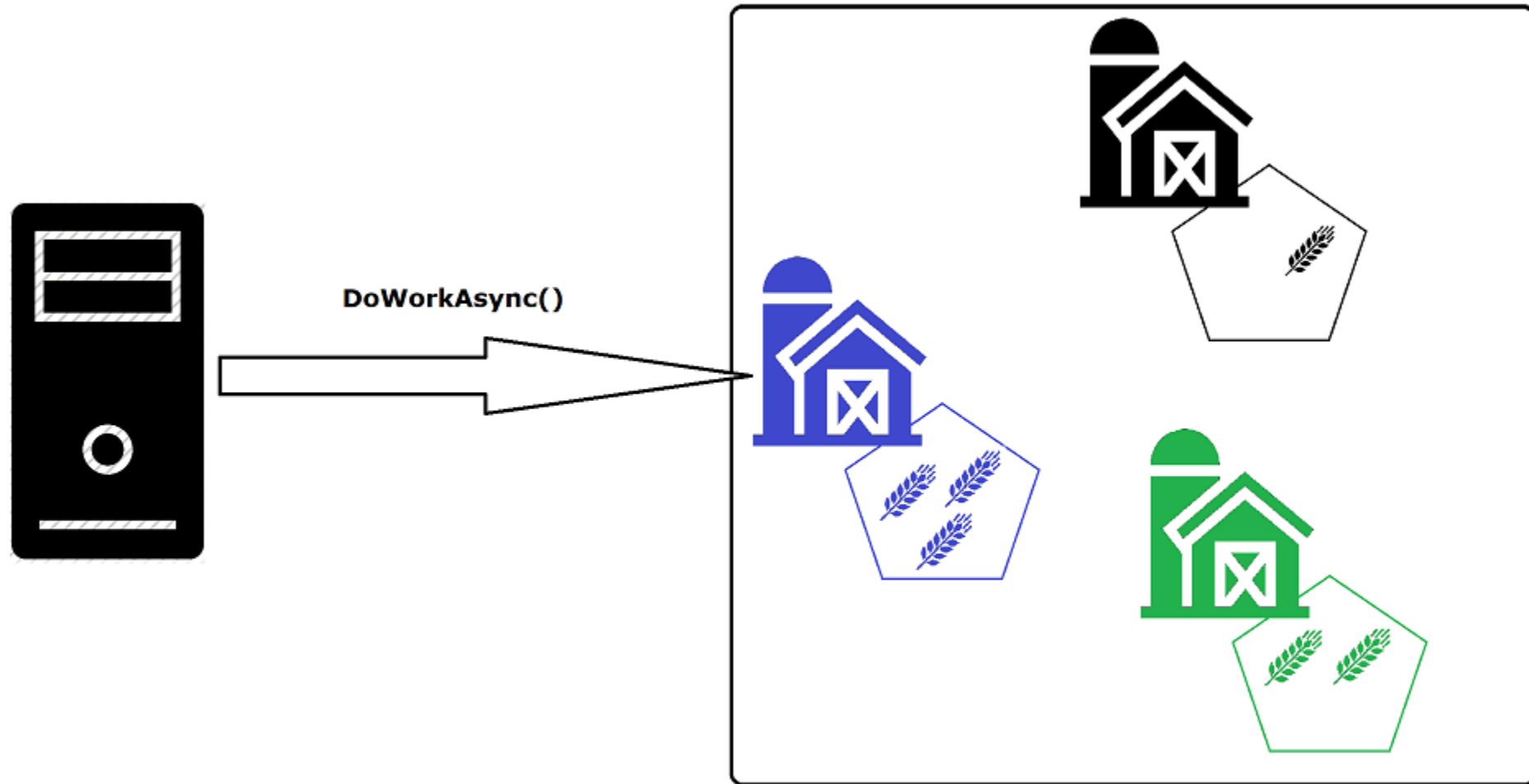


# Silo Failure

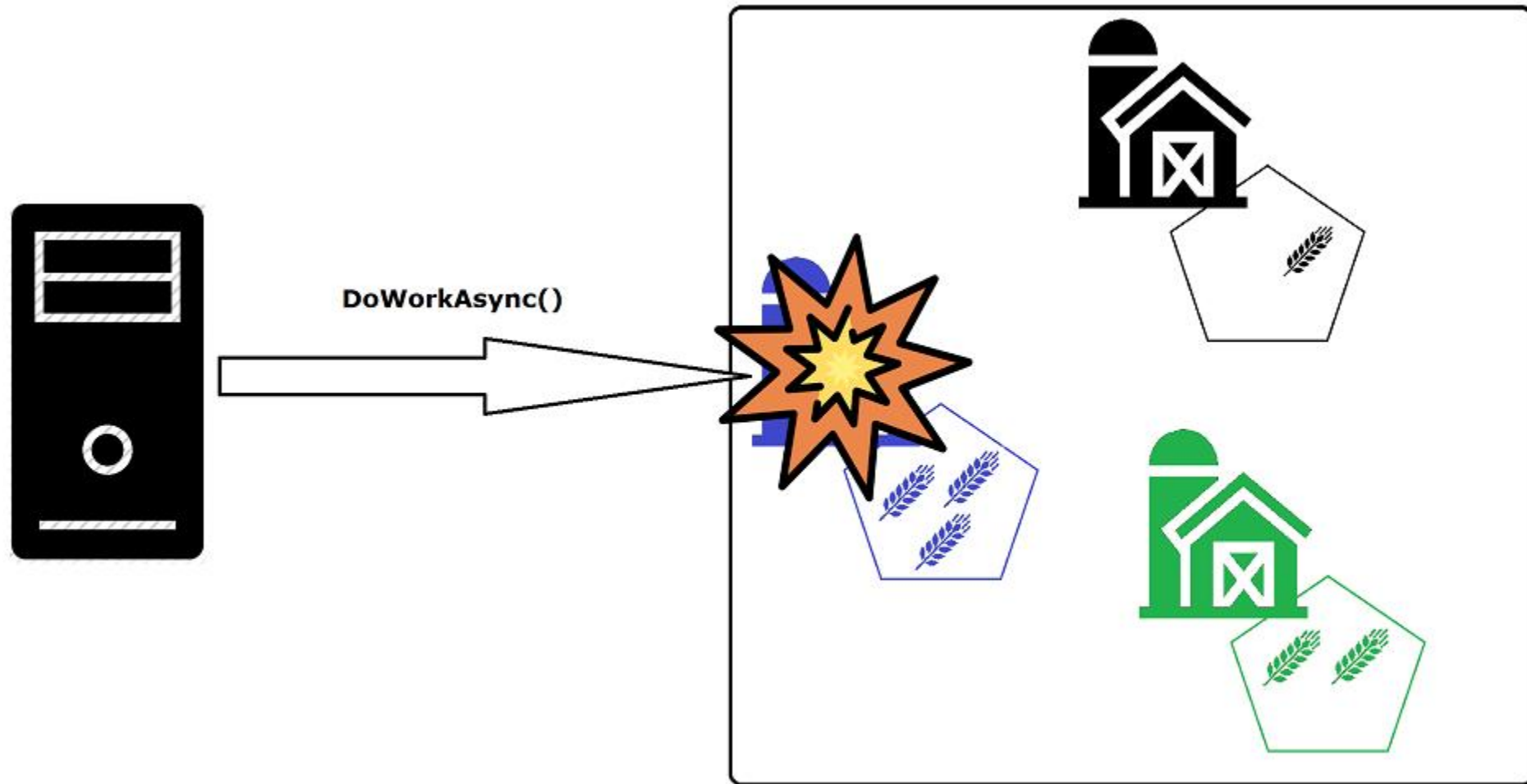




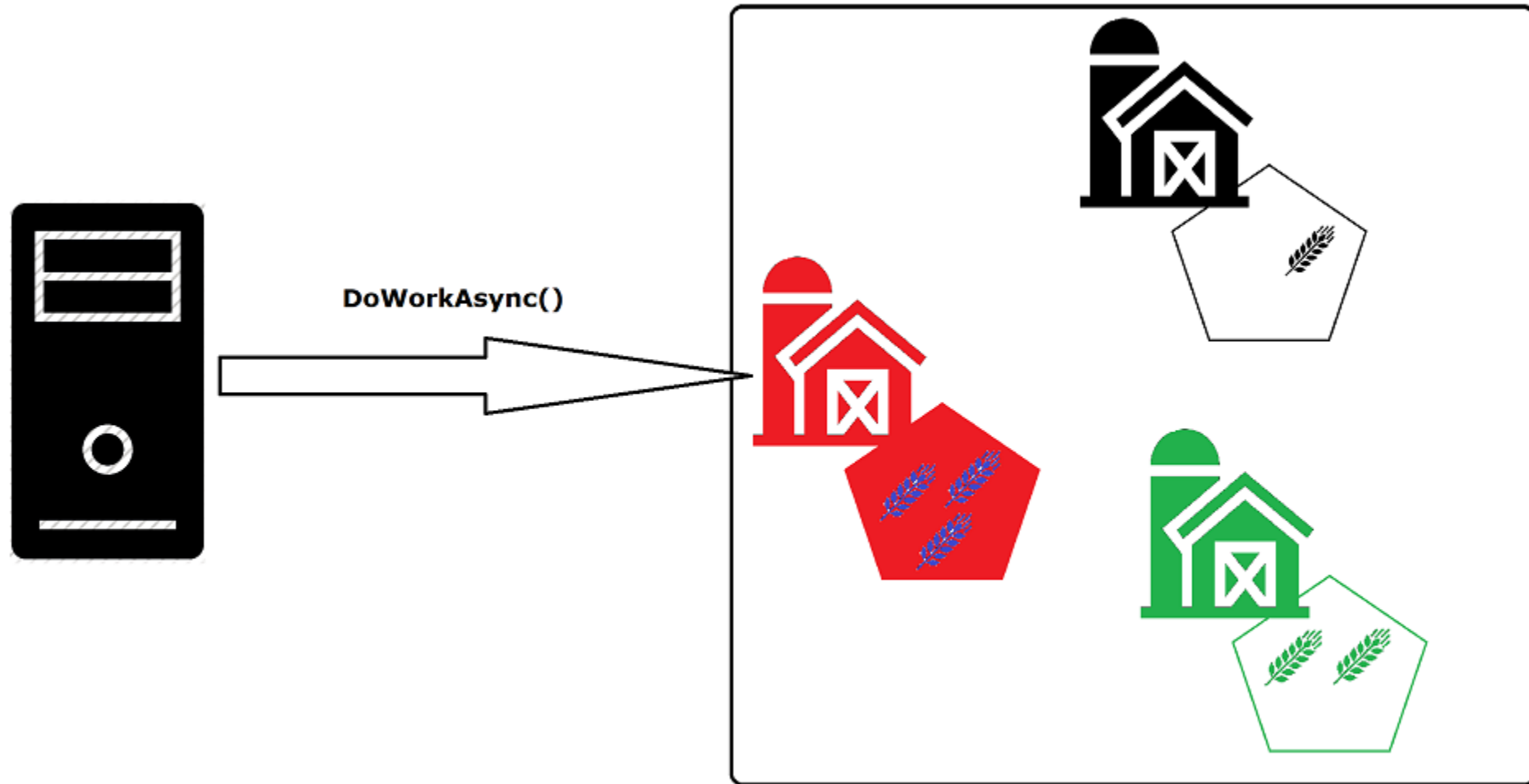
# Silo Failure



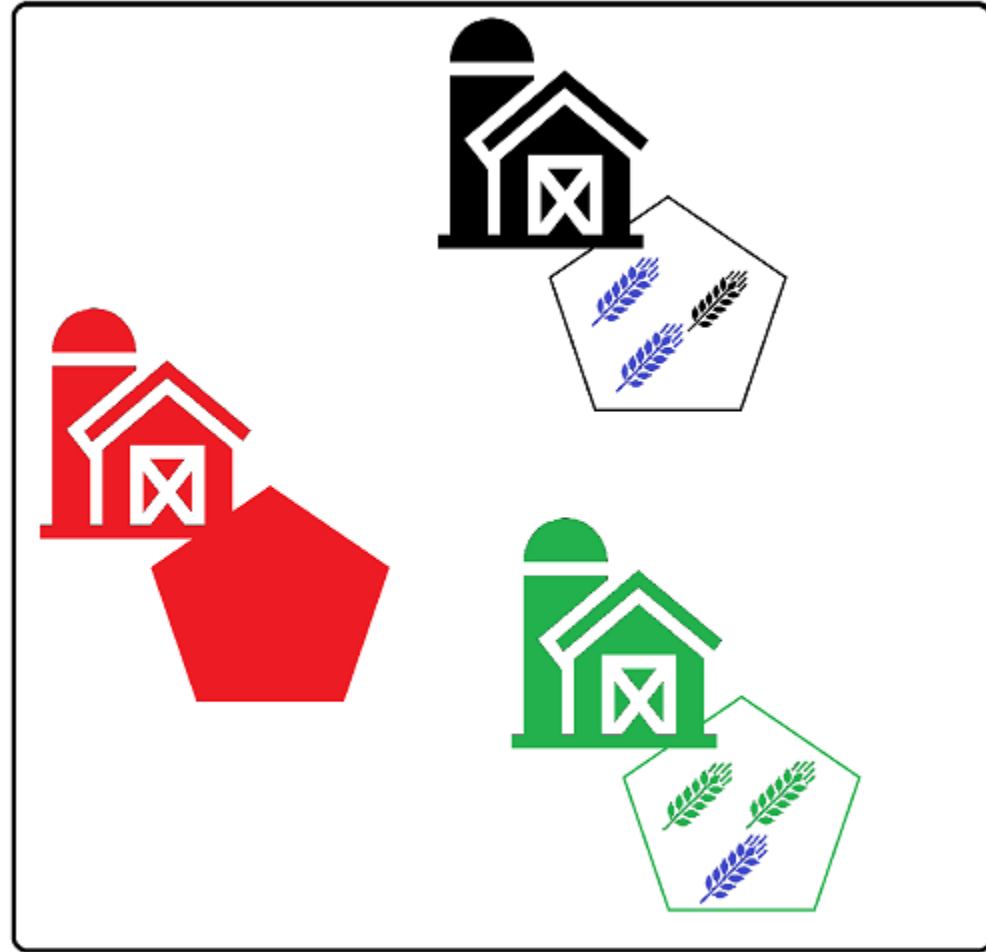
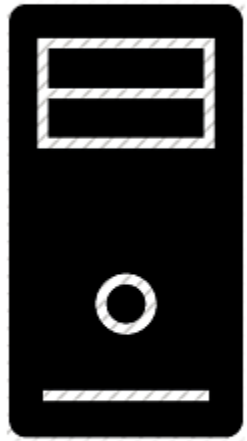
# Silo Failure



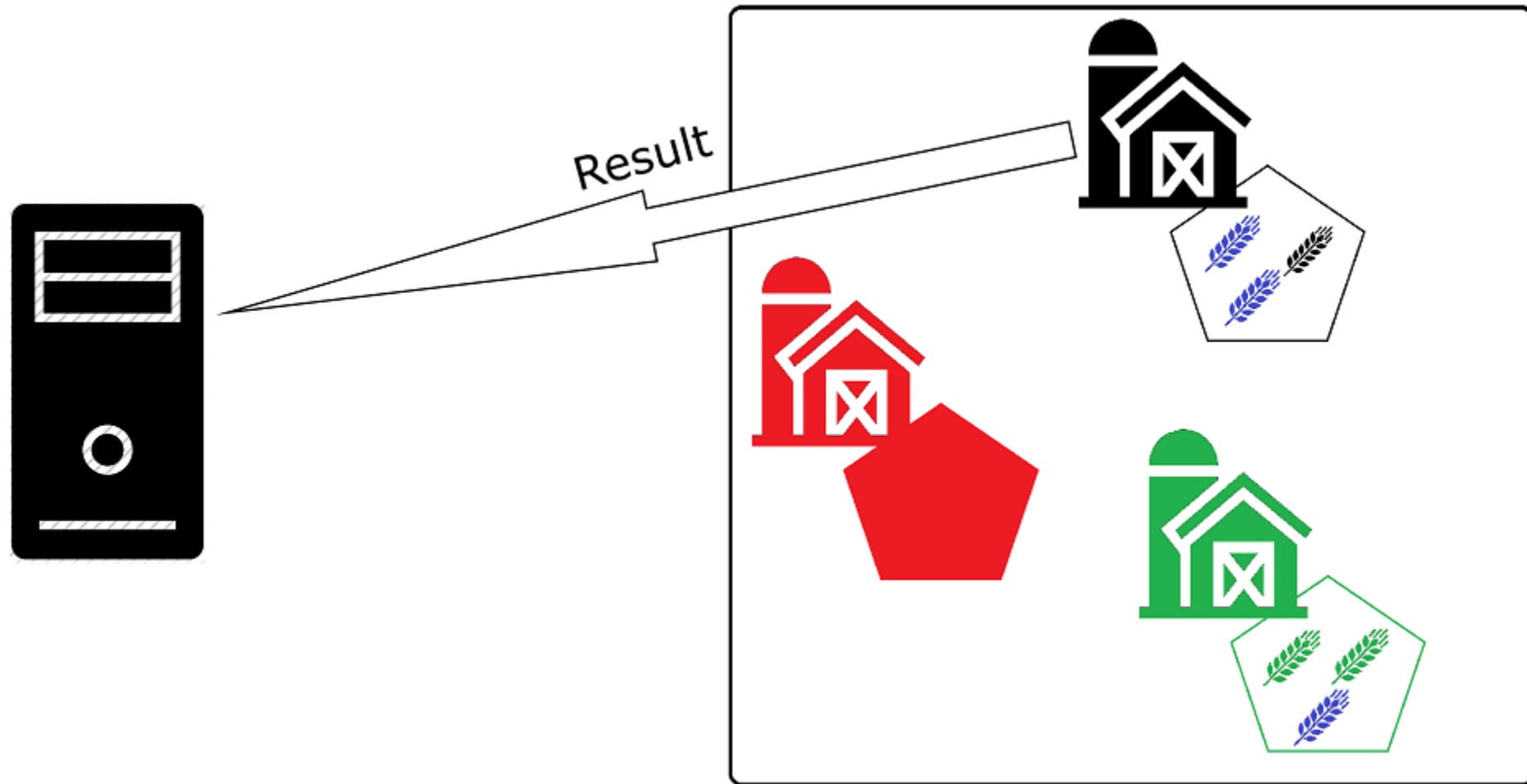
# Silo Failure



# Silo Failure



# Silo Failure

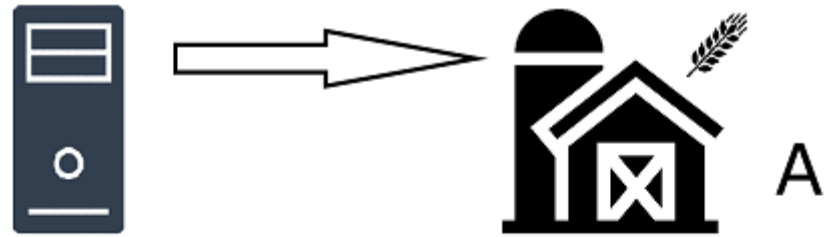


# Error handling

---

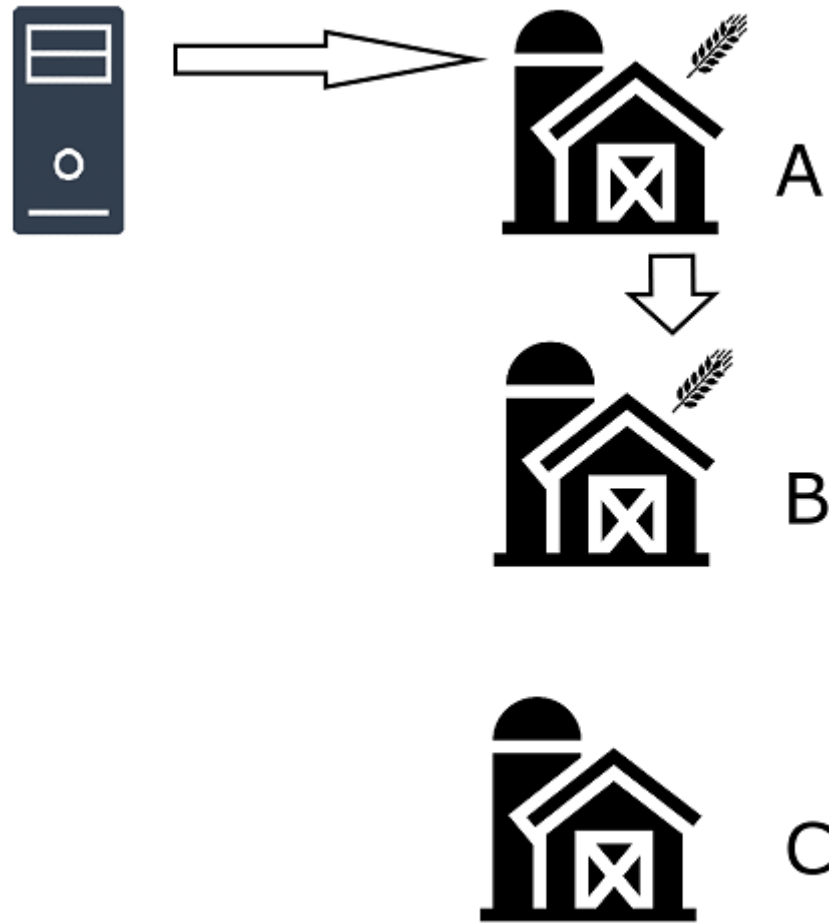


# Error handling

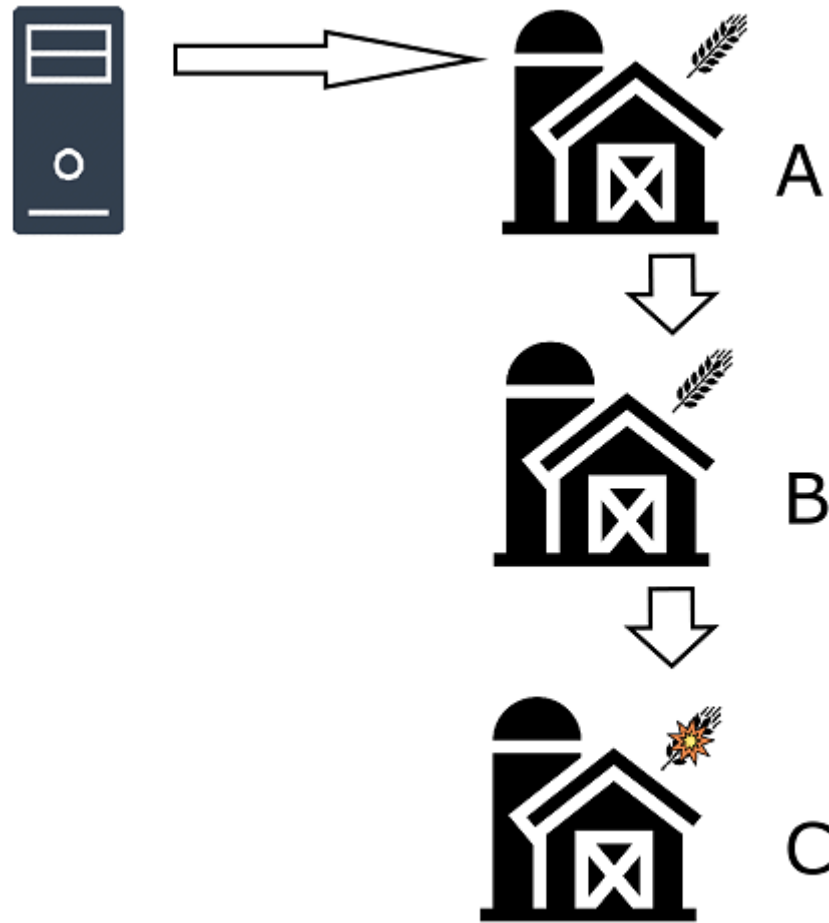




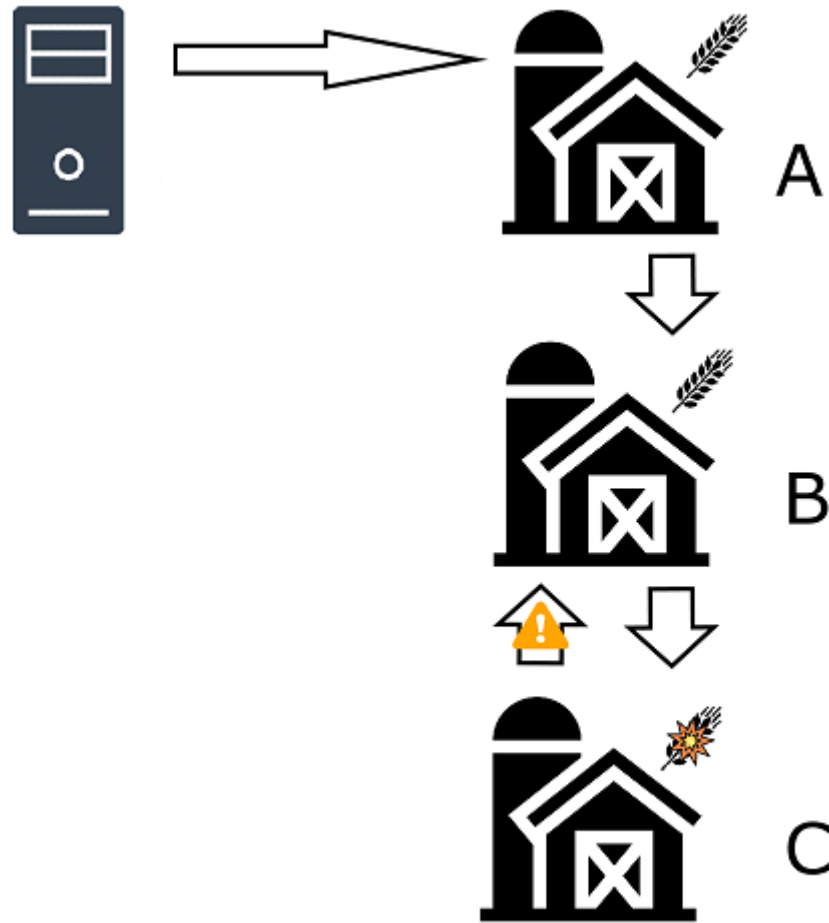
# Error handling



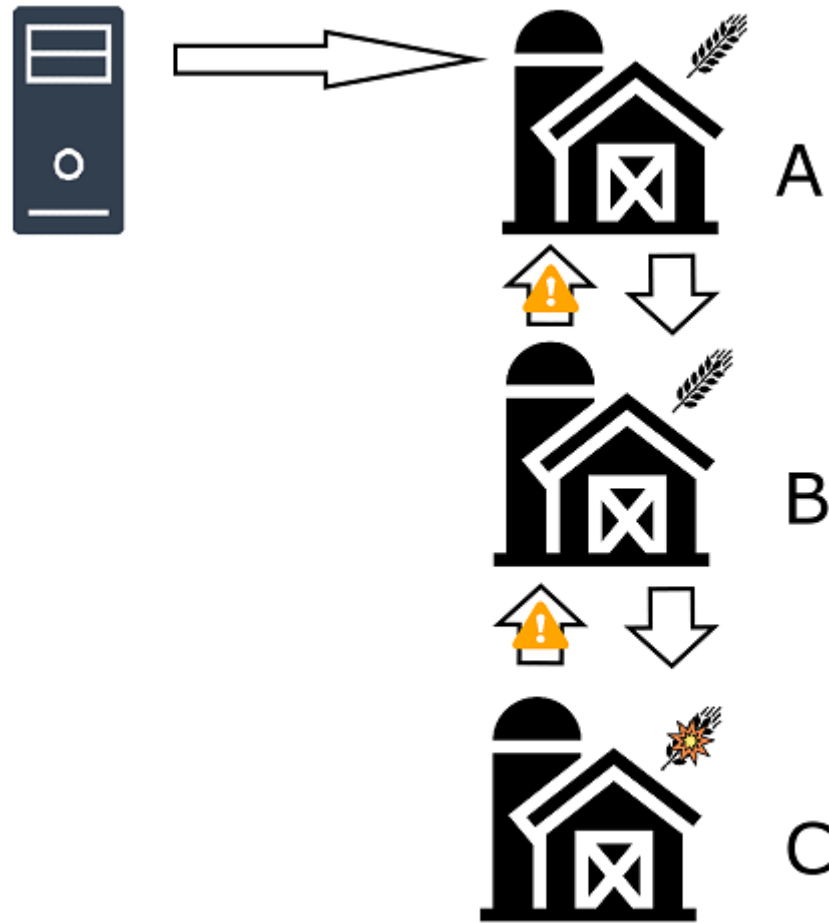
# Error handling



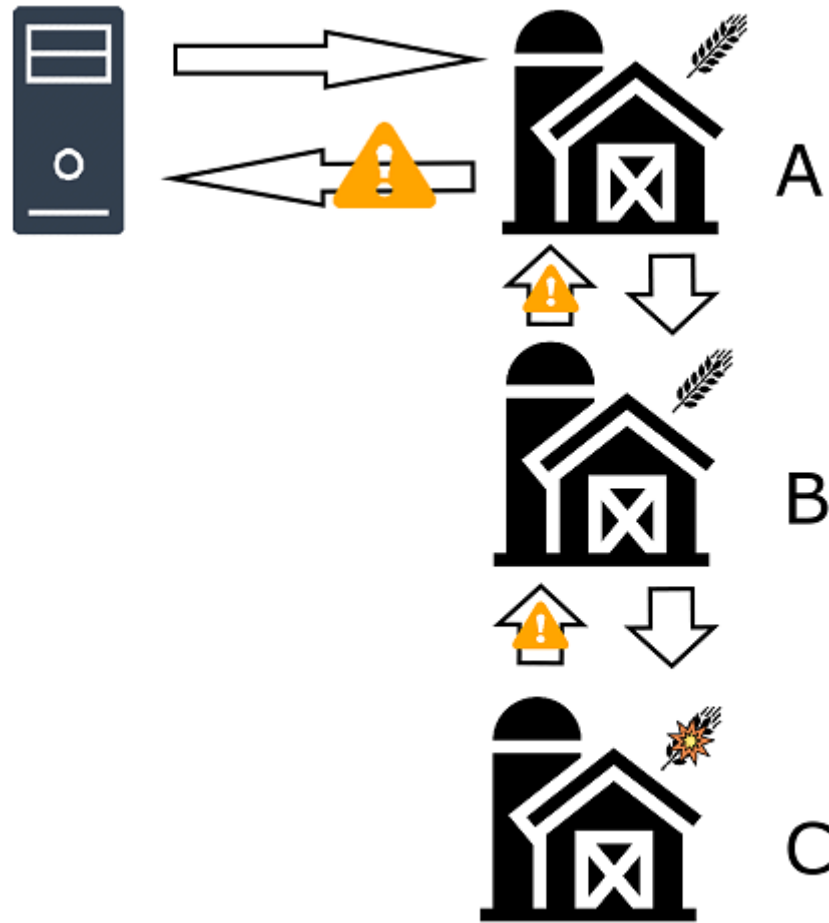
# Error handling



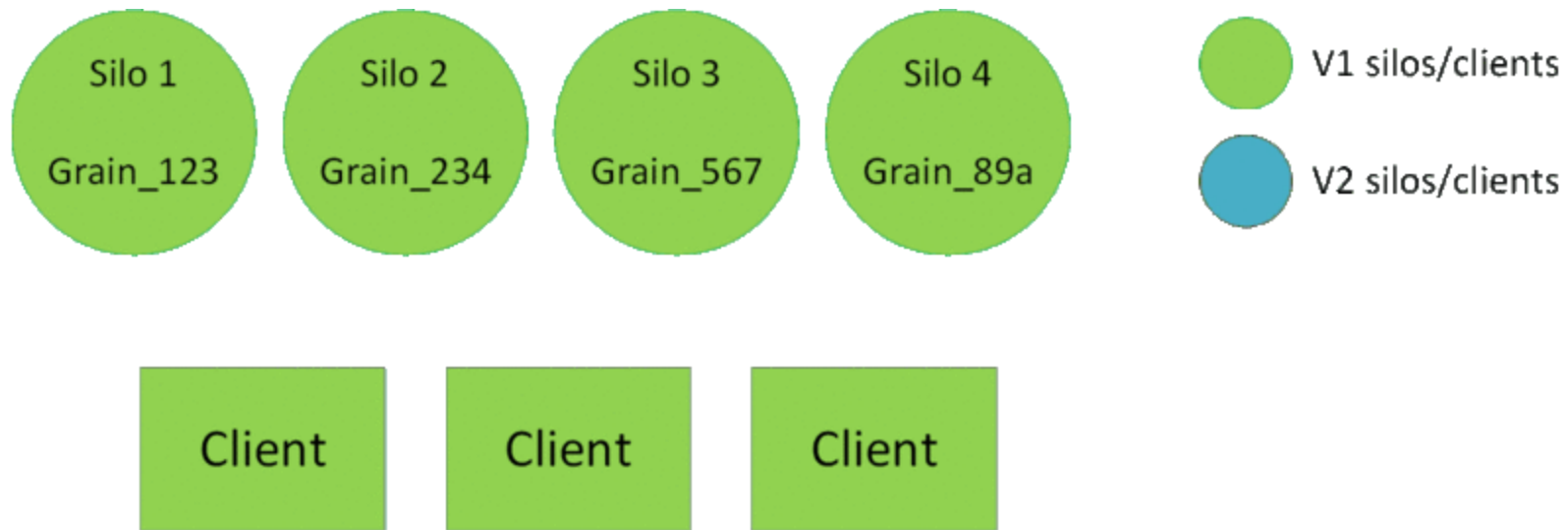
# Error handling



# Error handling



# Grain Versioning



# Orleans history

---

- Started in Microsoft Research - 2009
- In production – 2011
- Halo 4 – 2012
- Public preview – 2014
- GitHub (MIT) - 2015



# Other implementations



## What is Orbit?

Orbit is a framework to write distributed systems using virtual actors on the JVM. A virtual actor is an object that interacts with the world using asynchronous messages.

At any time an actor may be active or inactive. Usually the state of an inactive actor will reside in the database. When a message is sent to an inactive actor it will be activated somewhere in the pool of backend servers. During the activation process the actor's state is read from the database.

Actors are deactivated based on timeout and on server resource usage.

It is heavily inspired by the [Microsoft Orleans](#) project.



## erleans

FAILED


codecov 60%

## Components

### Grains

Stateful grains are backed by persistent storage and referenced by a primary key set by the grain. An activation of a grain is a single Erlang process in on an Erlang node (silo) in an Erlang cluster. Activation placement is handled by Erleans and communication is over standard Erlang distribution. If a grain is sent a message and does not have a current activation one is spawned.

## Introduction to Service Fabric Reliable Actors

11/01/2017 • 11 minutes to read •  +5

Reliable Actors is a Service Fabric application framework based on the [Virtual Actor](#) pattern. The Reliable Actors API provides a single-threaded programming model built on the scalability and reliability guarantees provided by Service Fabric.

# Links

---

- [A Universal Modular ACTOR Formalism for Artificial Intelligence](#)
- [Orleans: Distributed Virtual Actors for Programmability and Scalability](#)
- [Microsoft Orleans](#)
- [How to build real-world applications with Orleans](#)
- [Distributed Transactions are dead](#)

# Contacts

---



@KiloOscarTango



@KiloOscarTango



[github.com/Rizzen](https://github.com/Rizzen)