

# Building a scalable backend with Akka.NET

---

Hannes Lowette

## Disclaimer

***I am in this workshop for you, not the other way around.***

So please, don't hesitate to:

- Ask questions
- Interrupt me
- Tell me when you need a break
- etc.

## About me

### Hannes Lowette

- .NET Coach at Axxes
- Father of 3 kids
- Amateur luthier (guitar builder)
- Lover of whisky & absinthe
- Mountain biker

 @hannes\_lowette

 #20086521



# Agenda

- A bit of history
  - Introduction to Akka.NET
  - The problem domain
  - How Akka.NET fits in
  - Coding together
  - Further implementation details
  - Beyond this workshop

# AXXES



# Why we even have Akka.NET

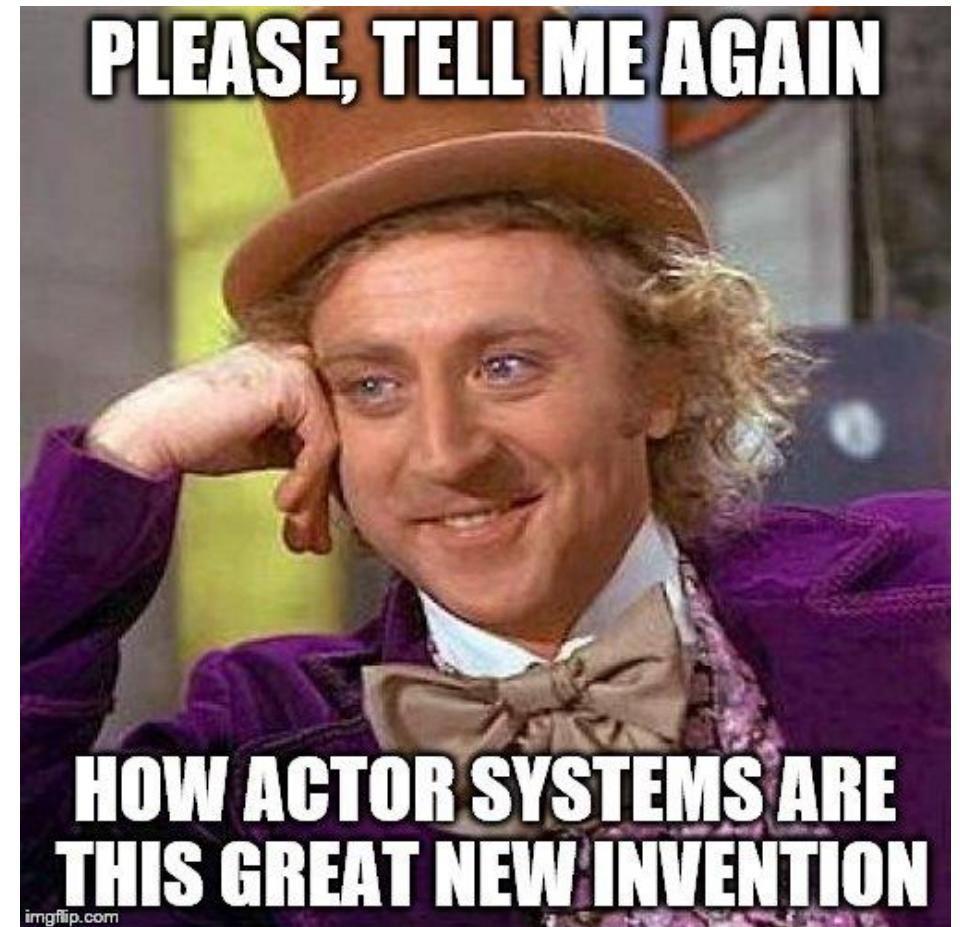
---

History

## Origin of the Actor model

Designing software inspired by physics:

- 1973
- Carl Hewitt, Peter Bishop & Richard Steiger
- Theoretical model
- Many independent microprocessors
- Further refinement in the 70's and early 80's



AXXES\_

## First put into practice

Ericsson AXD 301 Telco System:

- Invention of Erlang
- Fault-Tolerant
- Distributed
- Concurrent
- 2 million lines of code
- 99,999999% uptime (9 nines)  
~ 31ms downtime per year

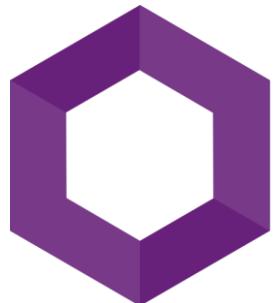


ERICSSON

AXXES

## 2015 – Year of the .NET actors

- Feb 2015: Project Orleans v 1.0.0
- April 2015: Akka.NET v 1.0.0
- April 2015: Service Fabric Reliable Actors v 1.0.x



Orleans



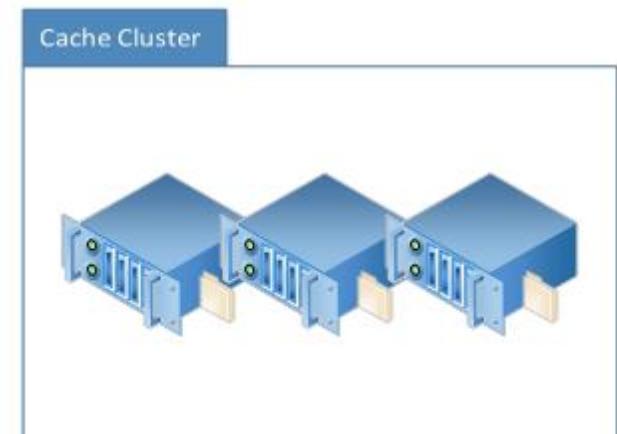
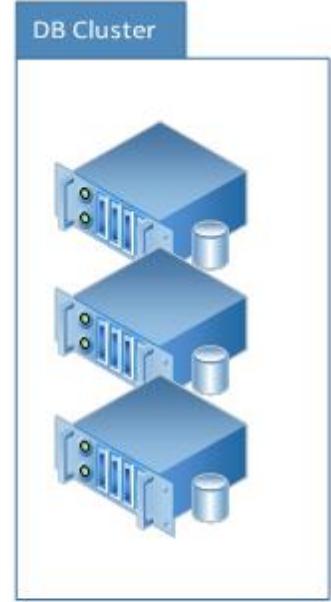
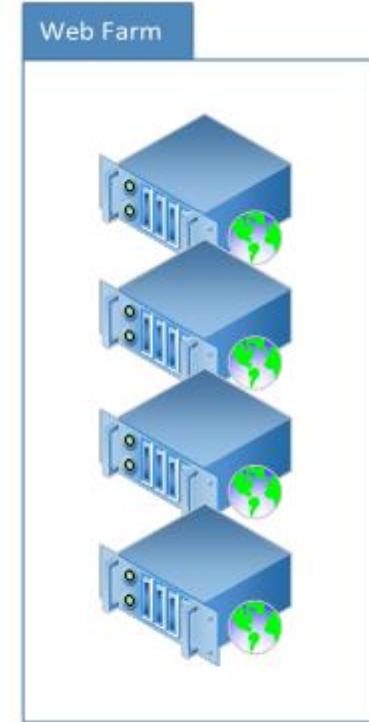
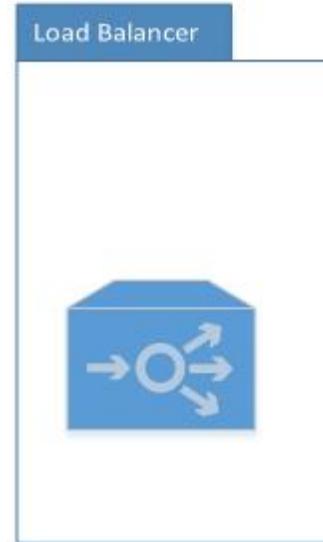
Microsoft Azure  
Service Fabric



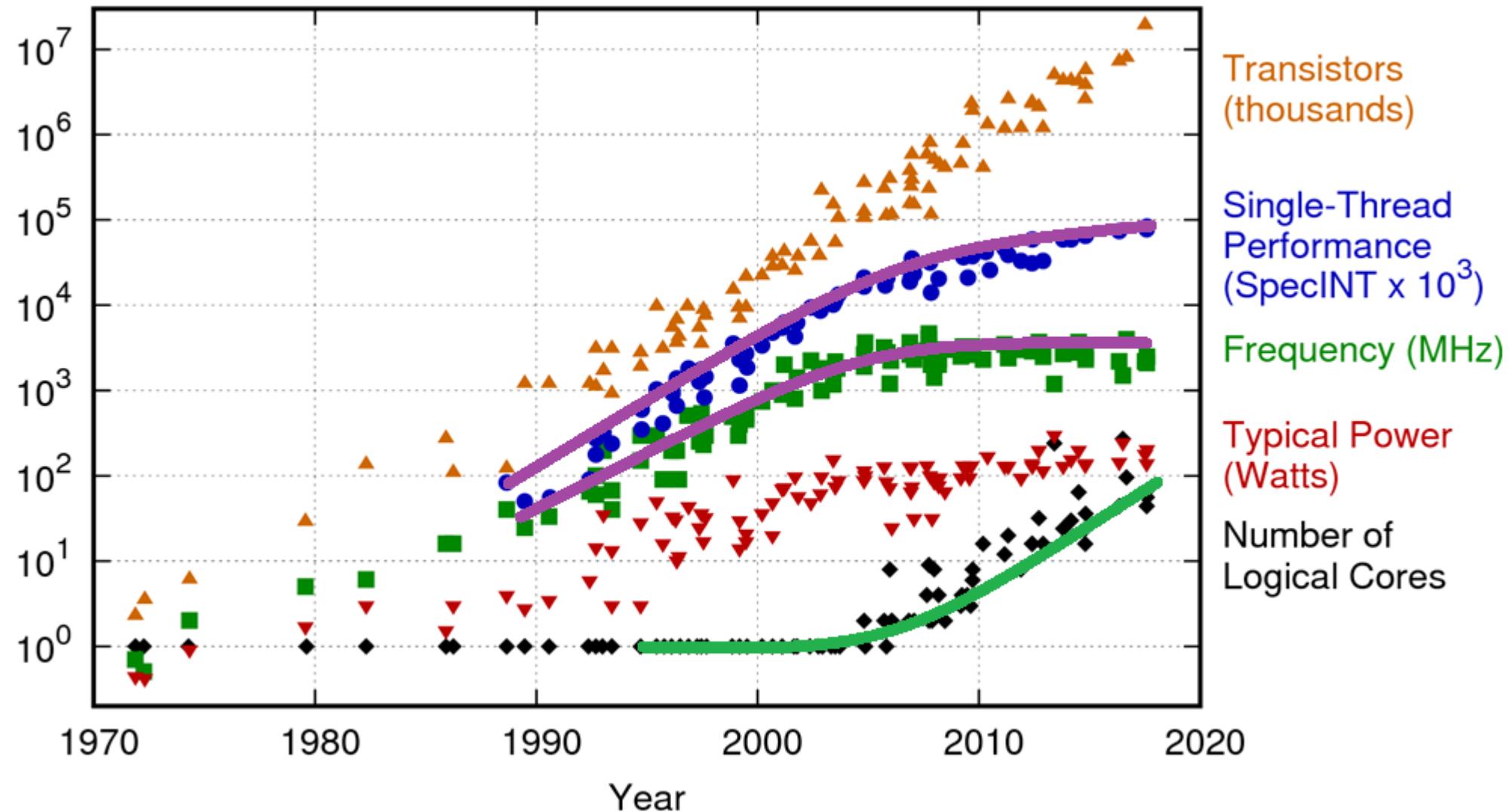
Why 2015?

## Classic scaling under stress

- Explosion of the web
- Smartphones
- Internet of things



Axes

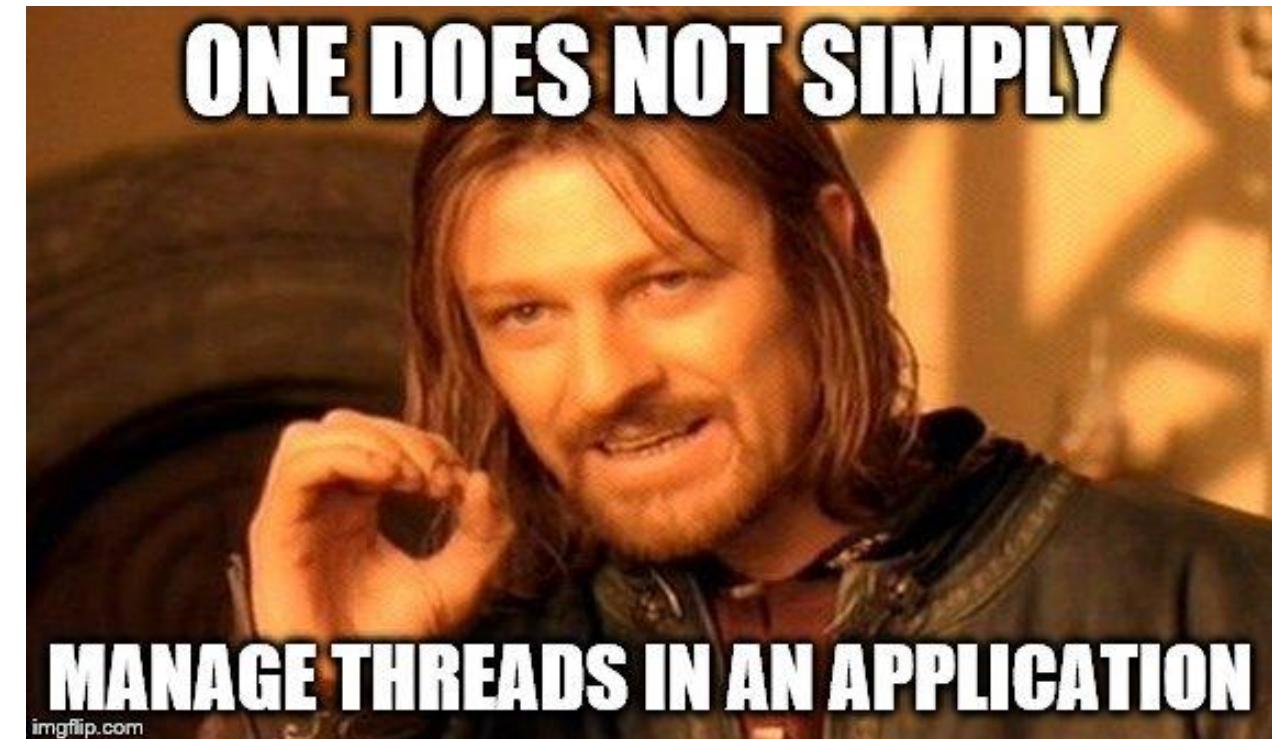


AXES

## Parallelism is salvation

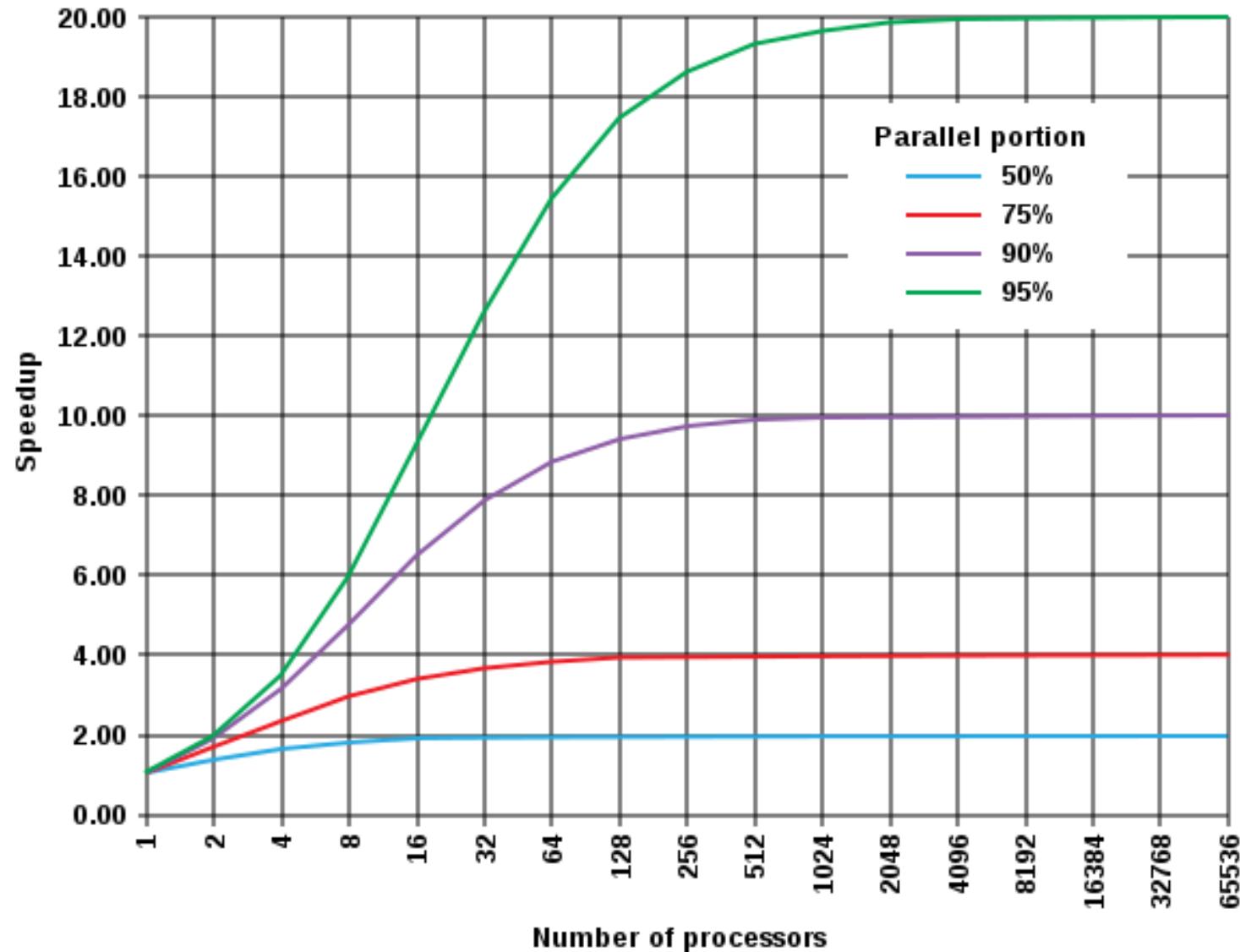
### Problems with parallelization:

- ➊ Shared State
  - Race Conditions
  - Blocking calls
  - Deadlocks
- ➋ Serialized code



AXXES

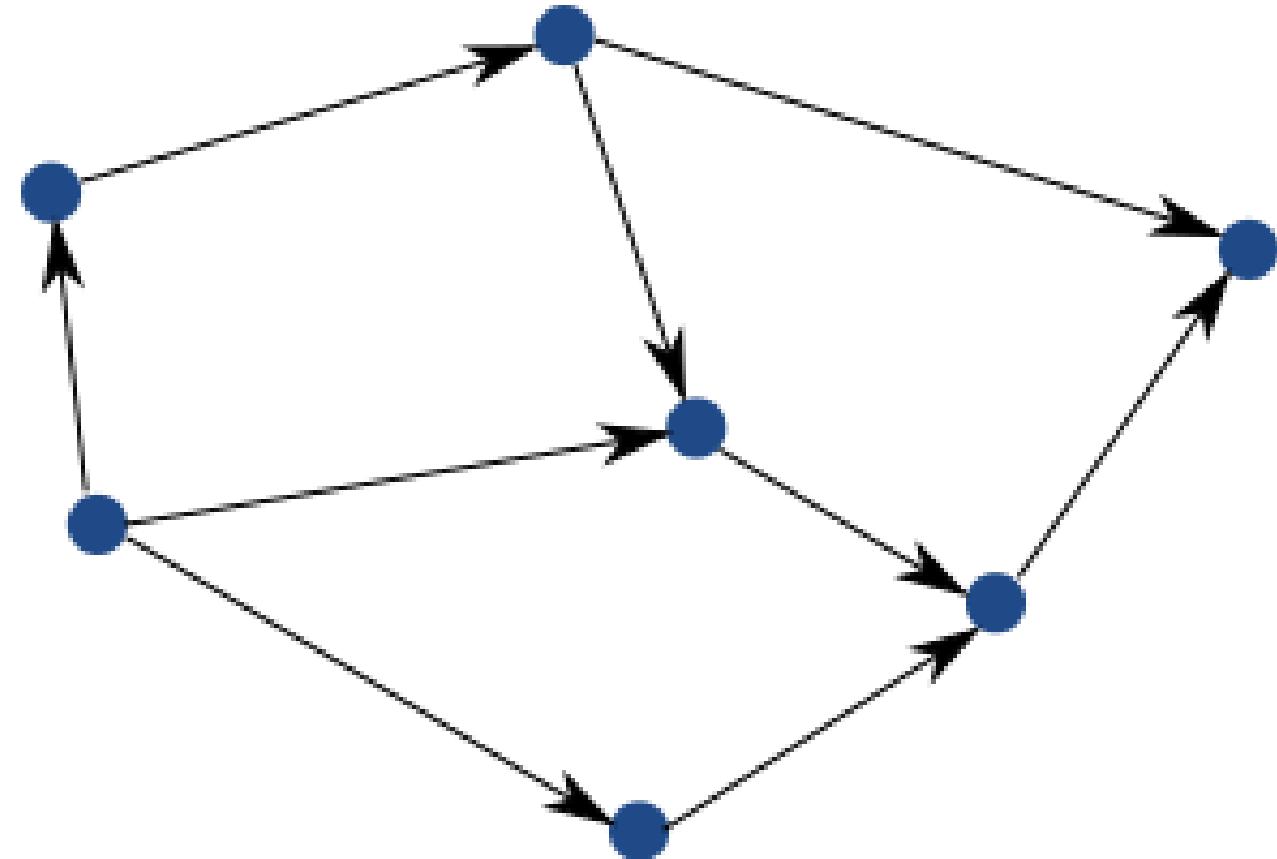
## Amdahl's Law



AXES

## Promises

- High parallelization
- For stateful systems
- Reactive Patterns
- Fault tolerance (self healing)



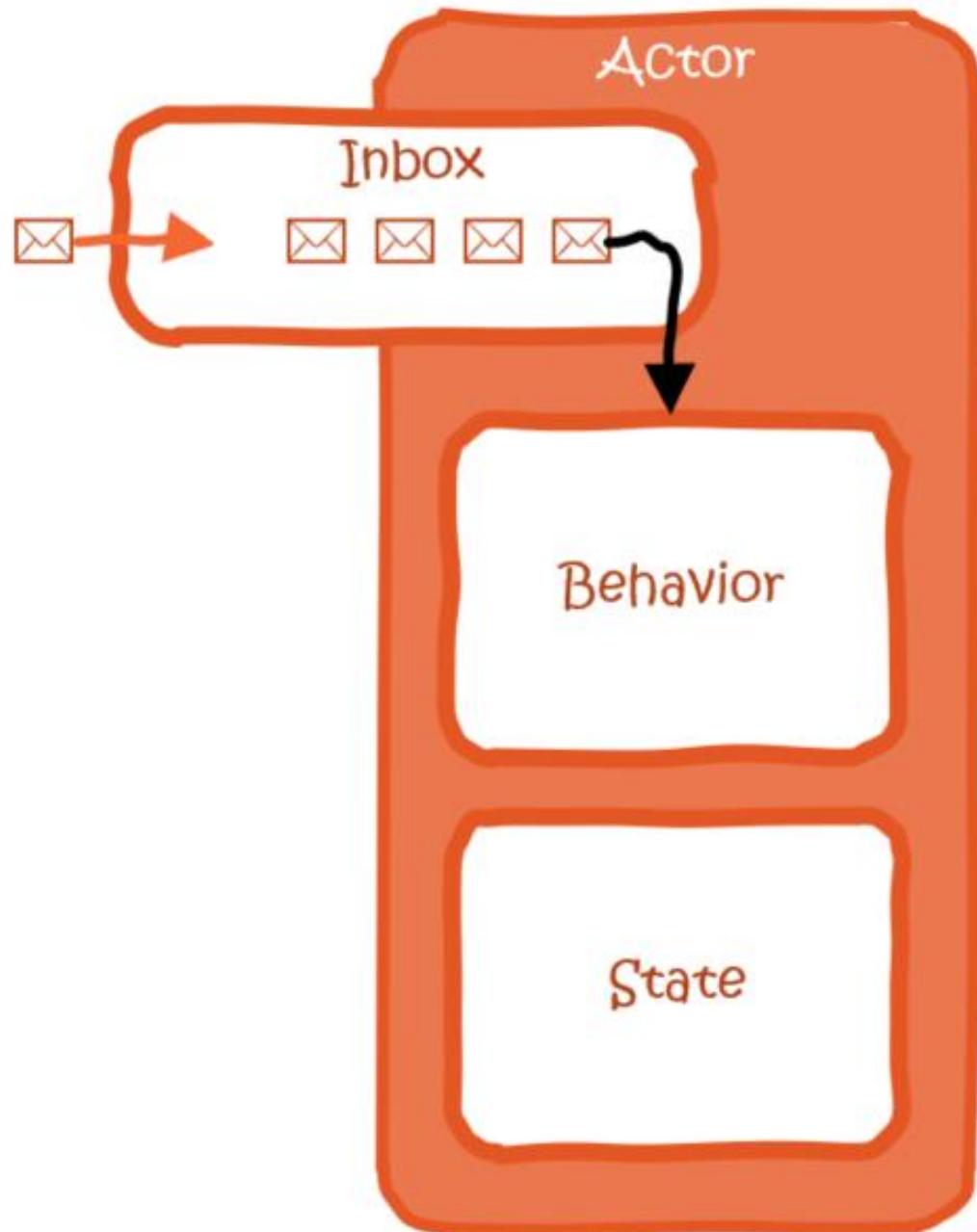
Actors interacting with each other  
by sending messages to each other

AXES

# Introduction to Akka.NET

---

How stuff works



## The Actor

### Simple object

- Holds its own state (no shared state)
- Inbox:
  - Messages (the only input)
  - Processed in order
- 1 message at a time

→ Guaranteed single threaded

## The simplest actor

```
public class MyActor : UntypedActor
{
    protected override void OnReceive(object message)
    {
        if (message is MyMessage myMessage)
            DoSomething(myMessage);
    }

    private void DoSomething(MyMessage myMessage)
    {
        // TODO: handle the message here
    }
}
```

AXES

## Messages

### Simple objects

- Immutable!
  - Akka.NET does not enforce this
  - DO NOT try to exploit this
- Might cross machine boundaries

### Throughput:

- Claimed: 50 M/s on a single machine
- Well over 1 M/s on my laptop

Axes



## An immutable message

```
public class MyMessage
{
    public int IntProperty { get; }
    public string StringProperty { get; }
    public ImmutableArray<decimal> Values { get; }

    public MyMessage(int intProperty, string stringProperty, ImmutableArray<decimal> values)
    {
        IntProperty = intProperty;
        StringProperty = stringProperty;
        Values = values;
    }
}
```

Axes

## The ActorSystem



### Manages:

- Actor life cycles
- Messaging
- Inboxes
- Thread scheduling
- The system event bus
- ...

## Creating an ActorSystem

```
static void Main(string[] args)
{
    ActorSystem system = ActorSystem.Create("MyActorSystem");

    Props myProps = Props.Create<MyActor>();

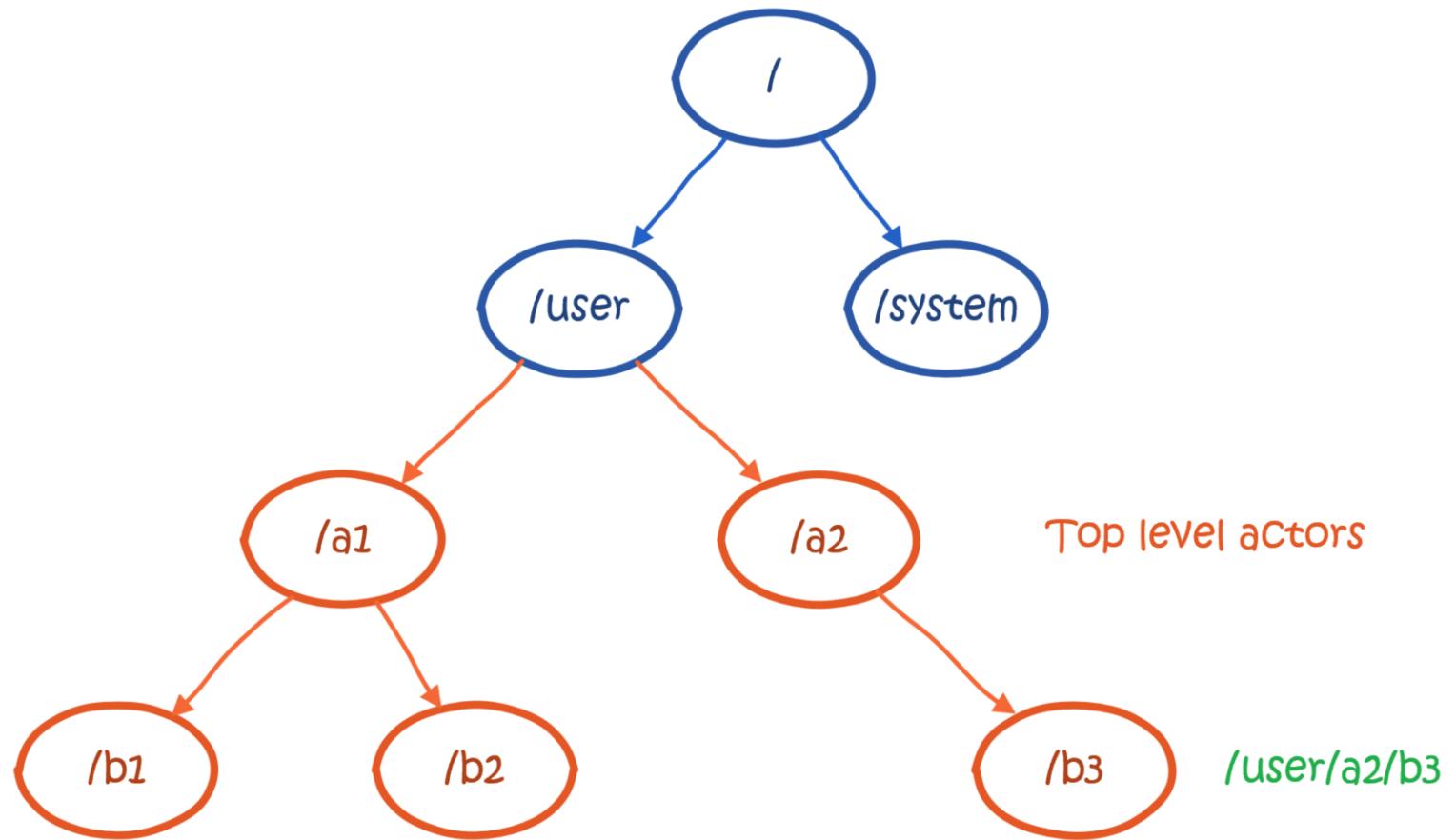
    IActorRef myActorRef = system.ActorOf(myProps, "my-actor-name");

    myActorRef.Tell(new MyMessage("hello"));
}
```

Axes

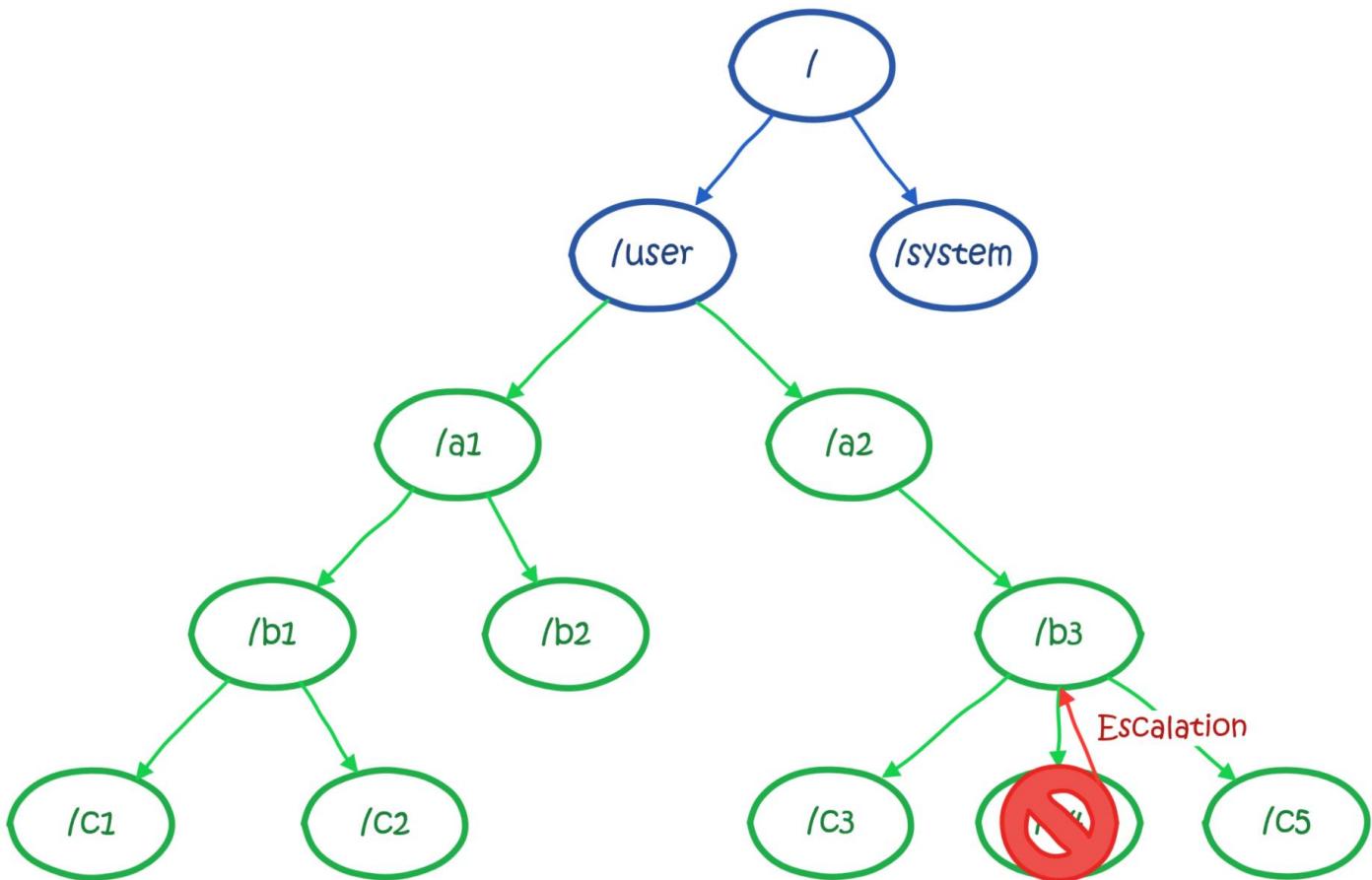
## Actor hierarchy

- Actors can have children
- Position = address
- 3 default actors:
  - /
  - /user
  - /system



AXES

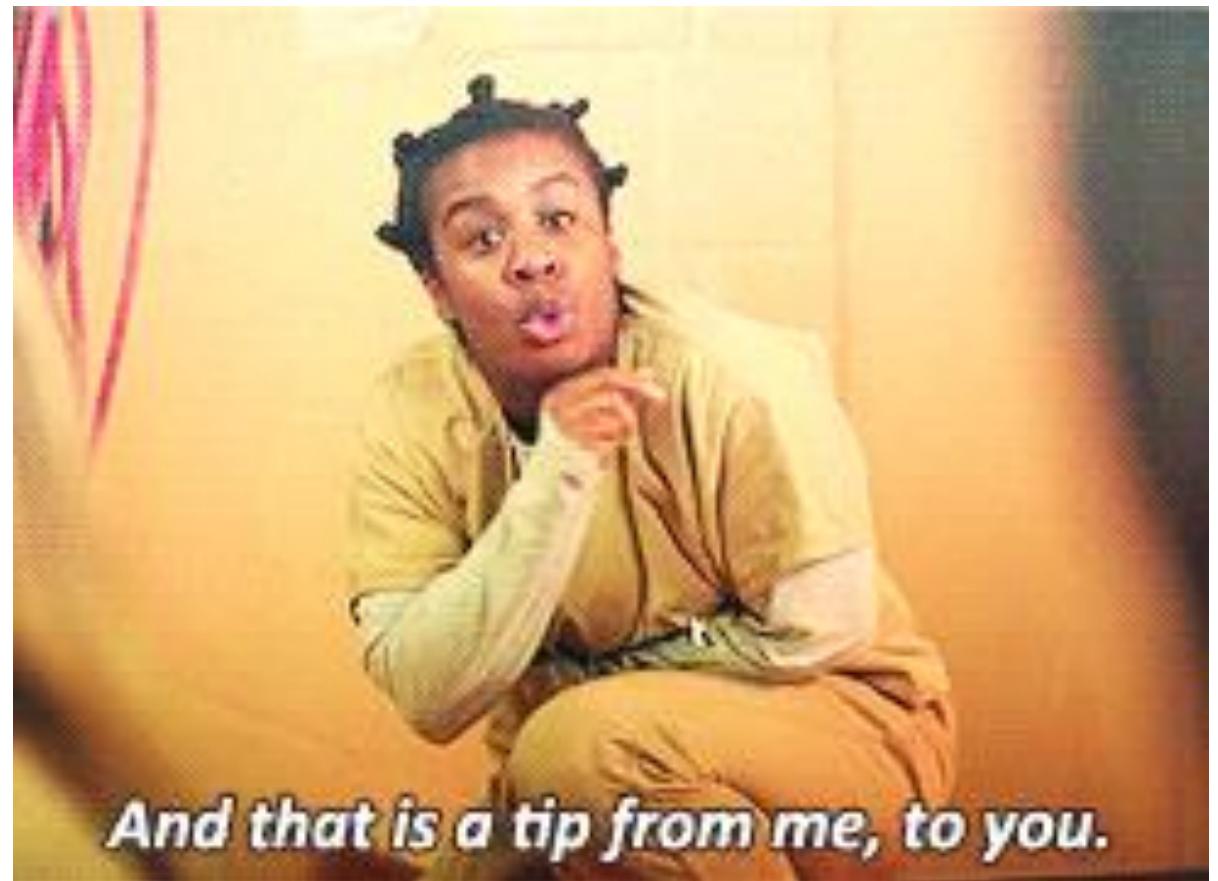
## Supervision



- Errors are escalated to the parent
- Parent decides OR escalates further
- Action:
  - Resume
  - Stop
  - Restart
- Strategy:
  - OneForOne: only the failing actor
  - OneForAll: all children

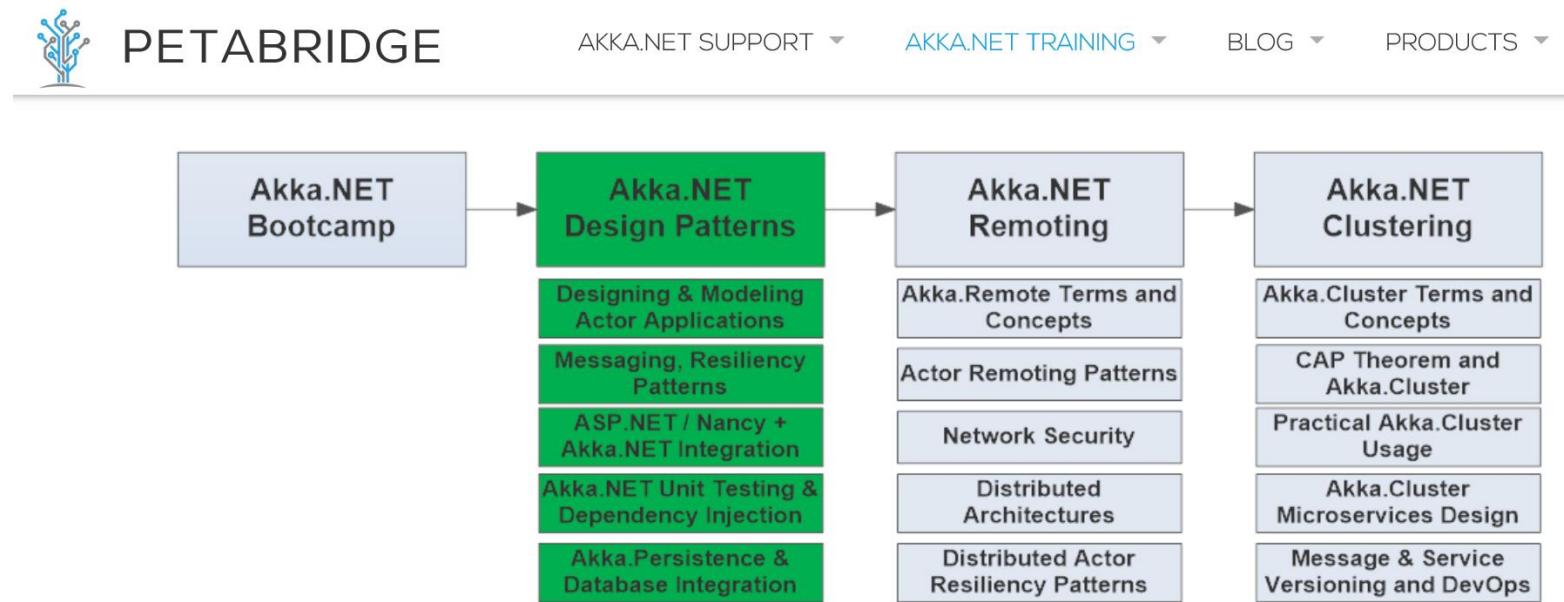
## Development ideas

- Split workloads into small chunks
- Make separate actors for every task
- Push risk to the edges,  
handle faults there
- Avoid ‘bottleneck actors’



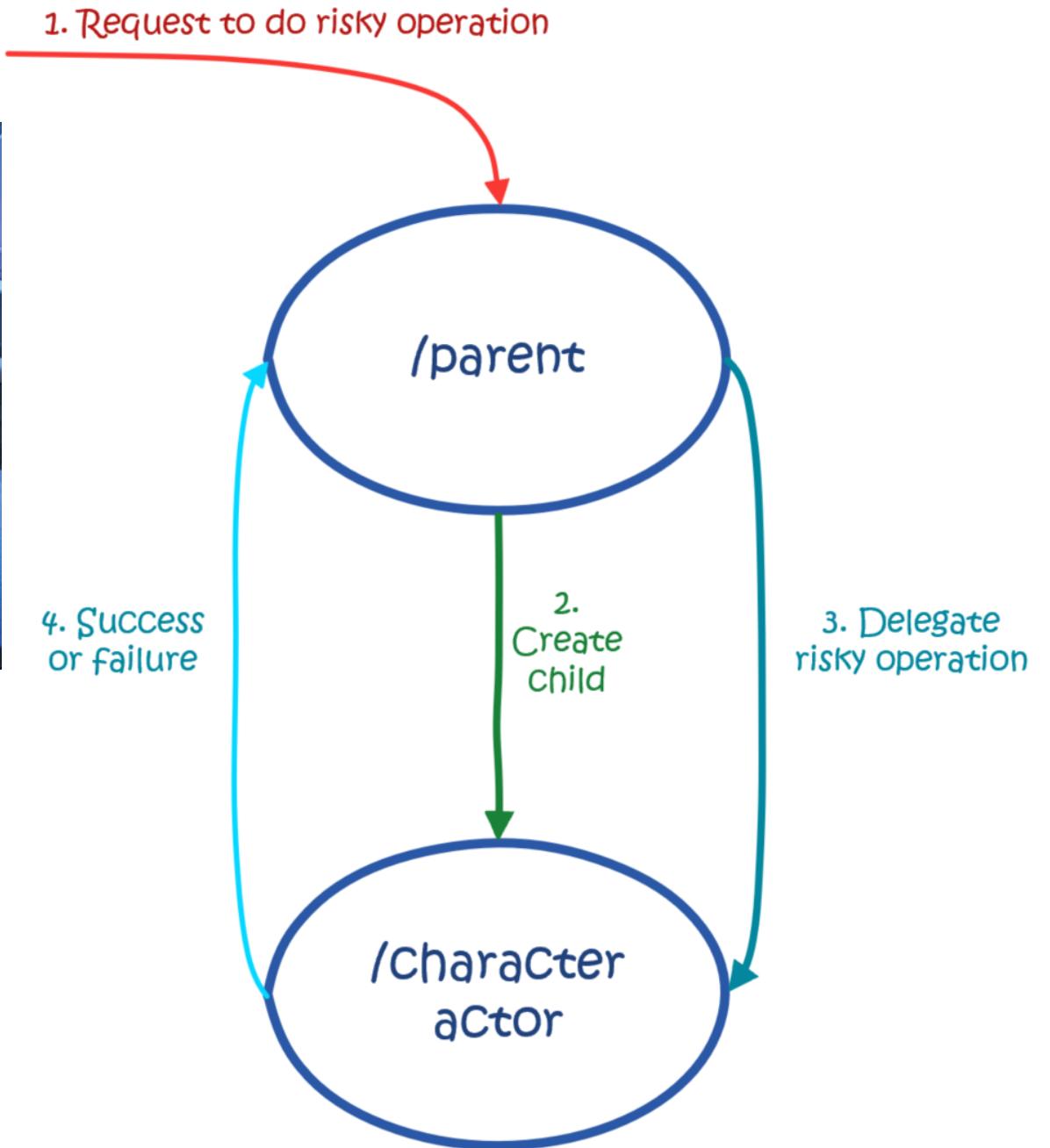
# Design Patterns

- Fan-out Pattern
- Parent Proxy Pattern
- Consensus Pattern
- Character Actor
- ...



AXXES\_

## The character actor



AXES\_

## The problem domain

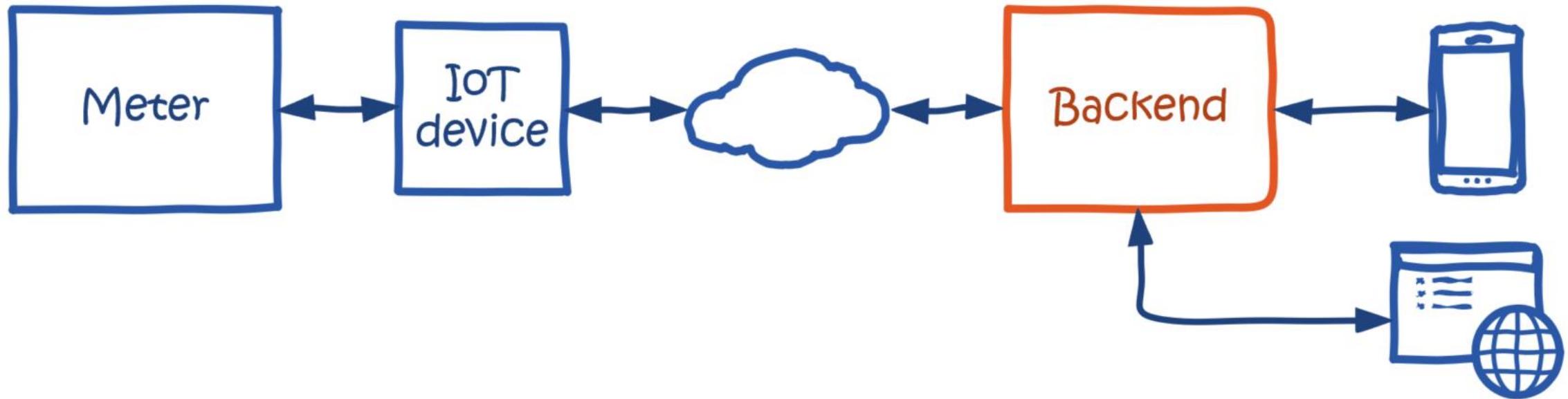
---

What are we trying to solve?



AXXES

## Connection situation



AXXES

## What do we want?

### Storage of historic usage

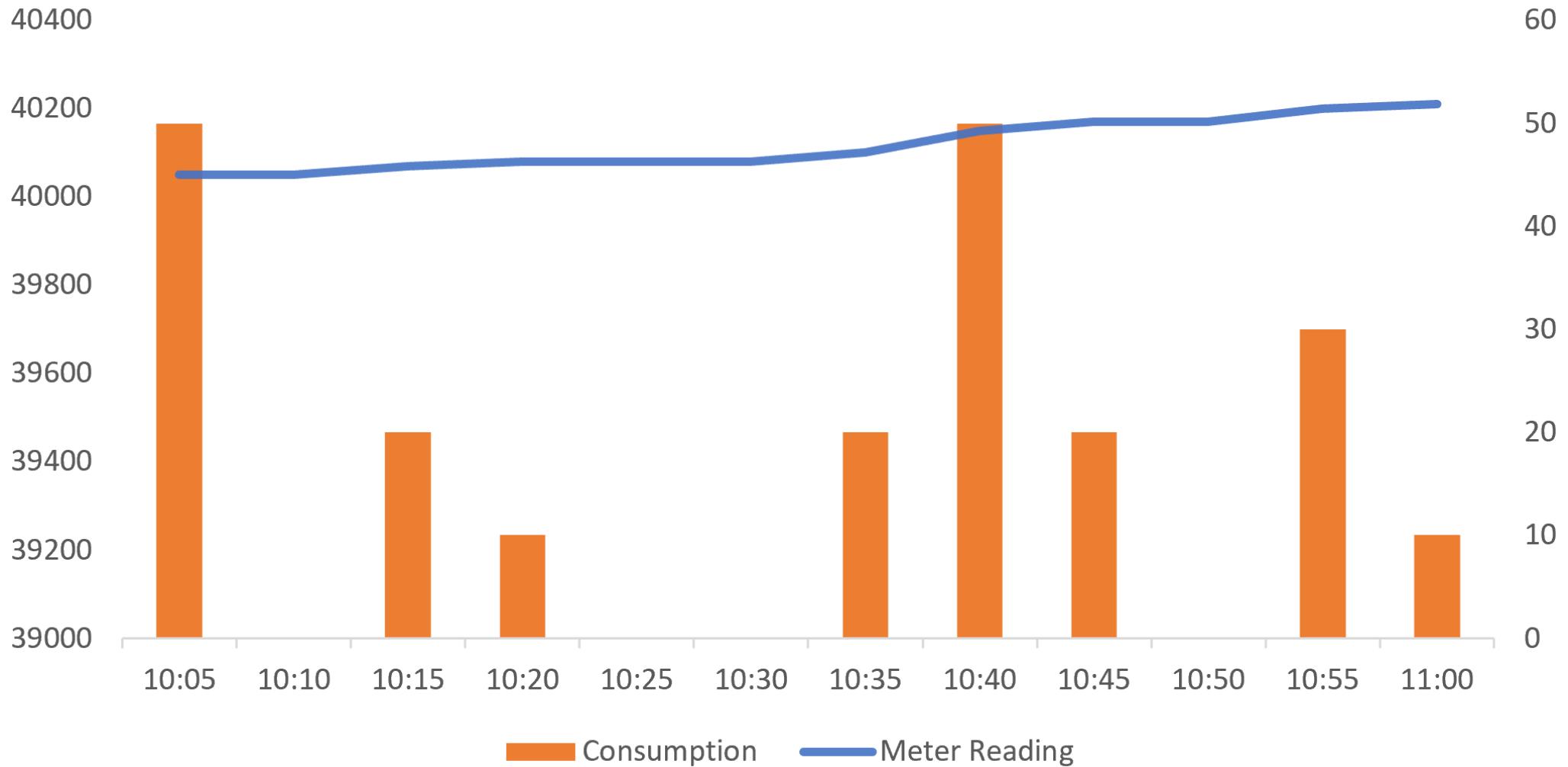
- Storage of (normalized) values
- Plotting of consumption graphs
- Comparison of time periods

### Alerting

- Momentary consumption threshold
- Periodic consumption threshold

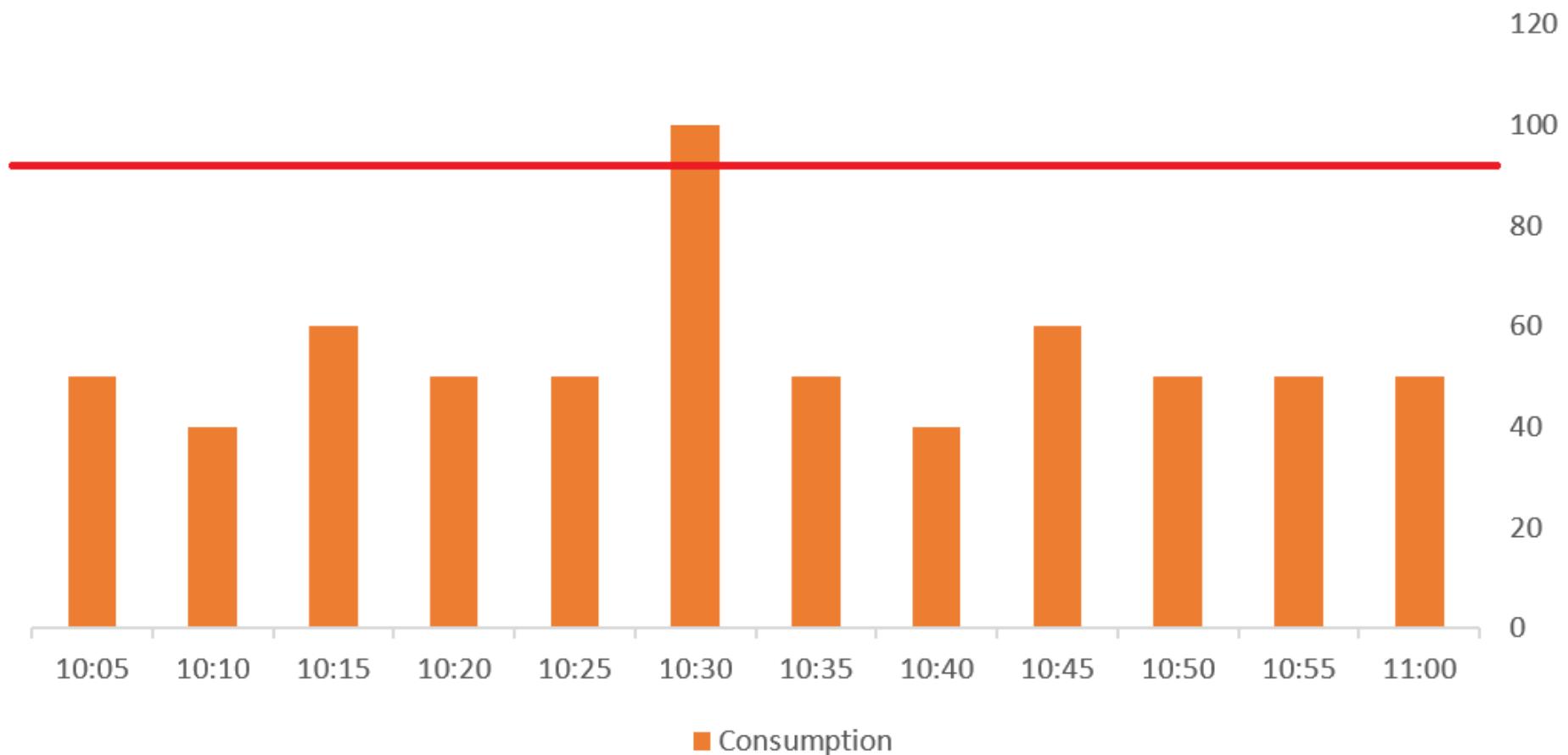
Axes

## Reading vs Consumption



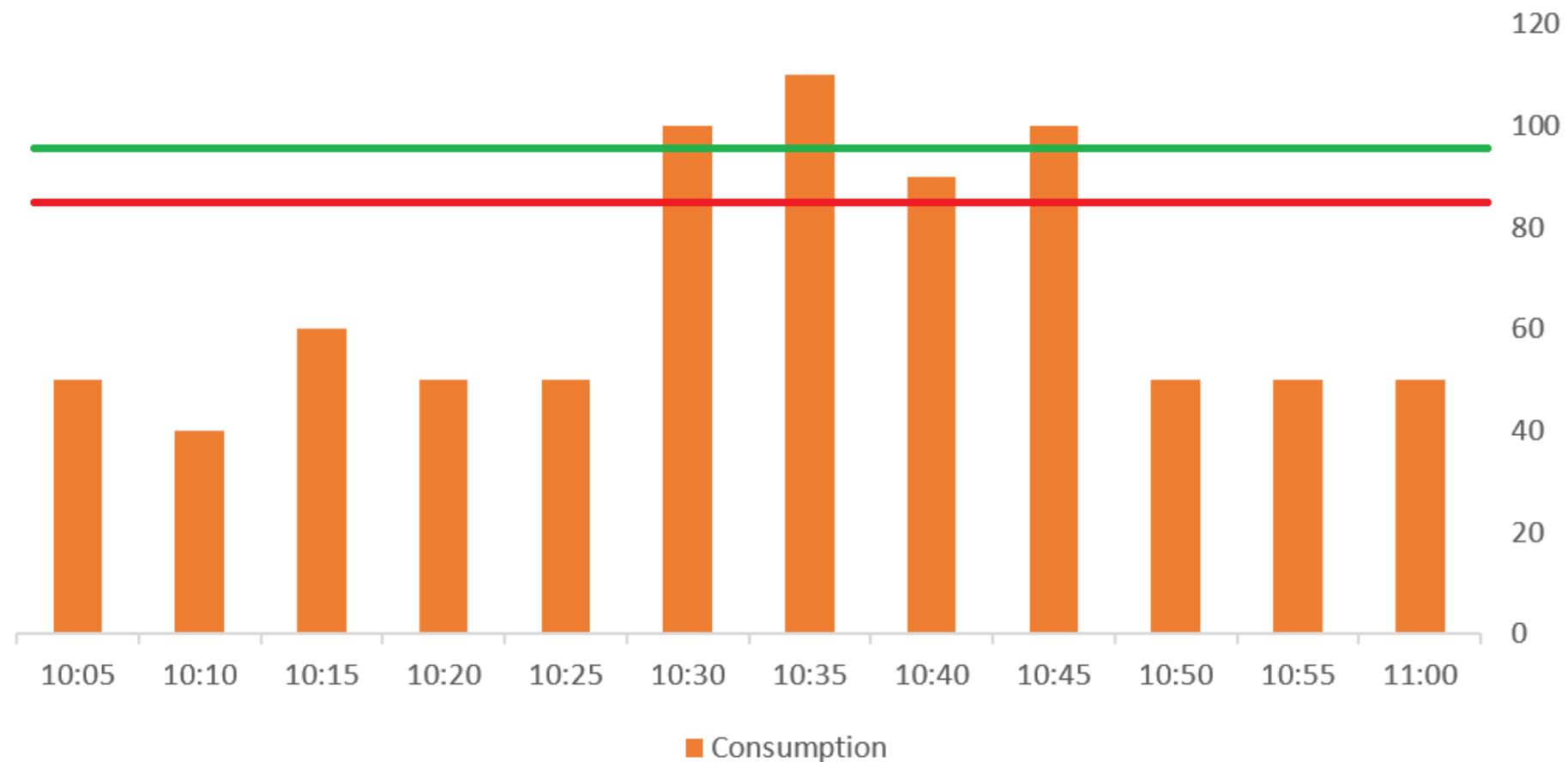
AXXES

## Momentary threshold alert



AXXES

## Momentary threshold alert



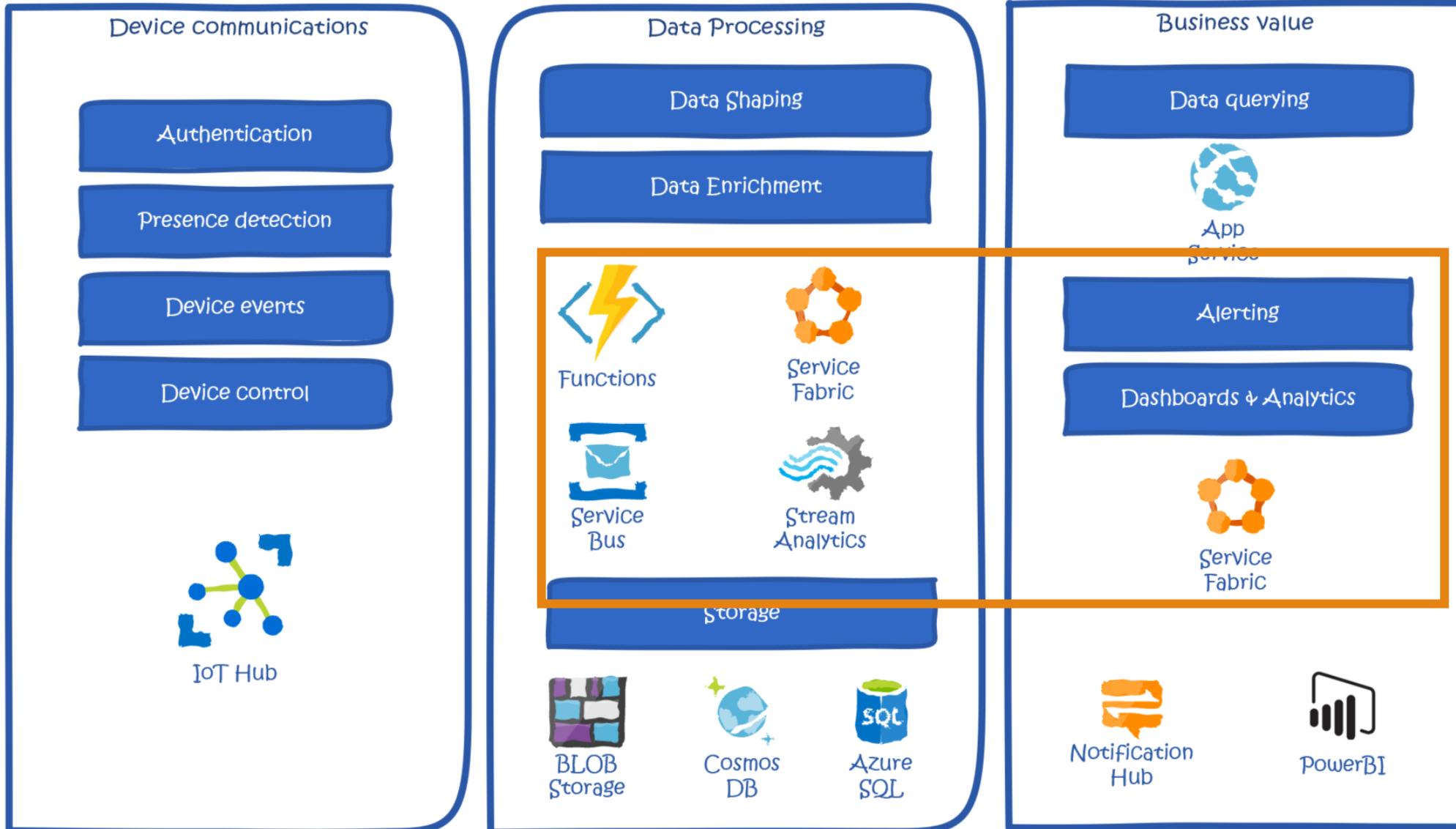
AXXES

## How Akka.NET fits in

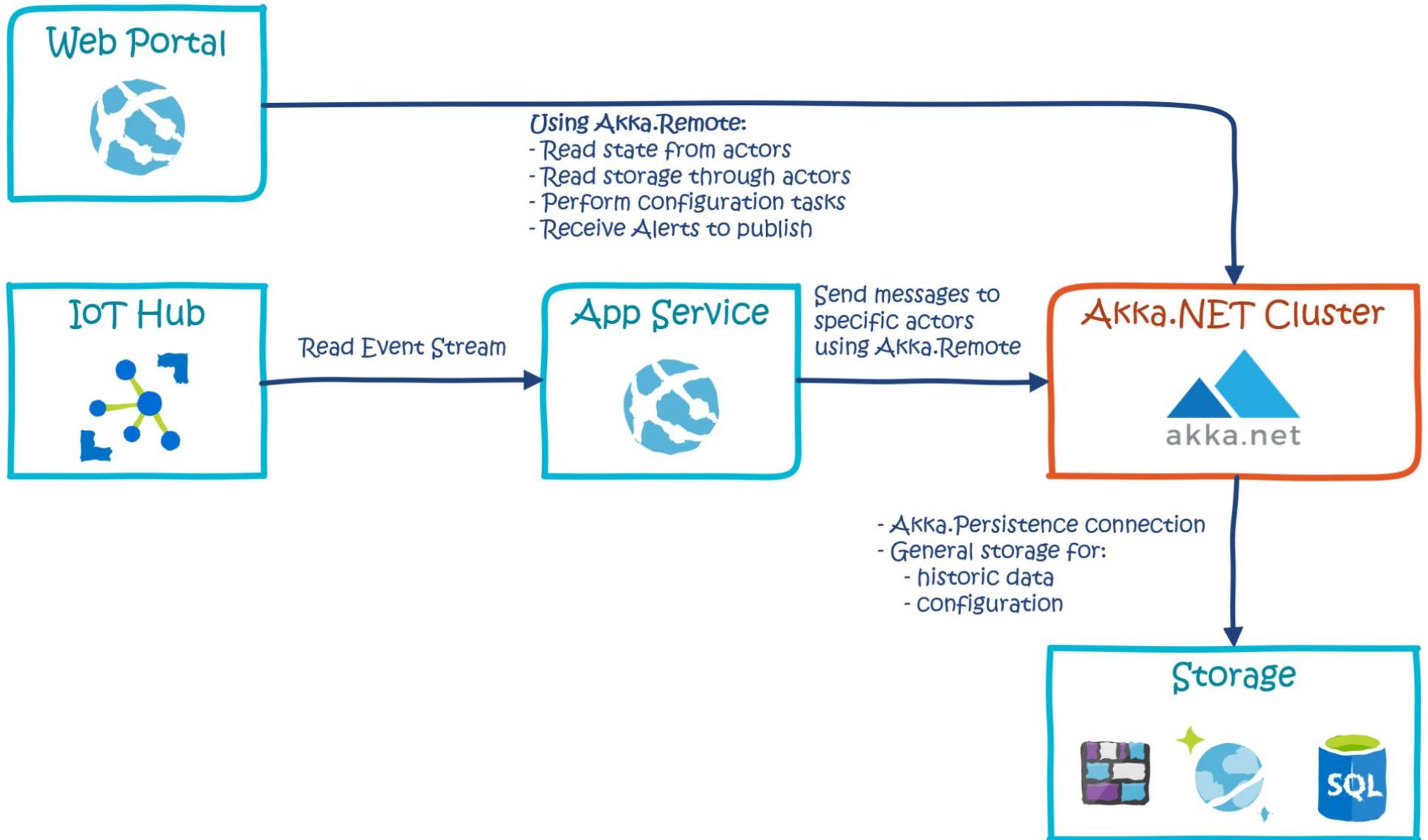
---

What part of the solution can Akka.NET provide?

# Your typical IoT stack



## Backend



AXXES

## Don't be a magpie!

### Good fits:

- Gaming backends
- Trading systems
- Internet of Things
- Parallelizable calculations
- ... any stateful high throughput application

It doesn't have to be the whole solution!

Axes



## Coding together

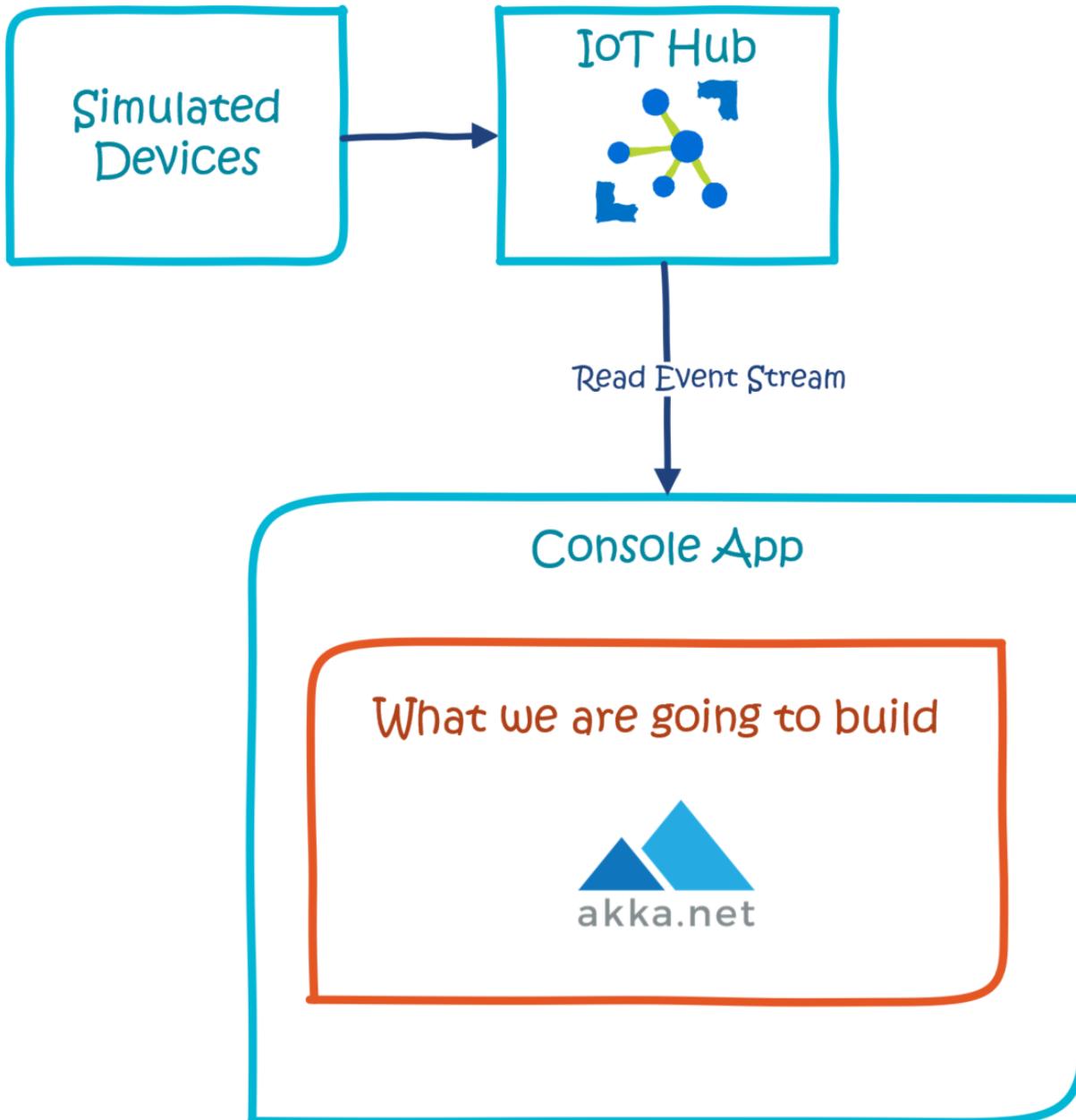
---

Enough chit-chat, let's see this thing in action!

## Our first Actors

---

Let's spin up an ActorSystem and make it do something



# Normalizing Measurements

---

-

Making sure actors get consistent data

## Why normalization?

**Writing logic is easier with consistent values:**

- Exact timestamps
- No gaps
- Incorrect values filtered
- ...

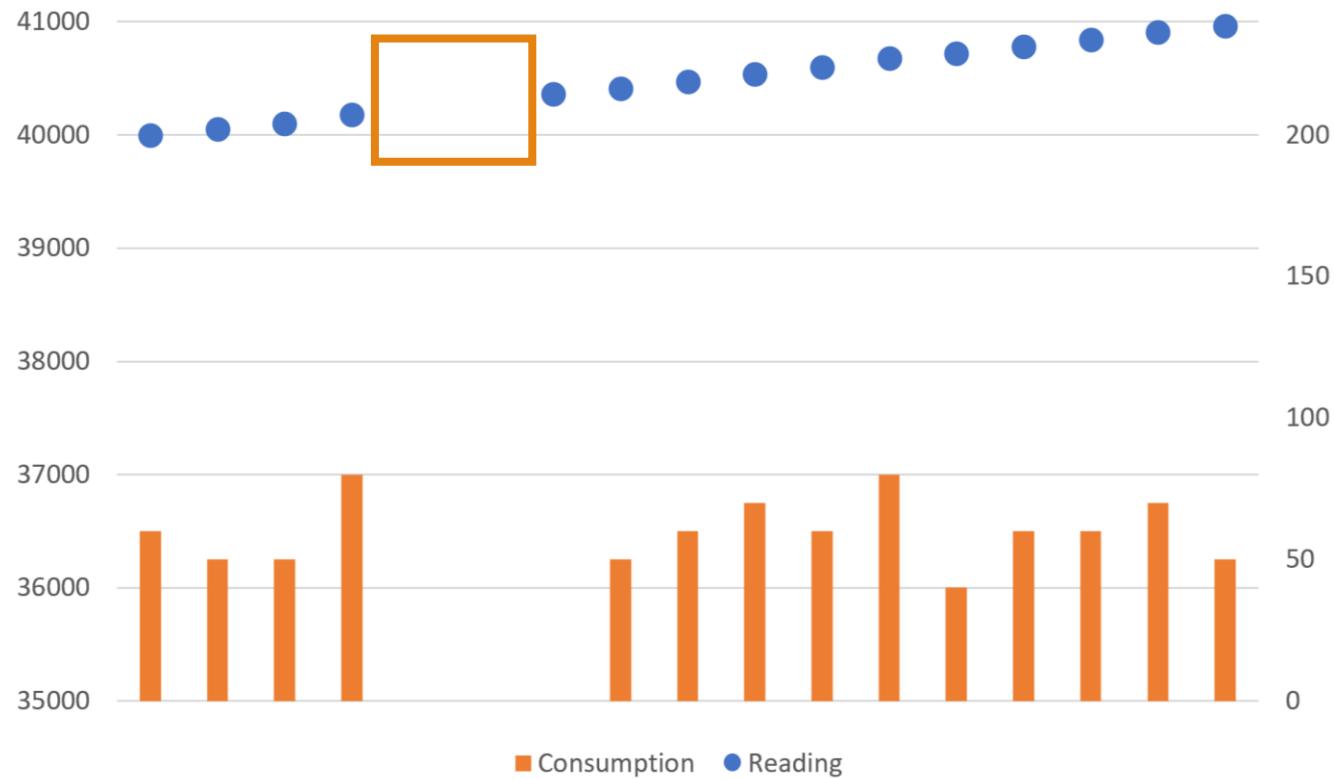
**Deal with it in one place**



## Timestamp correction & buckets

RAW			NORMALIZED		
Timestamp	Reading	Consumption	Timestamp	Reading	Consumption
9:59:25	40000				
10:00:25	40060	60	10:00:00	40035	
10:01:25	40120	60			
10:02:25	40180	60			
10:03:25	40240	60			
10:04:25	40300	60			
10:05:25	40360	60	10:05:00	40335	300
10:06:25	40420	60			
10:07:25	40480	60			
10:08:25	40540	60			
10:09:25	40600	60	10:10:00	40635	300
10:10:25	40660	60			
10:11:25	40720	60			
10:12:25	40780	60			
10:13:25	40840	60			
10:14:25	40900	60			
10:15:25	40960	60	10:15:00	40935	300

AXES



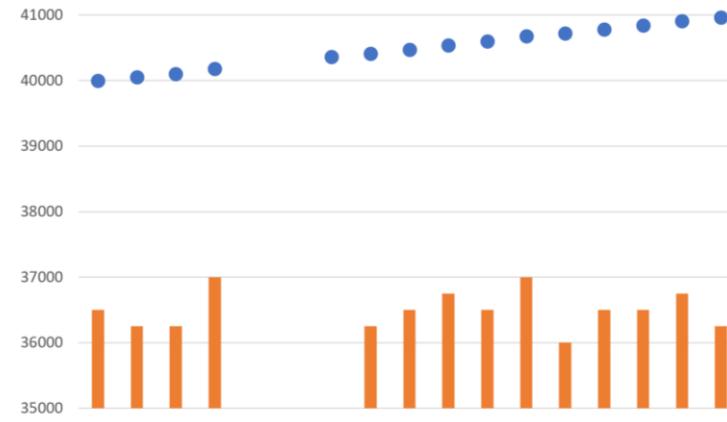
## Gap filling

- Do we want to fill this gap?
- If so, how?
- Do other Actors need to know?  
If yes, add a flag to the message

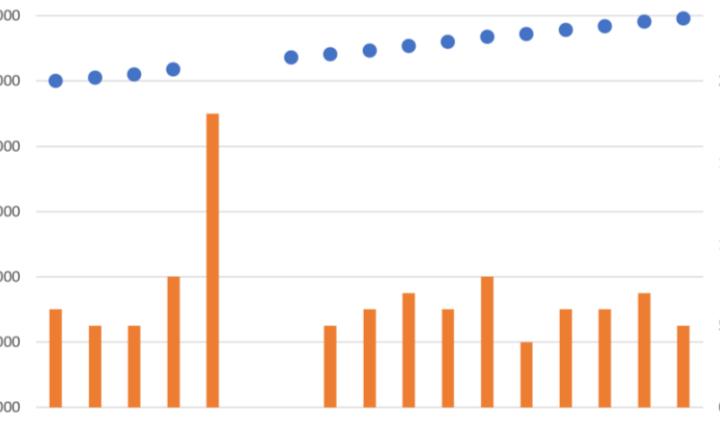
**There is no 'right' answer**

# Possible solutions

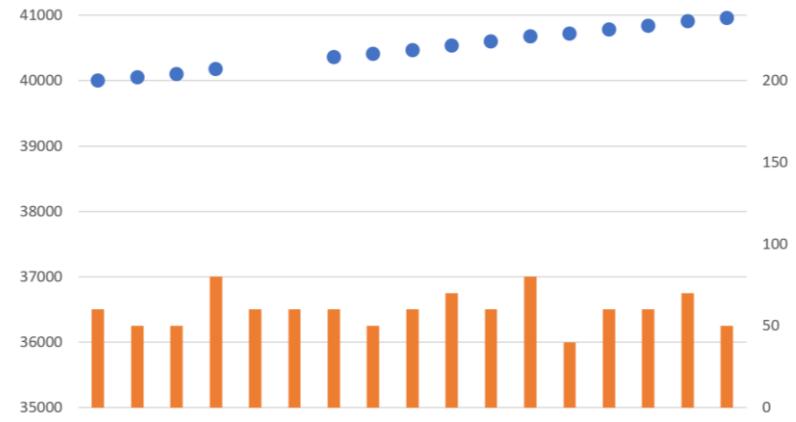
Not filled



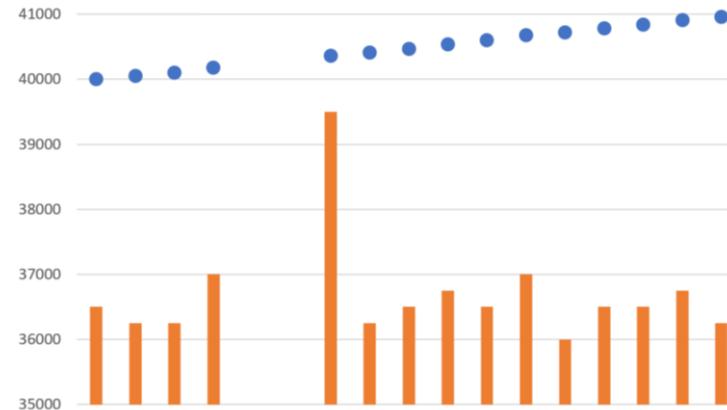
## Peak at start



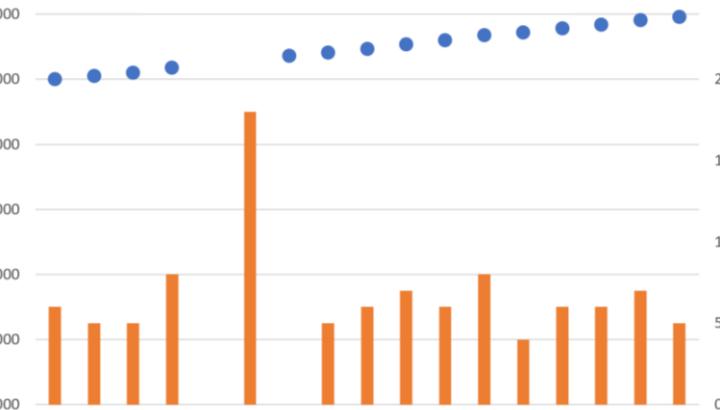
# Evenly split



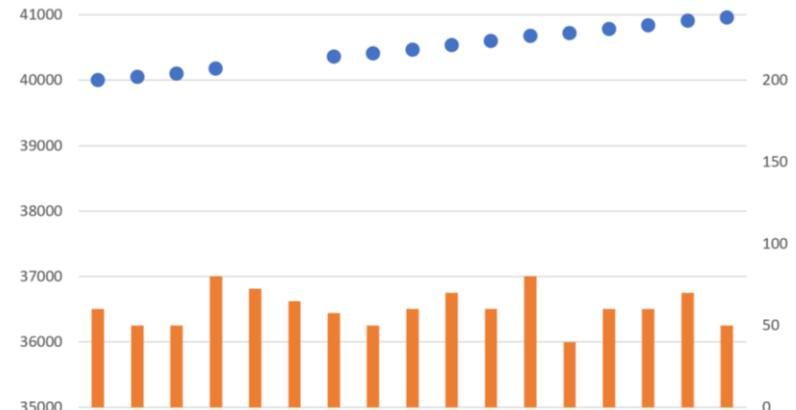
## Peak at end



## Peak in center

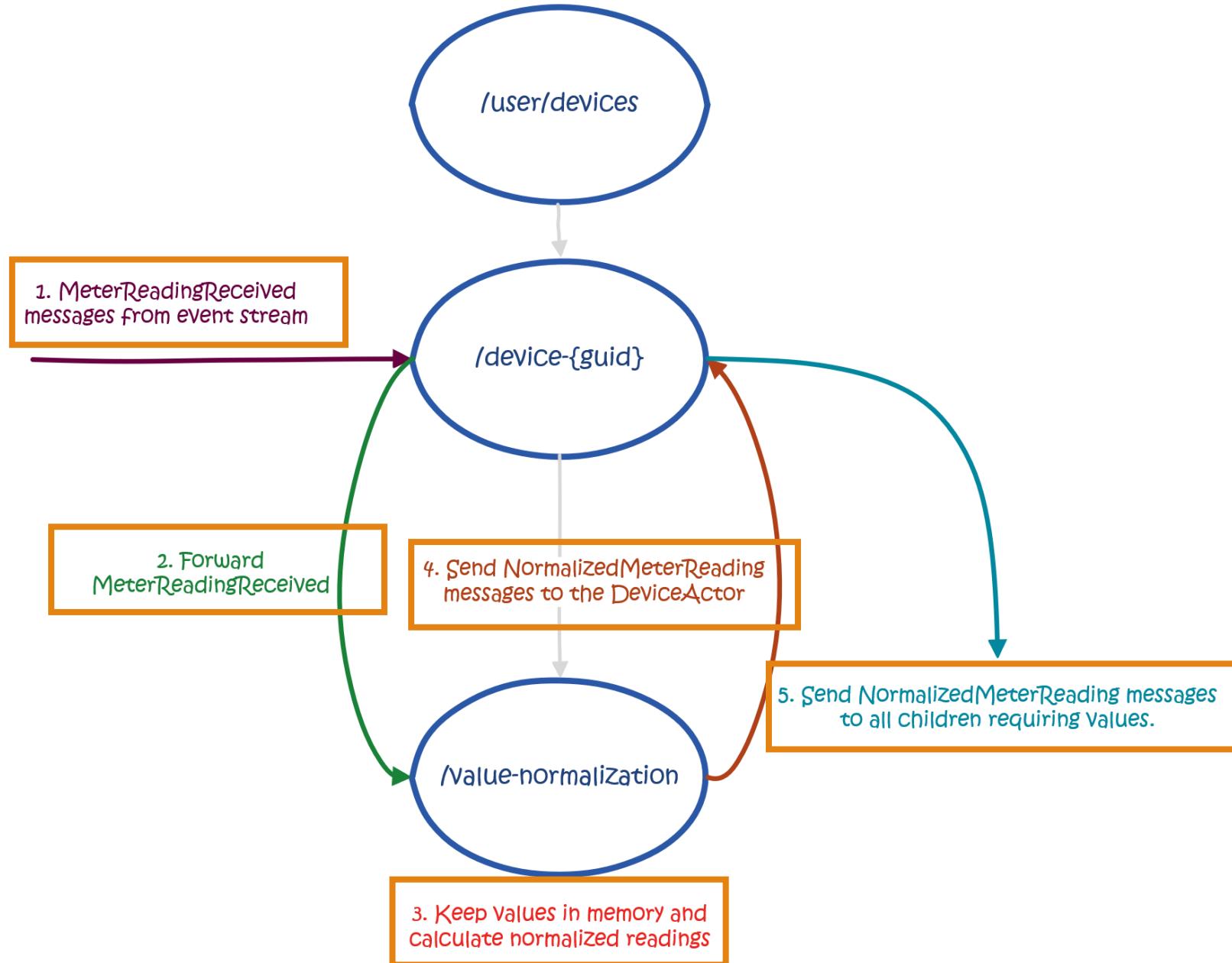


## Trend line



# AXXES

## Actors



AXXES

# Persisting Data

---

-

Saving what cannot be lost

## Akka.Persistence

Actors that recover their state when (re-)created:

- Inherit from `PersistentActor`
- Give it a unique `PersistenceId`
- Persist Events with the `Persist(...)` command
- Persist snapshots with the `SaveSnapshot(...)` command
- Register `Recover<T>(...)` handlers to restore state

```
public class MyPersistedActor : ReceivePersistentActor
{
    // Any PersistentActor needs a unique key!
    public override string PersistenceId { get; }

    // Grouping state into a state object is a good idea
    private MyState _state = new MyState();

    public MyPersistedActor(Guid id)
    {
        PersistenceId = $"my-persisted-actor-{id}";

        // There's a difference between 'Commands' and 'Recovers'
        Command<MyMessage>(HandleCommand);
        Recover<MyMessage>(HandleMessageInternal);

        // Snapshot events
        Recover<SnapshotOffer>(HandleSnapshotOffer);
        Command<SaveSnapshotSuccess>(HandleSnapshotSuccess);
        Command<SaveSnapshotFailure>(HandleSnapshotFailure);
    }
}
```

AXES

```
private int _msgSinceLastSnapshot = 0;

private void HandleCommand(MyMessage command)
{
    // Persists the message to the store and the actor simultaneously.
    Persist<MyMessage>(command, HandleMessageInternal);

    // Save a snapshot every 100 messages
    if (_msgSinceLastSnapshot == 100)
    {
        SaveSnapshot(_state);
        _msgSinceLastSnapshot = 0;
    }
}

private void HandleMessageInternal(MyMessage message)
{
    // In recovery, we call this directly, no need to persist it again.
    _state.Add(message);
    _msgSinceLastSnapshot++;
}
```

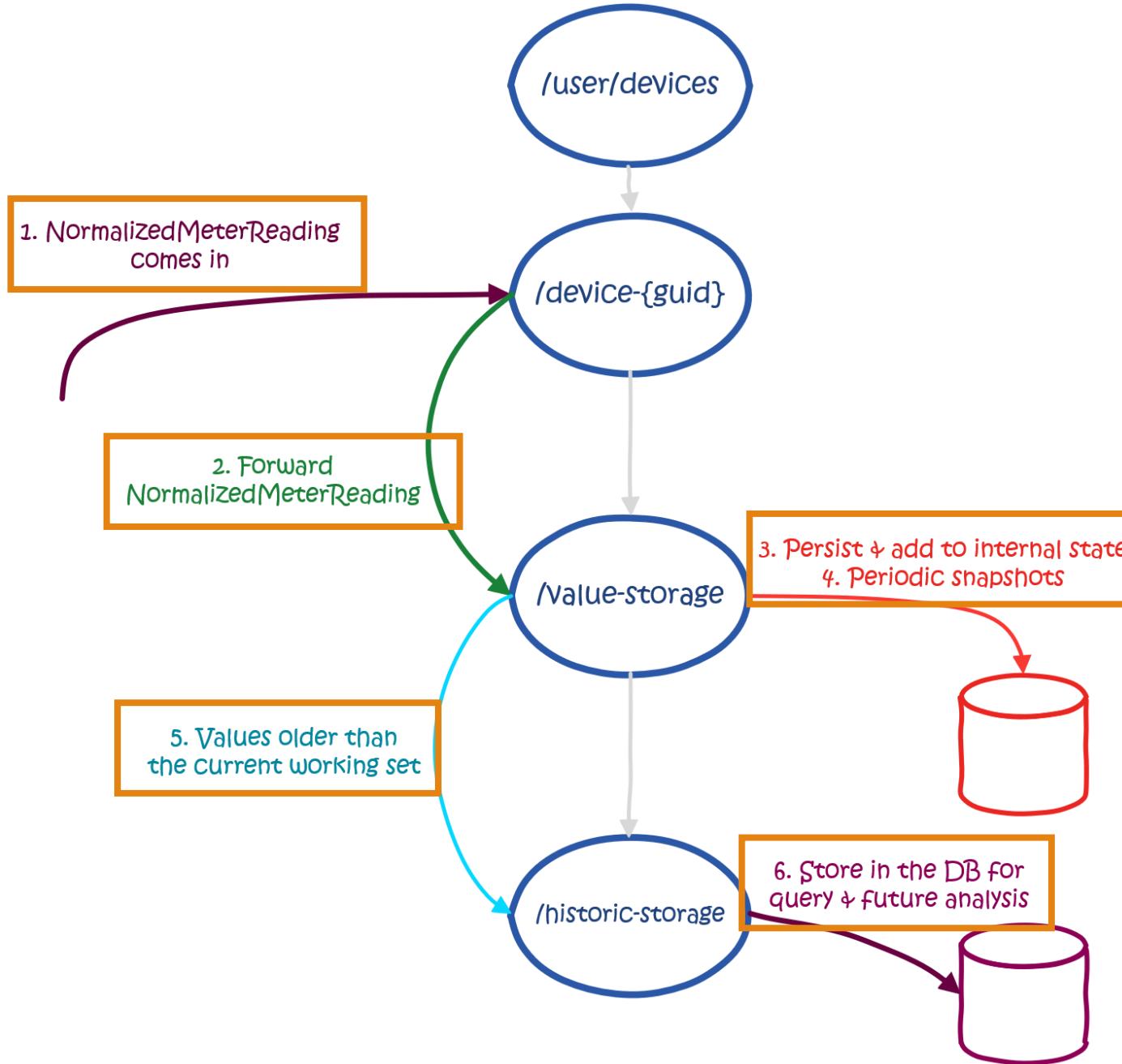
```
private void HandleSnapshotOffer(SnapshotOffer offer)
{
    if (offer.Snapshot is MyState newState)
        _state = newState;
}

private void HandleSnapshotSuccess(SaveSnapshotSuccess success)
{
    // Handle a successful snapshot save
}

private void HandleSnapshotFailure(SaveSnapshotFailure failure)
{
    // Handle the failure to save a snapshot
}
```

AXES

## Actors



AXXES

## Further implementation details

---

What else would you encounter if you wanted to build this?

# Getting messages to the ActorSystem

---

Akka.NET Remoting  
Proxy Actors

# Akka.Remote

## ActorSystems can talk to other ActorSystems

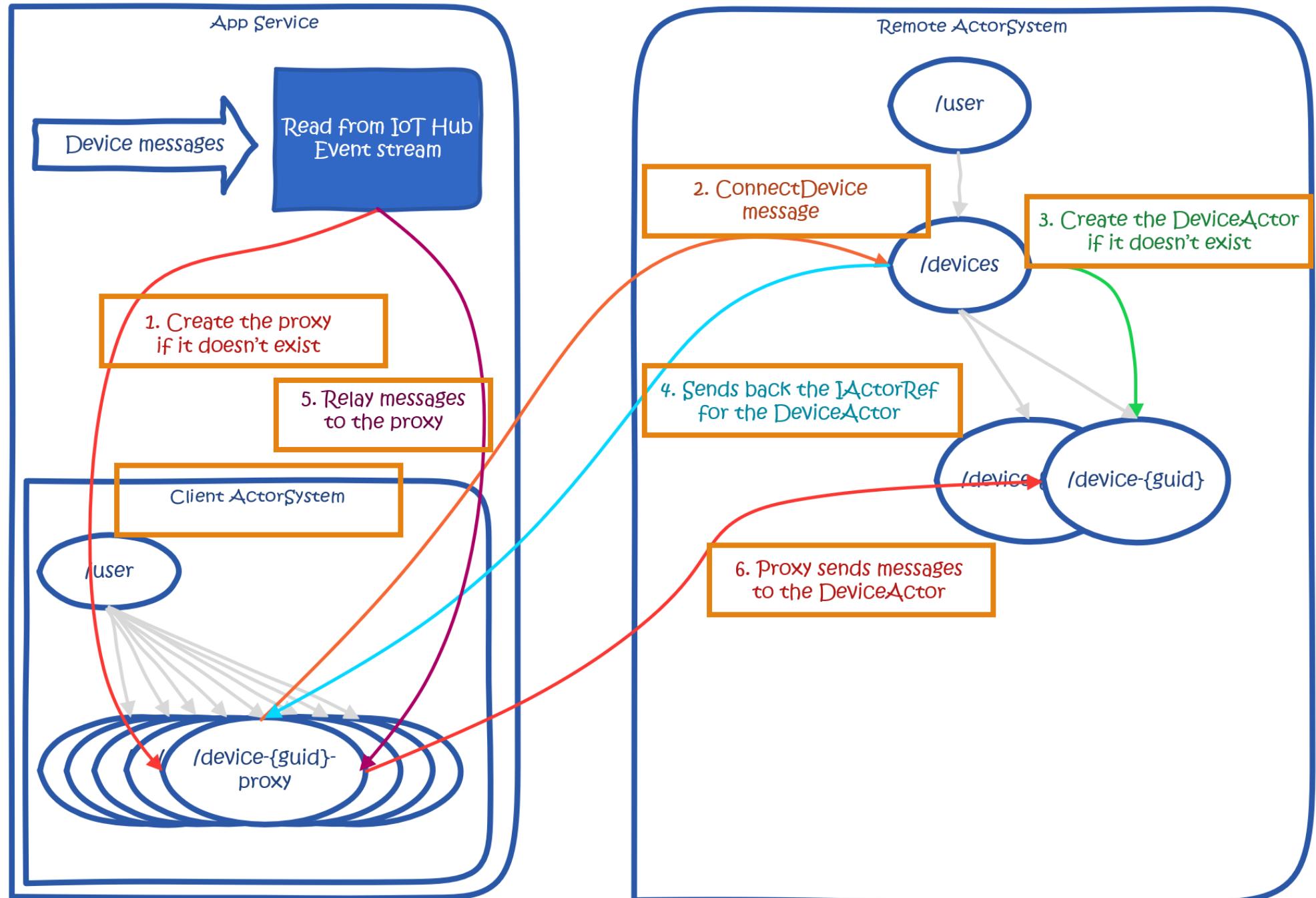
- Remote addressing
- Remote deployment
- Remote messaging
- Location Transparency
- Multiple transports

All parts form an "ActorPath"



AXES

AXXES



## DeviceActorProxy

```
class DeviceActorProxy : ReceiveActor
{
    private readonly Guid _deviceId;
    private IActorRef _deviceActor;

    public DeviceActorProxy(Guid deviceId) ...

    protected override void PreStart()
    {
        var devicesActorPath = $"{Constants.RemoteActorSystemAddress}/user/devices";
        var devicesActor = Context.ActorSelection(devicesActorPath);

        var request = new ConnectDevice(_deviceId);
        devicesActor.Tell(request);
    }

    private void HandleDeviceConnected(DeviceConnected message)
    {
        _deviceActor = message.DeviceRef;
    }
}
```

## DeviceActorProxy ... continued

```
class DeviceActorProxy : ReceiveActor
{
    private readonly Guid _deviceId;
    private IActorRef _deviceActor;

    public DeviceActorProxy(Guid deviceId)
    {
        _deviceId = deviceId;
        Receive<MeterReadingReceived>(HandleMeterReadingReceived);
        Receive<DeviceConnected>(HandleDeviceConnected);
    }

    protected override void PreStart()...

    private void HandleDeviceConnected(DeviceConnected message)...

    private void HandleMeterReadingReceived(MeterReadingReceived message)
    {
        _deviceActor?.Tell(message);
    }

    public static Props CreateProps(Guid deviceId)
    {
        return Props.Create<DeviceActorProxy>(deviceId);
    }
}
```

## DevicesActor

```
private void HandleConnectDevice(ConnectDevice request)
{
    if (!_deviceActors.ContainsKey(request.Id))
    {
        CreateDeviceActor(request.Id);
    }
    var response = new DeviceConnected(_deviceActors[request.Id]);
    Sender.Tell(response);
}

private void CreateDeviceActor(Guid deviceId)
{
    var props = DeviceActor.CreateProps(deviceId);
    var name = $"device-{deviceId}";
    var deviceActorRef = Context.ActorOf(props, name);

    _deviceActors[deviceId] = deviceActorRef;
}
```

# Restart behavior

---

-

How to get going again after a restart

## After a system restart

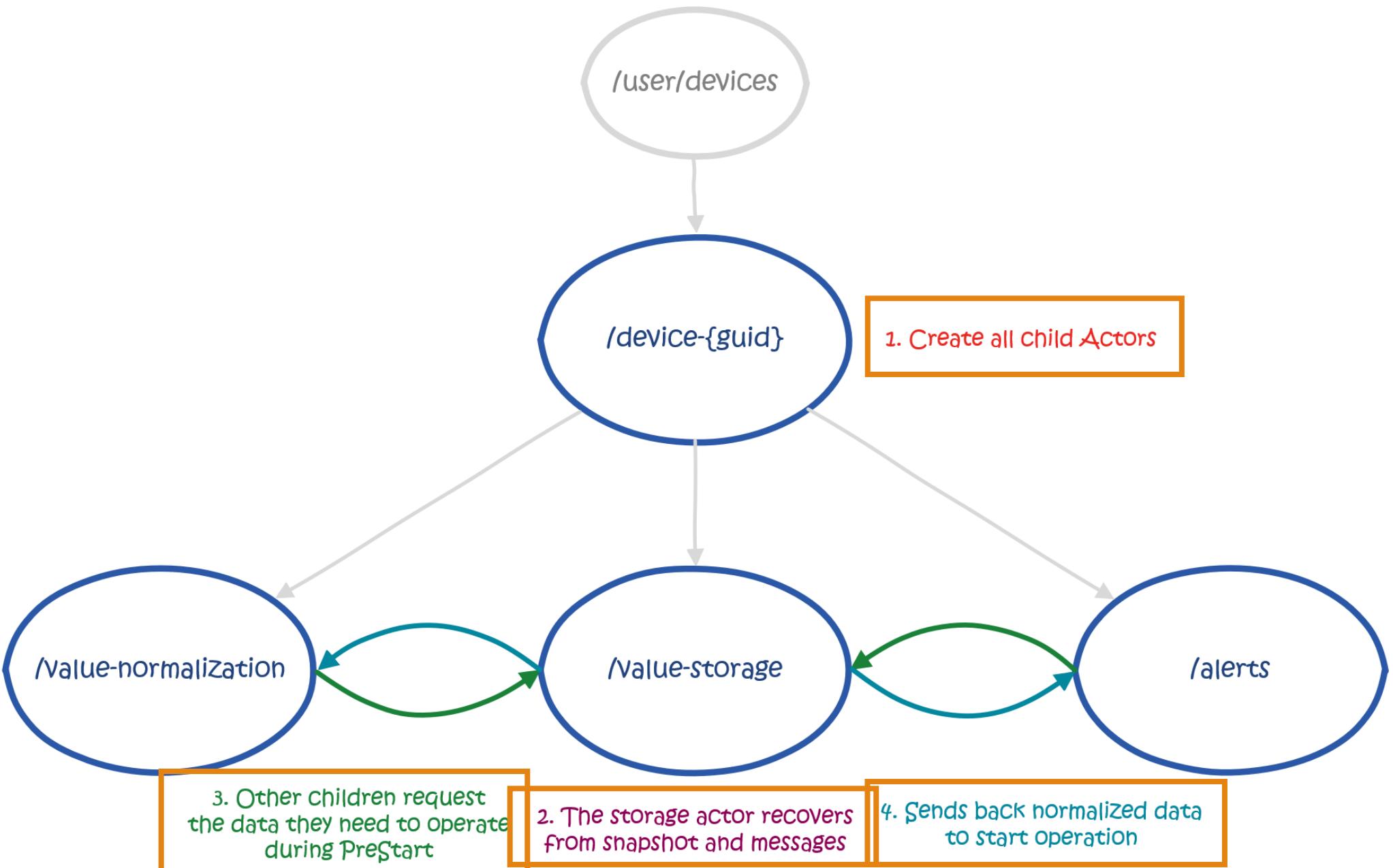
### Recreating Actors:

- Query the DB on startup
- Create the required Actors

### How to get Actor state back:

- Minimize the number of actors that need to recover state
- 1 PersistedActor per device = ideal
- Other actors query that actor for the state they need

## Actors



AXXES

## Beyond this workshop

---

The stuff that we didn't talk about ...

# Making Akka.NET production ready

- **Configuration:**  
HOCON
- **Clustering:**  
Run across multiple machines
- **Logging:**  
Adapters for Nlog, SeriLog, etc.
- **Dependency Injection:**  
Akka.NET supports DI for your actors  
(anti-pattern!)
- **Production monitoring:**  
Phobos





## Start learning

- 1. FREE Akka.NET Bootcamp by Petabridge:**  
<https://github.com/petabridge/akka-bootcamp>
- 2. PluralSight courses:**  
There are some good courses available!
- 3. Petabridge blog:**  
<https://petabridge.com/blog/>
- 4. Petabridge remote training (paid):**  
Worth it when you have serious questions

## Deployment

1. Pause the process that reads from the event stream
2. Wait for processing to end
3. Deploy the Akka.NET cluster
4. Re-create actors (triggering Persistence restores)
5. Resume sending from the event stream

→ When done right, you can do this without losing data!

→ **AUTOMATE THIS!**

## Conclusion

1. Check if your problem domain is a fit for Actors
2. Decide which part of the solution will be Akka.NET
3. Design your actor hierarchies appropriately
4. Normalizing data helps a lot
5. Think about deployment & recycles

EP