# Scala in Depth

Joshua D. Suereth

MANNING

# 9 Actors.

The chapter covers general design principles using Actors and how to apply these to Scala's standard library. The following topics will be covered:

- Knowing the difference between react and receive
- Using typed communication and sealed message protocols
- Limiting failures to zones using Supervisors
- Limiting starvation to zones using Schedulers

## 9.1 Know when to use Actors

Actors are an abstraction on aynchronous processes. They communicate to the external world by sending and receiving messages. An actor will process received messages sequentially in the order they are received. An actor will only handle one message at a time. This fact is critical, because it means that Actors can maintain state without explicit locks. Actors can also be asynchronous or synchronous. Most actors will not block a thread when waiting for messages, although this can be done if desired. The default behavior for actors is to share threads amongst each other when handling messages. This means a small set of threads could support a large number of actors, given the right behavior.

In fact, actors are great state machines. They accept a limited number of input messages and update their internal state. All communication is done through messages and each actor is standalone.

Actors are not the magic concurrency pill that will solve all issues your system is currently facing.

Actors are not paralleization factories. Actors process their messages in single threaded fashion. They work best when work is conceptually split and each actor can handle a portion of the work. If the application needs to farm many similar tasks out for processing, this requires a large pool of actors to see any concurrency benefits.

Actors and I/O should interleaved carefully. Asynchronous I/O and actors are a naturally pairing, as they execution models for these are very similar. Using an actor to perform blocking I/O is asking for trouble. That actor can starve other actors during this processing. This can be mitigated, as will be discussed in section 9.4.

While many problems can be successfully modelled in Actors, some will be more successful. The architecture of a system designed to use actors will also change fundamentally. Rather than relying on classic Model-View-Controller and client-based paralllelism, an Actors system parallelizes pieces of the architecture and performs all communication asynchronously.

Let's look at a cannonical eample of a good system design using actors. This example uses a lot of the tools found in the old Message Passing Interface (MPI) specification used in supercomputing. MPI is worth a look, as it holds a lot of concepts that have naturally translated into Actor-based systems.

### *9.1.1 An example*

Let's design a classic search program. This program has a set of documents that live in some kind of search index. Queries are accepted from users and the index is searched. Documents are scored and the highest scored documents are returned to the users. To optimise the query time, a Scatter Gather approach is used.

The Scatter Gather approach involves two phases of the query: Scatter and Gather.

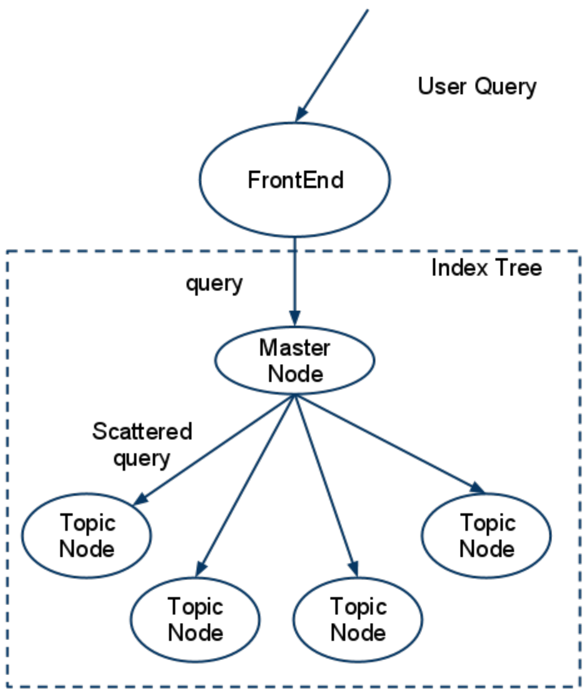

**Figure 9.1 Scatter Phase**

The first phase, Scatter, is when the query is farmed out to a set of sub nodes. Classically, these sub nodes are divided topically and store documents about their topic. These nodes are responsible for finding relevant documents for the query and returning the results.

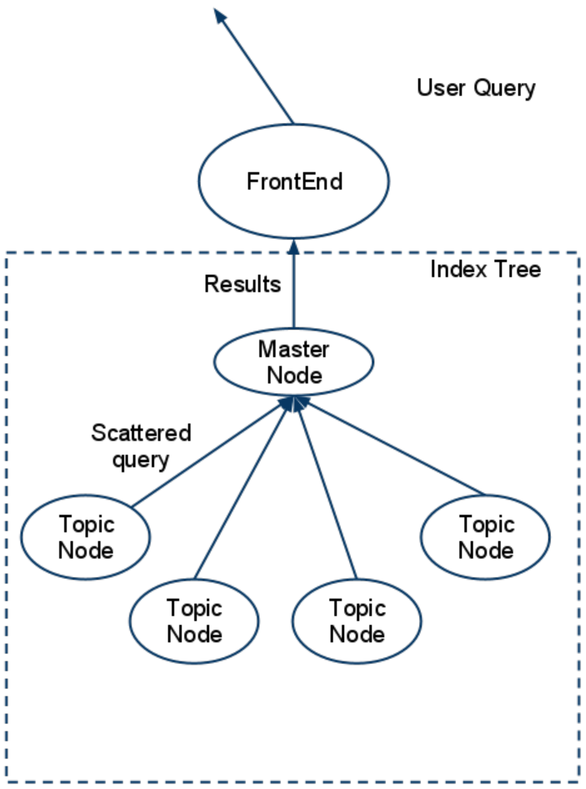

**Figure 9.2 Gather Phase**

The second phase, Gather, is when all the topic nodes respond to the main node with their results. These are then pruned and returned for the entire query.

Let's start by creating a SearchQuery message that can be sent amongs the actors.

```
case class SearchQuery(query : String, maxResults : Int)
```

The `SearchQuery` class has two paramters. The first is the actual query and the second is the maximum number of results that should be returned. Let's implement one of the topic nodes to handle this message.

```
trait SearchNode extends Actor {
  type ScoredDocument = (Double, String)
  val index : HashMap[String, Seq[ScoredDocument]] = ...
  override def act = Actor.loop {
    react {
      case SearchQuery(query, maxResults) =>
        reply index.get(query).getOrElse(Seq()).take(maxResults)
    }
  }
}
```

The Search node defines the type Scored Document to be a tuple of a double score and a string document. The index is defined as a HashMap of query string to scored documents. The index is implemented such that it pulls in a different set of values for each SearchNode created. The full implementation of the index is included in the source code for the book.

The act method on `SearchNode` contains its core behavior. When it receives a `SearchQuery` message, it looks for results in its index. It replies to the sender of the SearchQuery all of these results truncated so that only `maxResults` are returned.

**NOTE** **react vs. receive**
The SearchNode actor uses the `react` method for accepting messages. The actors library also supports a `receive` method. These methods differ in that `react` will defer the execution of the actor until there is a message available. The `receive` method will block the current thread until a message is available. Unlesss absolutely necessary, `receive` should be avoided to improve the paralleism in the system.

Now let's implement the HeadNode actor that's responsible for scattering queries and gathering results.

```
trait HeadNode extends Actor {
  val nodes : Seq[SearchNode] = ...
  override def act = Actor.loop {
    react {
      case s @ SearchQuery(query, maxResults) =>
        val futureResults = nodes map (n => n !! s)
        def combineResults(current : Seq[(Double, String)],
                           next : Seq[(Double, String)]) =
          (current ++ next).view.sortBy(_._1).take(maxDocs).force
        reply futureResults.foldLeft(Seq[ScoredDocument]()) {
          (current, next) =>
            combineResults(current,
                           next().asInstanceOf[Seq[ScoredDocument])
        }
    }
  }
}
```

The HeadNode actor is a bit more complicated. It defines a member containing all the SearchNodes that it can scatter to. It then defines its core behavior in the act method. The HeadNode wait for SearchQuery messages. When it receives one, it sends it to all the SearcNode children awaiting a future result. The !! method on actors will send a message and expect a reply at some future time. This reply is called a `Future`. The `headNode` can block until the reply is received by cally the `apply` method on the `Future`. This is exactly what it does in teh foldLeft over these futures. The `HeadNode` is aggregating the `next` future result with the current query results result to produce the final result list. This final result list is sent to the original query sender using the reply method.

The system now has a Scatter/Gather search tree for optimal searching. However, there is still a lot to be desired. For example, the casting of the result type in the HeadNode actor is less than ideal in a statically-typed language like Scala. Also, the HeadNode blocks for an entire SearchQuery. This means that amount of parallelism in the system could be expanded so that slow-running queries don't starve faster queries. Finally, the search tree has no failure handling currently. In the event of a bad index or query string, the whole system will crash.

Using actors, these downsides can be improved. Let's start with fixing the type-safety issues.

## *9.2 Use typed, transparent references*

One of the biggest dangers in the Scala standard actors library is to give actors references to each other. This can lead to accidentally calling a method defined on another actor instead of sending a message to that actor. While that may seem innocuous to some, this behavior can break an actors system, especially if locking is used. Actors are optimised by minimizing locking to a few minor locations, such as when scheduling and working with a message buffer. Introducing more locking can easily lead to deadlocks and heartache.

Another disadvantage to passing direct references of actors is that of transparency. That is, the location of an actors is 'tied in' to another actor. This locks them in place where they are. The actors can no longer migrate to other locations, either in memory or on the network. This severly limits the ability of a system to handle failure. This will be discussed in detail in section 9.3.

Another downside to sending actors directly, in the scala standard library, is that actors are untyped. This means that all the handy type system utilities one could leverage are thrown out the window when using raw actors. Specifically, the ability of the compiler to find exhausing pattern matches using sealed traits.

**NOTE** **Using sealed traits for message APIs**

It's a 'best practice' in Scala to define message APIs for actors within a sealed trait hierarchy. This has the benefit of defining every message that an actor can handle and keeping them in a central location for easy lookup. With a bit of machinery, the compiler can be coerced to warn when an actor does not handle its complete messaging API.

The Scala standard library provides two mechanisms for enforcing type safety and decoupling references from directly using an actor. They are the `InputChannel` and `OutputChannel` traits.

The `OutputChannel` trait is used to send messages to actors. This is the interface that should be passed to other actors. This interface looks as follows:

```
trait OutputChannel[-Msg] {
  def !(msg: Msg @unique): Unit
  def send(msg: Msg @unique, replyTo: OutputChannel[Any]): Unit
  def forward(msg: Msg @unique): Unit
  def receiver: Actor
}
```

The `OutputChannel` trait is templatized by the type of Messages that can be sent to it. It supports sending messages vai three methods: `!`, `send` and `forward`. The `!` method sends a message to an actor and does not expect a reply. The send method will send a message to an actor and also attaches an outputchannel that the actor can respond to. The forward method is used to send a message to another actor such that the original reply channel is preserved.

The receiver method on `OutputChannel` returns the raw Actor used by the OutputChannel. This method should be avoided.

Notice the methods that `OutputChannel` does *not* have: !! and !? method. In the scala standard library, !! and !? are used to send messages and expect a reply in the current scope. This is done through the creation of an anonymous actor which can receive the response. This anonymous actor is used as the replyTo argument for a send call. The !? method blocks the current thread until a response is received. The `!!` method create a `Future` object. The Future object stores the result when it occurs. Any attempt to retreive the result blocks the current thread until the result is available. `Futures` do provide a `map` method. This attaches a function that can be run on the value in the future when it is available without blocking the current thread.

In general, using !! and !? is discouraged. The potential for deadlocking a thread is great. When used lightly, or with caution they can be very helpful. It's important to understand the size/scope of the project and the type of problem being solved. If the problem is too complex to ensure !! and !? behave appropriately, then it is best to avoid their use altogether.

Let's modify the Scatter Gather example to communicate using OutputChannels.

### *9.2.1 Scatter Gather with OutputChannel*

The scatter gather example requires two changes to promote lightweight typesafe references: Removing the direct Actor references in HeadNode and change the query responses to go through a collection channel. The first change is simple.

```
/** The head node for the scatter/gather algorithm. */
trait HeadNode extends Actor {
  val nodes : Seq[OutputChannel[SearchNodeMessage]]
  override def act : Unit = {
    ...
  }
}
```

The nodes member of the HeadNode actor is changed to be a `Seq[OutputChannel[SearchNodeMessage]]`. This change ensures that the HeadNode will only ever send `SearchNodeMessage` messages to `SearchNode`s. The SearchNodeMessage type is a new sealed trait that will contain all messages that can be send to SearchNodes.

The second change is a bit more involved. Rather than directly responding to the sender of the `SearchQuery`, let's allow an output channel to be passed along with the `SearchQuery` that can receive results.

```
sealed trait SearchNodeMessage
case class SearchQuery(query : String,
                       maxDocs : Int,
                       gatherer : OutputChannel[QueryResponse])
  extends SearchNodeMessage
```

The `SearchQuery` message now has three parameters: The query, the maximum number of results and the output channel that will recieve the query results. The SearchQuery messaage now extends form the SearcNodeMessage. The new SearchNodeMessage trait is sealed, ensuring that all messages that can be sent to the SearcNode are defined in the same file. Let's update the SearchNodes to handle the updated SearchQuery message.

```
trait SearchNode extends Actor {
  lazy val index : HashMap[String, Seq[(Double, String)]] = ...

  override def act = Actor.loop {
    react {
      case SearchQuery(q, maxDocs, requester) =>
        val result = for {
          results <- index.get(q).toList
          resultList <- results
        }  yield resultList
        requester ! QueryResponse(result.take(maxDocs))
    }
  }
}
```

The `SearchNode` trait is the same as before except for the last line in the react call. Instead of calling reply with the `QueryResponse`, the `SearchNode` sends the response to the requestor parameter of the query.

This new behavior means that the head node cannot just send the same `SearchQuery` message to the `SearchNode`s. Let's rework the communication of the system.

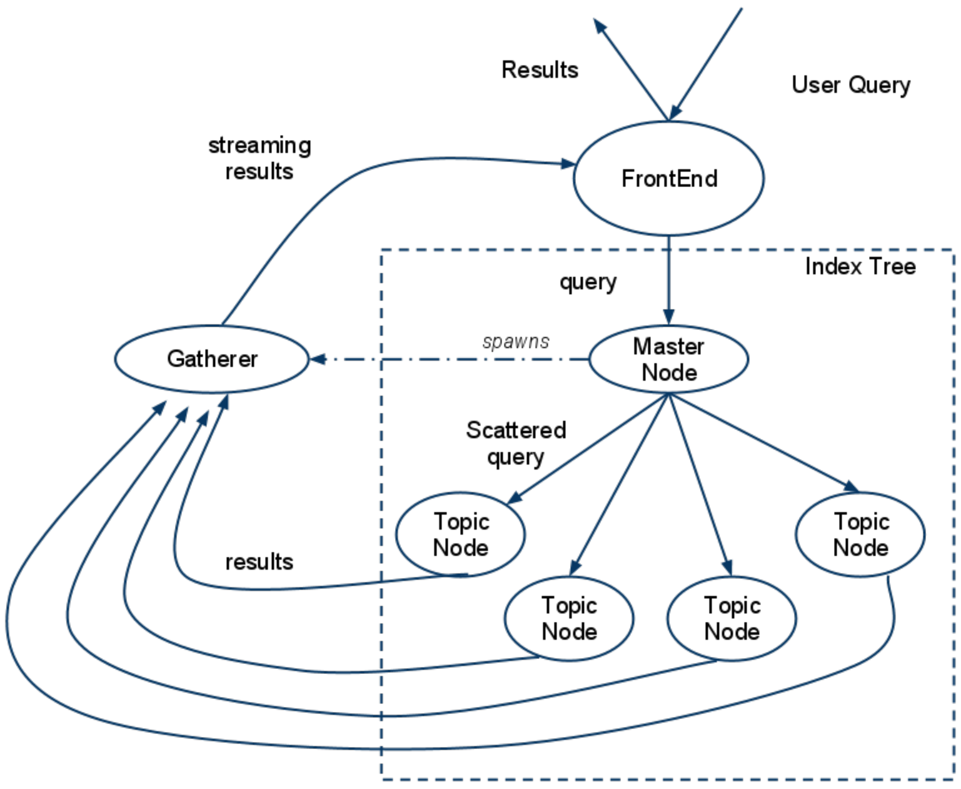

**Figure 9.3 Modified Scatter Gather search**

The new design has a Gatherer actor. This actor is responsible for receiving all results from SearchNodes and aggregating them before sending back to the front end. The Gatherer could be implemented in many ways. One advanced implementation could use prediction to stream results to the front end as they are returned, attempting to ensure high priority results get sent immediately. For now, let's implement the Gatherer node such that it aggregates all results first and then sends them to the front end.

```
// An actor which receives distributed results and aggregates/responds to the origin
trait GathererNode extends Actor {
  val maxDocs : Int
  val maxResponses : Int
  val client : OutputChannel[QueryResponse]
  ..
}
```

The GathererNode is defined as an Actor. It has three members. The maxDocs member is the maximum number of documents to return from a query. The maxResponses member is the maximum number of nodes that can respond before sending resutls for a query. The client member is the OutputChannel where results should be sent. The GathererNode should be tolerant of errors or timeouts in the search tree. To do this, it should wait a maximum of one second for each response before returning the query results. Let's implement the act method for the GathererNode.

```
def act = {
  def combineResults(current : Seq[(Double, String)], next : Seq[(Double, String)])
    (current ++ next).view.sortBy(_._1).take(maxDocs).force

  def bundleResult(curCount : Int, current : Seq[(Double, String)]) : Unit =
    if (curCount < maxResponses) {
      receiveWithin(1000L) {
        case QueryResponse(results) =>
          bundleResult(curCount+1, combineResults(current, results))
        case TIMEOUT =>
          bundleResult(maxResponses, current)
      }
    } else {
      client ! QueryResponse(current)
    }
  bundleResult(0, Seq())
}
```

The act method defines the core behavior of this actor. The combineResults helper method is used to take two sets of query results and aggregate them such that the highest scored results remain. This method also limits the number of results returned to be the same as the maxDocs member variable.

The bundleResult method is the core behavior of this actor. The curCount parameter is the number of responses seen so far. The current parameter is the aggregate of all collected query results from all nodes. The bundleResult method first checks to see if the number of responses is less than the maximum expected results. If so, then it calls `receiveWithin` to wait for another response. The `receiveWithin` method will wait for a given time for messages before sending the special `scala.actors.TIMEOUT` message. If another query result is received, the method combines the result with the previous set of results and recursively calls itself with bumped values. In the event receiving the message times out, the bundleResult method calls itself with the number of responses set to the maximum value. In the event the number of responses is at or above the maximum, the current query results are sent to the client.

Finally, the act method is implemented by calling the bundleResult method with an intial count of zero and an empty `Seq` of results.

The GathererNode stops trying to receive messages after the query results have been sent. This effectively 'ends' the life of the actor and allows the node to become garabage collected. The scala standard library actors library implements its own garbage collection routine that will have to remove references to the GathererNode before the JVM garabage collection can recover memory.

The last piece of implementation required is to adapt the HeadNode to use the GathererNode instead of collecting all the results in futures.

```
trait HeadNode extends Actor {

  val nodes : Seq[OutputChannel[SearchNodeMessage]]

  override def act : Unit = {
    this.react {
      case SearchQuery(q, max, responder) =>
        val gatherer = new GathererNode {
          val maxDocs = max
          val maxResponses = nodes.size
          val client = responder
        }
        gatherer.start
        for (node <- nodes) {
          node ! SearchQuery(q, max, gatherer)
        }
        act
    }
  }
  override def toString = "HeadNode with {\n" +
    "\t" + nodes.size + " search nodes\n" +
    nodes.mkString("\t", "\n\t", "\n}")
}
```

The HeadNode has been changed so that when it receives a SearchQuery it constructs a new GathererNode. The gatherer is instantiated using the parameters from the SearchQuery. The gatherer must also be started so that it can receive messages. The last piece is to send a new SearchQuery message to all the SearchNodes with the OutputChannel set to the gatherer.

Splitting the scatter and gather computations into different actors can help with throughput in the whole system. The HeadNode actor only has to deal with incoming messages and do any potential pre-processing of querys before scattering them. The GathererNode can focus on receiving responses from the search tree. A Gatherer node could even be implemented such that it stopped SearchNodes from performing lookups if enough quality results were received. Most importantly, if there is any kind of error gathering the results of one particular query it will not adversly affect any other query in the system.

This is a key design issue with actors. Failures should be isolated as much as possible. This can be done through the creation of failure zones.

## *9.3 Limit failures to zones*

Architecting and rationalizing distributed architecture can be difficult. Joe Armstrong, the creator of Erlang, popularized the notion of actors and how to handle failure. The recommended strategy for working with actors is to let them fail and let another actor, called a supervisor handle that failure. The supervisor is responsible for bringing the system it manages back into a working state.

Looking at supervisors and actors from a topological point of view, supervisors create 'zones' of failure for the actors they manage. That is, the actors in a system can be partitioned up by the supervisors such that if one section of the system goes down, the supervisor has a chance to prevent the failure from reaching the rest of the system. Each supervisor actor can itself have a supervisor actor, creating nested zones of failure.

The error handling of supervisors is similar to exception handling. A supervisor should handle any failure that it knows how to, and bubble up those it does not to outer processes. If no supervisor is able to handle the error, then this would bring down the entire system, so bubbling up errors should be done carefully!

Supervisors can be simpler to write than exception handling code. With exception handling, it's very difficult to know if a try-catch block contained any state-changing code and whether or not it can be retired. With supervisors, if an actor is misbehaving, then it can restart the portion of the system that is dependent on that actor. Each actor can be passed an initial 'good' state and continue processing messages.

Notice the relationship between the supervisor of an actor and the creator of the actor. If the supervisor needs to recreate an actor upon destruction, the supervisor is also the ideal candidate to start the actor when the system initializes. This allows all the initialization logic to live in the same location. Supervisors may also need to act as 'proxy' to the subsytem they manage. In the event of failure, the supervisor may need to buffer messages to a subsystem until after it has recovered and can begin processing again.

Supervisors are created differently in the various Scala actors libraries. In the core library, supervisors are created through the `link` method. The Akka actors library, designed for larger scale systems, provides many default supervisor implementations and mechanisms of wiring actors and supervisors together. One thing that is common across actors libraries is that supervisors are supported and failure zones are encouraged.

### *9.3.1 Scatter Gather Failure zones*

Let's adapt the Scatter Gather example to include failure zones. The first failure zone should cover the `HeadNode` and `SearchNode` actors. Upon failure, the supervisor can reload a failing search node and wire it back into the head node. The second failure zone should cover the FrontEnd actor and the supervisors of the first failure zone. In the event of failure in this outer zone, the supervisor can restart any failed inner zones and inform the front end of the new actors. Let's take a look at a topological view of this failure handling:

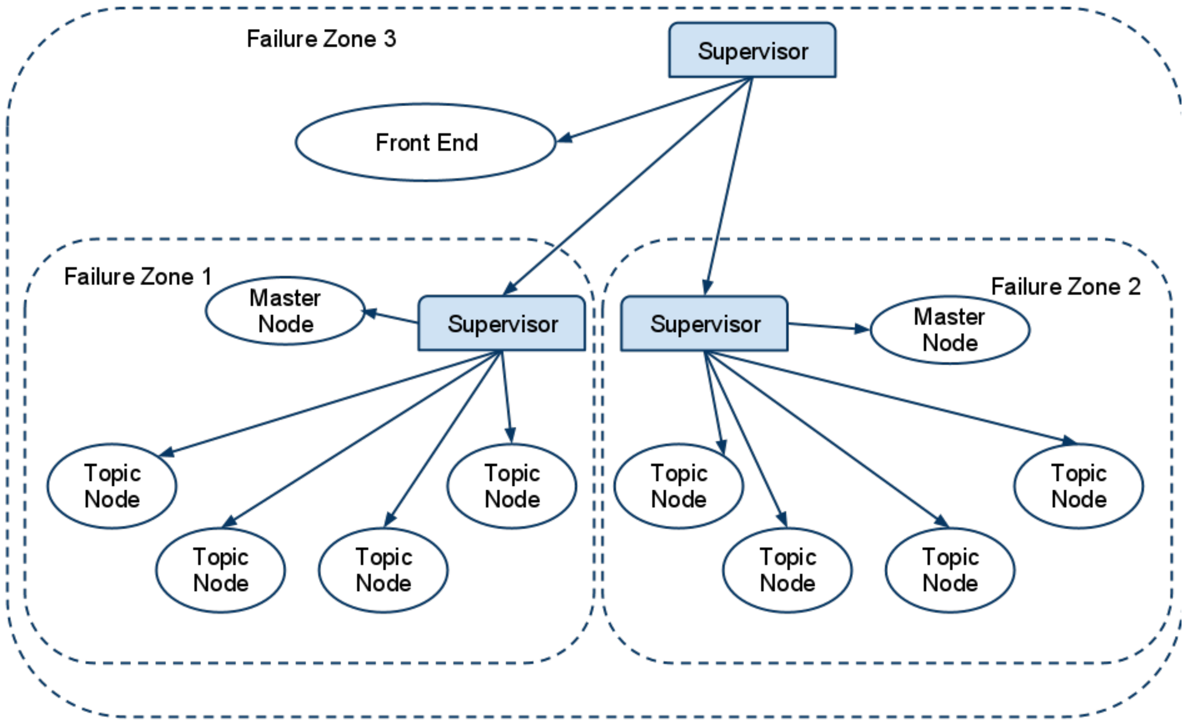

**Figure 9.4 Failure zones for Scatter Gather example**

Failure Zone 1 and 2 in the diagram show the Head Node and SearchNode failure zones for two parallel search hierarchies. The supervisor for these zones is responsible for restarting the entire tree, or a particular SearchNode, on failure. Zones 1 and 2 are each encompassed in Zone 3. This Zone manages the search front end. In the event of failure, it restarts the underlying search trees or the front end as needed.

Let's start by defining the supervisor for the search nodes.

**Listing 9.1 Supervisor for search nodes**

```
trait SearchNodeSupervisor extends Actor {
  val numThreadsForSearchTree = 5

  private def createSearchTree(size : Int) = {
    val searchNodes = for(i <- 1 to size) yield {
      val tmp = new SearchNode {
        override val id = i
      }
      SearchNodeSupervisor.this link tmp
      tmp.start
      tmp
    }
```

**1 Subtree constructor**

**2 Supervise sub-nodes**

```
    val headNode = new HeadNode {
      val nodes = searchNodes
      override val scheduler = s
    }
    this link headNode                                          ❷
    headNode.start
    headNode
  }
  def act() : Unit = {
    trapExit = true                                             ❸ Catch errors on
    def run(head : Actor) : Nothing = react {                     linked actors
                                                                ❹ Wait for messages
      case Exit(deadActor, reason) =>                           ❺ Restart on failure
        run(createSearchTree(10))
      case x =>
        head ! x
        run(head)
    }
    run(createSearchTree(10))
  }
}
```

The SearchNodeSupervisor contains two methods, createSearchTree and act. The createSearchTree is responsible for instantiating nodes of the search tree and returning the top node. This method iterates over the desired size of tree and creates the SearchNode class from the previous examples. Remember that each SearchNode uses their assigned id to load a set indexed documents and make them available for queries. Each search node created is linked to the supervisor. In the scala standard library actors, linking is what creates a supervisor hierarchy. Linking two actors means that if one fails, both are killed. It also allows one of them to trap errors from the other. This is done from the call to trapExit = true in the act method.

**NOTE** **Common linking pitfalls**

The `link` method has two restrictions that simplify it use.

- It must be called from inside a 'live' actor. That is from the act method or one of the continuations passed to react.
- It should be called on the supervisor with the other actor as the method argument.

Because link alters the behavior of failure handling, it needs to lock both actors it operates against. Because of this synchronization, it is possible to deadlock when waiting for locks. Therefore ordering the locking behavior can prevent this behavior. The link method also requires, through runtime asserts, that it is called against the current 'live' actor. That is, the actor must be actively running in its scheduled thread. This means linking should not be done external to the supervisor actor, but internal. This is why all the topological code is pushed down into the supervisor and why it acts as a natural proxy to the actors it manages.

The second method is the standard library actor's act method. This defines the core behavior of the Supervisor actor. The first line here is the trapExit = true, which allows this actor to catch errors from others. The next line is a helper function called run. The run function accepts one parameter, the current head actor. The run function calls react, which will block waiting for messages. The first message it handles is the special Exit message. An Exit message is passed if one of the linked actors fails. Notice that values that come with an Exit message: deadActor and reason. The deadActor link allows the supervisor to attempt to pull any partial state from the deadActor if needed, or remove it from any control structures as needed. Note that the deadActor is already gone and will not be scheduled anymore at the time of receiving this message.

For the SearchNodeSupervisor, when handling errors the entire search tree is reconstructed and passed back into the run method. This may not be ideal in a real life situation because reconstructing the entire tree could be expensive or the tree might be sprawled over several machines. In that case, the SearchNodeSupervisor could just restart the failed node and notify the search tree of the replacement.

If any other message is encountered by the SearchNodeSupervisor it is forwarded to the current HeadNode. This means that the supervisor itself can block incoming messages when restarting the system. When the main node crashes, the

Supervisor receives the Exit message and stops processing messages while it fixes the system. After restoring things, it will again pull messages from its queue and delegate them down to the search tree.

### *9.3.2 General Failure Handling Practices*

The supervisor for the Scatter Gatther search system demonstrates very simply ways to handle the issues of failure in an actors system. When designing an actors based system and outlining failure zones, the following list helps make decisions appropriate for that module:

**Table 9.1 Actor Design Decisions**

| Decision | Scatter Gather Example | Other Options |
|---|---|---|
| Providing transparent way to restart failed components | Forward messages through the supervisor. If supervisor fails, restart outer failure zone. | Update nameserivce with references to actors.<br><br>Directly communicate new location to connected components. |
| Granularity of failure zones | The entire search tree fails and restarts. | Single Search node inner failure zone with Search Tree outer failure zone. |
| Recovery of failed actor state | Actor data is statically pulled from disk. Does not change during its lifetime. | Periodic snapshoting to persistent store<br><br>Pulling 'live' state from dead actor and 'sanitizing'<br><br>Persisting state after every handled message |

These three decisions are crucial in defining robust concurrent actor systems. The first point is the most important. Creating a fail safe zone implies insuring that if that zone were to crash and restart it should not affect external zones. The scala actors library makes it very easy to loose transparency for actors. This can be done by passing the reference to a specific actor rather than a proxy or namespace reference.

The second decision can affect the messaging API for actors. If a subsystem needs to tolerate failure of one of its actors, then the other actors need to be updated to communicate with the replacement actor. Again, transparent actor references can be a boon here. For the scala standard library, using the supervisors as proxies to sub-components is the simplest way to provide transparency. This means that for fine-grained failure zones, many supervisors must be created, possibly one per actor.

The third decision is one not discussed in the example, that of state recovery. Most real life actors maintain some form of state during their lifetimes. This state may or may not need to be reconstructed for the system to continue functioning. Although not directly supported in the Scala standard library, one way to ensure state sticks around would be to periodically snapshot the actor by dumping its state to a persistent store. This could then be recovered later.

A second method of keeeping state would be to pull the last known state from a dead actor and 'sanitizing' it for the reconstructed actor. This method is risky as the state of a previous actor is not in a consistent state and the sanitization process may not be able to recover. The sanitization process could also be very hard to reason through and write. This mechanism is not recommended.

Another mechanism for handling state is to persist the state after every message an actor receives. While not directly supported by the scala standard library, this could easily be added through a subclass of actor.

**NOTE** **Akka Transactors**

The Akka actors library provides many ways to synchronize the state of live actors, one of which is Transactors. Transactors are actors whose message handling functions are executed within a transactional context and whose state is persisted after every message.

There's one item not on this list and that is threading strategies. Because actors share threads, an actor that is failing to handle its incoming messages could ruin the performance of other actors that share the same threading resources. The solution to this is to split actors into scheduling zones, similar to splitting them into failure zones.

## *9.4 Limit overload using Scheduler Zones*

One type of failure that a supervisor cannot handle well is thread starvation of actors. If one actor is receiving a lot of messages and spending a lot of CPU time processing them, it can starve other actors. The actor schedulers also don't have any notion of priority. There may be a high-priority actor in the system that must respond as quickly as possible. This actor could get bogged down by lower priority actors stealing all the resources.

Schedulers are the solution to this problem. A scheduler is the component responsible for 'sharing' actors amongst threads. The scheduler selects the next actor to run and assigns it to a particular thread. In the Scala actors library, a scheduler implements the IScheduler interface.

There are also a variety of scheduling mechanisms available for the standard library actors. Here's a table of a few key schedulers:

**Table 9.2 Schedulers**

| **Scheduler** | **Purpose** |
|---|---|
| ForkJoinScheduler | Parallelization optimized for tasks that are split up, parallelized and recovered. In other words, things that are forked for processing then joined together. |
| ResizableThreadPoolScheduler | Starts up a persistent thread pool for actors. If load is increased, it will automatically create new threads up to an environment-specified limit. |
| ExecutorScheduler | Uses a `java.util.concurrent.Executor` to schedule actors. This allows actors to use any of the standard java thread pools. This is the recommended way to assign fixed size thread pool. |

The `ForkJoinScheduler` is the default scheduler for scala actors. This is done through a nifty work-stealing algorithm where every thread has its own scheduler. Tasks created in a thread are added to its own scheduler. If a thread runs out of tasks, it steals work from another thread's scheduler. This provides great performance for a lot of situations. The Scatter Gather example is perfect fit for fork join. Queries are distributed to each SearchNode for executions and results are aggregated to create the final query results. The work-stealing pulls and dsitributes the 'forked' work for a query. If the system is bogged down, it could degrade to performing similarly to a single threaded query engine. While generally efficient, the ForkJoinScheduler is not optimal in situtations where task sizes are largely variable.

The `ResizableThreadPoolScheduler` constructs a pool of threads which share the processing of message for a set of Actors. Scheduling is done on a first come, first serve basis. If the work load starts to grow beyond what the current thread pool can handle, the scheduler will increase the available threads in the pool up until a maximum pool size. This can help a system handle a large increase in messaging throughput and back-off resources during downtime.

The `ExecutorScheduler` is a scheduler which defers scheduling actors to a `java.util.Executor` service. There are many implementations of `java.util.Executor` in the java standard library as well as some common alternatives. One of these, from my own codebases, was an `Executor` which would schedule tasks on the AWT rendering thread. Using this scheduler for an actor guarantees that it handles messages within a GUI context. This allowed the creation of GUIs where actors could be used to respond to backend events and update UI state.

Each of these schedulers may be appropriate to one or more components in a system. Not only that, some components scheduling may need to be completely isolated from other components. This is why Scheduling Zones are important.

### *9.4.1 Scheduling Zones*

Scheduling Zones are groupings of actors that share the same scheduler. Just as failure zones isolate failure recovery, so do scheduling zones isolate starvation and contention of subsystems. Not only that, scheduling zones can optimise the scheduler to the component.

Let's take a look at what a scheduling zone design might be for the Scatter Gather example:

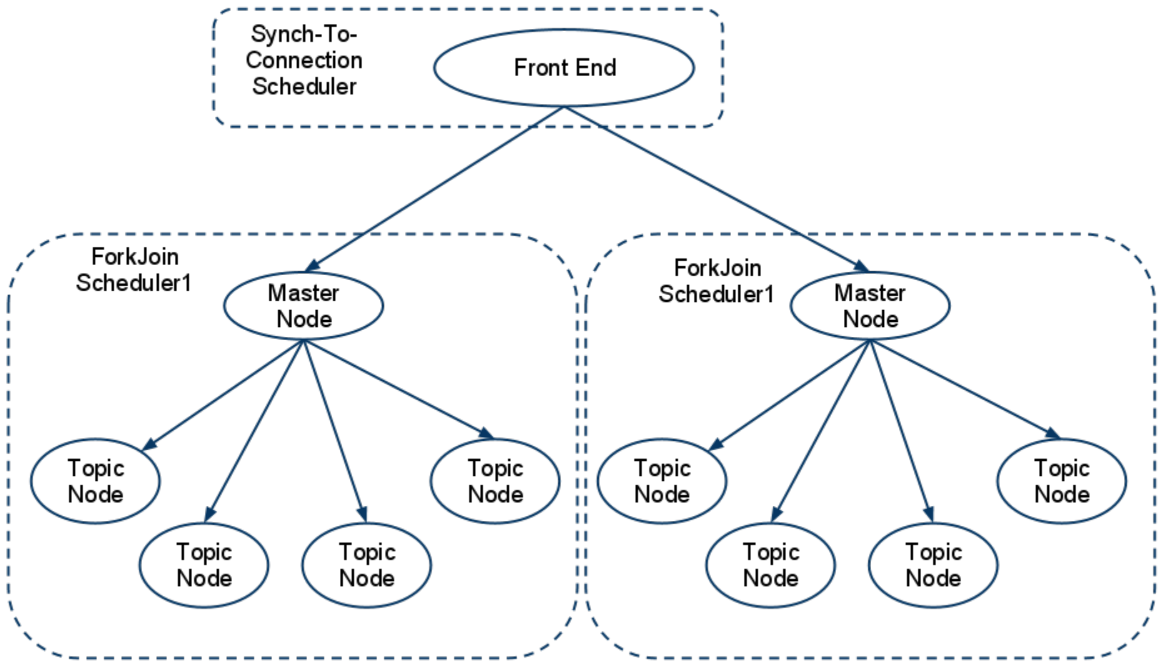

**Figure 9.5 Scatter Gather Scheduling Zones**

The Scatter Gather search service can be split into four scheduling zones: Search Tree 1, Search Tree 2, Front End and Supervisor

The first scheduling zone handles all actors in a search tree. The ForkJoinScheduler is optimized for the same behavior as the scatter gather algorithm, so it makes an ideal choice of scheduler for this zone. The replicated Searc tree uses its own ForkJoinScheduler to isolate failures and load between the two trees.

The Front End scheduling zone uses a customized scheduler which ties its execution to an asynchronous HTTP server. That is, the handling of messages is done on the same thread as input is taken and the results are streamed back into the appropriate socket using one of the front-end threads. These actors could also use there own thread pool. This would be ideal if the HTTP server accepting incoming connections used a thread pool of the same size.

The last scheduling zone, not shown, is the scheduling of erorr recovery. Out of habit, I tend to place these on a separate scheduling routine so they don't interfer with any other subcomponent. This isn't strictly necessary. Error recovery, when it happens, is the highest priority task for a given subcomponent and should not steal more important work from other threads. However, if more than one subcomponent is sharing the same scheduling zone, then I prefer to keep recovery work separate from 'core' work.

Let's add scheduling zones to the Scatter Gather search tree example. The only changes required are in the constructor function defined on the supervisor. Let's take a look:

```
private def createSearchTree(size : Int) = {
    val numProcessors =
      java.lang.Runtime.getRuntime.availableProcessors
    val s = new ForkJoinScheduler(
      initCoreSize = numProcessors,
      maxSize = numThreadsForSearchTree,
      daemon = false, fair = true)
    val searchNodes = for(i <- 1 to size) yield new SearchNode {
      override val id = i
      override val scheduler = s
    }
    searchNodes foreach this.link
    searchNodes.foreach(_.start)
    val headNode = new HeadNode {
      val nodes = searchNodes
      override val scheduler = s
    }
    this link headNode
    headNode.start
    headNode
  }
```

The original code has two new additons. The first is the creation of the ForkJoinScheduler. This scheduler takes four arguments. The initCoreSize and maxSize arguments are the minimum and maximum number of threads it should store in its thread pool. The daemon argument specifies whether or not threads should be constructed as daemons. This scheduler has the ability to shut itself down if the actors within are no longer performing any work. The last argument is whether or not the scheduler should attempt to enforce fairness in the work-stealing algorithm.

The second additions are the overriden scheduler member of the SearchNode and HeadNode actors. This override causes the actor to use the new scheduler for all of its behavior. This can only be done at creation time, so the scheduling zones must be known a-priori.

That's it, the actors are now operating within their own fork-join pool, isolated from load in other actors.

## 9.5 Conclussion

Actors provide a simpler parallelization model than traditional locking and threading. A well behaved actors system can be fault tolerant and resistant to total system slowdown. Actors provide an excellent abstraction for designing high performance servers, where throughput and uptime are of the utmost improtance. For these systems, designing failure zones and failuring handling behaviors can help keep a system running even in the event of critical failures. Splitting actors into Scheduling zones can ensure that input overload to any one portion of the system will not bring the rest of the system down. Finally, when designing with actors, it is recommended to use the Akka library for large scale system.

The Akka library differs from the standard library in a few key areas:

- Clients of an actor can never obtain a direct reference to that actor. This drastically simplifies scaling an Akka system to multiple servers since there is no chance an actor requires the direct reference to another.
- Messages are handles in the order received. If the current message handling routine cannot handle an input message, it is dropped (or hanlded by the unknown message handler). This prevents out of memory errors due to message buffers filling up.
- All core actors library code is designed to allow user code to handle failures without causing more. For example, Akka goes to great lengths to avoid causing out of memory exceptions within the core library. This allows user code, your code, to handle failures as needed.
- Akka provides most of the basic supervisor behaviors that can be used as building blocks for complex supervision strategies.
- Akka provides several means of persisting state "out of the box".

So, while the Scala actors library is an excellent resource for creating small to medium size actors applications, the Akka library provides the features needed to make a large-scale application.

Actors and Actor-related system design is a rich subject. This chapter lightly covered a few of the key aspects to actor-related design. These should be enough to create a fault-tolerant high-performant actors system.

Next, let's look into a topic of great interest: Java interoperability with Scala.