	Scalaz Actors	Lift Actors	Scala Actors	Akka Actors
Design philosophy				
	Minimal complexity. Maximal generality, modularity, and extensibility.	Minimal complexity, Garbage collection by JVM rather than worrying about an explicit lifecycle, error handling behavior consistent with other Scala & Java programs, lightweight/small memory footprint, mailbox, syntactically similar to Scala Actors and Erlang actors, high performance	Provide the full Erlang actor model in Scala, lightweight/small memory footprint	Simple and transparently distributable, high performance, lightweight and highly adaptable
Versioning				
Current stable version	5	2.1	2.8.1	0.10
Minimum Scala version	2.8	2.7.7		2.8
Minimum Java version		1.5	1.5	1.6
Actor Model Support				
spawn new actors inside of actor	Yes	Yes	Yes	Yes
send messages to known actor	Yes	Yes	Yes	Yes
change behavior for next message	Actors are immutable	Yes	Yes: nested react/receive	Yes: become/unbecome
Supervision (link/trapExit)	Not provided	No	Actor: Yes, Reactor: No	Yes
Level of state isolation				
If user defines public methods on their Actors, are they callable from the outside?	n/a. Actor is a sealed trait	Yes	Yes	No, actor instance is shielded behind an ActorRef
Actor type				
	Actor[A] extends A => ()	LiftActor, SpecializeLiftActor[T]	Reactor[T], Actor extends Reactor[Any]	Actor[Any]
Actor lifecycle management				
Manual start	No	No	Yes	Yes
Manual stop	No	No	No	Yes
Restart-on-failure	n/a	Yes	Yes	Configurable per actor instance
Restart semantics		N/A	Rerun actor behavior	Restore actor to stable state by re-allocating it and throw away the old instance
Restart configurability		N/A	N/A	X times, X times within Y time
Lifecycle hooks provided		No (no lifecycle)	act	preStart, postStop, preRestart, postRestart
Message send modes				

	Scalaz Actors	Lift Actors	Scala Actors	Akka Actors
fire-forget	a ! message, or a (message)	actor ! msg	actor ! msg	actorRef ! message
send-receive-reply	Any function f becomes such an actor: { val a: Msg => Promise[Rep] = f. promise; val reply: Rep = a(msg).get }	actor !? msg actor !! msg	actor !? msg	actorRef !! message
send-receive-future	Any function f becomes such an actor: { val a = f. promise; val replyFuture = a(message) }		actor !! msg	actorRef !!! message
send-result-of-future	promise(message).to (actor)			future.onComplete(f => to ! f.result)
compose actor with function	Contravariant functor: actor comap f. Also Kleisli composition in Promise	No	No	No
Message reply modes				
reply-to-sender-in-message	{ case (msg,replyTo) => replyTo ! replyMessage }	N/A	{ case (msg, replyTo) => replyTo ! replyMessage }	{ case (msg,replyTo) => replyTo ! replyMessage }
reply-to-message	Promote ordinary function to Promise	{ case msg => reply (response) }	{ case msg => reply(response) }	{ case msg => self reply replyMessage }
Message processing				
Supports nested receives		Yes (with a little hand coding)	Yes, both thread-based receive and event-based react	No, nesting receives can lead to memory leaks and degraded performance over time.
Message Execution Mechanism				
Name for Execution Mechanism	Strategy	java.util.Concurrent	IScheduler	Dispatcher
Execution Mechanism is configurable	Yes	No	Yes	Yes
Execution Mechanism can be specified on a per-actor basis	Yes	No	Yes	Yes
Lifecycle of Execution Mechanism must be explicitly managed	Depends on Strategy	No	Depends on IScheduler	No
"thread-per-actor"-execution mechanism	Use one Strategy per actor with single-threaded Strategy	No	When calling receive, thread pool provides thread of calling actor	ThreadBasedDispatcher (deallocates backing Thread after inactivity timeout)

	Scalaz Actors	Lift Actors	Scala Actors	Akka Actors
"event-driven"-execution mechanism	Strategy.Executor	All Lift Actors are event driven	Actors are event-driven when no thread-blocking methods like receive are used.	ExecutorBasedEventDrivenDispatcher, HawtDispatcher, ExecutorBasedEventDrivenWorkstealingDispatcher
Mailbox type	ConcurrentLinkedQueue guarded by CountdownLatch	custom implementation of a doubly linked list that requires very few locks to access (no locks during dispatch)	Custom linked list that enables optimizations in the actor implementation.	Defined per Dispatcher, highly configurable
Supports transient mailboxes	Yes	Yes	Yes	Yes
Supports persistent mailboxes	No	No		In commercial offering
Distribution/Remote Actors				
Transparent remote actors	N/A	No	Yes	Yes
Transport protocol	N/A	N/A	Java serialization on top of TCP	Akka Remote Protocol (Protobuf on top of TCP)
Dynamic clustering	N/A	N/A	N/A	In commercial offering
Howtos				
Define an actor	val messageHandler: T => = t => action(t)	new MyActor extends LiftActor { def messageHandler = {case x => } }	class MyActor extends Actor { def act() { react { case x => } } }	class MyActor extends Actor { def receive = { case message => action } }
Create an actor instance	actor(messageHandler)	new MyActor	new MyActor	val myActor = actorOf[MyActor]
Start an actor instance	n/a -- no need to start or stop an actor. An actor will always process messages as long as you have a JVM reference to it	n/a -- no need to start or stop an actor. An actor will always process messages as long as you have a JVM reference to it	myActor.start	myActor.start
Stop an actor instance	n/a	N/A -- no need to start or stop an actor. An actor will always process messages as long as you have a JVM reference to it	N/A	myActor.stop