1. ***What is Akka ?***

**Akka** is a toolkit (a set of open-source libraries), used to implement applications on .NET framework. Using Akka framework, these applications will be:

* ***Scalable*** (the system has the ability to process data regardless of whether the system gets an increased workload),
* ***Fault-tolerant*** (the system handles errors without crashing),
* ***Distributed systems*** (systems running on several different machines / servers and communicating over a network).

**Akka** framework allows us to focus on meeting business needs (instead of writing low-level code) and to provide reliable behavior, fault tolerance, and high performance.

Also, **Akka.NET** provides:

* Multi-threaded behavior (we will solve the major problems related to memory visibility)
* Transparent remote communication between systems and their components (It will not be necessary to write or maintain difficult networking code).
* A clustered, high-availability architecture that is elastic, scales in or out, on demand.

1. ***Why would you like to use Akka?***

As IT activities spreads out in society, the amount of data sent back and forth also increases. At the same time, users constantly require shorter charge and response times. Also, IT systems tend to grow over time. This can result in dull systems, and these systems could have a strong and negative impact of user experience.

Building fast systems with good performance are therefore of great importance for the user experience and the demands for these kinds of systems is today higher.

This is the reason why Akka was designed, in order to meet all these requirements:

* With Akka, we can create systems that handle large amounts of data efficiently and which can handle errors that occur, without the entire system crashing.
* Also, Akka.NET exploits the actor model to provide a level of abstraction that makes it easier to write correct concurrent, parallel and distributed systems.

The actor model spans the set of Akka.NET libraries, providing us with a consistent way of understanding and using them. Thus, Akka.NET offers a depth of integration that we cannot achieve by picking libraries to solve individual problems and trying to piece them together.

* By learning Akka.NET and its actor model, we will gain access to a vast set of tools that solve difficult distributed systems problems in a uniform programming model where everything fits together tightly and efficiently.

Implementing Akka framework, we'll succeed:

- to manage asynchronous execution (queues up and the system can continue),

- to create a complex system easier to understand,

- to design systems that can work on multiple processor cores,

- to build systems that can execute on multiple physical computers / servers,

- to manage errors and to create a "self-healing" system too.

1. ***What can you use Akka for? Describe and give examples***

Akka.NET is suitable for systems which use high data throughput and short upload times such as:

* banking systems (where a larger amounts of data need to be handled simultaneously)
* trading systems,
* social media,
* simulations,
* games and bets systems,
* traffic analysis systems,
* healthcare systems,
* systems that work with data analysis.

All of these examples can be classified into several major categories:

* 1. **Intensive transactions :** Banking and finance, Statistics and accounting, Betting, game, Social media, Telecom
  2. **Batch:** Batch / bulk job - data sync between different environments
  3. **General systems:** Representational State Transfer for creating API’s, System Integration
  4. **Communication platforms:** Chat Application and another different types of real-time applications
  5. **Multiplayer games:** Manage players and the interaction between them
  6. **Calculations processes:** Business Intelligence, Data mining (searching through large data sets)
  7. **Traffic monitoring systems:** Traffic flow, Positioning / navigation
  8. **Internet of things:** Manage input from connected devices

***4. What is an Akka Actor ?***

Actors is a type of object used in Akka framework (it could be compared it with an object used in OOP). But unlike conventional objects that communicate via method calls, Actors communicates via various types of messages.

Actors are the most important components (building blocks) of the Akka system, where they perform small and well-defined tasks. All large tasks are broken down into many small ones.

An Actor:

- has an address (if we have no direct reference to an instance, we can always reach it via his address)

- should be immutable - so it cannot be changed after it has been created

- looks the same as one that is distributed

- Join in a hierarchy so that errors can be isolated and not take down the entire application

- can communicate with another Actor through messages. This means that the different parts are loosely connected and add stability to the system

- is lazy (it will do nothing if they do not receive messages)

- is small and efficient (leave small traces, about 300 bytes per instance in the normal case).

Actor

1. Receive and respond to messages

4. Change their behavior when it receives the next message

3. Send messages to other Actors

2. Create another Actors

The most important Actor's features are:

Actor

Inbox / Inkorg

All Messages come here to be considered.

The actor handles a message and does this according to the FIFO \* principle

Behavior / Beteende BeteendeBeteende

Receive and manage an incoming message

State / Tillstånd

The properties that currently define the Actor

An Actor may have children, who are also Actors.

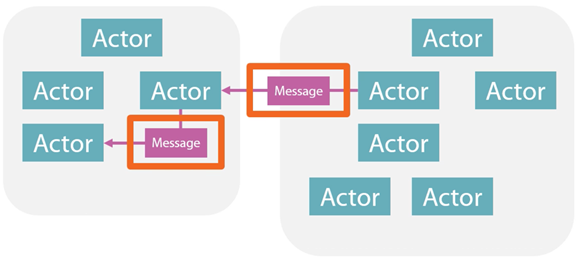
Child / Barn

Monitoring strategy /

Övervakningsstrategi

A strategy for managing children who have slipped.

Also, an Actor System contains more Actors that send messages between themselves (local actors). Actors can also send messages to other systems (remote actors).



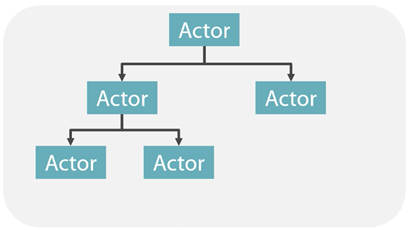
A remote message

A local message

En Actor - instances

Another Actor instance

An Actor can have children and is responsible for these. If it will occur a fault in a child, this fault will be flagged for the parent who is responsible to take a decision.

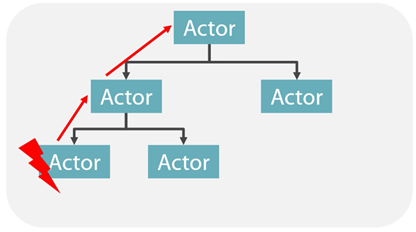


Parent

Childn

Actor System - Instance

Self-healing system - Actors are responsible for their Child-Actors and handle any errors that arise (with different error strategies).



Parent

der

Childn

rn

Actor System - Instance

Fault tolerance means that Actors themselves keep track of errors that occur in the system and try to solve them themselves.

This becomes possible through a hierarchical structure in which each parent (parent) Actor restarts or stops his child (child) Actor if a fault occurs.

If a restart or a stop does not resolve the error, the error is passed on to the next parent one step higher in the hierarchy.

This enables the creation of systems that can to a large extent handle errors without crashing and which through process restarts gives more attempts to perform a task.

1. ***What is an Akka message ?***

A single Actor cannot build a system, so to make things happen Actors communicates with each other.

For their help, they have Messages. To initiate a flow, process or execution, these messages are sent back and forth, and constitute the ecosystem of an Akka system.

A message:

* can contain an instruction that tells how an actor should perform a task
* is Immutable – it cannot change when it is created
* that is already sent, is placed in a queue in the receiving Actor's Mailbox and is waiting for it to be dealt with in turn.

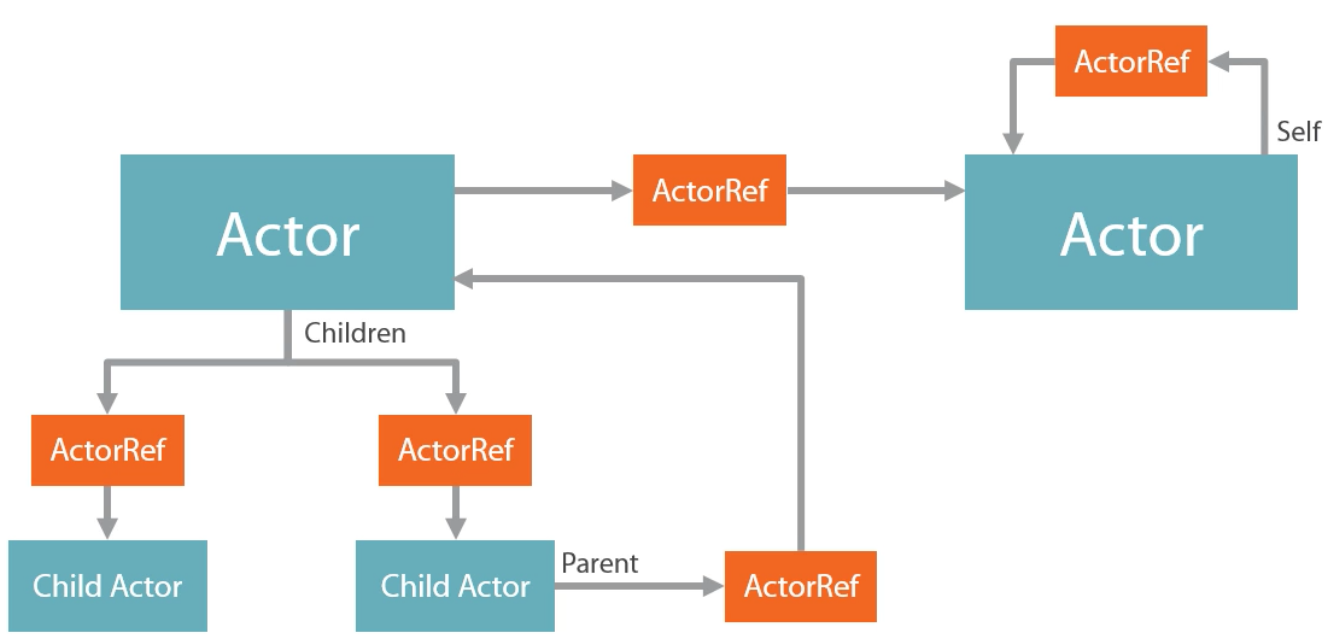
Also:

* Actors can change ways in which they respond to a message
* An Actor sends a message asynchronously

It is important to underline that Actors send messages to Actor References, not directly to Actors. So, Actors do not communicate directly with each other but instead via an ActorRef.

An ActorRef is a reference to an Akka class which in turn has a reference to the actual Actor.

Through the reference, Actor's message sends to each other, but also to himself.





There are two ways to create a reference to our Actor:

* When we create a new Actor via ActorOf ().
* Alternatively via a so-called lookup, an ActorSelection ()

Also, messages can be sent in two different ways:

- fire and forget (without response): Tell (),

- send and receive (an answer is awaited from the recipient): Ask ()

Actor

Actor

Caller

Caller

\_\_\_ \_\_ \_  
\_\_ \_\_ \_\_  
\_\_

\_\_\_ \_\_ \_  
\_\_ \_\_ \_\_  
\_\_

\_\_\_ \_\_ \_  
\_\_ \_\_ \_\_  
\_\_

Tell()

Ask()

Which are the differences between Tell() and Ask() ?

|  |  |
| --- | --- |
| **Tell** | **Ask** |
| Tells something to the host Actor  Expect no answer  Fire and forget  Don't wait for an answer  Better scaling and parallelism  More disconnected systems  Use by default | Ask another actor something  Expecting an answer  Receiving Actor must respond  Timeout (Task Cancelled Exception)  Inferior scaling and parallelism  More dependencies between Actors and systems  Just use if you have to |

How we can send a message ? We used the reference to send a few messages via Tell ().

playbackActorRef.Tell("Karate kid");

playbackActorRef.Tell(new PlayMovieMessage("Die Hard", 11));

We can instead create a so-called Actor Selection and use it to send messages to the Actor or those who match a selection that we define (Actor Selection).

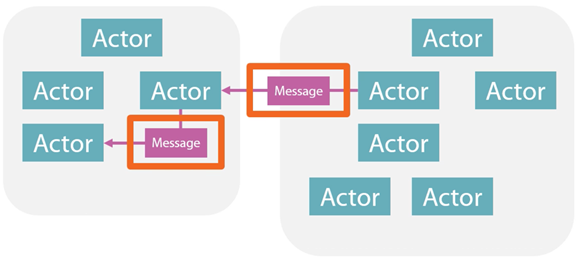


var message = new PlayMovieMessage(movieTitle, userId); actorSystem.ActorSelection("/user/Playback/UserCoordinator").Tell(message);

1. ***What is an Akka.Remote ?***

Akka.NET uses the "Home Depot" extensibility model. If the base Akka NuGet package provides all of the capabilities that you need to create actors, the Akka.NET project ships dozens of additional modules which take the capabilities of Akka.NET and extend them to do new things!

Akka.Remote is the most powerful of all of these additional packages, as it is what brings the capability to build an ActorSystem across multiple processes over a computer network. This means that we can execute the same code regardless of whether it is locally installed or installed on a computer elsewhere in the network. So, actors can also send messages to other systems (remote actors).



A remote message

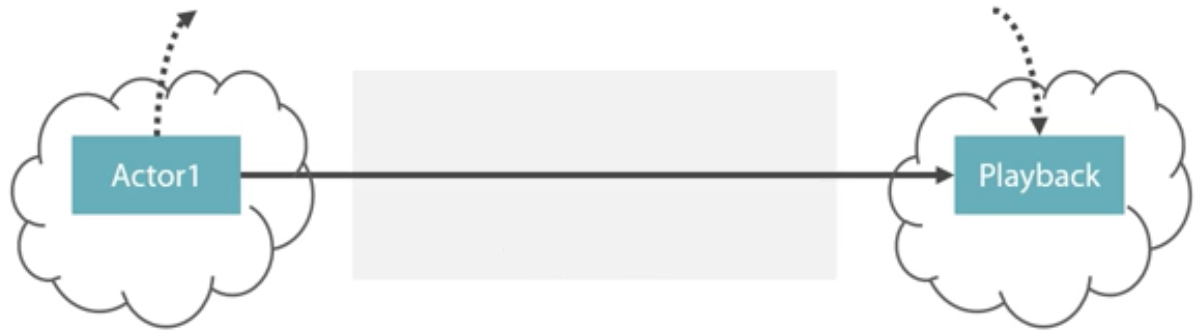
An Actor instance

A local message

Another Actor instance

In order to better understand, we will design an Actor System, also a Console Application. This system and the first one will then communicate with each other and help to execute our streaming service. Finally (via HCON), can configure our two systems so that they work together.

In the image below we have two Akka systems that execute different processes (locally or remotely), but using the same principle. If Actor1 wants to communicate with the Playback Actor which is located elsewhere, it sends a message as usual. The message is serialized, converted to a data stream, and sent using the TCP to the Playback actor. Then, the message is deserialized on the receiving side.



Transportprotokoll (TCP)

*akka.tcp://MovieStreamingActorSystem@172.20.10.3:9191/user/Playback*

This is the server part of our Akka system and it Known IP address and port number

This is our client system that will initiate the communication with the server.

In addition to sending messages between each other, we can get Akka systems to both install and start an Actor on another system.

An Akka system uses its configuration to find out where it can send its request in order to create a new Actor instance. Everything that the Actor needs to know about the remote system is defined there.



**Distributions-konfiguration**

**Actor rreferens**

**Create-Actor request**

When an instance of the Actor is created on the remote system, the reference will be handed over to the person who initiated the distribution.

-----------------------------------------------------------.----------------------------------------------------------------------------

Akka.Remote introduces the following ***capabilities to Akka.NET applications***:

* ***Location transparency with RemoteActorRef*** - write code that looks like it's communicating with local actors, but with just a few configuration settings your actors can begin communicating with actors hosted in remote processes in a way that's fully location transparent to your code.
* ***Remote addressing*** - Akka.Remote extends the Address and ActorPath components of Akka.NET to also now include information about how to connect to remote processes via ActorSelection.
* ***Remote messaging*** - send messages, transparently, to actors running in remote ActorSystems elsewhere on the network.
* ***Remote deployment*** - remotely deploy actors via the ActorOf method onto remote ActorSystem instances, anywhere on the network! The location of your actors on the network becomes a deployment detail in Akka.Remote.
* ***Multiple network transports*** - out of the box Akka.Remote ships with support for TCP, but has the ability to plugin third party transports and active multiple of them at the same time.

2) Define short and relevant, following concepts:

* 1. ***Concurrency*** means that two or more tasks are making progress even though they might not be executing simultaneously. This can for example be realized with time slicing where parts of tasks are executed sequentially and mixed with parts of other tasks.



For example, Akka's actor-based structure facilitates transaction-based processes, such as banks where larger amounts of data need to be handled simultaneously / concurrently - concurrency.

* 1. ***Parallelism*** arises when the execution can be made simultaneous.

This is difference between concurrency and parallelism concepts.



* 1. ***Asynchronous*** call allows the caller to progress after a finite number of steps, and the completion of the method may be signaled via some additional mechanism (it might be a registered callback, a Future, a Task or a message).

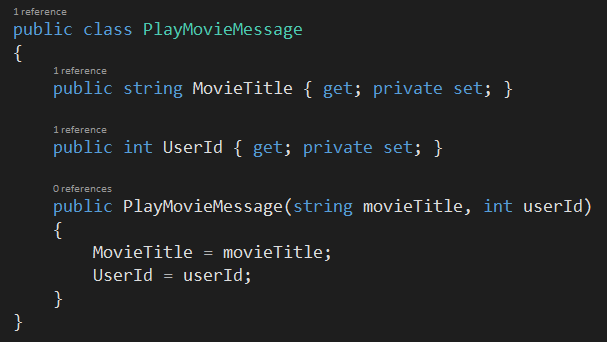
Also, I have to mention that Asynchronous APIs are preferred, as they guarantee that the system is able to progress. It is well known that actors are asynchronous by nature and an actor can progress after a message send without waiting for the actual delivery to happen.

* 1. ***Synchronous:*** in general, a method call is considered to be synchronous if the caller cannot make progress until the method returns a value or throws an exception. A synchronous API may use blocking to implement synchrony, but this is not a necessity. A very CPU intensive task might give a similar behavior as blocking.
  2. ***Non-blocking*** means that no thread is able to indefinitely delay others.

Non-blocking operations are preferred to blocking ones, as the overall progress of the system is not trivially guaranteed when it contains blocking operations.

* 1. ***Blocking of threads*** means the delay of one thread can indefinitely delay some of the other threads. A good example is a resource which can be used exclusively by one thread using mutual exclusion. In this way, if a thread holds on to the resource indefinitely (for example accidentally running an infinite loop) other threads waiting on the resource cannot progress.
  2. ***Immutability/immutable*** means that an Actor cannot be changed after it has been created. Also, the messages cannot be changed when they are created

By example, when we declare some properties, note that they are write-protected (immutable).



* 1. ***Location Transparency*** means is that whenever you send a message to an actor, you don’t need to know where they are within an actor system, which might span hundreds of computers. You just have to know that actors’ address.”

I have to mention that a local application can be split into distributed components that can be scaled independently. This can be achieved by changing only startup code, configuration and deployment, but leaving the main code untouched.

Also, everything in Akka is designed to work in a distributed setting: all interactions of actors use purely message passing and everything is asynchronous. This effort has been undertaken to ensure that all functions are available equally when running within a single JVM or on a cluster of hundreds of machines. The key for enabling this is to go from remote to local by way of optimization instead of trying to go from local to remote by way of generalization.

* 1. ***Deadlock*** arises when several participants are waiting on each other to reach a specific state to be able to progress. As none of them can progress without some other participant to reach a certain state all affected subsystems stall. Deadlock is closely related to blocking, as it is necessary that a participant thread be able to delay the progression of other threads indefinitely.
  2. ***Race Condition*** appears when an assumption about the ordering of a set of events might be violated by external non-deterministic effects. Race conditions often arise when multiple threads have a shared mutable state, and the operations of thread on the state might be interleaved causing unexpected behavior. While this is a common case, shared state is not necessary to have race conditions.

One example could be a client sending unordered packets (e.g. UDP datagrams) P1, P2 to a server. As the packets might potentially travel via different network routes, it is possible that the server receives P2 first and P1 afterwards. If the messages contain no information about their sending order it is impossible to determine by the server that they were sent in a different order. Depending on the meaning of the packets this can cause race conditions.

* 1. **Dependency Injection** means that if an Actor has a constructor that takes parameters then those need to be part of the Props as well.

But there are cases when a factory method must be used, for example when the actual constructor arguments are determined by a dependency injection framework.

// Create your DI container of preference

var someContainer = ... ;

// Create the actor system

var system = ActorSystem.Create("MySystem");

// Create the dependency resolver for the actor system

IDependencyResolver resolver = new XyzDependencyResolver(someContainer, system);

Also, when creating actorRefs straight off your ActorSystem instance, you can use the DI() Extension.

// Create the Props using the DI extension on your ActorSystem instance

var worker1Ref = system.ActorOf(system.DI().Props<TypedWorker>(), "Worker1");

var worker2Ref = system.ActorOf(system.DI().Props<TypedWorker>(), "Worker2");

When you want to create child actors from within your existing actors using Dependency Injection you can use the Actor Content extension just like in the following example.

protected override void PreStart()

{

var actorProps = Context.DI().Props<MyActor>()

.WithRouter(/\* options here \*/);

var myActorRef = Context.ActorOf(actorProps, "myChildActor");

}