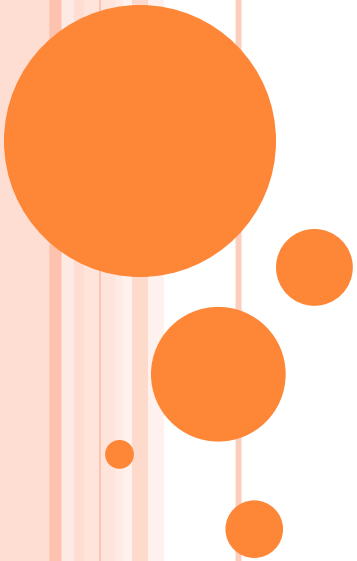


Network Programming: Actor Programming

COMP 30220: Distributed Systems

Lecturer: Rem Collier

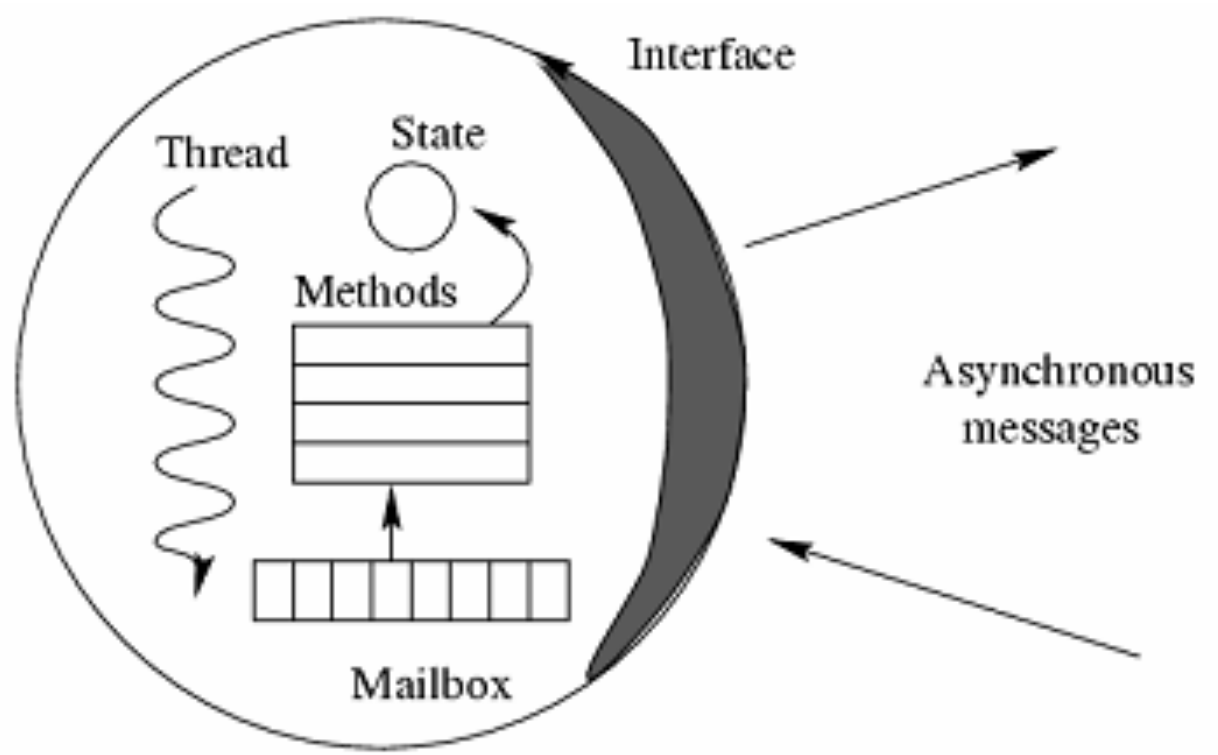
Email: rem.collier@ucd.ie

Actors

- Initially Proposed in the mid 1970s as a alternative model of concurrent computation.
  - Systems are made up of actor components that interact by passing messages.
  - Computation is realised through the handling of messages by:
    - Making local decisions
    - Creating more actors
    - Sending messages to other actors
  - Actors communicate **asynchronously:**
    - An actor does not wait for a response when sending a message.
  - Actors are **stateful** and **isolated:**
    - They maintain local (private) state
    - They share state by sending messages (no shared memory)
  - Actors are **location independent:**
    - Names do not reflect their physical location.

Anatomy of an Actor

- Actor = state + behaviour + mailbox + autonomy

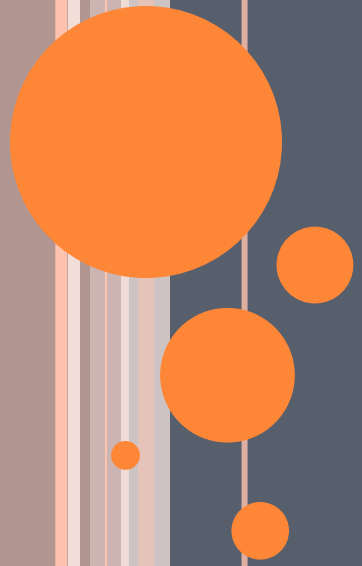


Actor Supervision

- Every actor created has a **supervisor** - the actor that created it
  - This means that actor systems are organised hierarchically.
  - Apart from the root actor, ever actor has exactly 1 supervisor and 0 or more children.
- Supervisors create actors to perform tasks either on their behalf (delegation) or in response to a larger system need.
- Supervisors play a key role in failure handling:
  - When an actor fails, it suspends all of its children and notifies its supervisor of the failure.
  - The supervisor chooses how to respond to the failure by either resuming the actor (and its children) or restarting the actor.
  - When restarted, the complete actor hierarchy rooted in the failed actor is destroyed.

Actor Languages / Libraries

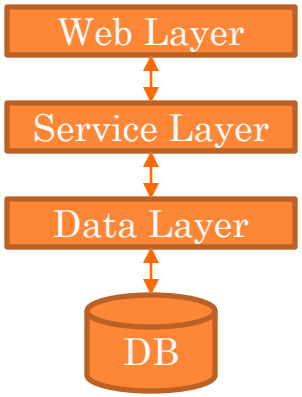
- Erlang (Telephone exchange programming)
- RevActor (Ruby)
- SALSA
- Kilim
- Reactors
- Akka (Scala and Java)
- Project Orleans (Microsoft Virtual Actors)
- …



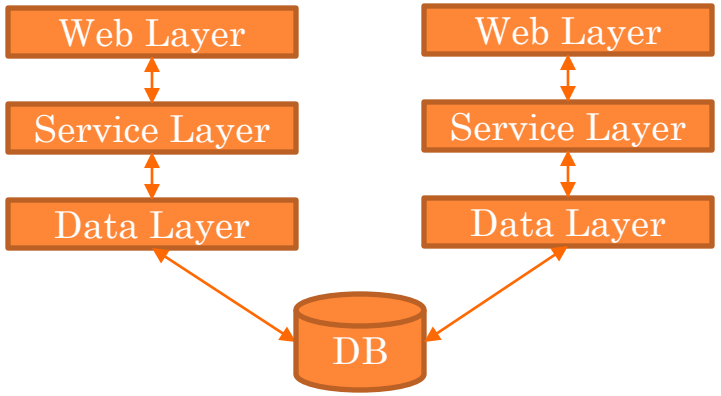
The Reactive Manifesto

Why Consider Actors?

- 3-tier architecture proven solution to web applications:
  - Data Layer: Code to access the database
  - Service Layer: Main business functions
  - Web Layer: Front end (views and controllers)



- Scaling can be achieved by replicating the 3 layers as necessary.
  - But this does not work for the DB…
    - Can be scaled somewhat using master-slave, clustering, …
    - Ultimately there is still a single master that must be updated sequentially…



THE TRADITIONAL APPROACH IS NO LONGER ADEQUATE FOR MODERN SYSTEMS

The Reactive Manifesto (http://www.reactivemanifesto.org/)

- Published 16th September, 2014 (> 20,000 signatures)
- Current approaches to building large-scale software systems is inadequate for the task at hand.
  - The last 10 years has seen a move from systems containing 10s of servers handling gigabytes of data to cloud-based systems that expand to consume 1,000s of servers handling petabytes of data.
- Increased scale makes manual failure monitoring and recovery impractical.
  - Need to develop systems that are designed to automatically detect and recover from component failures.
- The profusion of mobile devices and global nature of many web-based services has led to widely varying usage patterns that exhibit rapid changes in demand for services.
  - Systems need to be designed that can handle wild fluctuations in workload by (re-)allocating resources dynamically and seamlessly

The Reactive Manifesto (http://www.reactivemanifesto.org/)

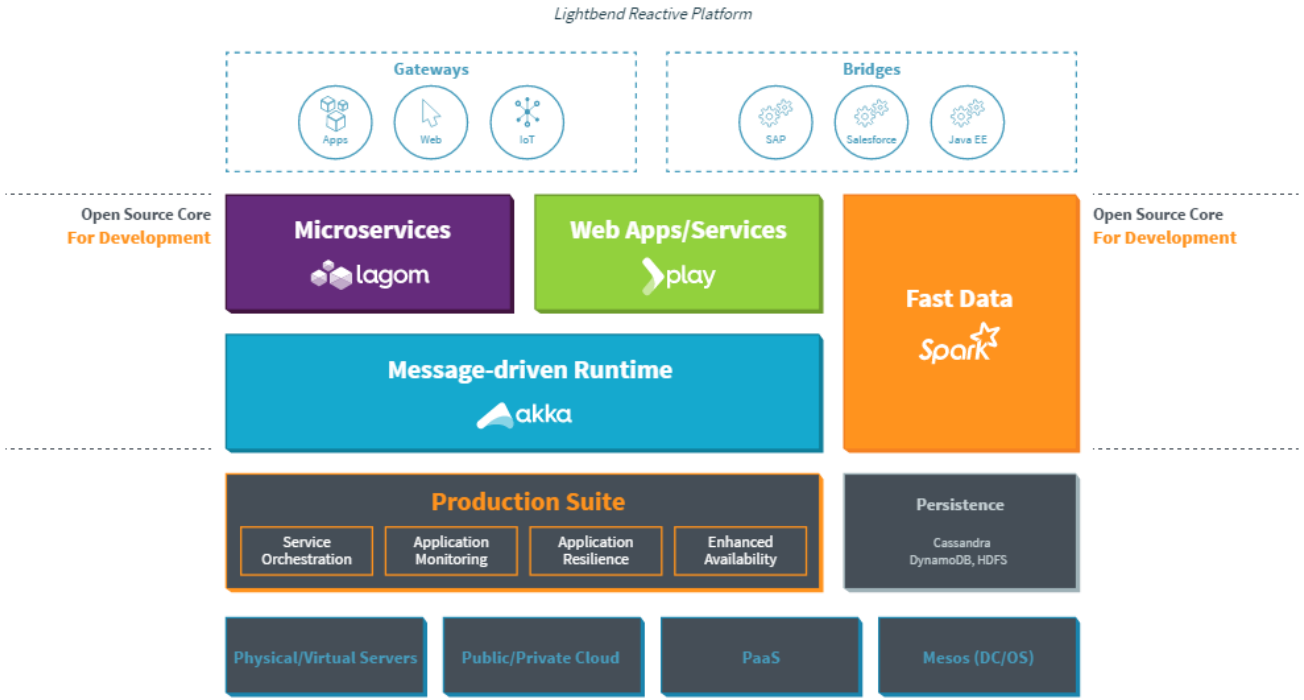
- The solution is to promote architectures that are:
  - **Responsive**: Systems should respond in a timely and consistent manner; where possible, providing guarantees on response time.
  - **Resilient**: Failure should be expected and handled transparently by systems while maintaining expected levels of responsiveness.
  - **Elastic**: Systems should remain responsive even as workload varies through dynamic allocation of resource as required.
  - **Message-driven**: Asynchronous message passing promotes loosely-coupled systems. Use of message-passing promotes load-management, elasticity, and flow control through monitoring and management of message queues.

Reactive (new Web) Technologies

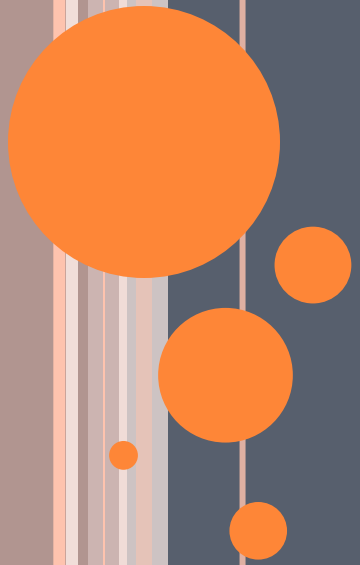
- Non-blocking IO (NIO):
  - Remove the need to block while waiting to read/write through the use of callbacks / event-queues (streams).
  - *Example instead of blocking, you insert a retry event set for some point in the future. This allows the system to continue to process other events in the meantime….*
- Reactive Programming (Rx)
  - Programming paradigm based on data flow (stream) processing.
  - Handlers perform operations on streams, such as consuming, merging, aggregating, filtering, creating, …
- Example: Netty (http://netty.io)
  - Non-blocking IO/Reactive Programming implementation for Java

Example: Lightbend Framework

- Typesafe's "New Web" Reactive Framework



https://www.lightbend.com/platform/development



Introducing Akka

Akka
(http://akka.io/)

- A toolkit and runtime for Actor Systems
  - Supported in both Java, Scala, and .NET.
  - Provided as a core library of Scala.
  - Part of the Lightbend framework.
  - Simple to use…
- Widely used in industry:
  - Some Case Studies: http://www.lightbend.com/resources/case-studies-and-stories
  - A good presentation on Akka and "the web": https://www.infoq.com/presentations/Akka-Actors
- Freely available and open-source…

Akka Concepts

- To create an actor in Java, you simply extend the `akka.actor.UntypedActor` class and implement the `onReceive(…)` method:

```
public class Hello extends UntypedActor {
    @Override
    public void onReceive(Object message) {
    }
}
```

- The `onReceive(…)` method is invoked whenever there is an unhandled message in the actors message queue and the actor is idle (not processing another message).

- It is important to note that messages are Java objects…

Hello Akka

- To create a "hello world" actor system, we need to do a couple of things:
  1. Create a Hello message that will be sent to the Hello actor:

```
public class HelloMessage {
}
```

  2. Write some code in the `onReceive(…)` method to handle the message and print out the text:

```
public class Hello extends UntypedActor {
    @Override
    public void onReceive(Object message) {
        if (message instanceof HelloMessage) {
            System.out.println("Hello World");
        }
    }
}
```

  3. Write some code to deploy our actor and send the message…

Hello Akka

3. Write some code to deploy our actor and send the message…

```
public class Main {
    public static void main(String[] args) {
        ActorSystem system =
            ActorSystem.create("ContentSystem");

        final ActorRef hello =
            system.actorOf(
                Props.create(Hello.class),
                "hello");

        hello.tell(new HelloMessage(), null);
    }
}
```

Hello Akka

3. Write some code to deploy our actor and send the message…

```
public class Main {
    public static void main(String[] args) {
        ActorSystem system =
            ActorSystem.create("ContentSystem");

        final ActorRef hello =
            system.actorOf(
                Props.create(Hello.class),
                "hello");

        hello.tell(new HelloMessage(), null);
    }
}
```

1. Create the Actor System

2. Create the "Hello" Actor

3. Send the initial message

Aside: Setting Up An Akka Project

- Download the Akka distribution from http://akka.io
- Create a new Java Project
- Add the following jar files from the akka distribution to your classpath (XXX does not need to be the same for each file):
  - akka-actor-XXX.jar
  - scala-library-XXX.jar
  - config-XXX.jar
- You are ready to go ☺

A Bigger Example

- Problem: Concurrent Summation of Values
  - Take a list of integers, break it into sub-lists that are distributed to actors who return the sum of the values in each sub-list.
- Solution:
  - Two actor types: Master (problem creator) and Worker (problem solver)
  - Master will receive and Init message defining:
    - How many random numbers
    - How many workers
  - Master will create the array of numbers and the workers
  - Master breaks the list into segments and passes one segment to each worker using a Problem message
  - Workers sum the values in the segment and return the result using a Result message.
  - Master combines the answers to give the final total

A Bigger Example: Messages

- Init Message:

```
public class Init {
    public int numbers;
    public int workers;

    public Init(int numbers, int workers) {
        this.numbers = numbers;
        this.workers = workers;
    }
}
```

- Problem Message:

```
public class Problem {
    public int[] list;
    public int start;
    public int end;

    public Problem(int[] list, int start, int end) {
        this.list = list;
        this.start = start;
        this.end = end;
    }
}
```

A Bigger Example: Messages

- Result Message:

```
public class Result {
    public int value;

    public Result(int value) {
        this.value = value;
    }
}
```

A Bigger Example: Worker

```
public class Worker extends UntypedActor {
    @Override
    public void onReceive(Object message) throws Throwable {
        if (message instanceof Problem) {
            Problem problem = (Problem) message;

            // Do the calculation
            int sum = 0;
            for (int i=problem.start; i<problem.end; i++) {
                sum += problem.list[i];
            }

            // Send the result
            getSender().tell(new Result(sum), getSelf());
            getSelf().tell(Kill.getInstance(), getSelf());
        }
    }
}
```

A Bigger Example: Master

```
public class Master extends UntypedActor {
    Random random = new Random();
    ActorRef[] workers = null;
    int count;
    int sum;
    long startTime;

    @Override
    public void onReceive(Object message) throws Throwable {
        if (message instanceof Init) {
            Init init = (Init) message;

            // Create the workers
            workers = new ActorRef[init.workers];
            for (int i=0; i<init.workers; i++) {
                workers[i] = getContext().actorOf(
                    Props.create(Worker.class), "worker_"+i);
            }
```

A Bigger Example: Master

```
// Create the list
int[] list = new int[init.numbers];
for (int i=0; i<init.numbers;i++) {
    list[i]=random.nextInt(100);
}

// Local Summation
int lsum = 0;
long st = System.currentTimeMillis();
for (int i=0; i<init.numbers;i++) {
    lsum+= list[i];
}
System.out.println("local result = " + lsum);
System.out.println("local time = " +
   (System.currentTimeMillis()-st) + "ms");

// Start the actor based approach
startTime = System.currentTimeMillis();
```

A Bigger Example: Master

```
            // Distribute the problems
            int segment = init.numbers / init.workers;
            for (int i=0; i<init.workers; i++) {
                workers[i].tell(
                    new Problem(list, i*segment, ((i+1)*segment)-1),
                    getSelf());
            }
            count = 0;sum = 0;
        } else if (message instanceof Result) {
            sum += ((Result) message).value; count++;
            if (count == workers.length) {
                System.out.println("result = " + sum);
                System.out.println("duration = " +
                   (System.currentTimeMillis()-startTime) +"ms");
                getSelf().tell(Kill.getInstance(), getSelf());
            }
        }
    }
}
```

What about Distribution?

- Akka supports a number of distribution models:
  - Remote: Linking of two akka runtimes based on TCP/IP.
    - Permits remote referencing of actor using ActorSelection class.
    - Permits remote creation of actors using Deploy class.
  - Apache Camel: Integration framework (supports SOAP, HTTP, JMS, …)
    - Akka integration allows you to easily create and deploy distributed actor systems.
  - Clustering: Decentralised peer-to-peer membership service.
    - Uses a gossip protocol to maintain membership information
    - Vector clocks for event ordering
    - Distributed failure monitoring used to monitor for failed runtimes