Concurrent and Distributed Systems

Talk Twelve

Actor Model of Concurrent Computations
Akka Toolkit

Maxim Mozgovoy

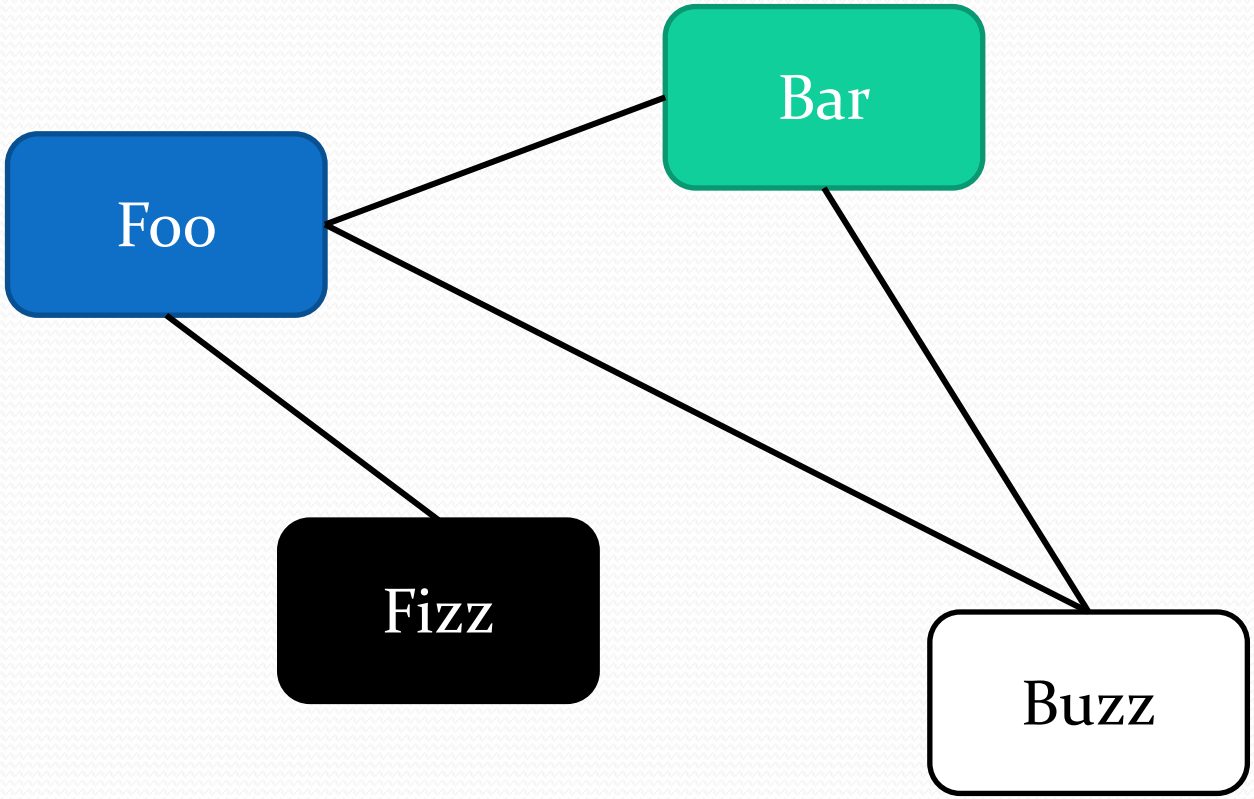
Actor Model

- Actor Model is yet another possible way to describe concurrent computations.
- It appeared in 1970s as a higher-level building block for concurrent systems (comparing to processes and threads).
- The basic idea is to represent a software system as an interaction of independent active objects (actors).

- Each actor lives in its own thread.
- Normally, an actor runs just one thread and thus has no concurrent code inside.
- Actors communicate by passing asynchronous messages to each other.

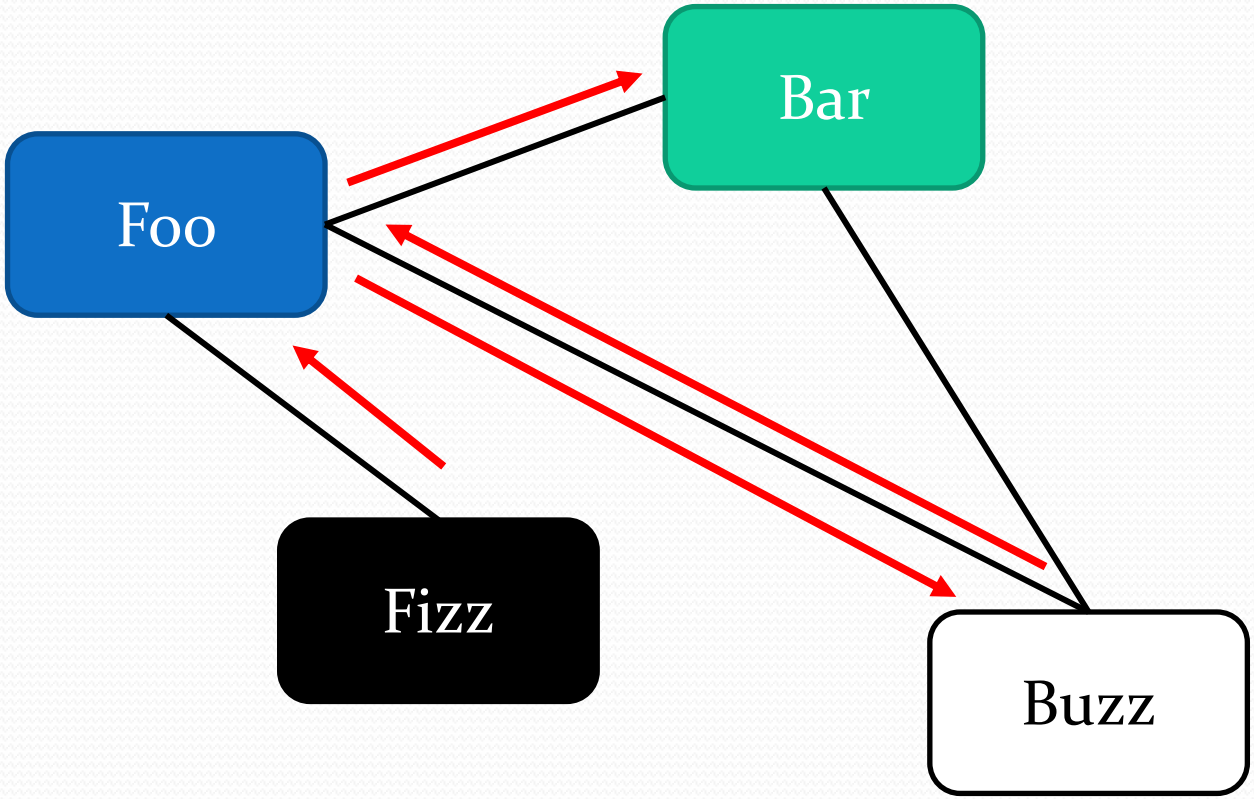
Actor Model: Motivation

Consider a typical multi-component system.
We can visualize the connections between its components:



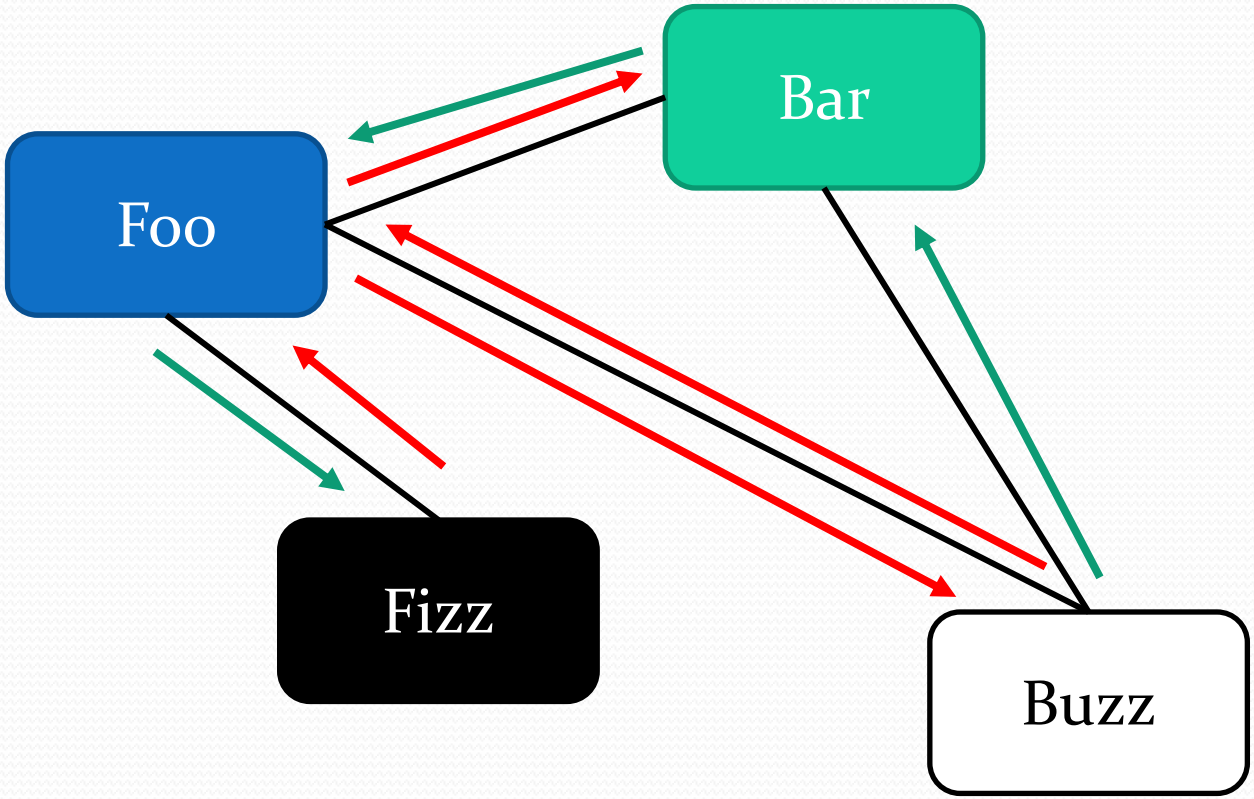
Actor Model: Motivation

This scheme does *not* show the actual control flow
(i.e, the runtime sequence of component invocation)



Actor Model: Motivation

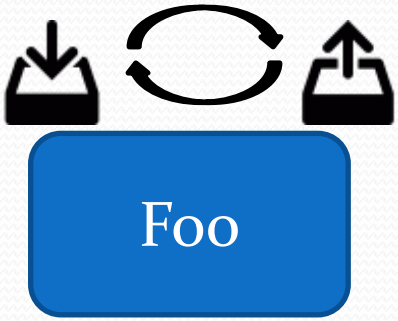
The situation becomes especially difficult to understand in case of *multiple* concurrent threads:

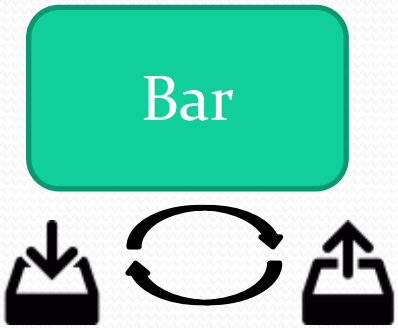


Actor Model: Motivation

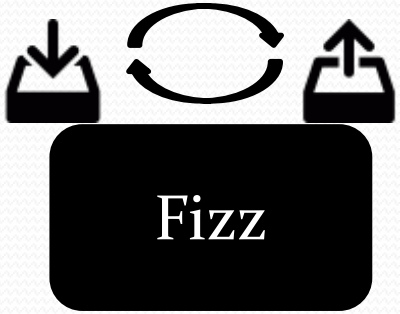
In Actor model, every component is assumed to operate as an independent thread that reacts only to event messages sent by other components asynchronously.

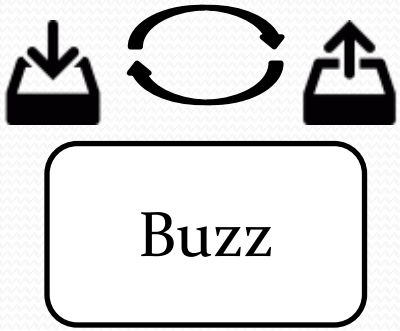
(Read mail ➔ Do something ➔ Repeat)





Events are generated by actors such as other components, system clock, input devices, etc.





Actor Model: Recap

Let's recap:

- We create a special actor object and specify what kinds of incoming messages it can handle and how.
- Event handling code is supposed to process a *single individual* incoming message (so there is no loop). Messages are not processed concurrently: an actor reads the next message only after handling the previous one.
- An actor can reside on a local or on a remote machine.
- An actor can send messages to other actors and optionally wait for message responses.
- Actors run in different threads.
  However, these threads are not "system threads", typically they are lightweight *coroutines*, so we run thousands of concurrent actors without performance penalties.

Some Notes

There are obvious parallels between actors and:

- threads, remote calls, executors, and fork/join operations.

Indeed, actors combine individual features of
the tools we considered previously.

- An actor is a thread-like object.
- Actors can be made remote / distributed.
- Thread creation and thread execution are automatic and invisible, just like in Java Executors.
- Actors can create other actors and process their responses, somewhat similarly to Fork/join framework.

Introducing Akka

To try actors, we will rely on Akka toolkit: https://akka.io

Akka is a modern production-level system, used in a variety of projects. It can be a bit hard to setup and use, but we will try some very simple examples, showing the basics.

Akka-based projects are easier to compile with Maven.
Take it from our Moodle, unzip into your home directory, and add its bin folder to your PATH:

MS Windows

```
SETX PATH "%PATH%";%USERPROFILE%\maven\bin
```

```
echo 'export PATH=$PATH:$HOME/maven/bin' >>
  ~/.bash_profile
```

Linux & Mac

Hello, World

Let's make sure you can compile and run the minimal example.

- Unzip `w12_helloAkka` sample from our Moodle.
- Go to the project folder and run
  `mvn compile`
- Maven will download some dependencies. It will take a while.
- Run the project with
  `mvn exec:java -Dexec.mainClass=w12_helloAkka`
- You should see some build info, followed by the line
  `Hello, world!`

On Project Structure

You can use the “Hello, world” example as a base for your code.

Maven build tool requires a predefined project structure:

```
<project_dir>/
    pom.xml                 – project configuration file
    src/
        main/
            java/
                *.java  – project source files
```

So you have to place your code under `/src/main/java` subdir.

On Building and Running

To build a Maven project, you should be inside the directory, where `pom.xml` is located. (You will not have to modify it).

To compile a project, use: `mvn compile`

To run a project, use:

```
mvn exec:java -Dexec.mainClass=<YourMainClassName>
```

You can also pass additional arguments to your code:

```
mvn exec ... -Dexec.args="arg1 arg2 arg3"
```

Inside "Hello, World"

Let's discuss the structure of our first program.

```
// here we define a custom actor type
class MyActor extends AbstractActor { ... }

public static void main(String[] args) {
    // initialize actor subsystem (one per machine)
    ActorSystem sys = ActorSystem.create();

    // create an actor of type MyActor
    ActorRef actor = sys.actorOf(Props.create(MyActor.class));

    // send it one textual and one integer-based message
    actor.tell("Hello, world!", null);
    actor.tell(0, null);
}
```

Actor System

Every Akka application has to instantiate one `ActorSystem` object, serving as a "root" of a tree of Actor objects.

Actors form a tree because every actor has a parent and can create new child actors.

ActorSystem can have an optional name, useful in distributed applications (so we can refer to remote systems by name).

You can terminate the ActorSystem to shutdown Akka app.

```
import akka.actor.ActorSystem;
ActorSystem sys = ActorSystem.create();
...
ActorSystem sys = ActorSystem.create("MySystem");
sys.terminate();
```

Creating Actors (1)

We will use a so-called "classic actors API", which is simpler.

An actor is an object derived from `AbstractActor`. To create an actor, specify its properties with `Props.create()`, then instantiate the object with `actorOf()`:

```
// (AbstractActor, Props, ActorRef, ActorSystem from akka.actor)
class MyActor extends AbstractActor { ... }
ActorSystem sys = ActorSystem.create();

// create a top-level actor (child of ActorSystem)
ActorRef actor = sys.actorOf(Props.create(MyActor.class));

// pass constructor arguments to MyActor if necessary
ActorRef actor =
  sys.actorOf(Props.create(MyActor.class, "argument1", 5, 3.7));
```

Creating Actors (2)

```
// Actors can have names (useful in distributed apps)
ActorRef actor = sys.actorOf(Props.create(...), "ActorName");

// to create child actors inside other actors, use getContext()
// use getContext().system() to access the ActorSystem object
ActorRef actor =
    getContext().actorOf(Props.create(MyActor.class));
```

Let's recap:

- You can create actors of any class derived from AbstractActor.
- You can pass any necessary arguments to its constructor.
- You can give names to actors.
- You can create them in the system namespace or as children of other actors (if an actor is shut down, it will also shut down its children automatically).

Actor Anatomy

A minimal actor must at least override the function

```
public Receive createReceive()
```

Its purpose is to bind possible incoming events to event handlers.

Its body normally follows the template structure below:

```
return receiveBuilder()
  .match...(args1)
  .match...(args2)
   ...
  .build();
```

The system tries to find a suitable handler for the next event in these `match…()` clauses and calls a handler if it is found ("handler not found" is an error)

Each `match…()` function is responsible for one message type or message value.

Match…() functions

The most common match…( ) functions include:

```
// calls handler if the next message has the given type
match(type, handler)

// calls handler if the next message has the given value
matchEquals(value, handler)

// calls handler if none of the previous match…() functions
// was able to find a match
matchAny(handler)

// a handler should be declared as
void handler_name(MessageType msg)

// or as a lambda function
```

Binding Handlers (1)

E.g., In our "Hello, world" example we bind `print_line()` to any message of type `String`:

```
return receiveBuilder()
.match(java.lang.String.class, this::print_line)
.matchAny(this::shutdown)
.build();


private void print_line(String msg)
{
    System.out.println(msg);
}
```

Binding Handlers (2)

Lambda functions are unnamed function objects that can be bound right inside `createReceive()`.

We can rewrite event handling in "Hello, world!" as follows:

```
return receiveBuilder()
.match(java.lang.String.class, msg -> {
    System.out.println(msg);
})
.matchAny(msg -> {
    getContext().system().terminate();
})
.build();
```

Sending Asynchronous Messages

The basic message-sending method in Akka is `tell()`:

```
target_actor.tell(message, sender)
```

It works asynchronously: the sender resumes its work immediately without waiting for an answer.

An actor can identify the current message sender by calling `getSender()`. An actor can get a reference to itself by calling `getSelf()`:

```
// say "thanks" to the sender of the last message
getSender().tell("thanks", getSelf());
```

If a sender object is not needed, you can pass `null`.

Example: The Game of Pig (w12_akkaPig)

Pig is a simple dice game for two players.

Players take turns as follows.

1. The player sets his "table" score to zero.
2. The player rolls a die. If "one" is rolled, he passes the turn to an opponent (who starts from step 1).
3. Otherwise, the player adds die points to his "table" score.
4. Next, the player must decide either to stop or to continue.
5. If he decides to stop, he adds his "table" score to his "final" score and passes the turn to an opponent (starts with step 1).
6. If he decides to continue, he returns to step 2.

The first player to score 100 "final" points wins.

Distributed Actors

Now, suppose we want to make our actors distributed. It's easy:

- Split the code into independent running modules.
- Provide names for actors and `ActorSystem` objects.
- Make sure your message objects are serializable. Akka uses its own serialization mechanism, but the built-in Java approach is also supported.
- Add init/shutdown code to each module.
- Use `actorSelection()` to get a reference to a remote actor.
- Provide an app configuration file specifying network settings (such as IP addresses and ports). You can provide one file per module or use a shared config with overridable settings.

Example: Online Pig (1)

Let's start with a configuration file. By default, it should be named `src/main/resources/application.conf`

```
akka {
actor { provider = remote # use remote actors
        allow-java-serialization = on     }
  remote {
    artery {
      transport = tcp
      canonical.hostname = "127.0.0.1" # Actor IP
      canonical.port = 25519           # Actor port
    }
  }
}
```

Config options are flexible; let's use the simplest approach for now.

Example: Online Pig (2)

```
// PigReferee.java
var sys = ActorSystem.create("PigReferee");
```

Akka uses `application.conf` settings automatically!

(it calls `ConfigFactory.load()` function)

The Referee actor will be instantiated at 127.0.0.1:25519

To override these settings, provide your values using `ConfigFactory.parseString()`, then supply the missing data with `withFallback()`:

```
import com.typesafe.config.ConfigFactory;
...
var portCfg = ConfigFactory.parseString("key=value");
var config = portCfg.withFallback(ConfigFactory.load());
```

Example: Online Pig (3)

Let's create actors like `PigPlayer0` and `PigPlayer1` on a custom port:

```
// PigPlayer.java
import com.typesafe.config.ConfigFactory;
public static void main(String[] args)
{
    var name = "Player" + args[0]; // 0 or 1
    var port = args[1];
    var portCfg = ConfigFactory.parseString
            ("akka.remote.artery.canonical.port=" + port);
    var config = portCfg.withFallback(ConfigFactory.load());

    var sys = ActorSystem.create("Pig" + name, config);
    var player = sys.actorOf(Props.create(Player.class), name);
}
```

Example: Online Pig (4)

Final changes:

```
// use ActorSelection in PigReferee.java:
ActorSelection[] players = new ActorSelection[2];
...
players[0] = getContext().actorSelection
             ("akka://PigPlayer0@127.0.0.1:25520/user/Player0");
players[1] = getContext().actorSelection
             ("akka://PigPlayer1@127.0.0.1:25521/user/Player1");
```

Example: Online Pig (5)

Final changes:

```
// ask players to shut down in PigReferee.java:
if(score[current] >= 100)
{
    System.out.println("Winner: " + current);
    players[0].tell("terminate", getSelf());
    players[1].tell("terminate", getSelf());
    getContext().system().terminate();
}
```

Example: Online Pig (6)

Final changes:

```
// Shutdown if "terminate" message is received in PigPlayer.java
return receiveBuilder()
.matchEquals("go", this::go)
.matchEquals("terminate", p -> {
    getContext().system().terminate();
})
.build();
```

Now we can run both players and the server:

```
mvn exec:java -Dexec.mainClass=PigPlayer -Dexec.args="0 25520"
mvn exec:java -Dexec.mainClass=PigPlayer -Dexec.args="1 25521"
mvn exec:java -Dexec.mainClass=PigReferee
```

(see w12_onlinePig)

Using Worker Actors

Akka can also be used in classic parallel computing tasks.

However, in this case it is often necessary to sync actors
(i.e., to allocate tasks, wait for the results and process them).

Thus, pure asynchronous messaging might be not enough.

A common way to organize worker actors
is to use the pipe-ask pattern.

ask() function (1)

The key piece for synchronizing actors is ask() function.

It sends a message to the given actor just like tell().

Unlike tell(), ask() returns a Future object
(we met Futures while discussing Java Executors).

A Future object can be used to check task progress
or simply to wait until the task is done.

In Akka, each task should be finished within a user-specified time interval (otherwise a timeout exception is thrown).

ask() function (2)

To call ask(), use the following syntax:

```
import akka.pattern.Patterns;
import java.time.Duration;
...
// send "hello" to myActor; should get reply within 2 sec
var myFuture = Patterns.ask(myActor, "hello",
                Duration.ofSeconds(2)).toCompletableFuture();
```

Here `myFuture` is `java.util.concurrent.CompletableFuture`
(you can read about it in Java documentation)

Futures Interface (1)

We can do various things with the Future object,
e.g., we wait for it to finish and get the task result:

```
var myResult = myFuture.join()  // or myFuture.get()
```

In Akka, a task is considered complete when the worker actor sends back a message to the master using tell().

```
getSender().tell(reply_message, getSelf());
```

Futures Interface (2)

The interface of Future class is quite rich.

One common combination of methods is `allOf()`... `thenApply()`.

It allows to wait until all given tasks are finished, and then apply the given function to the results obtained:

```
var f1 = Patterns.ask(...); // assume that task results
var f2 = Patterns.ask(...); // are Integer values

// calculate the sum
var myFuture = CompletableFuture.allOf(f1, f2).thenApply(v -> {
     return (Integer)f1.join() + (Integer)f2.join(); });
```

pipe() function (1)

The most common master-worker interaction scenario is to gather some results from workers, then send the aggregated answer to the master.

It is easy to synchronize actors earlier than necessary (which means forcing “waiting for operation to complete”)

An easy way to make sure the operations are performed asynchronously, and actors synchronize only to obtain the aggregated answer is to use pipe() function, specifically designed for this purpose.

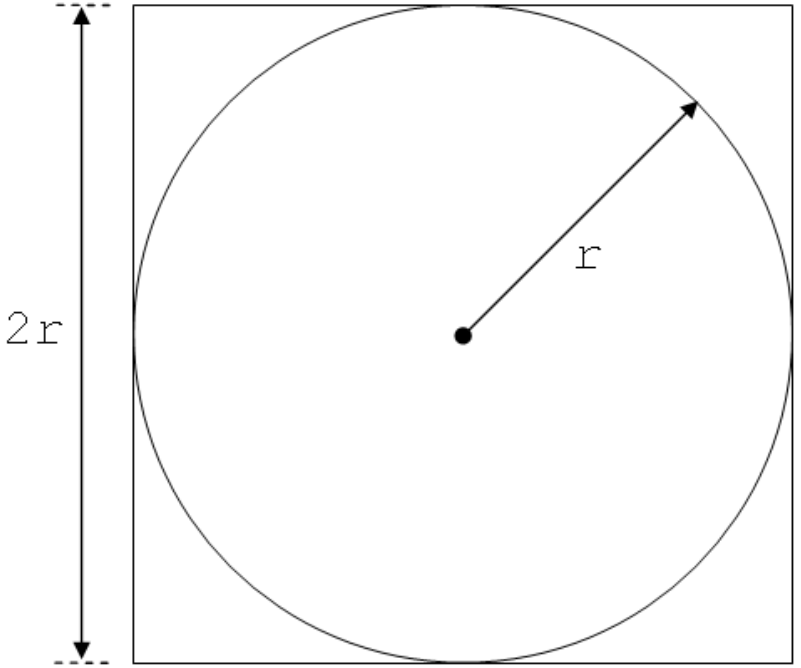
pipe() function (2)

```
import akka.pattern.Patterns;
...
var f1 = Patterns.ask(...); // assume that task results
var f2 = Patterns.ask(...); // are Integer values

var myFuture = CompletableFuture.allOf(f1, f2).thenApply(v -> {
     return (Integer)f1.join() + (Integer)f2.join(); });

// when myFuture is ready, send it value to targetActor
Patterns.pipe(myFuture,
    getContext().system().dispatcher()).to(targetActor);
```

Example: Finding the Value of Pi (1)

The value of Pi (3.14…) can be found with a "dartboard method":
Throw N darts into random places of the square board of width `2r`.

Then the value of Pi is approximately `4*N0 / N`,
where `N0` is the number of darts happened to be inside the circle.



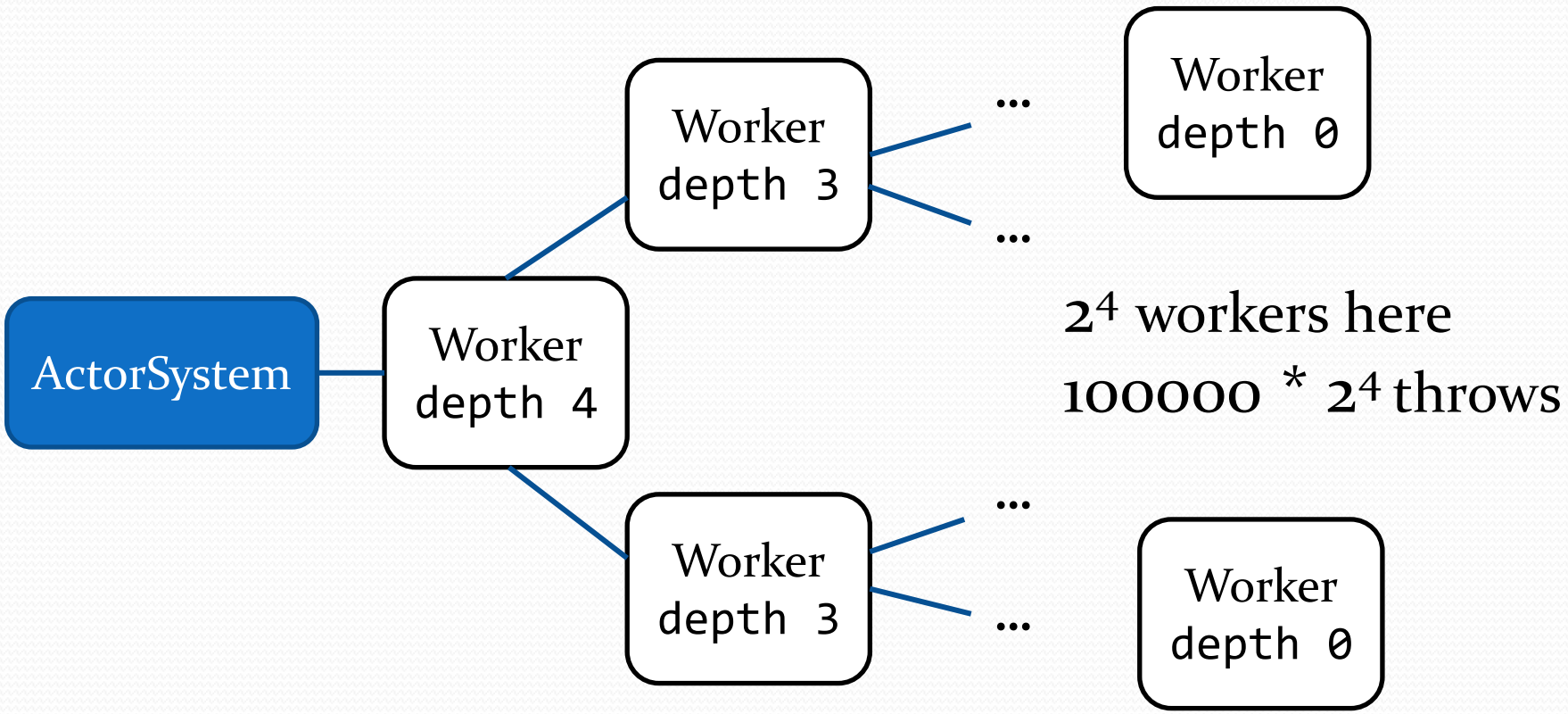
Example: Finding the Value of Pi (2)

The example solution (w12_akkaFindPi) works as follows.

- ActorSystem creates a worker actor with `depth = 4` and sends it a "go" message.
- A worker actor with `depth = 0` calculates the value of `N0` using 100000 dart throws and returns it to the caller.
- A worker actor with `depth > 0` creates two worker actors with `depth = depth – 1`, combines their results and sends back.
- The very first actor created by ActorSystem eventually gets the combined results from all other worker actors. It calculates the value of Pi and prints it on the screen.

Example: Finding the Value of Pi (3)

The resulting solution is conceptually similar to the "divide and conquer" approach of Fork/Join framework.



Conclusions

- Actor model suggests implementing a concurrent system as a network of communicating active objects (actors).
- Each actor lives in its own lightweight thread (a coroutine) and reacts to incoming messages by doing some work and sending messages to other actors.
- Messages are asynchronous: once a message is sent, the actor can continue its operation.
- Actors can be instantiated locally on operate over the net.
- Probably, the most popular actor model implementation for Java and Scala languages is Akka toolkit.
- Akka can cooperate with the standard concurrency primitives of Java (such as Futures).