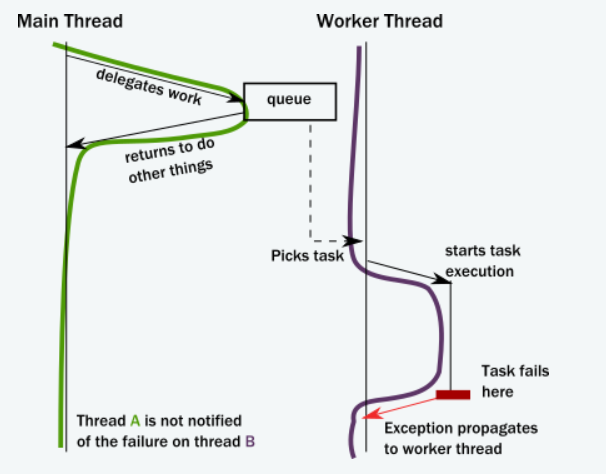
**Akka Tutorial**

Akka provides:

* Multi-threaded behavior without the use of low-level concurrency constructs like atomics or locks — relieving you from even thinking about memory visibility issues.
* Transparent remote communication between systems and their components — relieving you from writing and maintaining difficult networking code.
* A clustered, high-availability architecture that is elastic, scales in or out, on demand — enabling you to deliver a truly reactive system.

**Why concurrency with multithread fails ?**

* Objects can only guarantee encapsulation (protection of invariants) in the face of single-threaded access, multi-thread execution almost always leads to corrupted internal state. Every invariant can be violated by having two contending threads in the same code segment.
* While locks seem to be the natural remedy to uphold encapsulation with multiple threads, in practice they are inefficient and easily lead to deadlocks in any application of real-world scale.
* Locks work locally, attempts to make them distributed exist, but offer limited potential for scaling out.



The “main” thread needs to be notified somehow, but there is no call stack to unwind with an exception.

If this notification is not in place, the “caller” (main ) never gets notified of a failure and the task is lost! This is surprisingly similar to how networked systems work where messages/requests can get lost/fail without any notification.

In distributed system, this become worse, if any child thread fails in the same network system and we restart the system, internal states of the threads become lost and we cannot identify which thread was executing which task before the restart. Because after restart the state of the thread become un unpredictable.

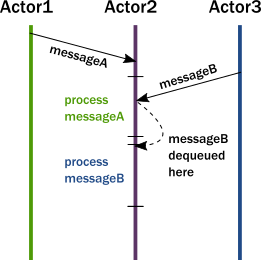
**Illusion of call stack**

Call stacks are local to each thread. If a thread delegates some of its work to another thread and suddenly if the second thread fails in middle of its execution and does not notify the calling thread, then calling thread might not know the exact situation. Concurrent systems with work delegation needs to handle service faults and have principled means to recover from them.

**Concurreny vs Multithread vs Asysnchronous programming**

<https://codewala.net/2015/07/29/concurrency-vs-multi-threading-vs-asynchronous-programming-explained/?fbclid=IwAR2P17OAXqbioCEgeB0_ZI-IhKgOS0Ga_yEkz6nJY6Q7YE-tOH6iT43STPk>

**What happens when an actor receives a message ?**



1. The actor adds the message to the end of a queue.
2. If the actor was not scheduled for execution, it is marked as ready to execute.
3. A (hidden) scheduler entity takes the actor and starts executing it.
4. Actor picks the message from the front of the queue.
5. Actor modifies its internal state, sends messages to other actors.
6. The actor is unscheduled.

To accomplish this behavior, actors have:

* A mailbox (the queue where messages end up).
* A behavior (the state of the actor, internal variables etc.).
* Messages (pieces of data representing a signal, similar to method calls and their parameters).
* An execution environment (the machinery that takes actors that have messages to react to and invokes their message handling code).
* An address (more on this later).

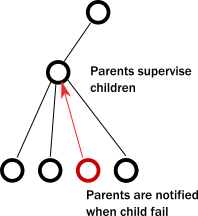
Messages go into actor mailboxes. The behavior of the actor describes how the actor responds to messages (like sending more messages and/or changing state). An execution environment orchestrates a pool of threads to drive all these actions completely transparently.

This is a very simple model and it solves the issues enumerated previously

**How actors handle errors?**

Since we no longer have a shared call stack between actors that send messages to each other, we need to handle error situations differently. There are two kinds of errors we need to consider:

* The first case is when the delegated task on the target actor failed due to an error in the task (typically some validation issue, like a non-existent user ID). In this case, the service encapsulated by the target actor is intact , it is only the task that itself is erroneous. The service actor should reply to the sender with a message, presenting the error case. There is nothing special here, errors are part of the domain and hence become ordinary messages.
* The second case is when a service itself encounters an internal fault. Akka enforces that all actors are organized into a tree-like hierarchy, i.e. an actor that creates another actor becomes the parent of that new actor. This is very similar how operating systems organize processes into a tree. Just like with processes, when an actor fails, its parent actor is notified and it can react to the failure. Also, if the parent actor is stopped, all of its children are recursively stopped, too. This service is called supervision and it is central to Akka.



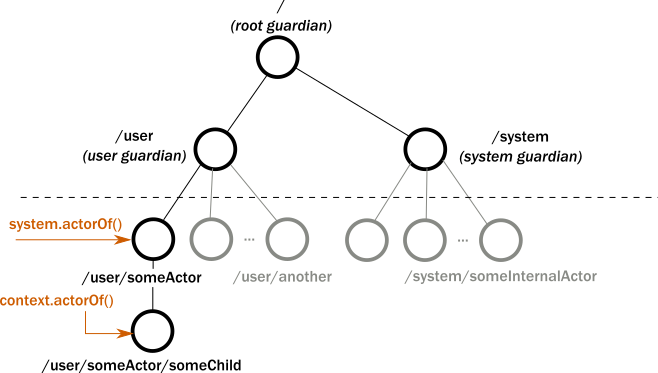
A supervisor (parent) can decide to restart its child actors on certain types of failures or stop them completely on others. Children never go silently dead (with the notable exception of entering an infinite loop) instead they are either failing and their parent can react to the fault, or they are stopped (in which case interested parties are automatically notified). There is always a responsible entity for managing an actor: its parent. Restarts are not visible from the outside: collaborating actors can keep continuing sending messages while the target actor restarts.

**What is prestart and poststop?**

The Akka actor API exposes many lifecycle hooks that you can override in an actor implementation. The most commonly used are preStart() and postStop().

* preStart() is invoked after the actor has started but before it processes its first message.
* postStop() is invoked just before the actor stops. No messages are processed after this point.

**Brief introduction on actor architecture**



If we add print an actor reference it will print a URL, which explains the above figure.

akka://testSystem/user/first-actor#1053618476

akka://testSystem/user/first-actor/second-actor#-1544706041

akka://testSystem/user/first-actor/second-sibling-actor#-1544706042

akka://testSystem/user/first-actor/ second-actor/ third-actor #-234567909

**What happen when we terminate an actorsystem ?**

actorSystem.terminate();

When you know everything is done for your application, you can call the terminate method of ActorSystem. That will stop the guardian actor, which in turn will recursively stop all its child actors, the system guardian.

Once an actor terminates, i.e. fails in a way which is not handled by a restart, stops itself or is stopped by its supervisor, it will free up its resources, draining all remaining messages from its mailbox into the system’s “dead letter mailbox” which will forward them to the EventStream as DeadLetters. The mailbox is then replaced within the actor reference with a system mailbox, redirecting all new messages to the EventStream as DeadLetters. This is done on a best effort basis, though, so do not rely on it in order to construct “guaranteed delivery”.

**Benefits of actor system ?**

an actor object needs to be shielded from the outside in order to benefit from the actor model. Therefore, actors are represented to the outside using actor references, which are objects that can be passed around freely and without restriction

* restarting an actor without needing to update references
* placing the actual actor object on remote hosts
* the most important aspect is that it is not possible to look inside an actor and get hold of its state from the outside, unless the actor unwisely publishes this information itself

**What is actor behavior ?**

Every time a message is processed, it is matched against the current behavior of the actor. Behavior means a function which defines the actions to be taken in reaction to the message at that point in time, say forward a request if the client is authorized, deny it otherwise. become and unbecome operations are the perfect example of actor behavior.

**What is actor mailbox ?**

The piece which connects sender and receiver is the actor’s mailbox: each actor has exactly one mailbox to which all senders enqueue their messages. There are different mailbox implementations to choose from, the default being a FIFO: the order of the messages processed by the actor matches the order in which they were enqueued. This is usually a good default, but applications may need to prioritize some messages over others. In this case, a priority mailbox will enqueue not always at the end but at a position as given by the message priority

**Supervising actors**

When any child actor failed, the supervisor actor can do these below options :

1. Resume the subordinate, keeping its accumulated internal state
2. Restart the subordinate, clearing out its accumulated internal state
3. Stop the subordinate permanently
4. Escalate the failure, thereby failing itself

**Restart of an actor**

When presented with an actor which failed while processing a certain message, causes for the failure fall into three categories:

* Systematic (i.e. programming) error for the specific message received
* (Transient) failure of some external resource used during processing the message
* Corrupt internal state of the actor

Restart of an actor is carried out by creating a new instance of the underlying Actor class and replacing the failed instance with the fresh one inside the child’s ActorRef; the ability to do this is one of the reasons for encapsulating actors within special references. The new actor then resumes processing its mailbox, meaning that the restart is not visible outside of the actor itself with the notable exception that the message during which the failure occurred is not re-processed

The precise sequence of events during a restart is the following:

1. suspend the actor (which means that it will not process normal messages until resumed), and recursively suspend all children
2. call the old instance’s preRestart hook (defaults to sending termination requests to all children and calling postStop)
3. wait for all children which were requested to terminate (using context.stop()) during preRestart to actually terminate; this—like all actor operations—is non-blocking, the termination notice from the last killed child will effect the progression to the next step
4. create new actor instance by invoking the originally provided factory again
5. invoke postRestart on the new instance (which by default also calls preStart)
6. send restart request to all children which were not killed in step 3; restarted children will follow the same process recursively, from step 2
7. resume the actor
8. Suppose if an exception occurred when an actor was processing a message and supervisory strategy called restart on this actor. After the restart we change the state of the actor using become() method to avoid the exception next time. But the actor is not going to process the old message after restart. Unless or until it receive any new message it is not going to process anything.

**Create an Actor**

* 1. Extending abstract actor:

**class** ArgumentActor **extends** AbstractActor{

**private** String arg1;

**private** **int** arg2;

**public** ArgumentActor(String arg1, **int** arg2) {

**this**.arg1 = arg1;

**this**.arg2 = arg2;

}

@Override

**public** Receive createReceive() {

**return** receiveBuilder().matchEquals("create", p->{

System.***out***.println("Inside Argument actor arguments are :"+arg1+" "+arg2);

}).build();

}

}

* 1. Extending UntypedAbstractActor :

**class** UntypedArgumentActor **extends** UntypedAbstractActor{

**private** **int** arg;

**static** Props props(**int** x) {

**return** Props.*create*(UntypedArgumentActor.**class**,()->**new** UntypedArgumentActor(x));

}

**public** UntypedArgumentActor(**int** arg) {

**this**.arg = arg;

}

@Override

**public** **void** onReceive(Object arg0) **throws** Throwable {

**if**(arg0 **instanceof** String) {

**if**(arg0.equals("create")) {

System.***out***.println("Inside no argument untyped actor "+arg);

}

} }}

* 1. Call to create an actor

ActorRef unTypedArgRef = actorSystem.actorOf(UntypedArgumentActor.props(10), "UnTypedArgumentactor");

unTypedArgRef.tell("create", ActorRef.noSender());

* 1. Props is a configuration class to specify options for the creation of actors, think of it as an immutable and thus freely shareable recipe for creating an actor including associated deployment information (e.g. which dispatcher to use, see more below). Here are some examples of how to create a Props instance.

For argument constructor , if we call Props from outside the actor class it may throw IllegalArgumentException if no or multiple matching constructors are found. So better way is to create a static method props() for every actor class and call the method to create props onject.

**Different match methods of receivebuilder ?**

* match(Calss<P> type, UnitApply<P> apply) : P is the type of the message class, UnitApply is a functional interface which accepts the message P as a single argument and return void.
* match(Calss<P> type, TypedPredicate<P> predicate, UnitApply<P> apply) : the second argument predicate is going to validate the message class type or the content.
* matchAny(UnitApply<P> apply) : It accepts any message of any type, P is only a generic type here.
* matchEquals(P object , UnitApply<P> apply) : when we are sure about the type of the message we directly check the content , like “create” is a string in above example.
* matchEquals(P object , TypedPredicate<P> predicate , UnitApply<P> apply) : this is same as the second case.

**What does unhandled(message) method do ?**

If the current actor behavior does not match a received message, unhandled is called, which by default publishes an akka.actor.UnhandledMessage(message, sender, recipient) on the actor system’s event stream (set configuration item akka.actor.debug.unhandled to on to have them converted into actual Debug messages).

**What is lifecycle monitoring ?**

Lifecycle Monitoring in Akka is usually referred to as DeathWatch.

In contrast to the special relationship between parent and child described above, each actor may monitor any other actor. Since actors emerge from creation fully alive and restarts are not visible outside of the affected supervisors, the only state change available for monitoring is the transition from alive to dead. Monitoring is thus used to tie one actor to another so that it may react to the other actor’s termination, in contrast to supervision which reacts to failure.

Lifecycle monitoring is implemented using a Terminated message to be received by the monitoring actor, where the default behavior is to throw a special DeathPactException if not otherwise handled. In order to start listening for Terminated messages, invoke ActorContext.watch(targetActorRef). To stop listening, invoke ActorContext.unwatch(targetActorRef).

Here is an example of deathwatch. Here I called stop method from the supervisor actor, we can stop from any other actor and apply watch method on it.

@Override

**public** Receive createReceive() {

* + 1. **return** receiveBuilder().matchEquals("create", p -> {
    2. ActorRef firstRef = getContext().actorOf(Props.*empty*(), "first-child-actor");
    3. System.***out***.println("First: " + firstRef);
    4. ActorRef secondRef = getContext().actorOf(Props.*empty*(), "second-child-actor");
    5. System.***out***.println("second: " + secondRef);
    6. getContext().watch(firstRef);
    7. getContext().watch(secondRef);
    8. getContext().stop(firstRef);
    9. getContext().stop(secondRef);
    10. }).match(Terminated.**class**, p->{
    11. System.***out***.println(" terminated message sent by "+getContext().sender());
    12. getContext().system().terminate();
    13. }).build(); } }

One important property is that the message will be delivered irrespective of the order in which the monitoring request and target’s termination occur, i.e. you still get the message even if at the time of registration the target is already dead. If we write line 6 and 7 after line 8 and 9, still it will receive the message.

Note : Actor monitoring is a relation between any actor, supervising is relation between parent and child actor.

**Backoff strategy pattern :**

Provided as a built-in pattern the akka.pattern.BackoffSupervisor implements the so-called *exponential backoff supervision strategy*, starting a child actor again when it fails, each time with a growing time delay between restarts.

This pattern is useful when the started actor fails [[1]](https://doc.akka.io/docs/akka/current/general/supervision.html#1) because some external resource is not available, and we need to give it some time to start-up again. One of the prime examples when this is useful is when a [PersistentActor](https://doc.akka.io/docs/akka/current/persistence.html) fails (by stopping) with a persistence failure - which indicates that the database may be down or overloaded, in such situations it makes most sense to give it a little bit of time to recover before the persistent actor is started.

Following is the code:

final Props childProps = Props.create(EchoActor.class);

final Props supervisorProps = BackoffSupervisor.props(

Backoff.onStop(

childProps,

"myEcho",

Duration.ofSeconds(3),

Duration.ofSeconds(30),

0.2)); // adds 20% "noise" to vary the intervals slightly

system.actorOf(supervisorProps, "echoSupervisor");

There are two classes of supervision strategies which come with Akka

* OneForOneStrategy
* AllForOneStrategy

The difference between them is that the former applies the obtained directive only to the failed child, whereas the latter applies it to all siblings as well. Normally, you should use the OneForOneStrategy, which also is the default if none is specified explicitly. The AllForOneStrategy is applicable in cases where the ensemble of children has such tight dependencies among them, that a failure of one child affects the function of the others

**What is actor path ?**

An actor path consists of an anchor, which identifies the actor system, followed by the concatenation of the path elements, from root guardian to the designated actor; the path elements are the names of the traversed actors and are separated by slashes.

**What is the difference actor reference and path ?**

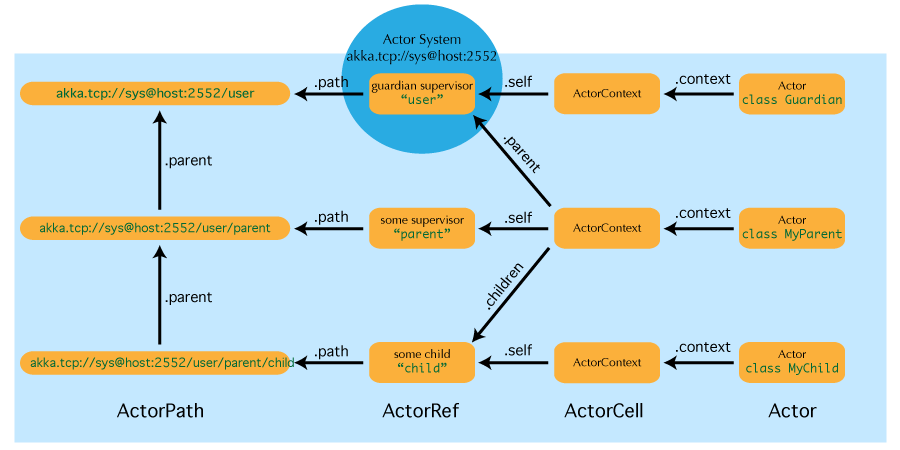
An actor reference designates a single actor and the life-cycle of the reference matches that actor’s life-cycle; an actor path represents a name which may or may not be inhabited by an actor and the path itself does not have a life-cycle, it never becomes invalid. You can create an actor path without creating an actor, but you cannot create an actor reference without creating corresponding actor.

You can create an actor, terminate it, and then create a new actor with the same actor path. The newly created actor is a new incarnation of the actor. It is not the same actor. An actor reference to the old incarnation is not valid for the new incarnation. Messages sent to the old actor reference will not be delivered to the new incarnation even though they have the same path.

"akka://my-sys/user/service-a/worker1" // purely local

"akka.tcp://my-sys@host.example.com:5678/user/service-b" // remote

**Actor path and the different methods**



**Difference between actroOf and actorSelection method in actorsystem ?**

* actorOf  only ever creates a new actor, and it creates it as a direct child of the context on which this method is invoked (which may be any actor or actor system).
* actorSelection only ever looks up existing actors when messages are delivered, i.e. does not create actors, or verify existence of actors when the selection is created
* actor selection can be used like this using relative path

context.actorSelection("../\*") or using absolute path context.actorSelection("user/parent/child")

* ActorSelection on the other hand points to the path (or multiple paths if wildcards are used) and is completely oblivious to which incarnation is currently occupying it. ActorSelection cannot be watched for this reason. It is possible to resolve the current incarnation’s ActorRef living under the path by sending an Identify message to the ActorSelection which will be replied to with an ActorIdentity containing the correct reference.

**Few terms related to actor path**

* "/user" is the guardian actor for all user-created top-level actors; actors created using ActorSystem.actorOf are found below this one.
* "/system" is the guardian actor for all system-created top-level actors, e.g. logging listeners or actors automatically deployed by configuration at the start of the actor system.
* "/deadLetters" is the dead letter actor, which is where all messages sent to stopped or non-existing actors are re-routed (on a best-effort basis: messages may be lost even within the local JVM).
* "/temp" is the guardian for all short-lived system-created actors, e.g. those which are used in the implementation of ActorRef.ask.
* "/remote" is an artificial path below which all actors reside whose supervisors are remote actor references

**What are the message delivery rules ?**

* **at-most-once** delivery means that for each message handed to the mechanism, that message is delivered once or not at all; in more casual terms it means that messages may be lost.
* **at-least-once** delivery means that for each message handed to the mechanism potentially multiple attempts are made at delivering it, such that at least one succeeds; again, in more casual terms this means that messages may be duplicated but not lost.
* **exactly-once** delivery means that for each message handed to the mechanism exactly one delivery is made to the recipient; the message can neither be lost nor duplicated.

The first one is the cheapest—highest performance, least implementation overhead—because it can be done in a fire-and-forget fashion without keeping state at the sending end or in the transport mechanism. Akka follows the first one at-most-once delivery process.

**Example of message delivery ordering in Akka**

1. If M1 is delivered it must be delivered before M2 and M3
2. If M2 is delivered it must be delivered before M3
3. If M4 is delivered it must be delivered before M5 and M6
4. If M5 is delivered it must be delivered before M6
5. A2 can see messages from A1 interleaved with messages from A3
6. Since there is no guaranteed delivery, any of the messages may be dropped, i.e. not arrive at A2

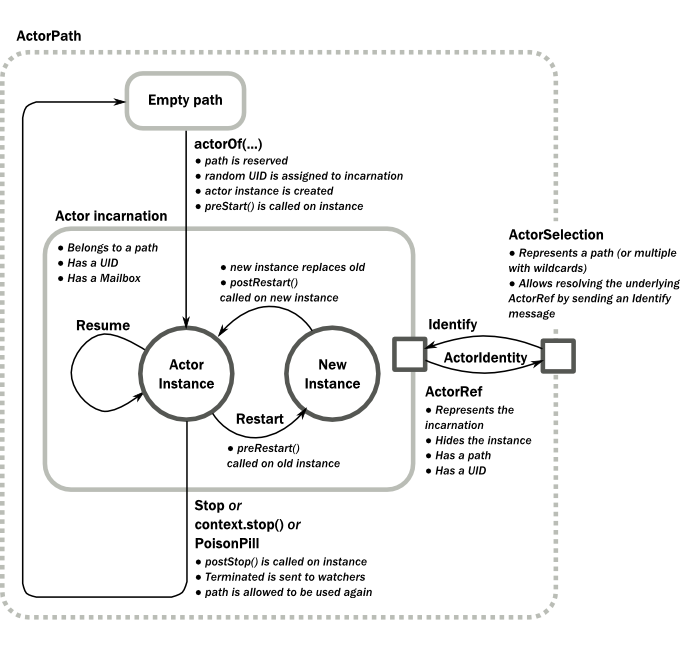
Please note, that the ordering guarantees discussed above only hold for user messages between actors. Failure of a child of an actor is communicated by special system messages that are not ordered relative to ordinary user messages. In particular: Child actor C sends message M to its parent P

Child actor fails with failure F

Parent actor P might receive the two events either in order M, F or F, M

The reason for this is that internal system messages have their own mailboxes therefore the ordering of enqueue calls of a user and system messages cannot guarantee the ordering of their dequeue times.

**Actor lifecycle**



**Difference between actor restart and stop method ?**

Restart and stop both are process inside a actor system. Stop process we can initiate for an actor reference by calling getcontext.stop(actor ref) or actorsystem.stop(actorref) . But for restart we don’t have any explicit method named restart. Wheather we can control restart process by hooking with prerestart() and postrestart() method.

A restart only swaps the Actor instance defined by the Props but the incarnation and hence the UID remains the same. As long as the incarnation is same, you can keep using the same ActorRef. Restart is handled by the [Supervision Strategy](https://doc.akka.io/docs/akka/2.5/fault-tolerance.html#creating-a-supervisor-strategy) of actor’s parent actor . That’s why an actor is encapsulated with actorreference, if actor instance is changed inside the reference, we are not aware of the change from outside.

The lifecycle of an incarnation ends when the actor is stopped. At that point the appropriate lifecycle events are called and watching actors are notified of the termination. After the incarnation is stopped, the path can be reused again by creating an actor with actorOf(). In this case the name of the new incarnation will be the same as the previous one but the UIDs will differ. An actor can be stopped by the actor itself, another actor or the ActorSystem. An ActorRef always represents an incarnation (path and UID) not just a given path. Therefore if an actor is stopped and a new one with the same name is created an ActorRef of the old incarnation will not point to the new one.

**Use of ask() method over tell**

* tell means “fire-and-forget”, e.g. send a message asynchronously and return immediately.
* ask sends a message asynchronously and returns a representing a possible reply.

Example of ask

Timeout timeout = Timeout.*create*(Duration.*ofSeconds*(5));

Future<Object> futureA = Patterns.*ask*(actorA, **new** Message("start", "A"), timeout);

Future<Object> futureB = Patterns.*ask*(actorB, **new** Message("start", "B"), timeout);

**try** {

Await.*result*(futureA, timeout.duration());

Await.*result*(futureB, timeout.duration());

}**catch**(Exception e) {

System.***out***.println(futureA.isCompleted()+" "+futureB.isCompleted());

}

**Stopping of an actor**

Termination of an actor proceeds in two steps: first the actor suspends its mailbox processing and sends a stop command to all its children, then it keeps processing the internal termination notifications from its children until the last one is gone, finally terminating itself (invoking postStop, dumping mailbox, publishing Terminated on the [DeathWatch](https://doc.akka.io/docs/akka/2.5/actors.html#deathwatch), telling its supervisor). This procedure ensures that actor system sub-trees terminate in an orderly fashion, propagating the stop command to the leaves and collecting their confirmation back to the stopped supervisor. If one of the actors does not respond (i.e. processing a message for extended periods of time and therefore not receiving the stop command), this whole process will be stuck.

Upon ActorSystem.terminate(), the system guardian actors will be stopped, and the aforementioned process will ensure proper termination of the whole system.

**What is graceful stop ?**

gracefulStop is useful if you need to wait for termination or compose ordered termination of several actors:

try {

CompletionStage<Boolean> stopped =

gracefulStop(actorRef, Duration.ofSeconds(5), Manager.SHUTDOWN);

stopped.toCompletableFuture().get(6, TimeUnit.SECONDS);

// the actor has been stopped

} catch (AskTimeoutException e) {

// the actor wasn't stopped within 5 seconds

}

When gracefulStop() returns successfully, the actor’s postStop() hook will have been executed: there exists a happens-before edge between the end of postStop() and the return of gracefulStop()

**Akka become/unbecome**

Akka supports hotswapping the Actor’s message loop (e.g. its implementation) at runtime: invoke the context.become method from within the Actor. become takes a PartialFunction<Object, BoxedUnit> that implements the new message handler. The hotswapped code is kept in a Stack which can be pushed and popped

The other way of using become does not replace but add to the top of the behavior stack. In this case care must be taken to ensure that the number of “pop” operations (i.e. unbecome) matches the number of “push” ones in the long run, otherwise this amounts to a memory leak (which is why this behavior is not the default).

Example :

Create receive method

**public** Receive createReceive() {

AbstractActor.Receive hello = receiveBuilder().matchEquals("Swap", x -> {

System.***out***.println("Hello");

getContext().unbecome();

}).build();

**return** receiveBuilder().matchEquals("Swap", s -> {

System.***out***.println("Hi");

getContext().become(hello,**false**);

}).build();

}

ActorRef swapper = system.actorOf(Props.*create*(Swapper.**class**), "swapper");

swapper.tell("Swap", ActorRef.*noSender*()); // Hi

swapper.tell("Swap", ActorRef.*noSender*()); // Hello

swapper.tell("Swap", ActorRef.*noSender*()); // Hi

for first tell call it will print “hi” and call become(hello) , which will push hello behavior on top of behavior stack. So for next call hello behavior will be imposed and print “hello”. But here we call unbecome, so the top of the stack is removed or popped. So next call it will again print “hi”

**Use of stash and unstash**

Invoking stash() adds the current message (the message that the actor received last) to the actor’s stash. It is typically invoked when handling the default case in the actor’s message handler to stash messages that aren’t handled by the other cases. It is illegal to stash the same message twice; to do so results in an IllegalStateException being thrown. The stash may also be bounded in which case invoking stash() may lead to a capacity violation, which results in a StashOverflowException. The capacity of the stash can be configured using the stash-capacity setting (an Int) of the mailbox’s configuration.

Invoking unstashAll() enqueues messages from the stash to the actor’s mailbox until the capacity of the mailbox (if any) has been reached (note that messages from the stash are prepended to the mailbox). In case a bounded mailbox overflows, a MessageQueueAppendFailedException is thrown. The stash is guaranteed to be empty after calling unstashAll().

The stash is backed by a scala.collection.immutable.Vector. As a result, even a very large number of messages may be stashed without a major impact on performance.

Note that the stash is part of the ephemeral actor state, unlike the mailbox. Therefore, it should be managed like other parts of the actor’s state which have the same property. The AbstractActorWithStash implementation of preRestart will call unstashAll(), which is usually the desired behavior.

Example :

public class ActorWithProtocol extends AbstractActorWithStash {

@Override

public Receive createReceive() {

return receiveBuilder()

.matchEquals("open", s -> {

getContext().become(receiveBuilder()

.matchEquals("write", ws -> { /\* do writing \*/ })

.matchEquals("close", cs -> {

unstashAll();

getContext().unbecome();

})

.matchAny(msg -> stash())

.build(), false);

})

.matchAny(msg -> stash())

.build();

}

}

If you want to enforce that your actor can only work with an unbounded stash, then you should use the AbstractActorWithUnboundedStash class instead.

**What happen when message is being processed and exception occurred ?**

It can happen that while a message is being processed by an actor, that some kind of exception is thrown, e.g. a database exception.

What happens to the Message

If an exception is thrown while a message is being processed (i.e. taken out of its mailbox and handed over to the current behavior), then this message will be lost. It is important to understand that it is not put back on the mailbox. So if you want to retry processing of a message, you need to deal with it yourself by catching the exception and retry your flow. Make sure that you put a bound on the number of retries since you don’t want a system to livelock (so consuming a lot of cpu cycles without making progress).

What happens to the mailbox

If an exception is thrown while a message is being processed, nothing happens to the mailbox. If the actor is restarted, the same mailbox will be there. So all messages on that mailbox will be there as well.

What happens to the actor

If code within an actor throws an exception, that actor is suspended and the supervision process is started (see [supervision](https://doc.akka.io/docs/akka/2.5/general/supervision.html)). Depending on the supervisor’s decision the actor is resumed (as if nothing happened), restarted (wiping out its internal state and starting from scratch) or terminated

**Fault Tolerence Handling**

Supervisory strategy can be handled like this

private static SupervisorStrategy strategy =

new OneForOneStrategy(

10,

Duration.ofMinutes(1),

DeciderBuilder.match(ArithmeticException.class, e -> SupervisorStrategy.resume())

.match(NullPointerException.class, e -> SupervisorStrategy.restart())

.match(IllegalArgumentException.class, e -> SupervisorStrategy.stop())

.matchAny(o -> SupervisorStrategy.escalate())

.build());

@Override

public SupervisorStrategy supervisorStrategy() {

return strategy;

}

Also, there are special values for these parameters. If you specify:

* -1 to maxNrOfRetries, and Duration.Inf() to withinTimeRange
  + then the child is always restarted without any limit
* -1 to maxNrOfRetries, and a non-infinite Duration to withinTimeRange
  + maxNrOfRetries is treated as 1
* a non-negative number to maxNrOfRetries and Duration.Inf() to withinTimeRange
  + withinTimeRange is treated as infinite duration (i.e.) no matter how long it takes, once the restart count exceeds maxNrOfRetries, the child actor is stopped

**What is default supervisory strategy ?**

Escalate is used if the defined strategy doesn’t cover the exception that was thrown.

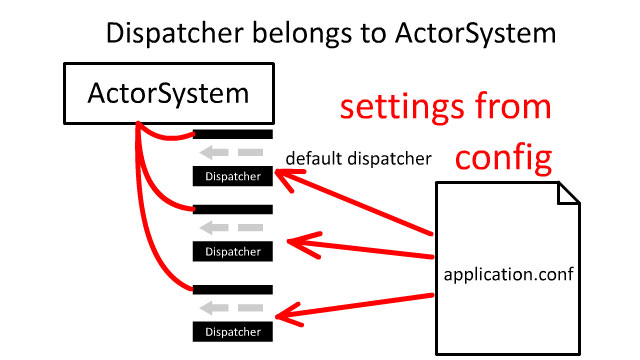
When the supervisor strategy is not defined for an actor the following exceptions are handled by default:

* ActorInitializationException will stop the failing child actor
* ActorKilledException will stop the failing child actor
* DeathPactException will stop the failing child actor
* Exception will restart the failing child actor
* Other types of Throwable will be escalated to parent actor

If the exception escalate all the way up to the root guardian it will handle it in the same way as the default strategy defined above.

**What is akka dispatcher ?**

If you look up the meaning of the word “dispatch” in a dictionary, you would find it is almost same as “send”. In akka, Dispatcher is, what sends messages, but something more than that.



Firstly, Dispatcher is configured for ActorSystem, typically in application.conf. There is at least default one, and you can [also configure multiple Dispatchers](https://doc.akka.io/docs/akka/current/dispatchers.html#dispatchers). By default this is a “fork-join-executor”, which gives excellent performance in most cases. Example of how to lookup a particular dispatcher context.

final ExecutionContext ex = system.dispatchers().lookup("my-dispatcher");

Example of application.conf

my-dispatcher {

# Dispatcher is the name of the event-based dispatcher

type = Dispatcher

# What kind of ExecutionService to use

executor = "fork-join-executor"

# Configuration for the fork join pool

fork-join-executor {

# Min number of threads to cap factor-based parallelism number to

parallelism-min = 2

# Parallelism (threads) ... ceil(available processors \* factor)

parallelism-factor = 2.0

# Max number of threads to cap factor-based parallelism number to

parallelism-max = 10

}

# Throughput defines the maximum number of messages to be

# processed per actor before the thread jumps to the next actor.

# Set to 1 for as fair as possible.

throughput = 100

}

**How many ways to define dispatcher in program ?**

 create the actor as usual and define the dispatcher in the deployment configuration.

ActorRef myActor = system.actorOf(Props.create(MyActor.class), "myactor");

akka.actor.deployment {

/myactor {

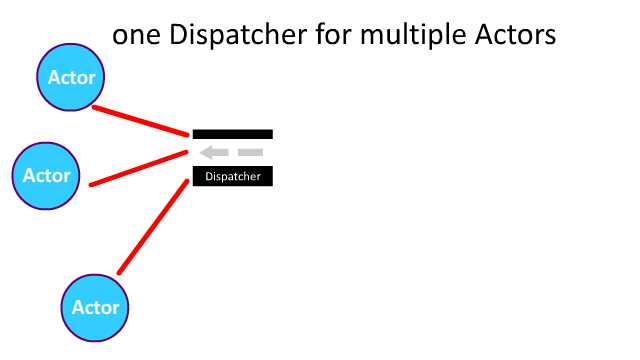
dispatcher = my-dispatcher

}

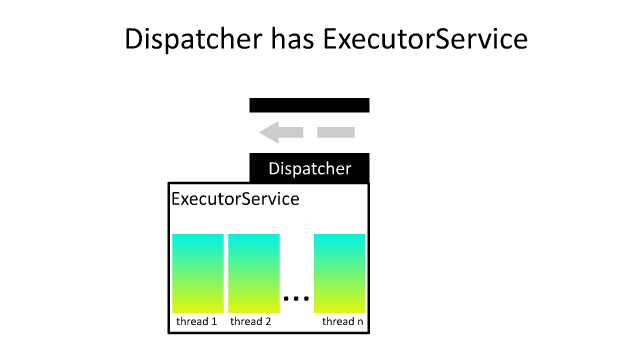
}

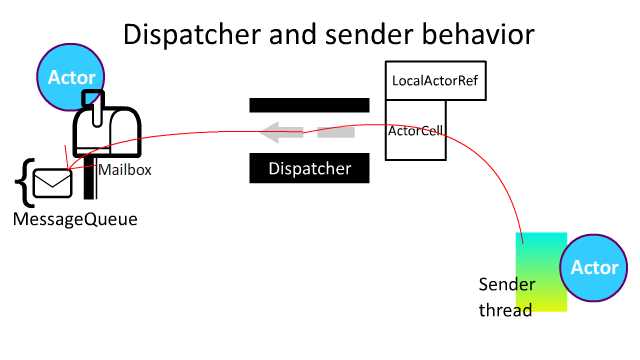
As a rule of thumb, the Dispatcher instance for the given name is created when the [lookup method of ActorSystem](https://github.com/akka/akka/blob/v2.5.9/akka-actor/src/main/scala/akka/dispatch/Dispatchers.scala#L79) is called for the first time. You don’t normally call it yourself, but this lookup is done by akka. Another thing is the default Dispatcher is already created upon ActorSystem initialization, as it calls lookup for the default internally.

Dispatcher is NOT part of Actor. One Dispatcher can send messages to multiple Actors. (NOTE: Dispatcher doesn’t have routing capabilities. Routing is done by akka [Router](https://doc.akka.io/docs/akka/2.5/routing.html#routing))



**Dispatcher and executor service**

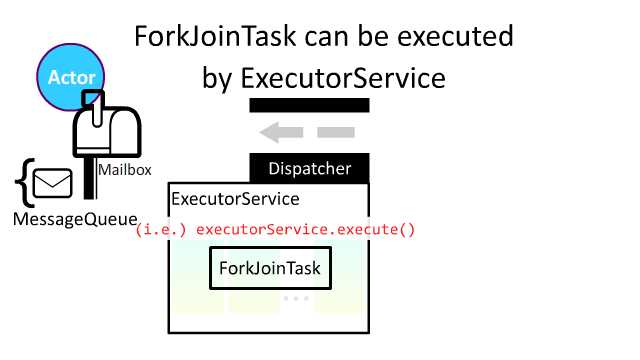


Dispatcher has ExecutorService, and ExecutorService is like a pool of threads where you can execute code (Runnable) concurrently. The pool of threads from ExecutorService is what invokes Actor’s receive method, 

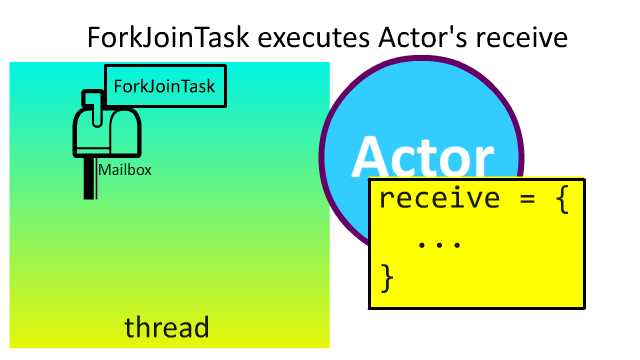
So when we do actorRef.tell("hello"), the  actorRef (whose type is ActorRef) already knows what Dispatcher to use via [ActorCell](https://github.com/akka/akka/blob/v2.5.9/akka-actor/src/main/scala/akka/actor/ActorCell.scala#L370). Also ActorCell extends [Dispatch trait](https://github.com/akka/akka/blob/v2.5.9/akka-actor/src/main/scala/akka/actor/dungeon/Dispatch.scala#L27) and it has a reference to Mailbox, so LocalActorRef also knows which Mailbox to send the massage, via ActorCell.

 Dispatcher’s excutorService is executing mbox: Mailbox, because [Mailbox](https://github.com/akka/akka/blob/v2.5.9/akka-actor/src/main/scala/akka/dispatch/Mailbox.scala#L56L57) extends ForkJoinTask, which can be execute-d by ExecutorService.

Mailbox extends ForkJoinTask. So it can be executed by executor service. Execution (i.e. processing) of Mailbox is run on a different Thread



it picks up a message from the message queue, and process it.



So this processMailbox method, called from ForkJoinTask’s run is what invokes your receive method you defined in your Actor.

**Why event bus ?**

If we want to send messages to the group of actors then we use Akka event bus. Unlike other actor message call EventBus does not preserve the sender of the published messages. If you need a reference to the original sender you have to provide it inside the message.

An event bus must define the following three type parameters:

* Event is the type of all events published on that bus
* Subscriber is the type of subscribers allowed to register on that event bus
* Classifier defines the classifier to be used in selecting subscribers for dispatching events.

like topic or channeled is the classifier.

**Different type of classifiers**

**LookupClassification** : The simplest classification is just to extract an arbitrary classifier from each event and maintaining a set of subscribers for each possible classifier. This can be compared to tuning in on a radio station.

extends LookupEventBus<MsgEnvelope, ActorRef, String>

MsgEnvelop is the event type

String is the topic type

Overridden methods

@Override

public String classify(MsgEnvelope event) {

return event.topic;

}

@Override

public void publish(MsgEnvelope event, ActorRef subscriber) {

subscriber.tell(event.payload, ActorRef.noSender());

}

@Override

public int compareSubscribers(ActorRef a, ActorRef b) {

return a.compareTo(b);

}

**SubChannel Classification :**

If classifiers form a hierarchy and it is desired that subscription be possible not only at the leaf nodes, this classification may be just the right one. It can be compared to tuning in on (possibly multiple) radio channels by genre. This classification has been developed for the case where the classifier is just the JVM class of the event and subscribers may be interested in subscribing to all subclasses of a certain class, but it may be used with any classifier hierarchy.

One subclassification class

class StartsWithSubclassification implements Subclassification<String> {

@Override

public boolean isEqual(String x, String y) {

return x.equals(y);

}

@Override

public boolean isSubclass(String x, String y) {

return x.startsWith(y);

}

}

Here topic type is String

And subchannel event bus

class SubchannelBusImpl extends SubchannelEventBus<MsgEnvelope, ActorRef, String>

Overridden method

@Override

public Subclassification<String> subclassification() {

return new StartsWithSubclassification();

}

@Override

public String classify(MsgEnvelope event) {

return event.topic;

}

@Override

public void publish(MsgEnvelope event, ActorRef subscriber) {

subscriber.tell(event.payload, ActorRef.noSender());

}

ScanningClassification :

this classifier is useful if there are overlapping classifiers which cover various parts of the event space without forming a hierarchy. It can be compared to tuning in on (possibly multiple) radio stations by geographical reachability (for old-school radio-wave transmission).

class ScanningBusImpl extends ScanningEventBus<String, ActorRef, Integer> {

// is needed for determining matching classifiers and storing them in an

// ordered collection

@Override

public int compareClassifiers(Integer a, Integer b) {

return a.compareTo(b);

}

// is needed for storing subscribers in an ordered collection

@Override

public int compareSubscribers(ActorRef a, ActorRef b) {

return a.compareTo(b);

}

// determines whether a given classifier shall match a given event; it is invoked

// for each subscription for all received events, hence the name of the classifier

@Override

public boolean matches(Integer classifier, String event) {

return event.length() <= classifier;

}

// will be invoked for each event for all subscribers which registered themselves

// for the event’s classifier

@Override

public void publish(String event, ActorRef subscriber) {

subscriber.tell(event, ActorRef.noSender());

}

}

**Actor Classification :**

This classification was originally developed specifically for implementing [DeathWatch](https://doc.akka.io/docs/akka/current/actors.html#deathwatch): subscribers as well as classifiers are of type ActorRef.

This classification requires an ActorSystem in order to perform book-keeping operations related to the subscribers being Actors, which can terminate without first unsubscribing from the EventBus. ManagedActorClassification maintains a system Actor which takes care of unsubscribing terminated actors automatically.

class Notification {

public final ActorRef ref;

public final int id;

public Notification(ActorRef ref, int id) {

this.ref = ref;

this.id = id;

}

}

lass ActorBusImpl extends ManagedActorEventBus<Notification> {

// the ActorSystem will be used for book-keeping operations, such as subscribers terminating

public ActorBusImpl(ActorSystem system) {

super(system);

}

// is used for extracting the classifier from the incoming events

@Override

public ActorRef classify(Notification event) {

return event.ref;

}

// determines the initial size of the index data structure

// used internally (i.e. the expected number of different classifiers)

@Override

public int mapSize() {

return 128;

}

}

**Refer this link** [**https://doc.akka.io/docs/akka/current/event-bus.html**](https://doc.akka.io/docs/akka/current/event-bus.html)

**What is Event stream ?**

The event stream is the main event bus of each actor system: it is used for carrying [log messages](https://doc.akka.io/docs/akka/current/logging.html) and [Dead Letters](https://doc.akka.io/docs/akka/current/event-bus.html#dead-letters) and may be used by the user code for other purposes as well. It uses [Subchannel Classification](https://doc.akka.io/docs/akka/current/event-bus.html#subchannel-classification) which enables registering to related sets of channels

Lets define an actor which handle deadletter

static class DeadLetterActor extends AbstractActor {

@Override

public Receive createReceive() {

return receiveBuilder()

.match(

DeadLetter.class,

msg -> {

System.out.println(msg);

})

.build();

}

It can be subscribed like this

final ActorSystem system = ActorSystem.create("DeadLetters");

final ActorRef actor = system.actorOf(Props.create(DeadLetterActor.class));

system.getEventStream().subscribe(actor, DeadLetter.class);

EventStream will automatically remove subscribers when they terminate

**What is Deadletter ?**

Messages queued when an actor terminates or sent after its death are re-routed to the dead letter mailbox, which by default will publish the messages wrapped in DeadLetter. This wrapper holds the original sender, receiver and message of the envelope which was redirected.