

Table of Contents

Introduction	1.1
Introduction	1.2
What is Akka.NET	1.2.1
What are actors	1.2.2
Library and Modules	1.2.3
Top-level Architecture	1.2.4
The Device Actor	1.2.5
Device Groups and Manager	1.2.6
Querying Group of Devices	1.2.7
Use-case and Deployment	1.2.8
Akka.NET Users and Use Cases	1.2.9
Concepts - Terminology	1.3
Actor Systems	1.3.1
What is an Actor	1.3.2
Supervision and Monitoring	1.3.3
Actor References, Paths and Addresses	1.3.4
Location Transparency	1.3.5
Message Delivery Reliability	1.3.6
Configuration	1.3.7
Actors - ReceivActor API	1.4
UntypedActor API	1.4.1
Routers	1.4.2
Dispatcher	1.4.3
Mailboxes	1.4.4
Inbox	1.4.5
Finite State Machines	1.4.6
Fault Tolerance	1.4.7
Dependency Injection	1.4.8
Testing Actor System	1.4.9
Coordinated Shutdown	1.4.10
Persistence - Architecture	1.5
Event sourcing	1.5.1
Persistence Views	1.5.2
Snapshot	1.5.3
At-Least-Once Delivery	1.5.4
Event Adapter	1.5.5
Persistent FSM	1.5.6

Storage plugins	1.5.7
Custom serialization	1.5.8
Persistence Query	1.5.9
Streams - introduction	1.6
Stream Quickstart Guide	1.6.1
Reactive Tweets	1.6.2
Design Principles	1.6.3
Basics and working with Flows	1.6.4
Working with Graphs	1.6.5
Modularity, Composition and Hierarchy	1.6.6
Buffers and working with rate	1.6.7
Dynamic stream handling	1.6.8
Custom stream processing	1.6.9
Integration	1.6.10
Error Handling in Streams	1.6.11
Working with streaming IO	1.6.12
Pipeline and Parallelism	1.6.13
Testing Streams	1.6.14
Overview of built-in stages and their semantics	1.6.15
Streams Cookbook	1.6.16
Configuration	1.6.17
Networking - I/O	1.7
Serialization	1.7.1
Multi-Node TestKit	1.7.2
Remoting	1.8
Transports	1.8.1
Remote Messaging	1.8.2
Deploying Actors Remotely	1.8.3
Network Failure Handling	1.8.4
Network Security	1.8.5
Clustering	1.9
Cluster Routing	1.9.1
Cluster Configuration	1.9.2
Cluster Actor System Extension	1.9.3
Cluster Singleton	1.9.4
Distributed Pub/Sub	1.9.5
Cluster Client	1.9.6
Custer Sharding	1.9.7
Distributed Data	1.9.8
Split Brain Resolver	1.9.9

Utilities - Event Bus	1.10
Logging	1.10.1
Serilog	1.10.2
Scheduler	1.10.3
Circuit Breaker	1.10.4
May Change	1.10.5
Configurations - Akka	1.11
Akka.Remote	1.11.1
Akka.Cluster	1.11.2
Akka.Persistence	1.11.3
Akka.Streams	1.11.4
Akka.TestKit	1.11.5
Community - books	1.12
Building Akka.Net	1.12.1
Online Resource	1.12.2
API Changes	1.12.3
Documentation Guideline	1.12.4
Contributor Guideline	1.12.5

Akka.NET



Builds status

	Windows	Linux (Mono)
Build	TC build success	TC build success
Unit Tests	TC build success	TC build success
MultiNode Tests	TC build success	
Perf Tests	TC build failed	

Akka.NET Current Roadmap

Akka.NET is a community-driven port of the popular Java/Scala framework Akka to .NET.

- Subscribe to the Akka.NET dev feed: [@AkkaDotNet](https://twitter.com/AkkaDotNet)
- Gitter chat: <https://gitter.im/akkadotnet/akka.net>
- Support forum: <https://groups.google.com/forum/#!forum/akkadotnet-user-list>
- Mail: hi@getakka.net
- Stack Overflow: <http://stackoverflow.com/questions/tagged/akka.net>

Documentation and resources

Akka.NET Community Site

Install Akka.NET via NuGet

If you want to include Akka.NET in your project, you can [install it directly from NuGet](#)

To install Akka.NET Distributed Actor Framework, run the following command in the Package Manager Console

```
PM> Install-Package Akka
PM> Install-Package Akka.Remote
```

And if you need F# support:

```
PM> Install-Package Akka.FSharp
```

Contributing

Where Can I Contribute?

All contributions are welcome! Please consider the [issues categorized in the Help! column](#) first, as they are areas we could really use your help :)

Contribution Guidelines

If you are interested in helping porting Akka to .NET please take a look at [Contributing to Akka.NET](#).

Our [docs](#) are always a work in progress—to contribute to docs, please see the [docs contribution guidelines here](#).

Builds

Please see [Building Akka.NET](#).

To access unstable nightly builds, please [see the instructions here](#).

Support



What is Akka.NET?

Welcome to Akka.NET, a set of open-source libraries for designing scalable, resilient systems that span processor cores and networks. Akka allows you to focus on meeting business needs instead of writing low-level code to provide reliable behavior, fault tolerance, and high performance.

Common practices and programming models do not address important challenges inherent in designing systems for modern computer architectures. To be successful, distributed systems must cope in an environment where components crash without responding, messages get lost without a trace on the wire, and network latency fluctuates. These problems occur regularly in carefully managed intra-datacenter environments - even more so in virtualized architectures.

To deal with these realities, Akka.NET provides:

- Multi-threaded behavior without the use of low-level concurrency constructs like atomics or locks. You do not even need to think about memory visibility issues.
- Transparent remote communication between systems and their components. You do not need to write or maintain difficult networking code.
- A clustered, high-availability architecture that is elastic, scales in or out, on demand.

All of these features are available through a uniform programming model: Akka.NET exploits the actor model to provide a level of abstraction that makes it easier to write correct concurrent, parallel and distributed systems. The actor model spans the set of Akka.NET libraries, providing you with a consistent way of understanding and using them. Thus, Akka.NET offers a depth of integration that you cannot achieve by picking libraries to solve individual problems and trying to piece them together.

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

What is the Actor Model?

The characteristics of today's computing environments are vastly different from the ones in use when the programming models of yesterday were conceived. Actors were invented decades ago by [Carl Hewitt](#). But relatively recently, their applicability to the challenges of modern computing systems has been recognized and proved to be effective.

The actor model provides an abstraction that allows you to think about your code in terms of communication, not unlike people in a large organization. The basic characteristic of actors is that they model the world as stateful entities communicating with each other by explicit message passing.

As computational entities, actors have these characteristics:

- They communicate with asynchronous messaging instead of method calls
- They manage their own state
- When responding to a message, they can:
 - Create other (child) actors
 - Send messages to other actors
 - Stop (child) actors or themselves

What problems does the actor model solve?

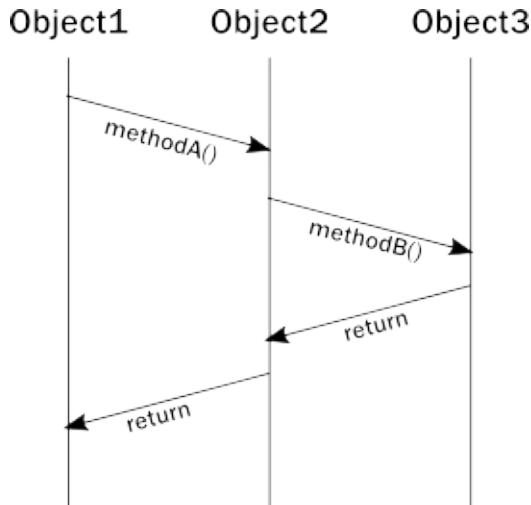
Akka.NET uses the actor model to overcome the limitations of traditional object-oriented programming models and meet the unique challenges of highly distributed systems. To fully understand why the actor model is necessary, it helps to identify mismatches between traditional approaches to programming and the realities of concurrent and distributed computing.

The illusion of encapsulation

Object-oriented programming (OOP) is a widely-accepted, familiar programming model. One of its core pillars is *encapsulation*. Encapsulation dictates that the internal data of an object is not accessible directly from the outside; it can only be modified by invoking a set of curated methods. The object is responsible for exposing safe operations that protect the invariant nature of its encapsulated data.

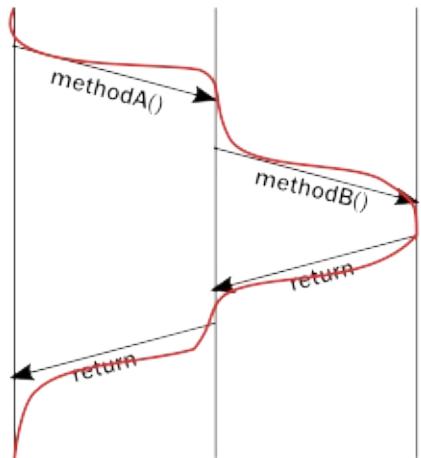
For example, operations on an ordered binary tree implementation must not allow violation of the tree ordering invariant. Callers expect the ordering to be intact and when querying the tree for a certain piece of data, they need to be able to rely on this constraint.

When we analyze OOP runtime behavior, we sometimes draw a message sequence chart showing the interactions of method calls. For example:



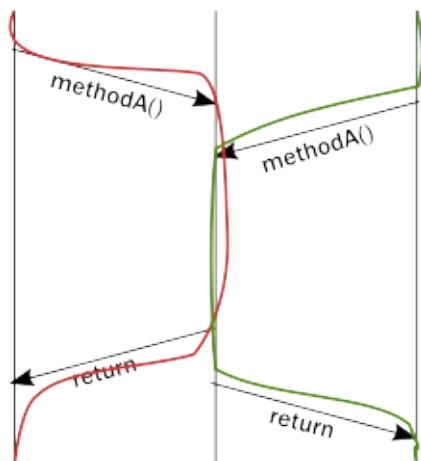
Unfortunately, the above diagram does not accurately represent the *lifelines* of the instances during execution. In reality, a *thread* executes all these calls, and the enforcement of invariants occurs on the same thread from which the method was called. Updating the diagram with the thread of execution, it looks like this:

Object1 Object2 Object3



The significance of this clarification becomes clear when you try to model what happens with *multiple threads*. Suddenly, our neatly drawn diagram becomes inadequate. We can try to illustrate multiple threads accessing the same instance:

Object1 Object2 Object3



There is a section of execution where two threads enter the same method. Unfortunately, the encapsulation model of objects does not guarantee anything about what happens in that section. Instructions of the two invocations can be interleaved in arbitrary ways which eliminate any hope for keeping the invariants intact without some type of coordination between two threads. Now, imagine this issue compounded by the existence of many threads.

The common approach to solving this problem is to add a lock around these methods. While this ensures that at most one thread will enter the method at any given time, this is a very costly strategy:

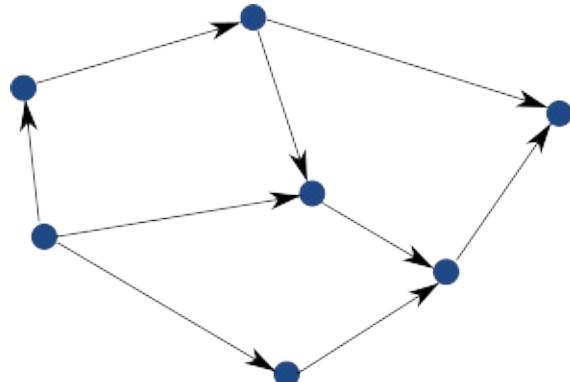
- Locks *seriously limit* concurrency, they are very costly on modern CPU architectures, requiring heavy-lifting from the operating system to suspend the thread and restore it later.
- The caller thread is now blocked, so it cannot do any other meaningful work. Even in desktop applications this is unacceptable, we want to keep user-facing parts of applications (its UI) to be responsive even when a long background job is running. In the backend, blocking is outright wasteful. One might think that this can be compensated by launching new threads, but threads are also a costly abstraction.
- Locks introduce a new menace: deadlocks.

These realities result in a no-win situation:

- Without sufficient locks, the state gets corrupted.
- With many locks in place, performance suffers and very easily leads to deadlocks.

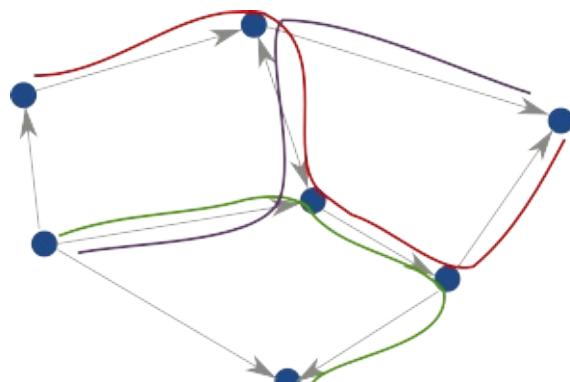
Additionally, locks only really work well locally. When it comes to coordinating across multiple machines, the only alternative is distributed locks. Unfortunately, distributed locks are several magnitudes less efficient than local locks and usually impose a hard limit on scaling out. Distributed lock protocols require several communication round-trips over the network across multiple machines, so latency goes through the roof.

In Object Oriented languages we rarely think about threads or linear execution paths in general. We often envision a system as a network of object instances that react to method calls, modify their internal state, then communicate with each other via method calls driving the whole application state forward:



Objects interacting with each other
by calling methods on each other

However, in a multi-threaded distributed environment, what actually happens is that threads "traverse" this network of object instances by following method calls. As a result, threads are what really drive execution:



Objects interacting with each other
Threads A, B, C, interacting with each other,
traversing method calls on objects

In summary:

- 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 illusion of shared memory on modern computer architectures

Programming models of the 80'-90's conceptualize that writing to a variable means writing to a memory location directly (which somewhat muddies the water that local variables might exist only in registers). On modern architectures - if we simplify things a bit - CPUs are writing to [cache lines](#) instead of writing to memory directly. Most of

these caches are local to the CPU core, that is, writes by one core are not visible by another. In order to make local changes visible to another core, and hence to another thread, the cache line needs to be shipped to the other core's cache.

On the JVM, we have to explicitly denote memory locations to be shared across threads by using `volatile` markers or `Atomic` wrappers. Otherwise, we can access them only in a locked section. Why don't we just mark all variables as `volatile`? Because shipping cache lines across cores is a very costly operation! Doing so would implicitly stall the cores involved from doing additional work, and result in bottlenecks on the cache coherence protocol (the protocol CPUs use to transfer cache lines between main memory and other CPUs). The result is magnitudes of slowdown.

Even for developers aware of this situation, figuring out which memory locations should be marked as `volatile`, or which atomic structures to use is a dark art.

In summary:

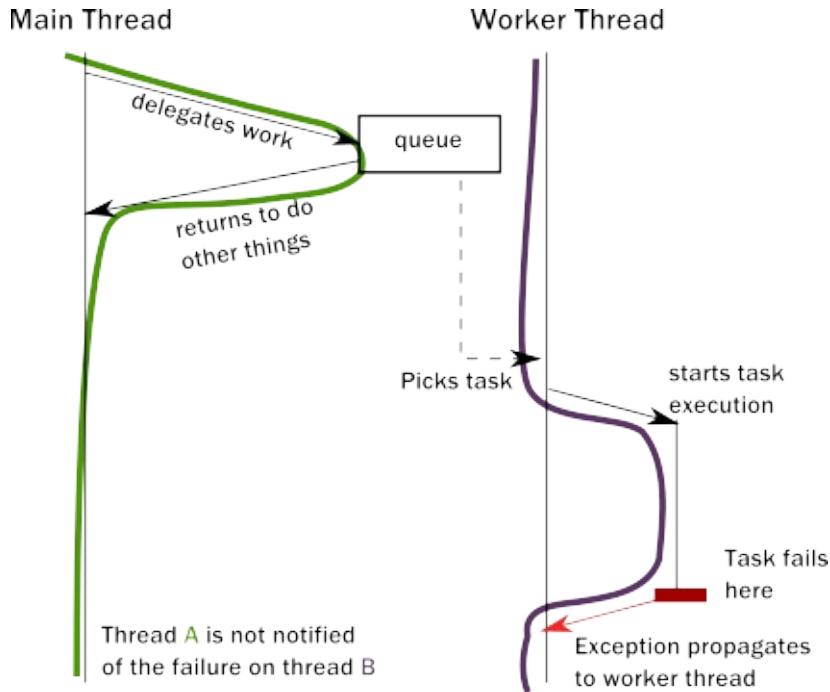
- There is no real shared memory anymore, CPU cores pass chunks of data (cache lines) explicitly to each other just as computers on a network do. Inter-CPU communication and network communication have more in common than many realize. Passing messages is the norm now be it across CPUs or networked computers.
- Instead of hiding the message passing aspect through variables marked as shared or using atomic data structures, a more disciplined and principled approach is to keep state local to a concurrent entity and propagate data or events between concurrent entities explicitly via messages.

The illusion of a call stack

Today, we often take call stacks for granted. But, they were invented in an era where concurrent programming was not as important because multi-CPU systems were not common. Call stacks do not cross threads and hence, do not model asynchronous call chains.

The problem arises when a thread intends to delegate a task to the "background". In practice, this really means delegating to another thread. This cannot be a simple method/function call because calls are strictly local to the thread. What usually happens, is that the "caller" puts an object into a memory location shared by a worker thread ("callee"), which in turn, picks it up in some event loop. This allows the "caller" thread to move on and do other tasks.

The first issue is, how can the "caller" be notified of the completion of the task? But a more serious issue arises when a task fails with an exception. Where does the exception propagate to? It will propagate to the exception handler of the worker thread completely ignoring who the actual "caller" was:



This is a serious problem. How does the worker thread deal with the situation? It likely cannot fix the issue as it is usually oblivious of the purpose of the failed task. The "caller" thread needs to be notified somehow, but there is no call stack to unwind with an exception. Failure notification can only be done via a side-channel, for example putting an error code where the "caller" thread otherwise expects the result once ready. If this notification is not in place, the "caller" 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.**

This bad situation gets worse when things go really wrong and a worker backed by a thread encounters a bug and ends up in an unrecoverable situation. For example, an internal exception caused by a bug bubbles up to the root of the thread and makes the thread shut down. This immediately raises the question, who should restart the normal operation of the service hosted by the thread, and how should it be restored to a known-good state? At first glance, this might seem manageable, but we are suddenly faced by a new, unexpected phenomena: the actual task, that the thread was currently working on, is no longer in the shared memory location where tasks are taken from (usually a queue). In fact, due to the exception reaching to the top, unwinding all of the call stack, the task state is fully lost! **We have lost a message even though this is local communication with no networking involved (where message losses are to be expected).**

In summary:

- To achieve any meaningful concurrency and performance on current systems, threads must delegate tasks among each other in an efficient way without blocking. With this style of task-delegating concurrency (and even more so with networked/distributed computing) call stack-based error handling breaks down and new, explicit error signaling mechanisms need to be introduced. Failures become part of the domain model.
- Concurrent systems with work delegation needs to handle service faults and have principled means to recover from them. Clients of such services need to be aware that tasks/messages might get lost during restarts. Even if loss does not happen, a response might be delayed arbitrarily due to previously enqueued tasks (a long queue), delays caused by garbage collection, etc. In face of these, concurrent systems should handle response deadlines in the form of timeouts, just like networked/distributed systems.

How the actor model meets the needs of concurrent, distributed systems

As described in the sections above, common programming practices cannot properly address the needs of modern concurrent and distributed systems. Thankfully, we don't need to scrap everything we know. Instead, the actor model addresses these shortcomings in a principled way, allowing systems to behave in a way that better matches our mental model.

In particular, we would like to:

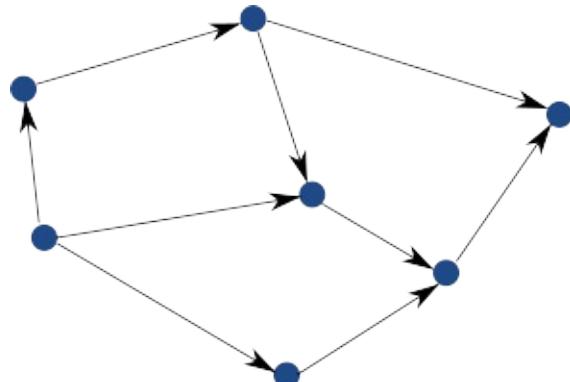
- Enforce encapsulation without resorting to locks.
- Use the model of cooperative entities reacting to signals, changing state and sending signals to each other to drive the whole application forward.
- Stop worrying about an executing mechanism which is a mismatch to our world view.

The actor model accomplishes all of these goals. The following topics describe how.

Usage of message passing avoids locking and blocking

Instead of calling methods, actors send messages to each other. Sending a message does not transfer the thread of execution from the sender to the destination. An actor can send a message and continue without blocking. It can, therefore, do more work, send and receive messages.

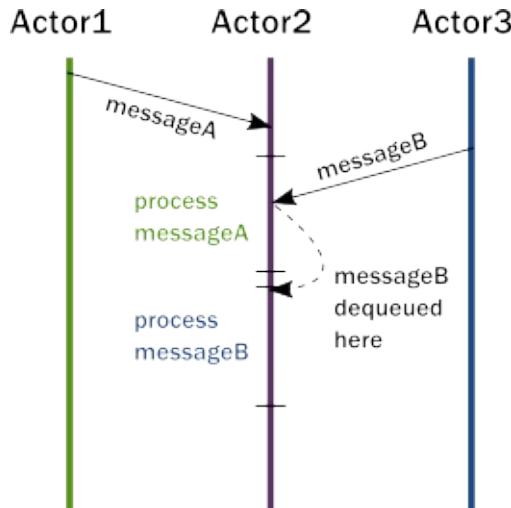
With objects, when a method returns, it releases control of its executing thread. In this respect, actors behave much like objects, they react to messages and return execution when they finish processing the current message. In this way, actors actually achieve the execution we imagined for objects:



Actors interacting with each other by sending messages to each other

An important difference of passing messages instead of calling methods is that messages have no return value. By sending a message, an actor delegates work to another actor. As we saw in [The illusion of a call stack](#), if it expected a return value, the sending actor would either need to block or to execute the other actor's work on the same thread. Instead, the receiving actor delivers the results in a reply message.

The second key change we need in our model is to reinstate encapsulation. Actors react to messages just like objects "react" to methods invoked on them. The difference is that instead of multiple threads "protruding" into our actor and wreaking havoc to internal state and invariants, actors execute independently from the senders of a message, and they react to incoming messages sequentially, one at a time. While each actor processes messages sent to it sequentially, different actors work concurrently with each other so an actor system can process as many messages simultaneously as many processor cores are available on the machine. Since there is always at most one message being processed per actor the invariants of an actor can be kept without synchronization. This happens automatically without using locks:



In summary, this is 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 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 are put into so-called Mailboxes of Actors. 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.

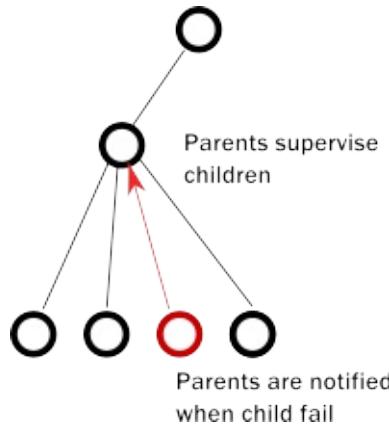
This is a very simple model and it solves the issues enumerated previously:

- Encapsulation is preserved by decoupling execution from signaling (method calls transfer execution, message passing does not).
- There is no need for locks. Modifying the internal state of an actor is only possible via messages, which are processed one at a time eliminating races when trying to keep invariants.
- There are no locks used anywhere, and senders are not blocked. Millions of actors can be efficiently scheduled on a dozen of threads reaching the full potential of modern CPUs. Task delegation is the natural mode of operation for actors.
- State of actors is local and not shared, changes and data is propagated via messages, which maps to how modern memory hierarchy actually works. In many cases, this means transferring over only the cache lines that contain the data in the message while keeping local state and data cached at the original core. The same model maps exactly to remote communication where the state is kept in the RAM of machines and changes/data is propagated over the network as packets.

Actors handle error situations gracefully

Since we have no longer 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.NET 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.NET.



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.

Akka.NET Libraries and Modules

Before we delve further into writing our first actors, we should stop for a moment and look at the set of libraries that come out-of-the-box. This will help you identify which modules and libraries provide the functionality you want to use in your system.

Actors (Akka Library, the Core)

The use of actors across Akka.NET libraries provides a consistent, integrated model that relieves you from individually solving the challenges that arise in concurrent or distributed system design. From a birds-eye view, actors are a programming paradigm that takes encapsulation, one of the pillars of OOP, to its extreme. Unlike objects, actors encapsulate not only their state but their execution. Communication with actors is not via method calls but by passing messages. While this difference may seem minor, it is actually what allows us to break clean from the limitations of OOP when it comes to concurrency and remote communication. Don't worry if this description feels too high level to fully grasp yet, in the next chapter we will explain actors in detail. For now, the important point is that this is a model that handles concurrency and distribution at the fundamental level instead of ad hoc patched attempts to bring these features to OOP.

Challenges that actors solve include:

- How to build and design high-performance, concurrent applications.
- How to handle errors in a multi-threaded environment.
- How to protect my project from the pitfalls of concurrency.

Remoting

Remoting enables actors that are remote, living on different computers, to seamlessly exchange messages. Remoting can be enabled mostly with configuration; it has only a few APIs. Thanks to the actor model, a remote and local message send looks exactly the same. The patterns that you use on local systems translate directly to remote systems. You will rarely need to use Remoting directly, but it provides the foundation on which the Cluster subsystem is built.

Some of the challenges Remoting solves are:

- How to address actor systems living on remote hosts.
- How to address individual actors on remote actor systems.
- How to turn messages to bytes on the wire.
- How to manage low-level, network connections (and reconnections) between hosts, detect crashed actor systems and hosts, all transparently.
- How to multiplex communications from an unrelated set of actors on the same network connection, all transparently.

Cluster

If you have a set of actor systems that cooperate to solve some business problem, then you likely want to manage these set of systems in a disciplined way. While Remoting solves the problem of addressing and communicating with components of remote systems, Clustering gives you the ability to organize these into a "meta-system" tied together by a membership protocol. **In most cases, you want to use the Cluster module instead of using Remoting directly.** Clustering provides an additional set of services on top of Remoting that most real world applications need.

The challenges the Cluster module solves, among others, are:

- How to maintain a set of actor systems (a cluster) that can communicate with each other and consider each other as part of the cluster.
- How to introduce a new system safely to the set of already existing members.
- How to reliably detect systems that are temporarily unreachable.
- How to remove failed hosts/systems (or scale down the system) so that all remaining members agree on the remaining subset of the cluster.
- How to distribute computations among the current set of members.
- How to designate members of the cluster to a certain role; in other words, to provide certain services and not others.

Cluster Sharding

Sharding helps to solve the problem of distributing a set of actors among members of an Akka.NET cluster. Sharding is a pattern that is mostly used together with Persistence to balance a large set of persistent entities (backed by actors) to members of a cluster and also migrate them to other nodes when members crash or leave.

The challenge space that Sharding targets:

- How to model and scale out a large set of stateful entities on a set of systems.
- How to ensure that entities in the cluster are distributed properly so that load is properly balanced across the machines.
- How to migrate entities from a crashed system without losing their state.
- How to ensure that an entity does not exist on multiple systems at the same time and is hence kept consistent.

Cluster Singleton

A common (in fact, a bit too common) use case in distributed systems is to have a single entity responsible for a given task which is shared among other members of the cluster and migrated if the host system fails. While this undeniably introduces a common bottleneck for the whole cluster that limits scaling, there are scenarios where the use of this pattern is unavoidable. Cluster singleton allows a cluster to select an actor system which will host a particular actor while other systems can always access said service independently from where it is.

The Singleton module can be used to solve these challenges:

- How to ensure that only one instance of a service is running in the whole cluster.
- How to ensure that the service is up even if the system hosting it currently crashes or shut down during the process of scaling down.
- How to reach this instance from any member of the cluster assuming that it can migrate to other systems over time.

Cluster Publish-Subscribe

For coordination among systems, it is often necessary to distribute messages to all, or one system of a set of interested systems in a cluster. This pattern is usually called publish-subscribe and this module solves this exact problem. It is possible to broadcast messages to all subscribers of a topic or send a message to an arbitrary actor that has expressed interest.

Cluster Publish-Subscribe is intended to solve the following challenges:

- How to broadcast messages to an interested set of parties in a cluster.
- How to send a message to a member from an interested set of parties in a cluster.
- How to subscribe and unsubscribe for events of a certain topic in the cluster.

Persistence

Just like objects in OOP, actors keep their state in volatile memory. Once the system is shut down, gracefully or because of a crash, all data that was in memory is lost. Persistence provides patterns to enable actors to persist events that lead to their current state. Upon startup, events can be replayed to restore the state of the entity hosted by the actor. The event stream can be queried and fed into additional processing pipelines (an external Big Data cluster for example) or alternate views (like reports).

Persistence tackles the following challenges:

- How to restore the state of an entity/actor when system restarts or crashes.
- How to implement a [CQRS system](#).
- How to ensure reliable delivery of messages in face of network errors and system crashes.
- How to introspect domain events that have lead an entity to its current state.
- How to leverage [Event Sourcing](#) in my application to support long-running processes while the project continues to evolve.

Distributed Data

In situations where eventual consistency is acceptable, it is possible to share data between nodes in an Akka.NET Cluster and accept both reads and writes even in the face of cluster partitions. This can be achieved using [Conflict Free Replicated Data Types](#) (CRDTs), where writes on different nodes can happen concurrently and are merged in a predictable way afterward. The Distributed Data module provides infrastructure to share data and a number of useful data types.

Distributed Data is intended to solve the following challenges:

- How to accept writes even in the face of cluster partitions.
- How to share data while at the same time ensuring low-latency local read and write access.

Streams

Actors are a fundamental model for concurrency, but there are common patterns where their use requires the user to implement the same pattern over and over. Very common is the scenario where a chain, or graph, of actors, need to process a potentially large, or infinite, stream of sequential events and properly coordinate resource usage so that faster processing stages does not overwhelm slower ones in the chain or graph. Streams provide a higher-level abstraction on top of actors that simplifies writing such processing networks, handling all the fine details in the background and providing a safe, typed, composable programming model. Streams is also an implementation of the [Reactive Streams standard](#) which enables integration with all third party implementations of that standard.

Streams solve the following challenges:

- How to handle streams of events or large datasets with high performance, exploiting concurrency and keep resource usage tight.
 - How to assemble reusable pieces of event/data processing into flexible pipelines.
 - How to connect asynchronous services in a flexible way to each other, and have good performance.
 - How to provide or consume Reactive Streams compliant interfaces to interface with a third party library.
-

The above is an incomplete list of all the available modules, but it gives a nice overview of the landscape of modules and the level of sophistication you can reach when you start building systems on top of Akka. All these modules integrate together seamlessly. For example, take a large set of stateful business objects (a document, a shopping cart, etc) that is accessed by on-line users of your website. Model these as sharded entities using Sharding and Persistence to keep them balanced across a cluster that you can scale out on-demand (for example during an

advertising campaign before holidays) and keep them available even if some systems crash. Take the real-time stream of domain events of your business objects with Persistence Query and use Streams to pipe it into a streaming BigData engine.

Has this made you interested? Keep on reading to learn more.

Part 1: Top-level Architecture

In this and the following chapters, we will build a sample Akka.NET application to introduce you to the language of actors and how solutions can be formulated with them. It is a common hurdle for beginners to translate their project into actors even though they don't understand what they do on the high-level. We will build the core logic of a small application and this will serve as a guide for common patterns that will help to kickstart Akka.NET projects.

The application we aim to write will be a simplified IoT system where devices, installed at the home of users, can report temperature data from sensors. Users will be able to query the current state of these sensors. To keep things simple, we will not actually expose the application via HTTP or any other external API, we will, instead, concentrate only on the core logic. However, we will write tests for the pieces of the application to get comfortable and proficient with testing actors early on.

Our Goals for the IoT System

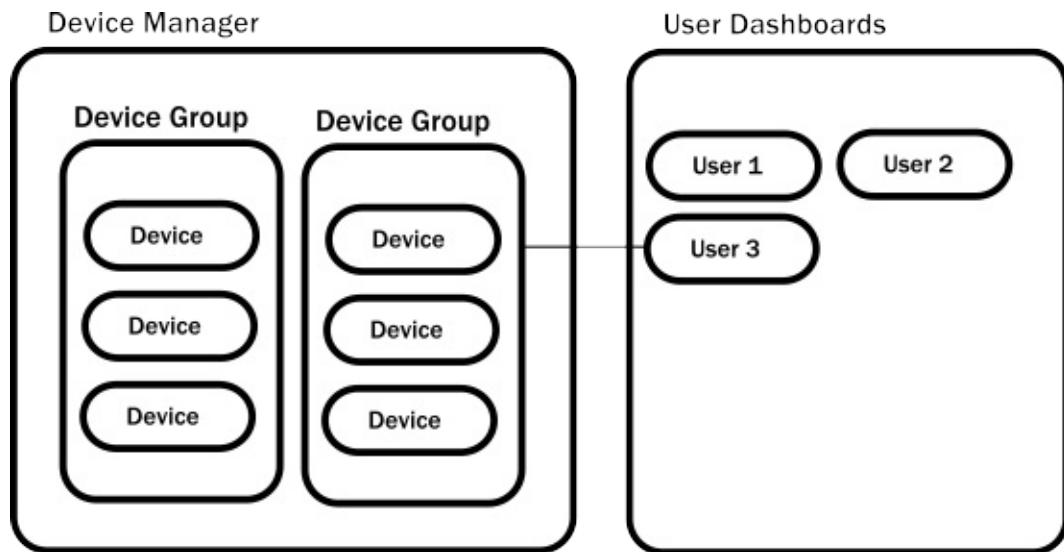
We will build a simple IoT application with the bare essentials to demonstrate designing an Akka.NET-based system. The application will consist of two main components:

- **Device data collection:** This component has the responsibility to maintain a local representation of the otherwise remote devices. The devices will be organized into device groups, grouping together sensors belonging to a home.
- **User dashboards:** This component has the responsibility to periodically collect data from the devices for a logged in user and present the results as a report.

For simplicity, we will only collect temperature data for the devices, but in a real application our local representations for a remote device, which we will model as an actor, would have many more responsibilities. Among others; reading the configuration of the device, changing the configuration, checking if the devices are unresponsive, etc. We leave these complexities for now as they can be easily added as an exercise.

We will also not address the means by which the remote devices communicate with the local representations (actors). Instead, we just build an actor based API that such a network protocol could use. We will use tests for our API everywhere though.

The architecture of the application will look like this:

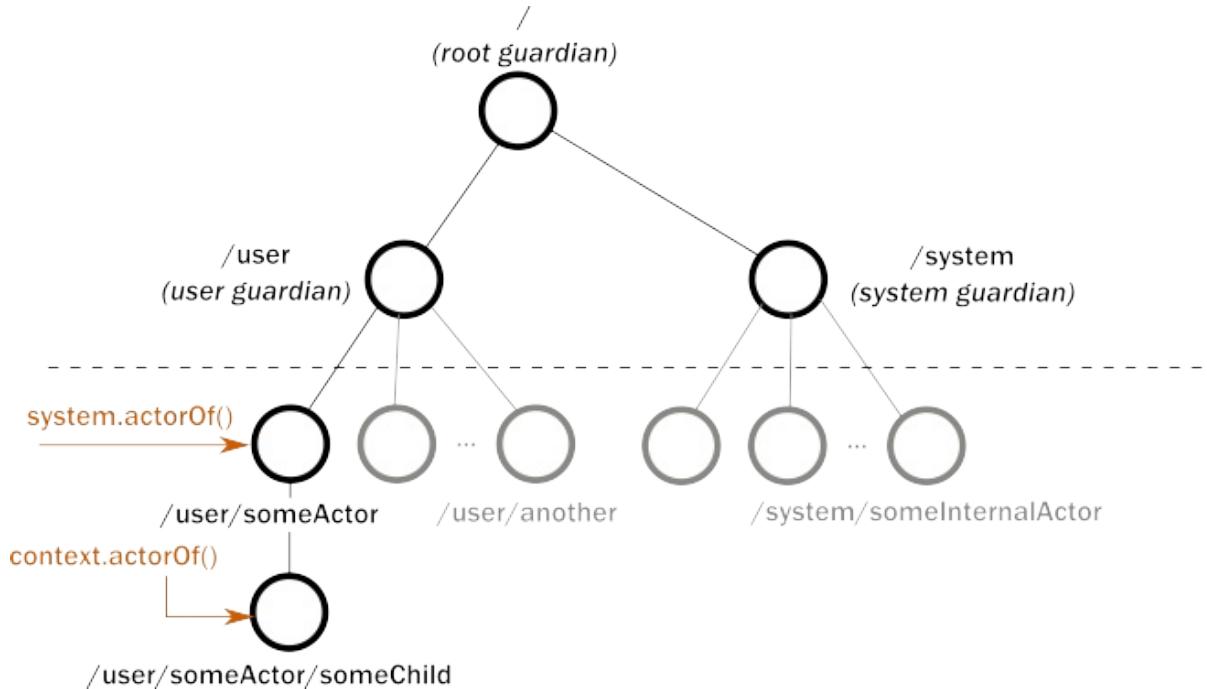


Top Level Architecture

When writing prose, the hardest part is usually to write the first couple of sentences. There is a similar feeling when trying to build an Akka.NET system: What should be the first actor? Where should it live? What should it do?

Fortunately, unlike with prose, there are established best practices that can guide us through these initial steps.

When one creates an actor in Akka.NET it always belongs to a certain parent. This means that actors are always organized into a tree. In general, creating an actor can only happen from inside another actor. This 'creator' actor becomes the *parent* of the newly created *child* actor. You might ask then, who is the parent of the *first* actor you create? To create a top-level actor one must first initialise an *actor system*, let's refer to this as the object `System`. This is followed by a call to `System.ActorOf()` which returns a reference to the newly created actor. This does not create a "freestanding" actor though, instead, it injects the corresponding actor as a child into an already existing tree:



As you see, creating actors from the "top" injects those actors under the path `/user/`, so for example creating an actor named `myActor` will end up having the path `/user/myActor`. In fact, there are three already existing actors in the system:

- `/` the so-called *root guardian*. This is the parent of all actors in the system, and the last one to stop when the system itself is terminated.
- `/user` the *guardian*. **This is the parent actor for all user created actors.** The name `user` should not confuse you, it has nothing to do with the logged in user, nor user handling in general. This name really means *userspace* as this is the place where actors that do not access Akka.NET internals live, i.e. all the actors created by users of the Akka.NET library. Every actor you will create will have the constant path `/user/` prepended to it.
- `/system` the *system guardian*.

The names of these built-in actors contain *guardian* because these are *supervising* every actor living as a child of them, i.e. under their path. We will explain supervision in more detail, all you need to know now is that every unhandled failure from actors bubbles up to their parent that, in turn, can decide how to handle this failure. These predefined actors are guardians in the sense that they are the final lines of defence, where all unhandled failures from user, or system, actors end up.

Does the root guardian (the root path `/`) have a parent? As it turns out, it has. This special entity is called the "Bubble-Walker". This special entity is invisible for the user and only has uses internally.

Structure of an IActorRef and Paths of Actors

The easiest way to see this in action is to simply print `IActorRef` instances. In this small experiment, we print the reference of the first actor we create and then we create a child of this actor, and print its reference. We have already created actors with `System.ActorOf()`, which creates an actor under `/user` directly. We call this kind of actors *top level*, even though in practice they are not on the top of the hierarchy, only on the top of the *user defined* hierarchy. Since in practice we usually concern ourselves about actors under `/user` this is still a convenient terminology, and we will stick to it.

Creating a non-top-level actor is possible from any actor, by invoking `Context.ActorOf()` which has the exact same signature as its top-level counterpart. This is how it looks like in practice:

```
public class PrintMyActorRefActor : UntypedActor
{
    protected override void OnReceive(object message)
    {
        switch (message)
        {
            case "printit":
                IActorRef secondRef = Context.ActorOf(Props.Empty, "second-actor");
                Console.WriteLine($"Second: {secondRef}");
                break;
        }
    }
}
```

```
var firstRef = Sys.ActorOf(Props.Create<PrintMyActorRefActor>(), "first-actor");
Console.WriteLine($"First: {firstRef}");
firstRef.Tell("printit", ActorRefs.NoSender);
```

We see that the following two lines are printed

```
First : Actor[akka://testSystem/user/first-actor#1053618476]
Second: Actor[akka://testSystem/user/first-actor/second-actor#-1544706041]
```

First, we notice that all of the paths start with `akka://testSystem/`. Since all actor references are valid URLs, there is a protocol field needed, which is `akka://` in the case of actors. Then, just like on the World Wide Web, the system is identified. In our case, this is `testSystem`, but could be any other name (if remote communication between multiple systems is enabled this name is the hostname of the system so other systems can find it on the network). Our two actors, as we have discussed before, live under `user`, and form a hierarchy:

- `akka://testSystem/user/first-actor` is the first actor we created, which lives directly under the user guardian, `/user`
- `akka://testSystem/user/first-actor/second-actor` is the second actor we created, using `Context.ActorOf`. As we see it lives directly under the first actor.

The last part of the actor reference, like `#1053618476` is a unique identifier of the actor living under the path. This is usually not something the user needs to be concerned with, and we leave the discussion of this field for later.

Hierarchy and Lifecycle of Actors

We have so far seen that actors are organized into a **strict hierarchy**. This hierarchy consists of a predefined upper layer of three actors (the root guardian, the user guardian, and the system guardian), thereafter the user created top-level actors (those directly living under `/user`) and the children of those. We now understand what the hierarchy looks like, but there are some nagging unanswered questions: *Why do we need this hierarchy? What is it used for?*

The first use of the hierarchy is to manage the lifecycle of actors. Actors pop into existence when created, then later, at user requests, they are stopped. Whenever an actor is stopped, all of its children are *recursively stopped* too. This is a very useful property and greatly simplifies cleaning up resources and avoiding resource leaks (like open sockets files, etc.). In fact, one of the overlooked difficulties when dealing with low-level multi-threaded code is the lifecycle management of various concurrent resources.

Stopping an actor can be done by calling `Context.Stop(actorRef)`. **It is considered a bad practice to stop arbitrary actors this way.** The recommended pattern is to call `Context.Stop(self)` inside an actor to stop itself, usually as a response to some user defined stop message or when the actor is done with its job.

The actor API exposes many lifecycle hooks that the actor implementation can override. 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.

Again, we can try out all this with a simple experiment:

```
public class StartStopActor1 : UntypedActor
{
    protected override void PreStart()
    {
        Console.WriteLine("first started");
        Context.ActorOf(Props.Create<StartStopActor2>(), "second");
    }

    protected override void PostStop() => Console.WriteLine("first stopped");

    protected override void OnReceive(object message)
    {
        switch (message)
        {
            case "stop":
                Context.Stop(Self);
                break;
        }
    }
}

public class StartStopActor2 : UntypedActor
{
    protected override void PreStart() => Console.WriteLine("second started");
    protected override void PostStop() => Console.WriteLine("second stopped");

    protected override void OnReceive(object message)
    {
    }
}
```

```
var first = Sys.ActorOf(Props.Create<StartStopActor1>(), "first");
first.Tell("stop");
```

After running it, we get the output

```
first started
second started
second stopped
first stopped
```

We see that when we stopped actor `first` it recursively stopped actor `second` and thereafter it stopped itself. This ordering is strict, *all* `PostStop()` hooks of the children are called before the `PostStop()` hook of the parent is called.

The family of these lifecycle hooks is rich, and we recommend reading [the actor lifecycle](#) section of the reference for all details.

Hierarchy and Failure Handling (Supervision)

Parents and children are not only connected by their lifecycles. Whenever an actor fails (throws an exception or an unhandled exception bubbles out from `receive`) it is temporarily suspended. The failure information is propagated to the parent, which decides how to handle the exception caused by the child actor. The default *supervisor strategy* is to stop and restart the child. If you don't change the default strategy all failures result in a restart. We won't change the default strategy in this simple experiment:

```
public class SupervisingActor : UntypedActor
{
    private IActorRef child = Context.ActorOf(Props.Create<SupervisedActor>(), "supervised-actor");

    protected override void OnReceive(object message)
    {
        switch (message)
        {
            case "failChild":
                child.Tell("fail");
                break;
        }
    }
}

public class SupervisedActor : UntypedActor
{
    protected override void PreStart() => Console.WriteLine("supervised actor started");
    protected override void PostStop() => Console.WriteLine("supervised actor stopped");

    protected override void OnReceive(object message)
    {
        switch (message)
        {
            case "fail":
                Console.WriteLine("supervised actor fails now");
                throw new Exception("I failed!");
        }
    }
}

var supervisingActor = Sys.ActorOf(Props.Create<SupervisingActor>(), "supervising-actor");
supervisingActor.Tell("failChild");
```

After running the snippet, we see the following output on the console:

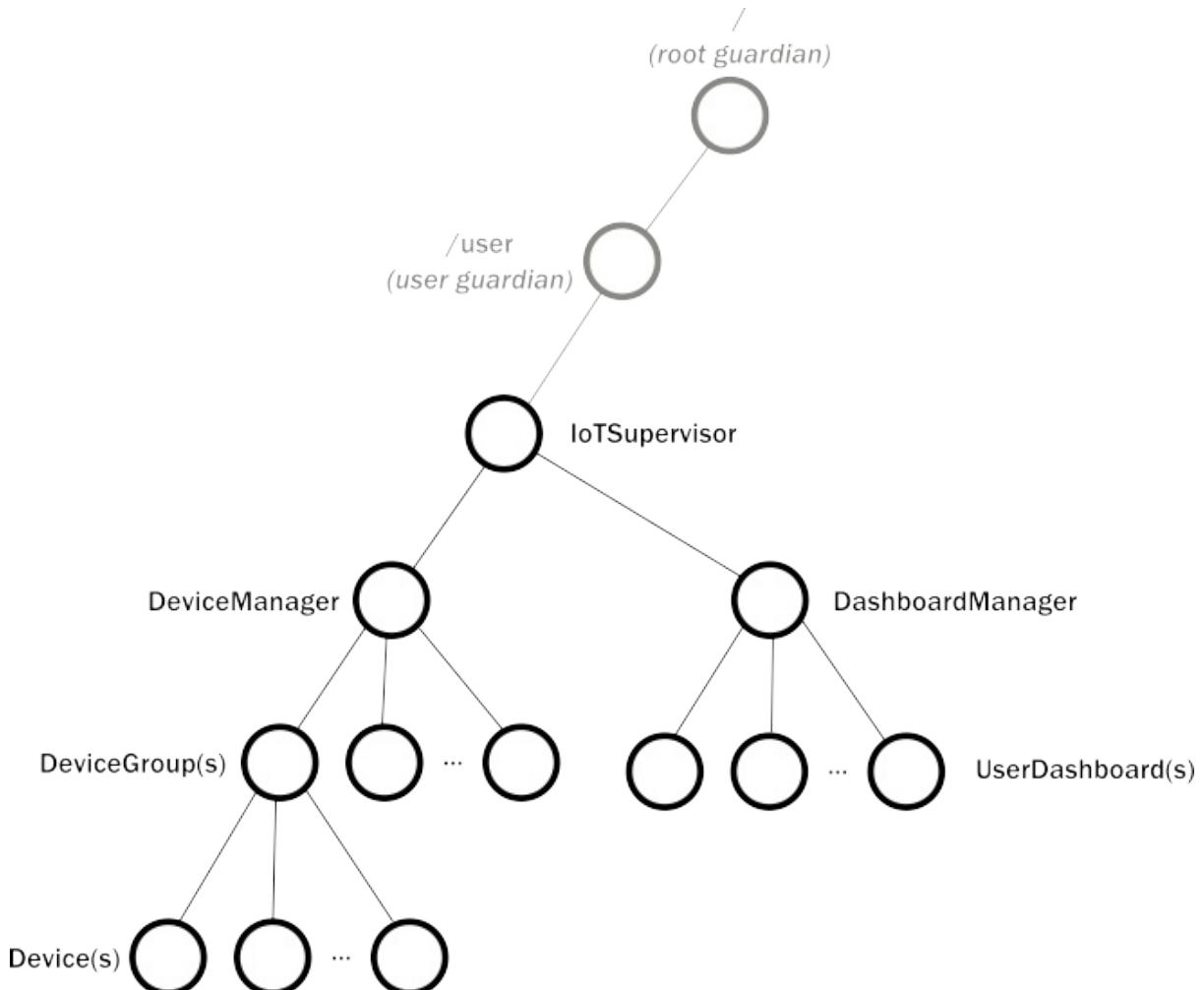
```
supervised actor started
supervised actor fails now
supervised actor stopped
[ERROR][05.06.2017 13:34:50][Thread 0003][akka://testSystem/user/supervising-actor/supervised-actor] I failed!
Cause: System.Exception: I failed!
at Tutorials.Tutorial1.SupervisedActor.OnReceive(Object message)
at Akka.Actor.UntypedActor.Receive(Object message)
at Akka.Actor.ActorBase.AroundReceive(Receive receive, Object message)
at Akka.Actor.ActorCell.ReceiveMessage(Object message)
at Akka.Actor.ActorCell.Invoke(Envelope envelope)
```

We see that after failure the actor is stopped and immediately started. We also see a log entry reporting the exception that was handled, in this case, our test exception. In this example we use `PreStart()` and `PostStop()` hooks which are the default to be called after and before restarts, so we cannot distinguish from inside the actor if it was started for the first time or restarted. This is usually the right thing to do, the purpose of the restart is to set the actor in a known-good state, which usually means a clean starting stage. **What actually happens though is that the `PreRestart()` and `PostRestart()` methods are called which, if not overridden, by default delegate to `PostStop()` and `PreStart()` respectively.** You can experiment with overriding these additional methods and see how the output changes.

For the impatient, we also recommend looking into the [supervision reference page](#) for more in-depth details.

The First Actor

Actors are organized into a strict tree, where the lifecycle of every child is tied to the parent and where parents are responsible for deciding the fate of failed children. At first, it might not be evident how to map our problem to such a tree, but in practice, this is easier than it looks. All we need to do is to rewrite our architecture diagram that contained nested boxes into a tree:



In simple terms, every component manages the lifecycle of the subcomponents. No subcomponent can outlive the parent component. This is exactly how the actor hierarchy works. Furthermore, it is desirable that a component handles the failure of its subcomponents. Together, these two desirable properties lead to the conclusion that the "contained-in" relationship of components should be mapped to the "children-of" relationship of actors.

The remaining question is how to map the top-level components to actors. It might be tempting to create the actors representing the main components as top-level actors. We instead, recommend creating an explicit component that represents the whole application. In other words, we will have a single top-level actor in our actor system and have the main components as children of this actor.

The first actor happens to be rather simple now, as we have not implemented any of the components yet. What is new is that we have dropped using `Console.WriteLine()` and instead use `ILoggingAdapter` which allows us to use the logging facility built into Akka.NET directly. Furthermore, we are using a recommended creational pattern for actors; define a static `Props()` method in the the actor:

```
public class IoTSupervisor : UntypedActor
{
    public ILoggingAdapter Log { get; } = Context.GetLogger();

    protected override void PreStart() => Log.Info("IoT Application started");
    protected override void PostStop() => Log.Info("IoT Application stopped");

    // No need to handle any messages
    protected override void OnReceive(object message)
    {

    }

    public static Props Props() => Akka.Actor.Props.Create<IoTSupervisor>();
}
```

All we need now is to tie this up with a class with the `main` entry point:

```
public class IoTApp
{
    public static void Init()
    {
        using (var system = ActorSystem.Create("iot-system"))
        {
            // Create top level supervisor
            var supervisor = system.ActorOf(Props.Create<IoTSupervisor>(), "iot-supervisor");
            // Exit the system after ENTER is pressed
            Console.ReadLine();
        }
    }
}
```

This application does very little for now, but we have the first actor in place and we are ready to extend it further.

What is next?

In the following chapters we will grow the application step-by-step:

1. We will create the representation for a device
2. We create the device management component
3. We add query capabilities to device groups

Part 2: The Device Actor

In part 1 we explained how to view actor systems *in the large*, i.e. how components should be represented, how actors should be arranged in the hierarchy. In this part, we will look at actors *in the small* by implementing an actor with the most common conversational patterns.

In particular, leaving the components aside for a while, we will implement an actor that represents a device. The tasks of this actor will be rather simple:

- Collect temperature measurements
- Report the last measured temperature if asked

When working with objects we usually design our API as *interfaces*, which are basically a collection of abstract methods to be filled out by the actual implementation. In the world of actors, the counterpart of interfaces is protocols. While it is not possible to formalize general protocols in the programming language, we can formalize its most basic elements: The messages.

The Query Protocol

Just because a device has been started it does not mean that it immediately has a temperature measurement. Hence, we need to account in our protocol for the case in which a temperature is not present. This, fortunately, means that we can test the query part of the actor without the write part present, as it can simply report an empty result.

The protocol for obtaining the current temperature from the device actor is rather simple:

1. Wait for a request for the current temperature.
2. Respond to the request with a reply containing the current temperature or an indication that it is not yet available.

We need two messages, one for the request, and one for the reply. A first attempt could look like this:

```
public sealed class ReadTemperature
{
    public static ReadTemperature Instance { get; } = new ReadTemperature();
    private ReadTemperature() { }
}

public sealed class RespondTemperature
{
    public RespondTemperature(double? value)
    {
        Value = value;
    }

    public double? Value { get; }
}
```

This is a fine approach, but it limits the flexibility of the protocol. To understand why we need to talk about message ordering and message delivery guarantees in general.

Message Ordering, Delivery Guarantees

In order to give some context to the discussion below, consider an application which spans multiple network hosts. The basic mechanism for communication is the same whether sending to an actor on the local CLR or to a remote actor, but of course, there will be observable differences in the latency of delivery (possibly also depending on the

bandwidth of the network link and the message size) and the reliability. In the case of a remote message send there are more steps involved which means that more can go wrong. Another aspect is that a local send will just pass a reference to the message inside the same CLR, without any restrictions on the underlying object which is sent, whereas a remote transport will place a limit on the message size.

It is also important to keep in mind that while sending inside the same CLR is significantly more reliable, if an actor fails due to a programmer error while processing the message, the effect is basically the same as if a remote, network request fails due to the remote host crashing while processing the message. Even though in both cases the service is recovered after a while (the actor is restarted by its supervisor, the host is restarted by an operator or by a monitoring system) individual requests are lost during the crash. **Writing your actors such that every message could possibly be lost is the safe, pessimistic bet.**

These are the rules in Akka.NET for message sends:

- At-most-once delivery, i.e. no guaranteed delivery.
- Message ordering is maintained per sender, receiver pair.

What Does "at-most-once" Mean?

When it comes to describing the semantics of a delivery mechanism, there are three basic categories:

- **At-most-once delivery** means that for each message handed to the mechanism, that message is delivered zero or one time; in more casual terms it means that messages may be lost, but never duplicated.
- **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 the state at the sending end or in the transport mechanism. The second one requires retries to counter transport losses, which means keeping the state at the sending end and having an acknowledgment mechanism at the receiving end. The third is most expensive, and has consequently worst performance: in addition to the second, it requires the state to be kept at the receiving end in order to filter out duplicate deliveries.

Why No Guaranteed Delivery?

At the core of the problem lies the question what exactly this guarantee shall mean, i.e. at which point is the delivery considered to be guaranteed:

1. When the message is sent out on the network?
2. When the message is received by the other host?
3. When the message is put into the target actor's mailbox?
4. When the message is starting to be processed by the target actor?
5. When the message is processed successfully by the target actor?

Most frameworks/protocols claiming guaranteed delivery actually provide something similar to point 4 and 5. While this sounds fair, **is this actually useful?** To understand the implications, consider a simple, practical example: A user attempts to place an order and we only want to claim that it has successfully processed once it is actually on disk in the database containing orders.

If we rely on the guarantees of such system it will report success as soon as the order has been submitted to the internal API that has the responsibility to validate it, process it and put it into the database. Unfortunately, immediately after the API has been invoked the following may happen:

- The host can immediately crash.
- Deserialization can fail.
- Validation can fail.
- The database might be unavailable.
- A programming error might occur.

The problem is that the **guarantee of delivery** does not translate to the **domain level guarantee**. We only want to report success once the order has been actually fully processed and persisted. **The only entity that can report success is the application itself, since only it has any understanding of the domain guarantees required. No generalized framework can figure out the specifics of a particular domain and what is considered a success in that domain.** In this particular example, we only want to signal success after a successful database write, where the database acknowledged that the order is now safely stored. **For these reasons Akka.NET lifts the responsibilities of guarantees to the application itself, i.e. you have to implement them yourself. On the other hand, you are in full control of the guarantees that you want to provide.**

Message Ordering

The rule is that for a given pair of actors, messages sent directly from the first to the second will not be received out-of-order. The word directly emphasizes that this guarantee only applies when sending with the tell operator directly to the final destination, but not when employing mediators.

If:

- Actor `A1` sends messages `M1`, `M2`, `M3` to `A2`.
- Actor `A3` sends messages `M4`, `M5`, `M6` to `A2`.

This means that:

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

For the full details on delivery guarantees please refer to the [reference page](#).

Revisiting the Query Protocol

There is nothing wrong with our first query protocol but it limits our flexibility. If we want to implement resends in the actor that queries our device actor (because of timed out requests) or want to query multiple actors it can be helpful to put an additional query ID field in the message which helps us correlate requests with responses.

Hence, we add one more field to our messages, so that an ID can be provided by the requester:

```
public sealed class ReadTemperature
{
    public ReadTemperature(long requestId)
    {
        RequestId = requestId;
    }

    public long RequestId { get; }

    public sealed class RespondTemperature
    {
        public RespondTemperature(long requestId, double? value)
```

```

    {
        RequestId = requestId;
        Value = value;
    }

    public long RequestId { get; }
    public double? Value { get; }
}

```

Our device actor has the responsibility to use the same ID for the response of a given query. Now we can sketch our device actor:

```

public class Device : UntypedActor
{
    private double? _lastTemperatureReading = null;

    public Device(string groupId, string deviceId)
    {
        GroupId = groupId;
        DeviceId = deviceId;
    }

    protected override void PreStart() => Log.Info($"Device actor {GroupId}-{DeviceId} started");
    protected override void PostStop() => Log.Info($"Device actor {GroupId}-{DeviceId} stopped");

    protected ILoggingAdapter Log { get; } = Context.GetLogger();
    protected string GroupId { get; }
    protected string DeviceId { get; }

    protected override void OnReceive(object message)
    {
        switch (message)
        {
            case MainDevice.ReadTemperature read:
                Sender.Tell(new RespondTemperature(read.RequestId, _lastTemperatureReading));
                break;
        }
    }

    public static Props Props(string groupId, string deviceId) =>
        Akka.Actor.Props.Create(() => new Device(groupId, deviceId));
}

```

We maintain the current temperature, initially set to `null`, and we simply report it back if queried. We also added fields for the ID of the device and the group it belongs to, which we will use later.

We can already write a simple test for this functionality @scala[(we use ScalaTest but any other test framework can be used with the Akka.NET Testkit):

```

[Fact]
public void Device_actor_must_reply_with_latest_temperature_reading()
{
    var probe = CreateTestProbe();
    var deviceActor = Sys.ActorOf(Device.Props("group", "device"));

    deviceActor.Tell(new RecordTemperature(requestId: 1, value: 24.0), probe.Ref);
    probe.ExpectMsg<TemperatureRecorded>(s => s.RequestId == 1);

    deviceActor.Tell(new ReadTemperature(requestId: 2), probe.Ref);
    var response1 = probe.ExpectMsg<RespondTemperature>();
    response1.RequestId.Should().Be(2);
    response1.Value.Should().Be(24.0);

    deviceActor.Tell(new RecordTemperature(requestId: 3, value: 55.0), probe.Ref);
    probe.ExpectMsg<TemperatureRecorded>(s => s.RequestId == 3);
}

```

```

        deviceActor.Tell(new ReadTemperature(requestId: 4), probe.Ref);
        var response2 = probe.ExpectMsg<RespondTemperature>();
        response2.RequestId.Should().Be(4);
        response2.Value.Should().Be(55.0);
    }
}

```

The Write Protocol

As a first attempt, we could model recording the current temperature in the device actor as a single message:

- When a temperature record request is received, update the `currentTemperature` property.

Such a message could possibly look like this:

```

public sealed class RecordTemperature
{
    public RecordTemperature(double value)
    {
        Value = value;
    }

    public double Value { get; }
}

```

The problem with this approach is that the sender of the record temperature message can never be sure if the message was processed or not. We have seen that Akka.NET does not guarantee delivery of these messages and leaves it to the application to provide success notifications. In our case, we would like to send an acknowledgment to the sender once we have updated our last temperature recording, e.g. `TemperatureRecorded`. Just like in the case of temperature queries and responses, it is a good idea to include an ID field to provide maximum flexibility.

Putting read and write protocol together, the device actor will look like this:

[!code-csharpMain]

```

public sealed class RecordTemperature
{
    public RecordTemperature(long requestId, double value)
    {
        RequestId = requestId;
        Value = value;
    }

    public long RequestId { get; }
    public double Value { get; }
}

public sealed class TemperatureRecorded
{
    public TemperatureRecorded(long requestId)
    {
        RequestId = requestId;
    }

    public long RequestId { get; }
}

public sealed class ReadTemperature
{
    public ReadTemperature(long requestId)
    {
        RequestId = requestId;
    }
}

```

```

        }

        public long RequestId { get; }

    }

    public sealed class RespondTemperature
    {
        public RespondTemperature(long requestId, double? value)
        {
            RequestId = requestId;
            Value = value;
        }

        public long RequestId { get; }
        public double? Value { get; }
    }

    public class Device : UntypedActor
    {
        private double? _lastTemperatureReading = null;

        public Device(string groupId, string deviceId)
        {
            GroupId = groupId;
            DeviceId = deviceId;
        }

        protected override void PreStart() => Log.Info($"Device actor {GroupId}-{DeviceId} started");
        protected override void PostStop() => Log.Info($"Device actor {GroupId}-{DeviceId} stopped");

        protected ILoggingAdapter Log { get; } = Context.GetLogger();
        protected string GroupId { get; }
        protected string DeviceId { get; }

        protected override void OnReceive(object message)
        {
            switch (message)
            {
                case RecordTemperature rec:
                    Log.Info($"Recorded temperature reading {rec.Value} with {rec.RequestId}");
                    _lastTemperatureReading = rec.Value;
                    Sender.Tell(new TemperatureRecorded(rec.RequestId));
                    break;
                case ReadTemperature read:
                    Sender.Tell(new RespondTemperature(read.RequestId, _lastTemperatureReading));
                    break;
            }
        }

        public static Props Props(string groupId, string deviceId) =>
            Akka.Actor.Props.Create(() => new Device(groupId, deviceId));
    }
}

```

We are also responsible for writing a new test case now, exercising both the read/query and write/record functionality together:

[!code-csharp>Main]

```

[Fact]
public void Device_actor_must_reply_with_latest_temperature_reading()
{
    var probe = CreateTestProbe();
    var deviceActor = Sys.ActorOf(Device.Props("group", "device"));

    deviceActor.Tell(new RecordTemperature(requestId: 1, value: 24.0), probe.Ref);
    probe.ExpectMsg<TemperatureRecorded>(s => s.RequestId == 1);
}

```

```
deviceActor.Tell(new ReadTemperature(requestId: 2), probe.Ref);
var response1 = probe.ExpectMsg<RespondTemperature>();
response1.RequestId.Should().Be(2);
response1.Value.Should().Be(24.0);

deviceActor.Tell(new RecordTemperature(requestId: 3, value: 55.0), probe.Ref);
probe.ExpectMsg<TemperatureRecorded>(s => s.RequestId == 3);

deviceActor.Tell(new ReadTemperature(requestId: 4), probe.Ref);
var response2 = probe.ExpectMsg<RespondTemperature>();
response2.RequestId.Should().Be(4);
response2.Value.Should().Be(55.0);
}
```

What is Next

So far, we have started designing our overall architecture, and we wrote our first actor directly corresponding to the domain. We now have to create the component that is responsible for maintaining groups of devices and the device actors themselves.

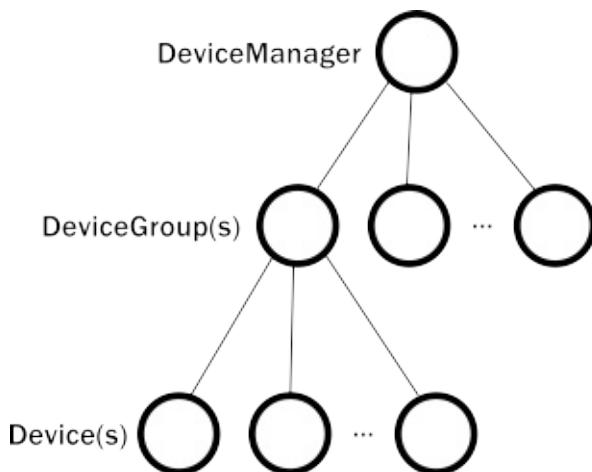
Part 3: Device Groups and Manager

In this chapter, we will integrate our device actors into a component that manages devices. When a new device comes online, there is no actor representing it. We need to be able to ask the device manager component to create a new device actor for us if necessary, in the required group (or return a reference to an already existing one).

Since we keep our tutorial system to the bare minimum, we have no actual component that interfaces with the external world via some networking protocol. For our exercise, we will just create the API necessary to integrate with such a component in the future. In a final system, the steps for connecting a device would look like this:

1. The device connects through some protocol to our system.
2. The component managing network connections accept the connection.
3. The ID of the device and the ID of the group that it belongs is acquired.
4. The device manager component is asked to create a group and device actor for the given IDs (or return an existing one).
5. The device actor (just been created or located) responds with an acknowledgment, at the same time exposing its `IActorRef` directly (by being the sender of the acknowledgment).
6. The networking component now uses the `IActorRef` of the device directly, avoiding going through the component.

We are only concerned with steps 4 and 5 now. We will model the device manager component as an actor tree with three levels:



- The top level is the supervisor actor representing the component. It is also the entry point to look up or create group and device actors.
- Device group actors are supervisors of the devices belonging to the group. Apart from supervising the device actors they also provide extra services, like querying the temperature readings from all the devices available.
- Device actors manage all the interactions with the actual devices, storing temperature readings for example.

When designing actor systems one of the main challenges is to decide on the granularity of the actors. For example, it would be perfectly possible to have only a single actor maintaining all the groups and devices in `Dictionary<String, ActorRef>` for example. It would be also reasonable to keep the groups as separate actors, but keep device state simply inside the group actor.

We chose this three-layered architecture for the following reasons:

- Having groups as individual actors:
 - Allows us to isolate failures happening in a group. If a programmer error would happen in the single actor that keeps all state, it would be all wiped out once that actor is restarted affecting groups that are otherwise

- non-faulty.
- Simplifies the problem of querying all the devices belonging to a group (since it only contains state related to the given group).
- Increases the parallelism of the system by allowing to query multiple groups concurrently. Since groups have dedicated actors, all of them can run concurrently.
- Having devices as individual actors:
 - Allows us to isolate failures happening in a device actor from the rest of the devices.
 - Increases the parallelism of collecting temperature readings as actual network connections from different devices can talk to the individual device actors directly, reducing contention points.

In practice, a system can be organized in multiple ways, all depending on the characteristics of the interactions between actors.

The following guidelines help to arrive at the right granularity:

- Prefer larger granularity to smaller. Introducing more fine-grained actors than needed causes more problems than it solves.
- Prefer finer granularity if it enables higher concurrency in the system.
- Prefer finer granularity if actors need to handle complex conversations with other actors and hence have many states. We will see a very good example for this in the next chapter.
- Prefer finer granularity if there is too much state to keep around in one place compared to dividing into smaller actors.
- Prefer finer granularity if the current actor has multiple unrelated responsibilities that can fail and be restored individually.

The Registration Protocol

As the first step, we need to design the protocol for registering a device and create an actor that will be responsible for it. This protocol will be provided by the `DeviceManager` component itself because that is the only actor that is known up front: device groups and device actors are created on-demand. The steps of registering a device are the following:

1. DeviceManager receives the request to track a device for a given group and device.
2. If the manager already has an actor for the device group, it forwards the request to it. Otherwise, it first creates a new one and then forwards the request.
3. The DeviceGroup receives the request to register an actor for the given device.
4. If the group already has an actor for the device, it forwards the request to it. Otherwise, it first creates a new one and then forwards the request.
5. The device actor receives the request and acknowledges it to the original sender. Since the device actor is the sender of the acknowledgment, the receiver, i.e. the device, will be able to learn its `IActorRef` and send direct messages to its device actor in the future.

Now that the steps are defined, we only need to define the messages that we will use to communicate requests and their acknowledgment:

[!code-csharpDeviceManager.scala]

```
public sealed class RequestTrackDevice
{
    public RequestTrackDevice(string groupId, string deviceId)
    {
        GroupId = groupId;
        DeviceId = deviceId;
    }

    public string GroupId { get; }
```

```

        public string DeviceId { get; }

    }

    public sealed class DeviceRegistered
    {
        public static DeviceRegistered Instance { get; } = new DeviceRegistered();
        private DeviceRegistered() { }
    }
}

```

As you see, in this case, we have not included a request ID field in the messages. Since registration is usually happening once, at the component that connects the system to some network protocol, we will usually have no use for the ID. Nevertheless, it is a good exercise to add this ID.

Add Registration Support to Device Actor

We start implementing the protocol from the bottom first. In practice, both a top-down and bottom-up approach can work, but in our case, we benefit from the bottom-up approach as it allows us to immediately write tests for the new features without mocking out parts.

At the bottom of our hierarchy are the `Device` actors. Their job in this registration process is rather simple: just reply to the registration request with an acknowledgment to the sender. *We will assume that the sender of the registration message is preserved in the upper layers.* We will show you in the next section how this can be achieved.

We also add a safeguard against requests that come with a mismatched group or device ID. This is how the resulting the code looks like:

[!code-csharpDevice.scala]

```

public sealed class RecordTemperature
{
    public RecordTemperature(long requestId, double value)
    {
        RequestId = requestId;
        Value = value;
    }

    public long RequestId { get; }
    public double Value { get; }
}

public sealed class TemperatureRecorded
{
    public TemperatureRecorded(long requestId)
    {
        RequestId = requestId;
    }

    public long RequestId { get; }
}

public sealed class ReadTemperature
{
    public ReadTemperature(long requestId)
    {
        RequestId = requestId;
    }

    public long RequestId { get; }
}

public sealed class RespondTemperature
{
}

```

```

public RespondTemperature(long requestId, double? value)
{
    RequestId = requestId;
    Value = value;
}

public long RequestId { get; }
public double? Value { get; }
}

public class Device : UntypedActor
{
    private double? _lastTemperatureReading = null;

    public Device(string groupId, string deviceId)
    {
        GroupId = groupId;
        DeviceId = deviceId;
    }

    protected override void PreStart() => Log.Info($"Device actor {GroupId}-{DeviceId} started");
    protected override void PostStop() => Log.Info($"Device actor {GroupId}-{DeviceId} stopped");

    protected ILoggingAdapter Log { get; } = Context.GetLogger();
    protected string GroupId { get; }
    protected string DeviceId { get; }

    protected override void OnReceive(object message)
    {
        switch (message)
        {
            case RequestTrackDevice req when req.GroupId.Equals(GroupId) && req.DeviceId.Equals(DeviceId):
                Sender.Tell(DeviceRegistered.Instance);
                break;
            case RequestTrackDevice req:
                Log.Warning($"Ignoring TrackDevice request for {req.GroupId}-{req.DeviceId}. This actor is responsible for {GroupId}-{DeviceId}.");
                break;
            case RecordTemperature rec:
                Log.Info($"Recorded temperature reading {rec.Value} with {rec.RequestId}");
                _lastTemperatureReading = rec.Value;
                Sender.Tell(new TemperatureRecorded(rec.RequestId));
                break;
            case ReadTemperature read:
                Sender.Tell(new RespondTemperature(read.RequestId, _lastTemperatureReading));
                break;
        }
    }

    public static Props Props(string groupId, string deviceId) => Akka.Actor.Props.Create(() => new Device(groupId, deviceId));
}

```

We should not leave features untested, so we immediately write two new test cases, one exercising successful registration, the other testing the case when IDs don't match:

[!NOTE] We used the `ExpectNoMsg()` helper method from `TestProbe`. This assertion waits until the defined time-limit and fails if it receives any messages during this period. If no messages are received during the waiting period the assertion passes. It is usually a good idea to keep these timeouts low (but not too low) because they add significant test execution time otherwise.

[!code-csharpDeviceSpec.scala]

```

[Fact]
public void Device_actor_must_reply_to_registration_requests()

```

```

{
    var probe = CreateTestProbe();
    var deviceActor = Sys.ActorOf(Device.Props("group", "device"));

    deviceActor.Tell(new RequestTrackDevice("group", "device"), probe.Ref);
    probe.ExpectMsg<DeviceRegistered>();
    probe.LastSender.Should().Be(deviceActor);
}

[Fact]
public void Device_actor_must_ignore_wrong_registration_requests()
{
    var probe = CreateTestProbe();
    var deviceActor = Sys.ActorOf(Device.Props("group", "device"));

    deviceActor.Tell(new RequestTrackDevice("wrongGroup", "device"), probe.Ref);
    probe.ExpectNoMsg(TimeSpan.FromMilliseconds(500));

    deviceActor.Tell(new RequestTrackDevice("group", "Wrongdevice"), probe.Ref);
    probe.ExpectNoMsg(TimeSpan.FromMilliseconds(500));
}

```

Device Group

We are done with the registration support at the device level, now we have to implement it at the group level. A group has more work to do when it comes to registrations. It must either forward the request to an existing child, or it should create one. To be able to look up child actors by their device IDs we will use a `Dictionary<string, IActorRef>`.

We also want to keep the original sender of the request so that our device actor can reply directly. This is possible by using `Forward` instead of the `Tell` operator. The only difference between the two is that `Forward` keeps the original sender while `Tell` always sets the sender to be the current actor. Just like with our device actor, we ensure that we don't respond to wrong group IDs:

[\[!code-csharpDeviceGroup.scala\]](#)

```

public class DeviceGroup : UntypedActor
{
    private Dictionary<string, IActorRef> deviceIdToActor = new Dictionary<string, IActorRef>();

    public DeviceGroup(string groupId)
    {
        GroupId = groupId;
    }

    protected override void PreStart() => Log.Info($"Device group {GroupId} started");
    protected override void PostStop() => Log.Info($"Device group {GroupId} stopped");

    protected ILoggingAdapter Log { get; } = Context.GetLogger();
    protected string GroupId { get; }

    protected override void OnReceive(object message)
    {
        switch (message)
        {
            case RequestTrackDevice trackMsg when trackMsg.GroupId.Equals(GroupId):
                if (deviceIdToActor.TryGetValue(trackMsg.DeviceId, out var actorRef))
                {
                    actorRef.Forward(trackMsg);
                }
                else
                {
                    Log.Info($"Creating device actor for {trackMsg.DeviceId}");
                    var deviceActor = Context.ActorOf(Device.Props(trackMsg.GroupId, trackMsg.DeviceId))
                }
        }
    }
}

```

```

        , $"device-{trackMsg.DeviceId}");
            deviceIdToActor.Add(trackMsg.DeviceId, deviceActor);
            deviceActor.Forward(trackMsg);
        }
        break;
    case RequestTrackDevice trackMsg:
        Log.Warning($"Ignoring TrackDevice request for {trackMsg.GroupId}. This actor is responsible for {GroupId}.");
        break;
    }
}

public static Props Props(string groupId) => Akka.Actor.Props.Create(() => new DeviceGroup(groupId));
}
}

```

Just as we did with the device, we test this new functionality. We also test that the actors returned for the two different IDs are actually different, and we also attempt to record a temperature reading for each of the devices to see if the actors are responding.

[!code-csharpDeviceGroupSpec.scala]

```

[Fact]
public void DeviceGroup_actor_must_be_able_to_register_a_device_actor()
{
    var probe = CreateTestProbe();
    var groupActor = Sys.ActorOf(DeviceGroup.Props("group"));

    groupActor.Tell(new RequestTrackDevice("group", "device1"), probe.Ref);
    probe.ExpectMsg<DeviceRegistered>();
    var deviceActor1 = probe.LastSender;

    groupActor.Tell(new RequestTrackDevice("group", "device2"), probe.Ref);
    probe.ExpectMsg<DeviceRegistered>();
    var deviceActor2 = probe.LastSender;
    deviceActor1.Should().NotBe(deviceActor2);

    // Check that the device actors are working
    deviceActor1.Tell(new RecordTemperature(requestId: 0, value: 1.0), probe.Ref);
    probe.ExpectMsg<TemperatureRecorded>(s => s.RequestId == 0);
    deviceActor2.Tell(new RecordTemperature(requestId: 1, value: 2.0), probe.Ref);
    probe.ExpectMsg<TemperatureRecorded>(s => s.RequestId == 1);
}

[Fact]
public void DeviceGroup_actor_must_ignore_requests_for_wrong_groupId()
{
    var probe = CreateTestProbe();
    var groupActor = Sys.ActorOf(DeviceGroup.Props("group"));

    groupActor.Tell(new RequestTrackDevice("wrongGroup", "device1"), probe.Ref);
    probe.ExpectNoMsg(TimeSpan.FromMilliseconds(500));
}

```

It might be, that a device actor already exists for the registration request. In this case, we would like to use the existing actor instead of a new one. We have not tested this yet, so we need to fix this:

[!code-csharpDeviceGroupSpec.scala] [Fact] public void
DeviceGroup_actor_must_return_same_actor_for_same_deviceId() { var probe = CreateTestProbe(); var groupActor = Sys.ActorOf(DeviceGroup.Props("group"));

```

        groupActor.Tell(new RequestTrackDevice("group", "device1"), probe.Ref);
        probe.ExpectMsg<DeviceRegistered>();
        var deviceActor1 = probe.LastSender;
    }
}
```

```

        groupActor.Tell(new RequestTrackDevice("group", "device1"), probe.Ref);
        probe.ExpectMsg<DeviceRegistered>();
        var deviceActor2 = probe.LastSender;

        deviceActor1.Should().Be(deviceActor2);
    }
}

```

So far, we have implemented everything for registering device actors in the group. Devices come and go, however, so we will need a way to remove those from the `Dictionary<string, IActorRef>`. We will assume that when a device is removed, its corresponding device actor is simply stopped. We need some way for the parent to be notified when one of the device actors are stopped. Unfortunately, supervision will not help because it is used for error scenarios, not graceful stopping.

There is a feature in Akka.NET that is exactly what we need here. It is possible for an actor to *watch* another actor and be notified if the other actor is stopped. This feature is called *Death Watch* and it is an important tool for any Akka.NET application. Unlike supervision, watching is not limited to parent-child relationships, any actor can watch any other actor given its `IActorRef`. After a watched actor stops, the watcher receives a `Terminated(ref)` message which also contains the reference to the watched actor. The watcher can either handle this message explicitly or, if it does not handle it directly it will fail with a `DeathPactException`. This latter is useful if the actor can no longer perform its duties after its collaborator actor has been stopped. In our case, the group should still function after one device have been stopped, so we need to handle this message. The steps we need to follow are the following:

1. Whenever we create a new device actor, we must also watch it.
2. When we are notified that a device actor has been stopped we also need to remove it from the `Dictionary<string, IActorRef>` which maps devices to device actors.

Unfortunately, the `Terminated` message contains only the `IActorRef` of the child actor but we do not know its ID, which we need to remove it from the map of existing device to device actor mappings. To be able to do this removal, we need to introduce another placeholder, `Dictionary<IActorRef, string>`, that allow us to find out the device ID corresponding to a given `IActorRef`. Putting this together the result is:

[!code-csharpDeviceGroup.scala]

```

public class DeviceGroup : UntypedActor
{
    private Dictionary<string, IActorRef> deviceIdToActor = new Dictionary<string, IActorRef>();
    private Dictionary<IActorRef, string> actorToDeviceId = new Dictionary<IActorRef, string>();

    public DeviceGroup(string groupId)
    {
        GroupId = groupId;
    }

    protected override void PreStart() => Log.Info($"Device group {GroupId} started");
    protected override void PostStop() => Log.Info($"Device group {GroupId} stopped");

    protected ILoggingAdapter Log { get; } = Context.GetLogger();
    protected string GroupId { get; }

    protected override void OnReceive(object message)
    {
        switch (message)
        {
            case RequestTrackDevice trackMsg when trackMsg.GroupId.Equals(GroupId):
                if (deviceIdToActor.TryGetValue(trackMsg.DeviceId, out var actorRef))
                {

```

```

        actorRef.Forward(trackMsg);
    }
    else
    {
        Log.Info($"Creating device actor for {trackMsg.DeviceId}");
        var deviceActor = Context.ActorOf(Device.Props(trackMsg.GroupId, trackMsg.DeviceId)
, $"device-{trackMsg.DeviceId}");
        Context.Watch(deviceActor);
        actorToDeviceId.Add(deviceActor, trackMsg.DeviceId);
        deviceIdToActor.Add(trackMsg.DeviceId, deviceActor);
        deviceActor.Forward(trackMsg);
    }
    break;
case RequestTrackDevice trackMsg:
    Log.Warning($"Ignoring TrackDevice request for {trackMsg.GroupId}. This actor is responsible for {GroupId}.");
    break;
case Terminated t:
    var deviceId = actorToDeviceId[t.ActorRef];
    Log.Info($"Device actor for {deviceId} has been terminated");
    actorToDeviceId.Remove(t.ActorRef);
    deviceIdToActor.Remove(deviceId);
    break;
}
}

public static Props Props(string groupId) => Akka.Actor.Props.Create(() => new DeviceGroup(groupId));
}
}

```

So far we have no means to get what devices the group device actor keeps track of and, therefore, we cannot test our new functionality yet. To make it testable, we add a new query capability `RequestDeviceList` that simply lists the currently active device IDs:

[!code-csharpDeviceGroup.scala]

```

public sealed class RequestDeviceList
{
    public RequestDeviceList(long requestId)
    {
        RequestId = requestId;
    }

    public long RequestId { get; }
}

public sealed class ReplyDeviceList
{
    public ReplyDeviceList(long requestId, ISet<string> ids)
    {
        RequestId = requestId;
        Ids = ids;
    }

    public long RequestId { get; }
    public ISet<string> Ids { get; }
}

public class DeviceGroup : UntypedActor
{
    private Dictionary<string, IActorRef> deviceIdToActor = new Dictionary<string, IActorRef>();
    private Dictionary<IActorRef, string> actorToDeviceId = new Dictionary<IActorRef, string>();

    public DeviceGroup(string groupId)
    {
        GroupId = groupId;
    }
}

```

```

protected override void PreStart() => Log.Info($"Device group {GroupId} started");
protected override void PostStop() => Log.Info($"Device group {GroupId} stopped");

protected ILoggingAdapter Log { get; } = Context.GetLogger();
protected string GroupId { get; }

protected override void OnReceive(object message)
{
    switch (message)
    {
        case RequestTrackDevice trackMsg when trackMsg.GroupId.Equals(GroupId):
            if (deviceIdToDeviceId.TryGetValue(trackMsg.DeviceId, out var actorRef))
            {
                actorRef.Forward(trackMsg);
            }
            else
            {
                Log.Info($"Creating device actor for {trackMsg.DeviceId}");
                var deviceActor = Context.ActorOf(Device.Props(trackMsg.GroupId, trackMsg.DeviceId)
, $"device-{trackMsg.DeviceId}");
                Context.Watch(deviceActor);
                actorToDeviceId.Add(deviceActor, trackMsg.DeviceId);
                deviceIdToActor.Add(trackMsg.DeviceId, deviceActor);
                deviceActor.Forward(trackMsg);
            }
            break;
        case RequestTrackDevice trackMsg:
            Log.Warning($"Ignoring TrackDevice request for {trackMsg.GroupId}. This actor is responsible for {GroupId}.");
            break;
        case RequestDeviceList deviceList:
            Sender.Tell(new ReplyDeviceList(deviceList.RequestId, new HashSet<string>(deviceIdToActor.Keys)));
            break;
        case Terminated t:
            var deviceId = actorToDeviceId[t.ActorRef];
            Log.Info($"Device actor for {deviceId} has been terminated");
            actorToDeviceId.Remove(t.ActorRef);
            deviceIdToActor.Remove(deviceId);
            break;
    }
}

public static Props Props(string groupId) => Akka.Actor.Props.Create(() => new DeviceGroup(groupId));
}
}

```

We almost have everything to test the removal of devices. What is missing is:

- Stopping a device actor from our test case, from the outside: any actor can be stopped by simply sending a special the built-in message, `PoisonPill`, which instructs the actor to stop.
- Be notified once the device actor is stopped: we can use the *Death Watch* facility for this purpose, too. Thankfully the `TestProbe` has two messages that we can easily use, `Watch()` to watch a specific actor, and `ExpectTerminated` to assert that the watched actor has been terminated.

We add two more test cases now. In the first, we just test that we get back the list of proper IDs once we have added a few devices. The second test case makes sure that the device ID is properly removed after the device actor has been stopped:

[!code-csharpDeviceGroupSpec.scala]

```

[Fact]
public void DeviceGroup_actor_must_be_able_to_register_a_device_actor()
{

```

```

        var probe = CreateTestProbe();
        var groupActor = Sys.ActorOf(DeviceGroup.Props("group"));

        groupActor.Tell(new RequestTrackDevice("group", "device1"), probe.Ref);
        probe.ExpectMsg<DeviceRegistered>();
        var deviceActor1 = probe.LastSender;

        groupActor.Tell(new RequestTrackDevice("group", "device2"), probe.Ref);
        probe.ExpectMsg<DeviceRegistered>();
        var deviceActor2 = probe.LastSender;
        deviceActor1.Should().NotBe(deviceActor2);

        // Check that the device actors are working
        deviceActor1.Tell(new RecordTemperature(requestId: 0, value: 1.0), probe.Ref);
        probe.ExpectMsg<TemperatureRecorded>(s => s.RequestId == 0);
        deviceActor2.Tell(new RecordTemperature(requestId: 1, value: 2.0), probe.Ref);
        probe.ExpectMsg<TemperatureRecorded>(s => s.RequestId == 1);
    }

    [Fact]
    public void DeviceGroup_actor_must_ignore_requests_for_wrong_groupId()
    {
        var probe = CreateTestProbe();
        var groupActor = Sys.ActorOf(DeviceGroup.Props("group"));

        groupActor.Tell(new RequestTrackDevice("wrongGroup", "device1"), probe.Ref);
        probe.ExpectNoMsg(TimeSpan.FromMilliseconds(500));
    }
}

```

Device Manager

The only part that remains now is the entry point for our device manager component. This actor is very similar to the device group actor, with the only difference that it creates device group actors instead of device actors:

[!code-csharpDeviceManager.scala]

```

public static partial class MainDeviceGroup
{
    #region device-manager-msgs
    public sealed class RequestTrackDevice
    {
        public RequestTrackDevice(string groupId, string deviceId)
        {
            GroupId = groupId;
            DeviceId = deviceId;
        }

        public string GroupId { get; }
        public string DeviceId { get; }
    }

    public sealed class DeviceRegistered
    {
        public static DeviceRegistered Instance { get; } = new DeviceRegistered();
        private DeviceRegistered() { }
    }
    #endregion

    public class DeviceManager : UntypedActor
    {
        private Dictionary<string, IActorRef> groupIdToActor = new Dictionary<string, IActorRef>();
        private Dictionary<IActorRef, string> actorToGroupId = new Dictionary<IActorRef, string>();

        protected override void PreStart() => Log.Info("DeviceManager started");
        protected override void PostStop() => Log.Info("DeviceManager stopped");
    }
}

```

```

protected ILoggingAdapter Log { get; } = Context.GetLogger();

protected override void OnReceive(object message)
{
    switch (message)
    {
        case RequestTrackDevice trackMsg:
            if (groupIdToActor.TryGetValue(trackMsg.GroupId, out var actorRef))
            {
                actorRef.Forward(trackMsg);
            }
            else
            {
                Log.Info($"Creating device group actor for {trackMsg.GroupId}");
                var groupActor = Context.ActorOf(DeviceGroup.Props(trackMsg.GroupId), $"group-{trackMsg.GroupId}");
                Context.Watch(groupActor);
                groupActor.Forward(trackMsg);
                groupIdToActor.Add(trackMsg.GroupId, groupActor);
                actorToGroupId.Add(groupActor, trackMsg.GroupId);
            }
            break;
        case Terminated t:
            var groupId = actorToGroupId[t.ActorRef];
            Log.Info($"Device group actor for {groupId} has been terminated");
            actorToGroupId.Remove(t.ActorRef);
            groupIdToActor.Remove(groupId);
            break;
    }
}

public static Props Props(string groupId) => Akka.Actor.Props.Create<DeviceManager>();
}
}

```

We leave tests of the device manager as an exercise as it is very similar to the tests we have written for the group actor.

What is Next?

We have now a hierarchical component for registering and tracking devices and recording measurements. We have seen some conversation patterns like:

- Request-respond (for temperature recordings).
- Delegate-respond (for registration of devices).
- Create-watch-terminate (for creating the group and device actor as children).

In the next chapter, we will introduce group query capabilities, which will establish a new conversation pattern of scatter-gather. In particular, we will implement the functionality that allows users to query the status of all the devices belonging to a group.

Part 4: Querying a Group of Devices

The conversational patterns we have seen so far were simple in the sense that they required no or little state to be kept in the actor that is only relevant to the conversation. Our device actors either simply returned a reading, which required no state change, recorded a temperature, which was required an update of a single field, or in the most complex case, managing groups and devices, we had to add or remove simple entries from a map.

In this chapter, we will see a more complex example. Our goal is to add a new service to the group device actor, one which allows querying the temperature from all running devices. Let us start by investigating how we want our query API to behave.

The very first issue we face is that the set of devices is dynamic, and each device is represented by an actor that can stop at any time. At the beginning of the query, we need to ask all of the device actors for the current temperature that we know about. However, during the lifecycle of the query:

- A device actor may stop and not respond back with a temperature reading.
- A new device actor might start up, but we missed asking it for the current temperature.

There are many approaches that can be taken to address these issues, but the important point is to settle on what is the desired behavior. We will pick the following two guarantees:

- When a query arrives at the group, the group actor takes a *snapshot* of the existing device actors and will only ask those for the temperature. Actors that are started *after* the arrival of the query are simply ignored.
- When an actor stops during the query without answering (i.e. before all the actors we asked for the temperature responded) we simply report back that fact to the sender of the query message.

Apart from device actors coming and going dynamically, some actors might take a long time to answer, for example, because they are stuck in an accidental infinite loop, or because they failed due to a bug and dropped our request. Ideally, we would like to give a deadline to our query:

- The query is considered completed if either all actors have responded (or confirmed being stopped), or we reach the deadline.

Given these decisions, and the fact that a device might not have a temperature to record, we can define four states that each device can be in, according to the query:

- It has a temperature available: `Temperature`.
- It has responded, but has no temperature available yet: `TemperatureNotAvailable`.
- It has stopped before answering: `DeviceNotAvailable`.
- It did not respond before the deadline: `DeviceTimedOut`.

Summarizing these in message types we can add the following to `DeviceGroup`:

[!code-csharpDeviceGroup.cs]

```
public sealed class RequestAllTemperatures
{
    public RequestAllTemperatures(long requestId)
    {
        RequestId = requestId;
    }

    public long RequestId { get; }

    public sealed class RespondAllTemperatures
    {
```

```

public RespondAllTemperatures(long requestId, Dictionary<string, ITemperatureReading> temperatures)
{
    RequestId = requestId;
    Temperatures = temperatures;
}

public long RequestId { get; }
public Dictionary<string, ITemperatureReading> Temperatures { get; }
}

public interface ITemperatureReading
{
}

public sealed class Temperature : ITemperatureReading
{
    public Temperature(double value)
    {
        Value = value;
    }

    public double Value { get; }
}

public sealed class TemperatureNotAvailable : ITemperatureReading
{
    public static TemperatureNotAvailable Instance { get; } = new TemperatureNotAvailable();
    private TemperatureNotAvailable() { }
}

public sealed class DeviceNotAvailable : ITemperatureReading
{
    public static DeviceNotAvailable Instance { get; } = new DeviceNotAvailable();
    private DeviceNotAvailable() { }
}

public sealed class DeviceTimedOut : ITemperatureReading
{
    public static DeviceTimedOut Instance { get; } = new DeviceTimedOut();
    private DeviceTimedOut() { }
}

```

Implementing the Query

One of the approaches for implementing the query could be to add more code to the group device actor. While this is possible, in practice this can be very cumbersome and error prone. When we start a query, we need to take a snapshot of the devices present at the start of the query and start a timer so that we can enforce the deadline. Unfortunately, during the time we execute a query *another query* might just arrive. For this other query, of course, we need to keep track of the exact same information but isolated from the previous query. This complicates the code and also poses some problems. For example, we would need a data structure that maps the `IActorRef`s of the devices to the queries that use that device, so that they can be notified when such a device terminates, i.e. a `Terminated` message is received.

There is a much simpler approach that is superior in every way, and it is the one we will implement. We will create an actor that represents a *single query* and which performs the tasks needed to complete the query on behalf of the group actor. So far we have created actors that belonged to classical domain objects, but now, we will create an actor that represents a process or task rather than an entity. This move keeps our group device actor simple and gives us better ways to test the query capability in isolation.

First, we need to design the lifecycle of our query actor. This consists of identifying its initial state, then the first action to be taken by the actor, then, the cleanup if necessary. There are a few things the query should need to be able to work:

- The snapshot of active device actors to query, and their IDs.
- The `requestID` of the request that started the query (so we can include it in the reply).
- The `IActorRef` of the actor who sent the group actor the query. We will send the reply to this actor directly.
- A timeout parameter, how long the query should wait for replies. Keeping this as a parameter will simplify testing.

Since we need to have a timeout for how long we are willing to wait for responses, it is time to introduce a new feature that we have not used yet: timers. Akka has a built-in scheduler facility for this exact purpose. Using it is simple, the `ScheduleTellOnceCancelable(time, actorRef, message, sender)` method will schedule the message `message` into the future by the specified `time` and send it to the actor `actorRef`. To implement our query timeout we need to create a message that represents the query timeout. We create a simple message `CollectionTimeout` without any parameters for this purpose. The return value from `ScheduleTellOnceCancelable` is a `ICancellable` which can be used to cancel the timer if the query finishes successfully in time. Getting the scheduler is possible from the `ActorSystem`, which, in turn, is accessible from the actor's context: `Context.System.Scheduler`.

At the start of the query, we need to ask each of the device actors for the current temperature. To be able to quickly detect devices that stopped before they got the `ReadTemperature` message we will also watch each of the actors. This way, we get `Terminated` messages for those that stop during the lifetime of the query, so we don't need to wait until the timeout to mark these as not available.

Putting together all these, the outline of our actor looks like this:

[!code-csharpDeviceGroupQueryInProgress.cs]

```
public class DeviceGroupQuery : UntypedActor
{
    private ICancellable queryTimeoutTimer;

    public DeviceGroupQuery(Dictionary<IActorRef, string> actorToDeviceId, long requestId, IActorRef requester, TimeSpan timeout)
    {
        ActorToDeviceId = actorToDeviceId;
        RequestId = requestId;
        Requester = requester;
        Timeout = timeout;

        queryTimeoutTimer = Context.System.Scheduler.ScheduleTellOnceCancelable(timeout, Self, Collecti
onTimeout.Instance, Self);
    }

    protected override void PreStart()
    {
        foreach (var deviceActor in ActorToDeviceId.Keys)
        {
            Context.Watch(deviceActor);
            deviceActor.Tell(new ReadTemperature(0));
        }
    }

    protected override void PostStop()
    {
        queryTimeoutTimer.Cancel();
    }

    protected ILoggingAdapter Log { get; } = Context.GetLogger();
    public Dictionary<IActorRef, string> ActorToDeviceId { get; }
    public long RequestId { get; }
    public IActorRef Requester { get; }
    public TimeSpan Timeout { get; }
}
```

```

protected override void OnReceive(object message)
{
}

public static Props Props(Dictionary<IActorRef, string> actorToDeviceId, long requestId, IActorRef requester, TimeSpan timeout) =>
    Akka.Actor.Props.Create(() => new DeviceGroupQuery(actorToDeviceId, requestId, requester, timeout));
}

```

The query actor, apart from the pending timer, has one stateful aspect about it: the actors that did not answer so far or, from the other way around, the set of actors that have replied or stopped. One way to track this state is to create a mutable field in the actor. There is another approach. It is also possible to change how the actor responds to messages. By default, the `OnReceive` block defines the behavior of the actor, but it is possible to change it, several times, during the life of the actor. This is possible by calling `Context.Become(newBehavior)` where `newBehavior` is anything with type `UntypedReceive`. A `UntypedReceive` is just a function (or an object, if you like) that can be returned from another function. We will leverage this feature to track the state of our actor.

As the first step, instead of defining `OnReceive` directly, we delegate to another function to create the `UntypedReceive`, which we will call `WaitingForReplies`. This will keep track of two changing values, a `Dictionary` of already received replies and a `HashSet` of actors that we still wait on. We have three events that we should act on. We can receive a `RespondTemperature` message from one of the devices. Second, we can receive a `Terminated` message for a device actor that has been stopped in the meantime. Finally, we can reach the deadline and receive a `CollectionTimeout`. In the first two cases, we need to keep track of the replies, which we now simply delegate to a method `ReceivedResponse` which we will discuss later. In the case of timeout, we need to simply take all the actors that have not yet replied yet (the members of the set `stillWaiting`) and put a `DeviceTimedout` as the status in the final reply. Then we reply to the submitter of the query with the collected results and stop the query actor:

[!code-csharpDeviceGroupQuery.cs]

```

public DeviceGroupQuery(Dictionary<IActorRef, string> actorToDeviceId, long requestId, IActorRef requester, TimeSpan timeout)
{
    ActorToDeviceId = actorToDeviceId;
    RequestId = requestId;
    Requester = requester;
    Timeout = timeout;

    queryTimeoutTimer = Context.System.Scheduler.ScheduleTellOnceCancelable(timeout, Self, Collecti
onTimeout.Instance, Self);

    Become(WaitingForReplies(new Dictionary<string, ITemperatureReading>(), new HashSet<IActorRef>(
ActorToDeviceId.Keys)));
}

protected override void PreStart()
{
    foreach (var deviceActor in ActorToDeviceId.Keys)
    {
        Context.Watch(deviceActor);
        deviceActor.Tell(new ReadTemperature());
    }
}

protected override void PostStop()
{
    queryTimeoutTimer.Cancel();
}

protected ILoggingAdapter Log { get; } = Context.GetLogger();
public Dictionary<IActorRef, string> ActorToDeviceId { get; }

```

```

public long RequestId { get; }
public IActorRef Requester { get; }
public TimeSpan Timeout { get; }

public UntypedReceive WaitingForReplies(
    Dictionary<string, ITemperatureReading> repliesSoFar,
    HashSet<IActorRef> stillWaiting)
{
    return message =>
    {
        switch (message)
        {
            case RespondTemperature response when response.RequestId == 0:
                var deviceActor = Sender;
                ITemperatureReading reading = null;
                if (response.Value.HasValue)
                {
                    reading = new Temperature(response.Value.Value);
                }
                else
                {
                    reading = TemperatureNotAvailable.Instance;
                }
                ReceivedResponse(deviceActor, reading, stillWaiting, repliesSoFar);
                break;
            case Terminated t:
                ReceivedResponse(t.ActorRef, DeviceNotAvailable.Instance, stillWaiting, repliesSoFar);
                break;
            case CollectionTimeout _:
                var replies = new Dictionary<string, ITemperatureReading>(repliesSoFar);
                foreach (var actor in stillWaiting)
                {
                    var deviceId = ActorToDeviceId[actor];
                    replies.Add(deviceId, DeviceTimedOut.Instance);
                }
                Requester.Tell(new RespondAllTemperatures(RequestId, replies));
                Context.Stop(Self);
                break;
        }
    };
}
}

```

What is not yet clear is how we will "mutate" the `answersSoFar` and `stillWaiting` data structures. One important thing to note is that the function `WaitingForReplies` **does not handle the messages directly. It returns an `UntypedReceive` function that will handle the messages.** This means that if we call `WaitingForReplies` again, with different parameters, then it returns a brand new `UntypedReceive` that will use those new parameters. We have seen how we can install the initial `UntypedReceive` by simply returning it from `OnReceive`. In order to install a new one, to record a new reply, for example, we need some mechanism. This mechanism is the method `Context.Become(newReceive)` which will *change* the actor's message handling function to the provided `newReceive` function. You can imagine that before starting, your actor automatically calls `Context.Become(receive)`, i.e. installing the `UntypedReceive` function that is returned from `OnReceive`. This is another important observation: **it is not `OnReceive` that handles the messages, it just returns a `UntypedReceive` function that will actually handle the messages.**

We now have to figure out what to do in `ReceivedResponse`. First, we need to record the new result in the map `repliesSoFar` and remove the actor from `stillWaiting`. The next step is to check if there are any remaining actors we are waiting for. If there is none, we send the result of the query to the original requester and stop the query actor. Otherwise, we need to update the `repliesSoFar` and `stillWaiting` structures and wait for more messages.

In the code before, we treated `Terminated` as the implicit response `DeviceNotAvailable`, so `ReceivedResponse` does not need to do anything special. However, there is one small task we still need to do. It is possible that we receive a proper response from a device actor, but then it stops during the lifetime of the query. We don't want this second event to overwrite the already received reply. In other words, we don't want to receive `Terminated` after we recorded the response. This is simple to achieve by calling `Context.Unwatch(ref)`. This method also ensures that we don't receive `Terminated` events that are already in the mailbox of the actor. It is also safe to call this multiple times, only the first call will have any effect, the rest is simply ignored.

With all this knowledge, we can create the `ReceivedResponse` method:

[!code-csharpDeviceGroupQuery.cs]

```
public void ReceivedResponse(
    IActorRef deviceActor,
    ITemperatureReading reading,
    HashSet<IActorRef> stillWaiting,
    Dictionary<string, ITemperatureReading> repliesSoFar)
{
    Context.Unwatch(deviceActor);
    var deviceId = ActorToDeviceId[deviceActor];
    stillWaiting.Remove(deviceActor);

    repliesSoFar.Add(deviceId, reading);

    if (stillWaiting.Count == 0)
    {
        Requester.Tell(new RespondAllTemperatures(RequestId, repliesSoFar));
        Context.Stop(Self);
    }
    else
    {
        Context.Become(WaitingForReplies(repliesSoFar, stillWaiting));
    }
}
```

It is quite natural to ask at this point, what have we gained by using the `Context.Become()` trick instead of just making the `repliesSoFar` and `stillWaiting` structures mutable fields of the actor? In this simple example, not that much. The value of this style of state keeping becomes more evident when you suddenly have *more kinds* of states. Since each state might have temporary data that is relevant itself, keeping these as fields would pollute the global state of the actor, i.e. it is unclear what fields are used in what state. Using parameterized `OnReceive` "factory" methods we can keep data private that is only relevant to the state. It is still a good exercise to rewrite the query using mutable fields instead of `Context.Become()`. However, it is recommended to get comfortable with the solution we have used here as it helps structuring more complex actor code in a cleaner and more maintainable way.

Our query actor is now done:

[!code-csharpDeviceGroupQuery.cs]

```
public sealed class CollectionTimeout
{
    public static CollectionTimeout Instance { get; } = new CollectionTimeout();
    private CollectionTimeout() { }

    public class DeviceGroupQuery : UntypedActor
    {
        private ICancelable queryTimeoutTimer;

        #region query-state
        public DeviceGroupQuery(Dictionary<IActorRef, string> actorToDeviceId, long requestId, IActorRef requester, TimeSpan timeout)
        {
```

```

ActorToDeviceId = actorToDeviceId;
RequestId = requestId;
Requester = requester;
Timeout = timeout;

queryTimeoutTimer = Context.System.Scheduler.ScheduleTellOnceCancelable(timeout, Self, Collecti
onTimeout.Instance, Self);

Become(WaitingForReplies(new Dictionary<string, ITemperatureReading>(), new HashSet<IActorRef>(
ActorToDeviceId.Keys)));
}

protected override void PreStart()
{
    foreach (var deviceActor in ActorToDeviceId.Keys)
    {
        Context.Watch(deviceActor);
        deviceActor.Tell(new ReadTemperature(0));
    }
}

protected override void PostStop()
{
    queryTimeoutTimer.Cancel();
}

protected ILoggingAdapter Log { get; } = Context.GetLogger();
public Dictionary<IActorRef, string> ActorToDeviceId { get; }
public long RequestId { get; }
public IActorRef Requester { get; }
public TimeSpan Timeout { get; }

public UntypedReceive WaitingForReplies(
    Dictionary<string, ITemperatureReading> repliesSoFar,
    HashSet<IActorRef> stillWaiting)
{
    return message =>
    {
        switch (message)
        {
            case RespondTemperature response when response.RequestId == 0:
                var deviceActor = Sender;
                ITemperatureReading reading = null;
                if (response.Value.HasValue)
                {
                    reading = new Temperature(response.Value.Value);
                }
                else
                {
                    reading = TemperatureNotAvailable.Instance;
                }
                ReceivedResponse(deviceActor, reading, stillWaiting, repliesSoFar);
                break;
            case Terminated t:
                ReceivedResponse(t.ActorRef, DeviceNotAvailable.Instance, stillWaiting, repliesSoFa
r);
                break;
            case CollectionTimeout _:
                var replies = new Dictionary<string, ITemperatureReading>(repliesSoFar);
                foreach (var actor in stillWaiting)
                {
                    var deviceId = ActorToDeviceId[actor];
                    replies.Add(deviceId, DeviceTimedOut.Instance);
                }
                Requester.Tell(new RespondAllTemperatures(RequestId, replies));
                Context.Stop(Self);
                break;
        }
    };
}

```

```
}
```

Testing

Now let's verify the correctness of the query actor implementation. There are various scenarios we need to test individually to make sure everything works as expected. To be able to do this, we need to simulate the device actors somehow to exercise various normal or failure scenarios. Thankfully we took the list of collaborators (actually a `Dictionary`) as a parameter to the query actor, so we can easily pass in `TestProbe` references. In our first test, we try out the case when there are two devices and both report a temperature:

[!code-csharpDeviceGroupQuerySpec.cs]

```
[Fact]
public void DeviceGroupQuery_must_return_temperature_value_for_working_devices()
{
    var requester = CreateTestProbe();

    var device1 = CreateTestProbe();
    var device2 = CreateTestProbe();

    var queryActor = Sys.ActorOf(DeviceGroupQuery.Props(
        actorToDeviceId: new Dictionary<IActorRef, string> { [device1.Ref] = "device1", [device2.Re
f] = "device2" },
        requestId: 1,
        requester: requester.Ref,
        timeout: TimeSpan.FromSeconds(3)
    ));

    device1.ExpectMsg<ReadTemperature>(read => read.RequestId == 0);
    device2.ExpectMsg<ReadTemperature>(read => read.RequestId == 0);

    queryActor.Tell(new RespondTemperature(requestId: 0, value: 1.0), device1.Ref);
    queryActor.Tell(new RespondTemperature(requestId: 0, value: 2.0), device2.Ref);

    requester.ExpectMsg<RespondAllTemperatures>(msg =>
        msg.Temperatures["device1"].AsInstanceOf<Temperature>().Value == 1.0 &&
        msg.Temperatures["device2"].AsInstanceOf<Temperature>().Value == 2.0 &&
        msg.RequestId == 1);
}
```

That was the happy case, but we know that sometimes devices cannot provide a temperature measurement. This scenario is just slightly different from the previous:

[!code-csharpDeviceGroupQuerySpec.cs]

```
[Fact]
public void DeviceGroupQuery_must_return_TemperatureNotAvailable_for_devices_with_no_readings()
{
    var requester = CreateTestProbe();

    var device1 = CreateTestProbe();
    var device2 = CreateTestProbe();

    var queryActor = Sys.ActorOf(DeviceGroupQuery.Props(
        actorToDeviceId: new Dictionary<IActorRef, string> { [device1.Ref] = "device1", [device2.Re
f] = "device2" },
        requestId: 1,
        requester: requester.Ref,
        timeout: TimeSpan.FromSeconds(3)
    ));

    device1.ExpectMsg<ReadTemperature>(read => read.RequestId == 0);
```

```

        device2.ExpectMsg<ReadTemperature>(read => read.RequestId == 0);

        queryActor.Tell(new RespondTemperature(requestId: 0, value: null), device1.Ref);
        queryActor.Tell(new RespondTemperature(requestId: 0, value: 2.0), device2.Ref);

        requester.ExpectMsg<RespondAllTemperatures>(msg =>
            msg.Temperatures["device1"] is TemperatureNotAvailable &&
            msg.Temperatures["device2"].AsInstanceOf<Temperature>().Value == 2.0 &&
            msg.RequestId == 1);
    }
}

```

We also know, that sometimes device actors stop before answering:

[!code-csharpDeviceGroupQuerySpec.cs]

```

[Fact]
public void DeviceGroupQuery_must_return_DeviceNotAvailable_if_device_stops_before_answering()
{
    var requester = CreateTestProbe();

    var device1 = CreateTestProbe();
    var device2 = CreateTestProbe();

    var queryActor = Sys.ActorOf(DeviceGroupQuery.Props(
        actorToDeviceId: new Dictionary<IActorRef, string> { [device1.Ref] = "device1", [device2.Re
f] = "device2" },
        requestId: 1,
        requester: requester.Ref,
        timeout: TimeSpan.FromSeconds(3)
    ));

    device1.ExpectMsg<ReadTemperature>(read => read.RequestId == 0);
    device2.ExpectMsg<ReadTemperature>(read => read.RequestId == 0);

    queryActor.Tell(new RespondTemperature(requestId: 0, value: 1.0), device2.Ref);
    device2.Tell(PoisonPill.Instance);

    requester.ExpectMsg<RespondAllTemperatures>(msg =>
        msg.Temperatures["device1"].AsInstanceOf<Temperature>().Value == 1.0 &&
        msg.Temperatures["device2"] is DeviceNotAvailable &&
        msg.RequestId == 1);
}

```

If you remember, there is another case related to device actors stopping. It is possible that we get a normal reply from a device actor, but then receive a `Terminated` for the same actor later. In this case, we would like to keep the first reply and not mark the device as `DeviceNotAvailable`. We should test this, too:

[!code-csharpDeviceGroupQuerySpec.cs]

```

[Fact]
public void DeviceGroupQuery_must_return_temperature_reading_even_if_device_stops_after_answering()
{
    var requester = CreateTestProbe();

    var device1 = CreateTestProbe();
    var device2 = CreateTestProbe();

    var queryActor = Sys.ActorOf(DeviceGroupQuery.Props(
        actorToDeviceId: new Dictionary<IActorRef, string> { [device1.Ref] = "device1", [device2.Re
f] = "device2" },
        requestId: 1,
        requester: requester.Ref,
        timeout: TimeSpan.FromSeconds(3)
    ));
}

```

```

device1.ExpectMsg<ReadTemperature>(read => read.RequestId == 0);
device2.ExpectMsg<ReadTemperature>(read => read.RequestId == 0);

queryActor.Tell(new RespondTemperature(requestId: 0, value: 1.0), device1.Ref);
queryActor.Tell(new RespondTemperature(requestId: 0, value: 2.0), device2.Ref);
device2.Tell(PoisonPill.Instance);

requester.ExpectMsg<RespondAllTemperatures>(msg =>
    msg.Temperatures["device1"].AsInstanceOf<Temperature>().Value == 1.0 &&
    msg.Temperatures["device2"].AsInstanceOf<Temperature>().Value == 2.0 &&
    msg.RequestId == 1);
}

```

The final case is when not all devices respond in time. To keep our test relatively fast, we will construct the `DeviceGroupQuery` actor with a smaller timeout:

[!code-csharpDeviceGroupQuerySpec.cs]

```

[Fact]
public void DeviceGroupQuery_must_return_DeviceTimedOut_if_device_does_not_answer_in_time()
{
    var requester = CreateTestProbe();

    var device1 = CreateTestProbe();
    var device2 = CreateTestProbe();

    var queryActor = Sys.ActorOf(DeviceGroupQuery.Props(
        actorToDeviceId: new Dictionary<IActorRef, string> { [device1.Ref] = "device1", [device2.Re
f] = "device2" },
        requestId: 1,
        requester: requester.Ref,
        timeout: TimeSpan.FromSeconds(1)
    ));

    device1.ExpectMsg<ReadTemperature>(read => read.RequestId == 0);
    device2.ExpectMsg<ReadTemperature>(read => read.RequestId == 0);

    queryActor.Tell(new RespondTemperature(requestId: 0, value: 1.0), device1.Ref);

    requester.ExpectMsg<RespondAllTemperatures>(msg =>
        msg.Temperatures["device1"].AsInstanceOf<Temperature>().Value == 1.0 &&
        msg.Temperatures["device2"] is DeviceTimedOut &&
        msg.RequestId == 1);
}

```

Our query works as expected now, it is time to include this new functionality in the `DeviceGroup` actor now.

Adding the Query Capability to the Group

Including the query feature in the group actor is fairly simple now. We did all the heavy lifting in the query actor itself, the group actor only needs to create it with the right initial parameters and nothing else.

[!code-csharpDeviceGroupQueryInProgress.cs]

```

public class DeviceGroup : UntypedActor
{
    private Dictionary<string, IActorRef> deviceIdToActor = new Dictionary<string, IActorRef>();
    private Dictionary<IActorRef, string> actorToDeviceId = new Dictionary<IActorRef, string>();
    private long nextCollectionId = 0L;

    public DeviceGroup(string groupId)
    {
    }
}

```

```

        GroupId = groupId;
    }

    protected override void PreStart() => Log.Info($"Device group {GroupId} started");
    protected override void PostStop() => Log.Info($"Device group {GroupId} stopped");

    protected ILoggingAdapter Log { get; } = Context.GetLogger();
    protected string GroupId { get; }

    protected override void OnReceive(object message)
    {
        switch (message)
        {
            case RequestAllTemperatures r:
                Context.ActorOf(DeviceGroupQuery.Props(actorToDeviceId, r.RequestId, Sender, TimeSpan.FromSeconds(3)));
                break;
        }
    }

    public static Props Props(string groupId) => Akka.Actor.Props.Create(() => new DeviceGroup(groupId));
}
}

```

It is probably worth reiterating what we said at the beginning of the chapter: By keeping the temporary state that is only relevant to the query itself in a separate actor we keep the group actor implementation very simple. It delegates everything to child actors and therefore does not have to keep state that is not relevant to its core business. Also, multiple queries can now run parallel to each other, in fact, as many as needed. In our case querying an individual device actor is a fast operation, but if this were not the case, for example, because the remote sensors need to be contacted over the network, this design would significantly improve throughput.

We close this chapter by testing that everything works together. This test is just a variant of the previous ones, now exercising the group query feature:

[!code-csharpDeviceGroupSpec.cs]

```

[Fact]
public void DeviceGroup_actor_must_be_able_to_collect_temperatures_from_all_active_devices()
{
    var probe = CreateTestProbe();
    var groupActor = Sys.ActorOf(DeviceGroup.Props("group"));

    groupActor.Tell(new RequestTrackDevice("group", "device1"), probe.Ref);
    probe.ExpectMsg<DeviceRegistered>();
    var deviceActor1 = probe.LastSender;

    groupActor.Tell(new RequestTrackDevice("group", "device2"), probe.Ref);
    probe.ExpectMsg<DeviceRegistered>();
    var deviceActor2 = probe.LastSender;

    groupActor.Tell(new RequestTrackDevice("group", "device3"), probe.Ref);
    probe.ExpectMsg<DeviceRegistered>();
    var deviceActor3 = probe.LastSender;

    // Check that the device actors are working
    deviceActor1.Tell(new RecordTemperature(requestId: 0, value: 1.0), probe.Ref);
    probe.ExpectMsg<TemperatureRecorded>(s => s.RequestId == 0);
    deviceActor2.Tell(new RecordTemperature(requestId: 1, value: 2.0), probe.Ref);
    probe.ExpectMsg<TemperatureRecorded>(s => s.RequestId == 1);
    // No temperature for device3

    groupActor.Tell(new RequestAllTemperatures(0), probe.Ref);
    probe.ExpectMsg<RespondAllTemperatures>(msg =>
        msg.Temperatures["device1"].AsInstanceOf<Temperature>().Value == 1.0 &&
        msg.Temperatures["device2"].AsInstanceOf<Temperature>().Value == 2.0 &&
    )
}

```

```
    msg.Temperatures["device3"] is TemperatureNotAvailable &&
    msg.RequestId == 0);
}
```

Use-case and Deployment Scenarios

Console Application

```
PM> install-package Akka
PM> install-package Akka.Remote

using Akka;
using Akka.Actor;
using Akka.Configuration;

namespace Foo.Bar
{
    class Program
    {
        static void Main(string[] args)
        {
            //configure remoting for localhost:8081
            var fluentConfig = FluentConfig.Begin()
                .StartRemotingOn("localhost", 8081)
                .Build();

            using (var system = ActorSystem.Create("my-actor-server", fluentConfig))
            {
                //start two services
                var service1= system.ActorOf<Service1>("service1");
                var service2 = system.ActorOf<Service2>("service2");
                Console.ReadKey();
            }
        }
    }
}
```

ASP.NET

Creating the Akka.NET resources

Hosting inside an ASP.NET application is easy. The Global.asax would be the designated place to start.

```
public class MvcApplication : System.Web.HttpApplication
{
    protected static ActorSystem ActorSystem;
    //here you would store your toplevel actor-refs
    protected static IActorRef MyActor;

    protected void Application_Start()
    {
        //your mvc config. Does not really matter if you initialise
        //your actor system before or after

        ActorSystem = ActorSystem.Create("app");
        //here you would register your toplevel actors
        MyActor = ActorSystem.ActorOf<MyActor>();
    }
}
```

As you can see the main point here is keeping a static reference to your `ActorSystem`. This ensures it won't be accidentally garbage collected and gets disposed and created with the start and stop events of your web application.

[!WARNING] Although hosting inside an ASP.NET Application is easy. A **word of caution**: When you are hosting inside of `IIS` the applicationpool your app lives in could be stopped and started at the whim of `IIS`. This in turn means your `ActorSystem` could be stopped at any given time.

Typically you use a very lightweight `ActorSystem` inside ASP.NET applications, and offload heavy-duty work to a separate Windows Service via Akka.Remote / Akka.Cluster

Interaction between Controllers and Akka.NET

In the sample below, we use an Web API Controller:

```
public class SomeController : ApiController
{
    //expose your endpoint as async
    public async Task<SomeResult> Post(SomeRequest someRequest)
    {
        //send a message based on your incoming arguments to one of the actors you created earlier
        //and await the result by sending the message to `Ask`
        var result = await MvcApplication.MyActor.Ask<SomeResult>(new SomeMessage(someRequest.SomeArg1, someRequest.SomeArg2));
        return result;
    }
}
```

Windows Service

For windows service deployment its recommended to use [TopShelf](#) to build your Windows Services. It radically simplifies hosting Windows Services.

The quickest way to get started with TopShelf is by creating a Console Application. Which would look like this:

Program.cs

```
using Akka.Actor;
using Topshelf;

class Program
{
    static void Main(string[] args)
    {
        HostFactory.Run(x =>
        {
            x.Service<MyActorService>(s =>
            {
                s.ConstructUsing(n => new MyActorService());
                s.WhenStarted(service => service.Start());
                s.WhenStopped(service => service.Stop());
                //continue and restart directives are also available
            });

            x.RunAsLocalSystem();
            x.UseAssemblyInfoForServiceInfo();
        });
    }
}
```

```

/// <summary>
/// This class acts as an interface between your application and TopShelf
/// </summary>
public class MyActorService
{
    private ActorSystem mySystem;

    public void Start()
    {
        //this is where you setup your actor system and other things
        mySystem = ActorSystem.Create("MySystem");
    }

    public async void Stop()
    {
        //this is where you stop your actor system
        await mySystem.Terminate();
    }
}

```

The above example is the simplest way imaginable. However there are also other styles of integration with TopShelf that give you more control.

Installing with Topshelf is as easy as calling `myConsoleApp.exe install` on the command line.

For all the options and settings check out their [docs](#).

Azure PaaS Worker Role

The following sample assumes that you have created a new Azure Paas Cloud Service that contains a single empty Worker Role. The Cloud Service project templates are added to Visual Studio by installing the [Azure .Net SDK](#).

The Worker Role implementation can be tested locally using the Azure Compute Emulator before deploying to the cloud. The MSDN Azure article "[Using Emulator Express to Run and Debug a Cloud Service Locally](#)" describes this in more detail.

The Azure PaaS Worker Role implementation is very similar to the [Akka.Net Windows Service Sample](#). The quickest way to get started with Akka.Net is to create a simple Worker Role which invokes the top-level user-actor in the `RunAsync()` method, as follows:

WorkerRole.cs

```

using Akka.Actor;

namespace MyActorWorkerRole
{
    public class WorkerRole : RoleEntryPoint
    {
        private readonly CancellationTokenSource cancellationTokenSource = new CancellationTokenSource();
        private readonly ManualResetEvent runCompleteEvent = new ManualResetEvent(false);

        private ActorSystem _actorSystem;

        public override bool OnStart()
        {
            // Setup the Actor System
            _actorSystem = ActorSystem.Create("MySystem");

            return (base.OnStart());
        }
    }
}

```

```
public override void OnStop()
{
    this.cancellationTokenSource.Cancel();
    this.runCompleteEvent.WaitOne();

    // Shutdown the Actor System
    _actorSystem.Shutdown();

    base.OnStop();
}

public override void Run()
{
    try
    {
        this.RunAsync(this.cancellationTokenSource.Token).Wait();
    }
    finally
    {
        this.runCompleteEvent.Set();
    }
}

private async Task RunAsync(CancellationToken cancellationToken)
{
    // Create an instance to the top-level user Actor
    var workerRoleActor = _actorSystem.ActorOf<WorkerRoleActor>("WorkerRole");

    // Send a message to the Actor
    workerRoleActor.Tell(new WorkerRoleMessage("Hello World!"));

    while (!cancellationToken.IsCancellationRequested)
    {
        await Task.Delay(1000);
    }
}
```

Akka.NET is used by many large organizations in a big range of industries all from investment and merchant banking, retail and social media, simulation, gaming and betting, automobile and traffic systems, health care, data analytics and much more. Any system that have the need for high-throughput and low latency is a good candidate for using Akka.NET.

There is a great discussion on use-cases for Akka.NET with some good write-ups by production users here

Akka.NET Users

Transaction processing (Online Gaming, Finance/Banking, Trading, Statistics, Betting, Social Media, Telecom)

- [CellularSales](#)

Service backend (any industry, any app)

IVC Business Systems:

Sam Covington, IVC Business Systems: We had an in-house "Actor" system that we replaced with Akka.Net, which allowed us to innovate and be productive elsewhere, and not reinvent the wheel(not to mention test it to death). This back end of Microservices forms the basis of all of our products and services. We're using it in our Enterprise Social Product, and our new Livescan Office product for Livescan fingerprinting customers.

Concurrency/parallelism (any app)

- [SNL Financial \(a subsidiary of McGraw Hill\): Akka.NET Goes to Wall Street](#)

Simulation

Master/Worker, Compute Grid, MapReduce etc.

Batch processing (any industry)

Camel integration to hook up with batch data sources Actors divide and conquer the batch workloads

Communications Hub (Telecom, Web media, Mobile media)

- [EventDay: real-time conference and event management at scale with Akka.NET](#)

Gaming and Betting (MOM, online gaming, betting)

Scale up, scale out, fault-tolerance / HA

Business Intelligence/Data Mining/general purpose crunching

- [Real-time Clickstream Processing at Domain.au with Octopus Deploy and Akka.NET](#)

Internet of Things

- [Synchromatics: Real-time public transit tracking and analytics using Akka.NET](#)

Complex Event Stream Processing

- [MarkedUp Analytics: Real-time Marketing Automation with Distributed Actor Systems and Akka.NET](#)

Terminology and Concepts

In this chapter we attempt to establish a common terminology to define a solid ground for communicating about concurrent, distributed systems which Akka.NET targets. Please note that, for many of these terms, there is no single agreed definition. We simply seek to give working definitions that will be used in the scope of the Akka.NET documentation.

Concurrency vs. Parallelism

Concurrency and parallelism are related concepts, but there are small differences. Concurrency means that two or more tasks are making progress even though they might not be executing simultaneously. This can for example be realized with time slicing where parts of tasks are executed sequentially and mixed with parts of other tasks. Parallelism on the other hand arise when the execution can be truly simultaneous.

Concurrency



Parallelism



Asynchronous vs. Synchronous

A method call is considered synchronous if the caller cannot make progress until the method returns a value or throws an exception. On the other hand, an asynchronous call allows the caller to progress after a finite number of steps, and the completion of the method may be signalled via some additional mechanism (it might be a registered callback, a Task, or a message).

A synchronous API may use blocking to implement synchrony, but this is not a necessity. A very CPU intensive task might give a similar behavior as blocking. In general, it is preferred to use asynchronous APIs, as they guarantee that the system is able to progress. Actors are asynchronous by nature: an actor can progress after a message send without waiting for the actual delivery to happen.

Non-blocking vs. Blocking

We talk about blocking if the delay of one thread can indefinitely delay some of the other threads. A good example is a resource which can be used exclusively by one thread using mutual exclusion. If a thread holds on to the resource indefinitely (for example accidentally running an infinite loop) other threads waiting on the resource can not progress. In contrast, non-blocking means that no thread is able to indefinitely delay others.

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

Deadlock vs. Starvation vs. Live-lock

Deadlock arises when several participants are waiting on each other to reach a specific state to be able to progress. As none of them can progress without some other participant to reach a certain state (a "Catch-22" problem) all affected subsystems stall. Deadlock is closely related to blocking, as it is necessary that a participant thread be able to delay the progression of other threads indefinitely.

In the case of deadlock, no participants can make progress, while in contrast Starvation happens, when there are participants that can make progress, but there might be one or more that cannot. Typical scenario is the case of a naive scheduling algorithm that always selects high-priority tasks over low-priority ones. If the number of incoming high-priority tasks is constantly high enough, no low-priority ones will be ever finished.

Livelock is similar to deadlock as none of the participants make progress. The difference though is that instead of being frozen in a state of waiting for others to progress, the participants continuously change their state. An example scenario when two participants have two identical resources available. They each try to get the resource, but they also check if the other needs the resource, too. If the resource is requested by the other participant, they try to get the other instance of the resource. In the unfortunate case it might happen that the two participants "bounce" between the two resources, never acquiring it, but always yielding to the other.

Race Condition

We call it a Race condition when an assumption about the ordering of a set of events might be violated by external non-deterministic effects. Race conditions often arise when multiple threads have a shared mutable state, and the operations of thread on the state might be interleaved causing unexpected behavior. While this is a common case, shared state is not necessary to have race conditions. One example could be a client sending unordered packets (e.g UDP datagrams) P1, P2 to a server. As the packets might potentially travel via different network routes, it is possible that the server receives P2 first and P1 afterwards. If the messages contain no information about their sending order it is impossible to determine by the server that they were sent in a different order. Depending on the meaning of the packets this can cause race conditions.

[!NOTE] The only guarantee that Akka.NET provides about messages sent between a given pair of actors is that their order is always preserved. see [Message Delivery Reliability](#)

Non-blocking Guarantees (Progress Conditions)

As discussed in the previous sections blocking is undesirable for several reasons, including the dangers of deadlocks and reduced throughput in the system. In the following sections we discuss various non-blocking properties with different strength.

Wait-freedom

A method is wait-free if every call is guaranteed to finish in a finite number of steps. If a method is bounded wait-free then the number of steps has a finite upper bound.

From this definition it follows that wait-free methods are never blocking, therefore deadlock can not happen. Additionally, as each participant can progress after a finite number of steps (when the call finishes), wait-free methods are free of starvation.

Lock-freedom

Lock-freedom is a weaker property than wait-freedom. In the case of lock-free calls, infinitely often some method finishes in a finite number of steps. This definition implies that no deadlock is possible for lock-free calls. On the other hand, the guarantee that some call finishes in a finite number of steps is not enough to guarantee that all of them eventually finish. In other words, lock-freedom is not enough to guarantee the lack of starvation.

Obstruction-freedom

Obstruction-freedom is the weakest non-blocking guarantee discussed here. A method is called obstruction-free if there is a point in time after which it executes in isolation (other threads make no steps, e.g.: become suspended), it finishes in a bounded number of steps. All lock-free objects are obstruction-free, but the opposite is generally not true.

Optimistic concurrency control (OCC) methods are usually obstruction-free. The OCC approach is that every participant tries to execute its operation on the shared object, but if a participant detects conflicts from others, it rolls back the modifications, and tries again according to some schedule. If there is a point in time, where one of the participants is the only one trying, the operation will succeed.

Actor Systems

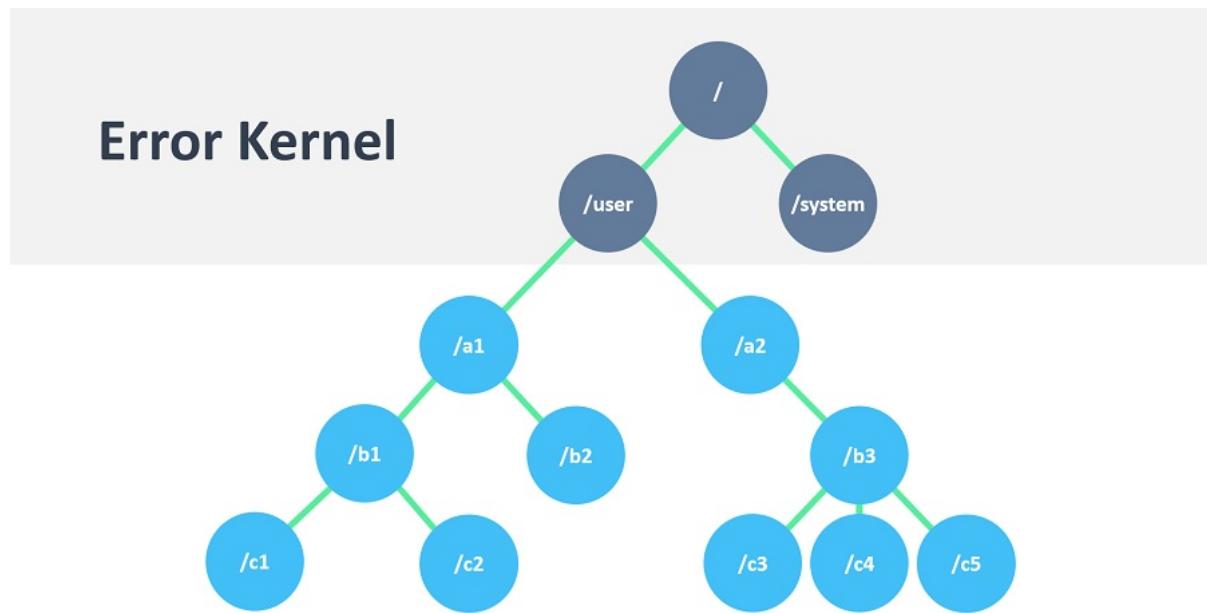
Actors are objects which encapsulate state and behavior, they communicate exclusively by exchanging messages which are placed into the recipient's mailbox. In a sense, actors are the most stringent form of object-oriented programming, but it serves better to view them as persons: while modeling a solution with actors, envision a group of people and assign sub-tasks to them, arrange their functions into an organizational structure and think about how to escalate failure (all with the benefit of not actually dealing with people, which means that we need not concern ourselves with their emotional state or moral issues). The result can then serve as a mental scaffolding for building the software implementation.

[!NOTE] An `ActorSystem` is a heavyweight structure that will allocate 1...N Threads, so create one per logical application.

Hierarchical Structure

Like in an economic organization, actors naturally form hierarchies. One actor, which is to oversee a certain function in the program might want to split up its task into smaller, more manageable pieces. For this purpose it starts child actors which it supervises. While the details of supervision are explained here, we shall concentrate on the underlying concepts in this section. The only prerequisite is to know that each actor has exactly one supervisor, which is the actor that created it.

The quintessential feature of actor systems is that tasks are split up and delegated until they become small enough to be handled in one piece. In doing so, not only is the task itself clearly structured, but the resulting actors can be reasoned about in terms of which messages they should process, how they should react normally and how failure should be handled. If one actor does not have the means for dealing with a certain situation, it sends a corresponding failure message to its supervisor, asking for help. The recursive structure then allows to handle failure at the right level.



Compare this to layered software design which easily devolves into defensive programming with the aim of not leaking any failure out: if the problem is communicated to the right person, a better solution can be found than if trying to keep everything "under the carpet".

Compare this to layered software design which easily devolves into defensive programming with the aim of not leaking any failure out: if the problem is communicated to the right person, a better solution can be found than if trying to keep everything "under the carpet".

Now, the difficulty in designing such a system is how to decide who should supervise what. There is of course no single best solution, but there are a few guidelines which might be helpful:

- If one actor manages the work another actor is doing, e.g. by passing on sub-tasks, then the manager should supervise the child. The reason is that the manager knows which kind of failures are expected and how to handle them.
- If one actor carries very important data (i.e. its state shall not be lost if avoidable), this actor should source out any possibly dangerous sub-tasks to children it supervises and handle failures of these children as appropriate. Depending on the nature of the requests, it may be best to create a new child for each request, which simplifies state management for collecting the replies. This is known as the "Error Kernel Pattern" from Erlang.
- If one actor depends on another actor for carrying out its duty, it should watch that other actor's liveness and act upon receiving a termination notice. This is different from supervision, as the watching party has no influence on the supervisor strategy, and it should be noted that a functional dependency alone is not a criterion for deciding where to place a certain child actor in the hierarchy.

There are of course always exceptions to these rules, but no matter whether you follow the rules or break them, you should always have a reason.

Configuration Container

The actor system as a collaborating ensemble of actors is the natural unit for managing shared facilities like scheduling services, configuration, logging, etc. Several actor systems with different configuration may co-exist within the same runtime without problems, there is no global shared state within Akka.NET itself. Couple this with the transparent communication between actor systems—within one node or across a network connection—to see that actor systems themselves can be used as building blocks in a functional hierarchy.

Actor Best Practices

1. Actors should be like nice co-workers: do their job efficiently without bothering everyone else needlessly and avoid hogging resources. Translated to programming this means to process events and generate responses (or more requests) in an event-driven manner. Actors should not block (i.e. passively wait while occupying a Thread) on some external entity—which might be a lock, a network socket, etc.—unless it is unavoidable; in the latter case see below.
2. Do not pass mutable objects between actors. In order to ensure that, prefer immutable messages. If the encapsulation of actors is broken by exposing their mutable state to the outside, you are back in normal .NET concurrency land with all the drawbacks.
3. Actors are made to be containers for behavior and state, embracing this means to not routinely send behavior within messages. One of the risks is to accidentally share mutable state between actors, and this violation of the actor model unfortunately breaks all the properties which make programming in actors such a nice experience.
4. Top-level actors are the innermost part of your Error Kernel, so create them sparingly and prefer truly hierarchical systems. This has benefits with respect to fault-handling (both considering the granularity of configuration and the performance) and it also reduces the strain on the guardian actor, which is a single point of contention if over-used.

Blocking Needs Careful Management

In some cases it is unavoidable to do blocking operations, i.e. to put a thread to sleep for an indeterminate time, waiting for an external event to occur. Examples are legacy RDBMS drivers or messaging APIs, and the underlying reason is typically that (network) I/O occurs under the covers. When facing this, you may be tempted to just wrap the blocking call inside a Future and work with that instead, but this strategy is too simple: you are quite likely to find bottlenecks or run out of memory or threads when the application runs under increased load.

The non-exhaustive list of adequate solutions to the "blocking problem" includes the following suggestions:

- Do the blocking call within an actor (or a set of actors managed by a [router](#)), making sure to configure a thread pool which is either dedicated for this purpose or sufficiently sized.
- Do the blocking call within a `Task`, ensuring an upper bound on the number of such calls at any point in time (submitting an unbounded number of tasks of this nature will exhaust your memory or thread limits).
- Do the blocking call within a `Task`, providing a thread pool with an upper limit on the number of threads which is appropriate for the hardware on which the application runs.
- Dedicate a single thread to manage a set of blocking resources and dispatch events as they occur as actor messages.

The first possibility is especially well-suited for resources which are single-threaded in nature, like database handles which traditionally can only execute one outstanding query at a time and use internal synchronization to ensure this. A common pattern is to create a router for N actors, each of which wraps a single DB connection and handles queries as sent to the router. The number N must then be tuned for maximum throughput, which will vary depending on which DBMS is deployed on what hardware.

[!NOTE] Configuring thread pools is a task best delegated to Akka.NET, simply configure in the application.conf and instantiate through an ActorSystem.

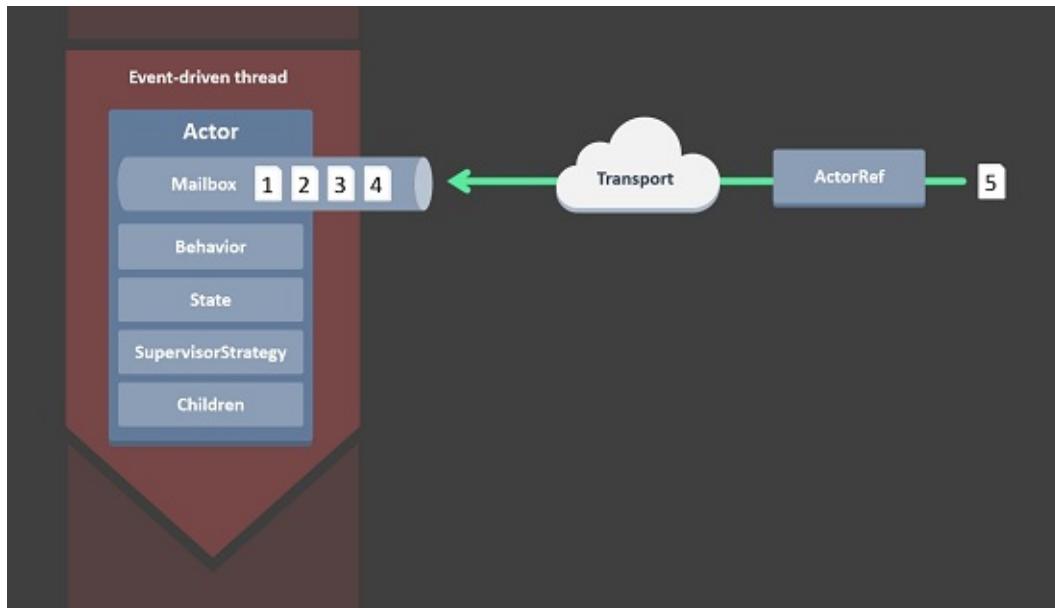
What you should not concern yourself with

An actor system manages the resources it is configured to use in order to run the actors which it contains. There may be millions of actors within one such system, after all the mantra is to view them as abundant and they weigh in at an overhead of only roughly 300 bytes per instance. Naturally, the exact order in which messages are processed in large systems is not controllable by the application author, but this is also not intended. Take a step back and relax while Akka.NET does the heavy lifting under the hood.

Actors

The previous section about [Actor Systems](#) explained how actors form hierarchies and are the smallest unit when building an application. This section looks at one such actor in isolation, explaining the concepts you encounter while implementing it. For a more in depth reference with all the details please refer to [F# API](#) or [C# API](#).

An actor is a container for [State](#), [Behavior](#), a [Mailbox](#), [Children](#) and a [Supervisor Strategy](#). All of this is encapsulated behind an [Actor Reference](#). One noteworthy aspect is that actors have an explicit lifecycle, they are not automatically destroyed when no longer referenced; after having created one, it is your responsibility to make sure that it will eventually be terminated as well—which also gives you control over how resources are released [When an Actor Terminates](#).



Actor Reference

As detailed below, 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. This split into inner and outer object enables transparency for all the desired operations: restarting an actor without needing to update references elsewhere, placing the actual actor object on remote hosts, sending messages to actors in completely different applications. But 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.

State

Actor objects will typically contain some variables which reflect possible states the actor may be in. This can be an explicit state machine (e.g. using the [FSM module](#)), or it could be a counter, set of listeners, pending requests, etc. These data are what make an actor valuable, and they must be protected from corruption by other actors. The good news is that Akka.NET actors conceptually each have their own light-weight thread, which is completely shielded from the rest of the system. This means that instead of having to synchronize access using locks you can just write your actor code without worrying about concurrency at all.

Behind the scenes Akka.NET will run sets of actors on sets of real threads, where typically many actors share one thread, and subsequent invocations of one actor may end up being processed on different threads. Akka.NET ensures that this implementation detail does not affect the single-threadedness of handling the actor's state.

Because the internal state is vital to an actor's operations, having inconsistent state is fatal. Thus, when the actor fails and is restarted by its supervisor, the state will be created from scratch, like upon first creating the actor. This is to enable the ability of self-healing of the system.

Optionally, an actor's state can be automatically recovered to the state before a restart by persisting received messages and replaying them after restart (see [Persistence](#)).

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. This behavior may change over time, e.g. because different clients obtain authorization over time, or because the actor may go into an "out-of-service" mode and later come back. These changes are achieved by either encoding them in state variables which are read from the behavior logic, or the function itself may be swapped out at runtime, see the become and unbecome operations. However, the initial behavior defined during construction of the actor object is special in the sense that a restart of the actor will reset its behavior to this initial one.

Mailbox

An actor's purpose is the processing of messages, and these messages were sent to the actor from other actors (or from outside the actor system). 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. Enqueuing happens in the time-order of send operations, which means that messages sent from different actors may not have a defined order at runtime due to the apparent randomness of distributing actors across threads. Sending multiple messages to the same target from the same actor, on the other hand, will enqueue them in the same order.

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, which might even be at the front. While using such a queue, the order of messages processed will naturally be defined by the queue's algorithm and in general not be FIFO.

An important feature in which Akka.NET differs from some other actor model implementations is that the current behavior must always handle the next dequeued message, there is no scanning the mailbox for the next matching one. Failure to handle a message will typically be treated as a failure, unless this behavior is overridden.

Child Actors

Each actor is potentially a supervisor: if it creates children for delegating sub-tasks, it will automatically supervise them. The list of children is maintained within the actor's context and the actor has access to it. Modifications to the list are done by creating (`Context.ActorOf(...)`) or stopping (`Context.Stop(child)`) children and these actions are reflected immediately. The actual creation and termination actions happen behind the scenes in an asynchronous way, so they do not "block" their supervisor.

Supervisor Strategy

The final piece of an actor is its strategy for handling faults of its children. Fault handling is then done transparently by Akka, applying one of the strategies described in [Supervision and Monitoring](#) for each incoming failure. As this strategy is fundamental to how an actor system is structured, it cannot be changed once an actor has been created.

Considering that there is only one such strategy for each actor, this means that if different strategies apply to the various children of an actor, the children should be grouped beneath intermediate supervisors with matching strategies, preferring once more the structuring of actor systems according to the splitting of tasks into sub-tasks.

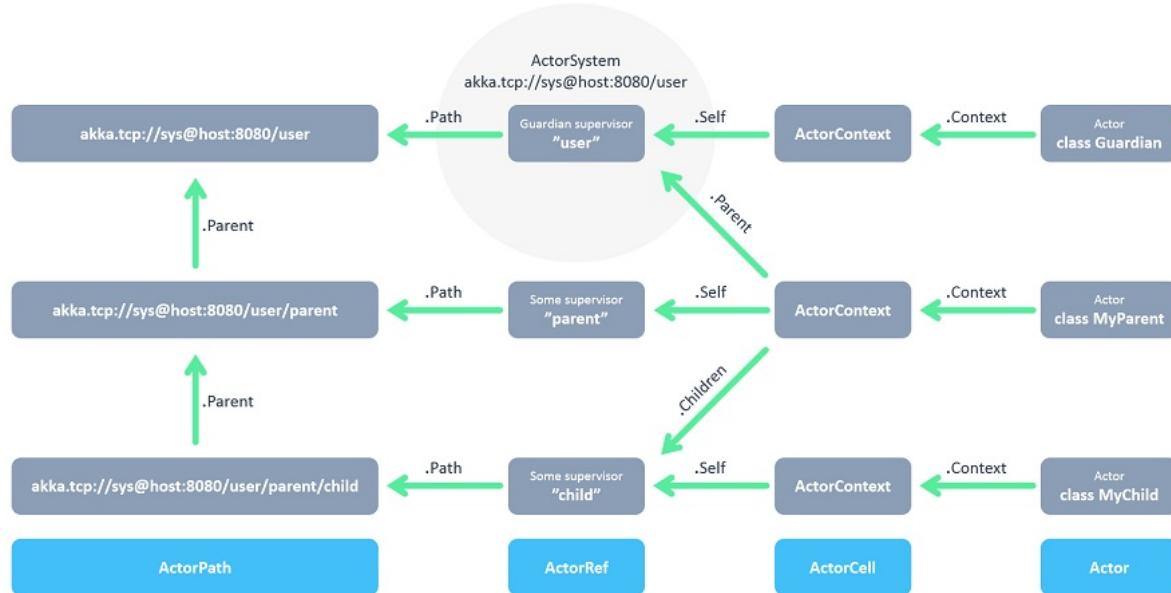
When an Actor Terminates

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".

The reason for not just silently dumping the messages was inspired by our tests: we register the `TestEventListener` on the event bus to which the dead letters are forwarded, and that will log a warning for every dead letter received—this has been very helpful for deciphering test failures more quickly. It is conceivable that this feature may also be of use for other purposes.

Actor References, Paths and Addresses

This chapter describes how actors are identified and located within a possibly distributed actor system. It ties into the central idea that [Actor Systems](#) form intrinsic supervision hierarchies as well as that communication between actors is transparent with respect to their placement across multiple network nodes.



The above image displays the relationship between the most important entities within an actor system, please read on for the details.

What is an Actor Reference?

An actor reference is a subtype of `ActorRef`, whose foremost purpose is to support sending messages to the actor it represents. Each actor has access to its canonical (local) reference through the `Self` property; this reference is also included as sender reference by default for all messages sent to other actors. Conversely, during message processing the actor has access to a reference representing the sender of the current message through the `sender` method.

There are several different types of actor references that are supported depending on the configuration of the actor system:

- Purely local actor references are used by actor systems which are not configured to support networking functions. These actor references will not function if sent across a network connection to a remote CLR.
- Local actor references when remoting is enabled are used by actor systems which support networking functions for those references which represent actors within the same CLR. In order to also be reachable when sent to other network nodes, these references include protocol and remote addressing information.
- There is a subtype of local actor references which is used for routers. Its logical structure is the same as for the aforementioned local references, but sending a message to them dispatches to one of their children directly instead.
- Remote actor references represent actors which are reachable using remote communication, i.e. sending messages to them will serialize the messages transparently and send them to the remote CLR.
- There are several special types of actor references which behave like local actor references for all practical

purposes:

- `PromiseActorRef` is the special representation of a `Task` for the purpose of being completed by the response from an actor. `ICanTell.Ask` creates this actor reference.
- `DeadLetterActorRef` is the default implementation of the dead letters service to which Akka routes all messages whose destinations are shut down or non-existent.
- `EmptyLocalActorRef` is what Akka returns when looking up a non-existent local actor path: it is equivalent to a `DeadLetterActorRef`, but it retains its path so that Akka can send it over the network and compare it to other existing actor references for that path, some of which might have been obtained before the actor died.
- And then there are some one-off internal implementations which you should never really see:
 - There is an actor reference which does not represent an actor but acts only as a pseudo-supervisor for the root guardian, we call it "the one who walks the bubbles of space-time".
 - The first logging service started before actually firing up actor creation facilities is a fake actor reference which accepts log events and prints them directly to standard output; it is `Logging.StandardOutLogger`.

What is an Actor Path?

Since actors are created in a strictly hierarchical fashion, there exists a unique sequence of actor names given by recursively following the supervision links between child and parent down towards the root of the actor system. This sequence can be seen as enclosing folders in a file system, hence we adopted the name "path" to refer to it, although actor hierarchy has some fundamental difference from file system hierarchy.

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 Between 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.

Actor Path Anchors

Each actor path has an address component, describing the protocol and location by which the corresponding actor is reachable, followed by the names of the actors in the hierarchy from the root up. Examples are:

```
"akka://my-sys/user/service-a/worker1"          // purely local
"akka.tcp://my-sys@host.example.com:5678/user/service-b" // remote
```

Here, `akka.tcp` is the default remote transport; other transports are pluggable. A remote host using UDP would be accessible by using `akka.udp`. The interpretation of the host and port part (i.e. `serv.example.com:5678` in the example) depends on the transport mechanism used, but it must abide by the URI structural rules.

Logical Actor Paths

The unique path obtained by following the parental supervision links towards the root guardian is called the logical actor path. This path matches exactly the creation ancestry of an actor, so it is completely deterministic as soon as the actor system's remoting configuration (and with it the address component of the path) is set.

Physical Actor Paths

While the logical actor path describes the functional location within one actor system, configuration-based remote deployment means that an actor may be created on a different network host than its parent, i.e. within a different actor system. In this case, following the actor path from the root guardian up entails traversing the network, which is a costly operation. Therefore, each actor also has a physical path, starting at the root guardian of the actor system where the actual actor object resides. Using this path as sender reference when querying other actors will let them reply directly to this actor, minimizing delays incurred by routing.

One important aspect is that a physical actor path never spans multiple actor systems or CLRs. This means that the logical path (supervision hierarchy) and the physical path (actor deployment) of an actor may diverge if one of its ancestors is remotely supervised.

Actor path alias or symbolic link?

As in some real file-systems you might think of a "path alias" or "symbolic link" for an actor, i.e. one actor may be reachable using more than one path. However, you should note that actor hierarchy is different from file system hierarchy. You cannot freely create actor paths like symbolic links to refer to arbitrary actors. As described in the above logical and physical actor path sections, an actor path must be either logical path which represents supervision hierarchy, or physical path which represents actor deployment.

How are Actor References obtained?

There are two general categories to how actor references may be obtained: by creating actors or by looking them up, where the latter functionality comes in the two flavours of creating actor references from concrete actor paths and querying the logical actor hierarchy.

Creating Actors

An actor system is typically started by creating actors beneath the guardian actor using the `ActorSystem.ActorOf` method and then using `ActorContext.ActorOf` from within the created actors to spawn the actor tree. These methods return a reference to the newly created actor. Each actor has direct access (through its `ActorContext`) to references for its parent, itself and its children. These references may be sent within messages to other actors, enabling those to reply directly.

Looking up Actors by Concrete Path

In addition, actor references may be looked up using the `ActorSystem.ActorSelection` method. The selection can be used for communicating with said actor and the actor corresponding to the selection is looked up when delivering each message.

To acquire an `ActorRef` that is bound to the life-cycle of a specific actor you need to send a message, such as the built-in `Identify` message, to the actor and use the `Sender` reference of a reply from the actor.

Absolute vs. Relative Paths

In addition to `ActorSystem.actorSelection` there is also `ActorContext.ActorSelection`, which is available inside any actor as `context.ActorSelection`. This yields an actor selection much like its twin on `ActorSystem`, but instead of looking up the path starting from the root of the actor tree it starts out on the current actor. `Path` elements consisting of two dots ("..") may be used to access the parent actor. You can for example send a message to a specific sibling:

```
Context.ActorSelection("../brother").Tell(msg);
```

Absolute paths may of course also be looked up on context in the usual way, i.e.

```
Context.ActorSelection("/user/serviceA").Tell(msg);
```

will work as expected.

Querying the Logical Actor Hierarchy

Since the actor system forms a file-system like hierarchy, matching on paths is possible in the same way as supported by Unix shells: you may replace (parts of) path element names with wildcards (`«*»` and `«?»`) to formulate a selection which may match zero or more actual actors. Because the result is not a single actor reference, it has a different type `ActorSelection` and does not support the full set of operations an `IActorRef` does. Selections may be formulated using the `ActorSystem.ActorSelection` and `IActorContext.ActorSelection` methods and do support sending messages:

```
Context.ActorSelection("../*").Tell(msg);
```

will send msg to all siblings including the current actor.

Summary: `ActorOf` vs. `ActorSelection`

[!NOTE] What the above sections described in some detail can be summarized and memorized easily as follows:

- `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 Reference and Path Equality

Equality of `ActorRef` match the intention that an `ActorRef` corresponds to the target actor incarnation. Two actor references are compared equal when they have the same path and point to the same actor incarnation. A reference pointing to a terminated actor does not compare equal to a reference pointing to another (re-created) actor with the same path. Note that a restart of an actor caused by a failure still means that it is the same actor incarnation, i.e. a restart is not visible for the consumer of the `ActorRef`.

If you need to keep track of actor references in a collection and do not care about the exact actor incarnation you can use the `ActorPath` as key, because the identifier of the target actor is not taken into account when comparing actor paths.

Reusing Actor Paths

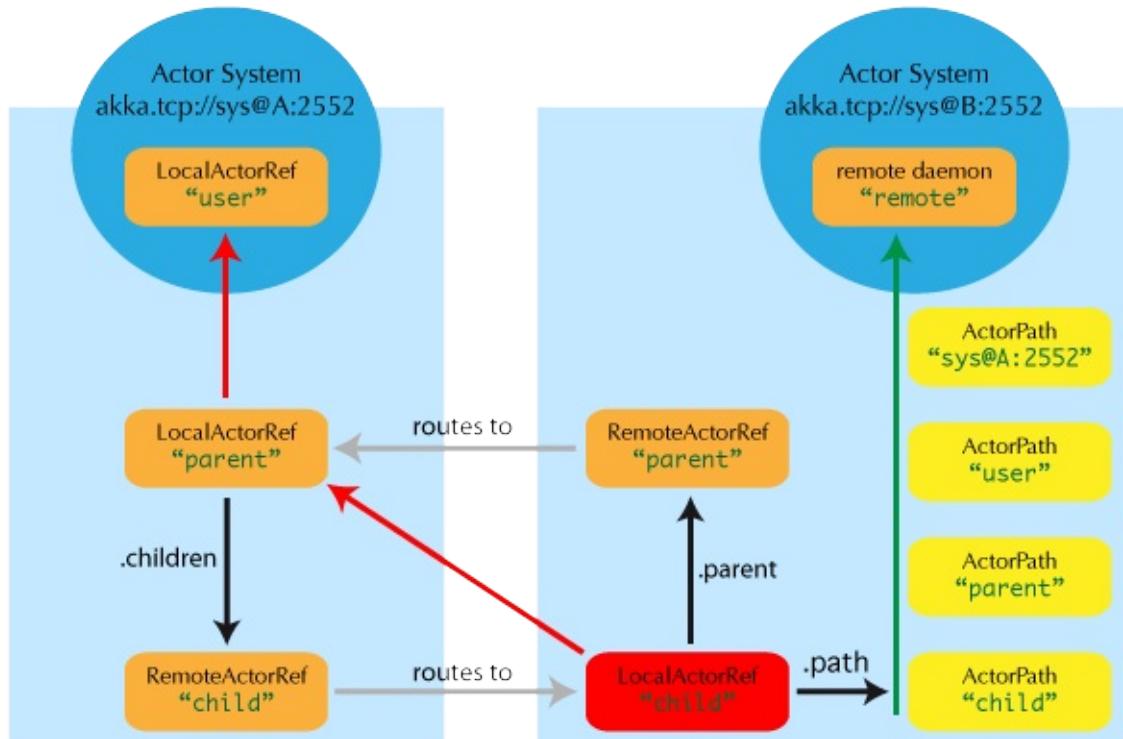
When an actor is terminated, its reference will point to the dead letter mailbox, `Deathwatch` will publish its final transition and in general it is not expected to come back to life again (since the actor life cycle does not allow this). While it is possible to create an actor at a later time with an identical path—simply due to it being impossible to enforce the opposite without keeping the set of all actors ever created available—this is not good practice: messages sent with `ActorSelection` to an actor which "died" suddenly start to work again, but without any guarantee of ordering between this transition and any other event, hence the new inhabitant of the path may receive messages which were destined for the previous tenant.

It may be the right thing to do in very specific circumstances, but make sure to confine the handling of this precisely to the actor's supervisor, because that is the only actor which can reliably detect proper deregistration of the name, before which creation of the new child will fail.

It may also be required during testing, when the test subject depends on being instantiated at a specific path. In that case it is best to mock its supervisor so that it will forward the `Terminated` message to the appropriate point in the test procedure, enabling the latter to await proper deregistration of the name.

The Interplay with Remote Deployment

When an actor creates a child, the actor system's deployer will decide whether the new actor resides in the same CLR or on another node. In the second case, creation of the actor will be triggered via a network connection to happen in a different CLR and consequently within a different actor system. The remote system will place the new actor below a special path reserved for this purpose and the supervisor of the new actor will be a remote actor reference (representing that actor which triggered its creation). In this case, `context.parent` (the supervisor reference) and `context.path.parent` (the parent node in the actor's path) do not represent the same actor. However, looking up the child's name within the supervisor will find it on the remote node, preserving logical structure e.g. when sending to an unresolved actor reference.



logical actor path: `akka.tcp://sys@A:2552/user/parent/child`

physical actor path: `akka.tcp://sys@B:2552/remote/sys@A:2552/user/parent/child`

What is the Address part used for?

When sending an actor reference across the network, it is represented by its path. Hence, the path must fully encode all information necessary to send messages to the underlying actor. This is achieved by encoding protocol, host and port in the address part of the path string. When an actor system receives an actor path from a remote node, it checks whether that path's address matches the address of this actor system, in which case it will be resolved to the actor's local reference. Otherwise, it will be represented by a remote actor reference.

Top-Level Scopes for Actor Paths

At the root of the path hierarchy resides the root guardian above which all other actors are found; its name is `"/"`. The next level consists of the following:

- `"/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 CLR).
- `"/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. The need to structure the name space for actors like this arises from a central and very simple design goal: everything in the hierarchy is an actor, and all actors function in the same way. Hence you can not only look up the actors you created, you can also look up the system guardian and send it a message (which it will dutifully discard in this case). This powerful principle means that there are no quirks to remember, it makes the whole system more uniform and consistent.

If you want to read more about the top-level structure of an actor system, have a look at [The Top-Level Supervisors](#).

Location Transparency

The previous section describes how actor paths are used to enable location transparency. This special feature deserves some extra explanation, because the related term "transparent remoting" was used quite differently in the context of programming languages, platforms and technologies.

Distributed by Default

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

Ways in which Transparency is Broken

What is true of Akka.NET need not be true of the application which uses it, since designing for distributed execution poses some restrictions on what is possible. The most obvious one is that all messages sent over the wire must be serializable. While being a little less obvious this includes closures which are used as actor factories (i.e. within `Props`) if the actor is to be created on a remote node.

Another consequence is that everything needs to be aware of all interactions being fully asynchronous, which in a computer network might mean that it may take several minutes for a message to reach its recipient (depending on configuration). It also means that the probability for a message to be lost is much higher than within one CLR, where it is close to zero (still: no hard guarantee!).

How is Remoting Used?

We took the idea of transparency to the limit in that there is nearly no API for the remoting layer of Akka: it is purely driven by configuration. Just write your application according to the principles outlined in the previous sections, then specify remote deployment of actor sub-trees in the configuration file. This way, your application can be scaled out without having to touch the code. The only piece of the API which allows programmatic influence on remote deployment is that `Props` contain a field which may be set to a specific `Deploy` instance; this has the same effect as putting an equivalent deployment into the configuration file (if both are given, configuration file wins).

Peer-to-Peer vs. Client-Server

Akka.NET Remoting is a communication module for connecting actor systems in a peer-to-peer fashion, and it is the foundation for Akka Clustering. The design of remoting is driven by two (related) design decisions:

Communication between involved systems is symmetric: if a system A can connect to a system B then system B must also be able to connect to system A independently. The role of the communicating systems are symmetric in regards to connection patterns: there is no system that only accepts connections, and there is no system that only initiates connections. The consequence of these decisions is that it is not possible to safely create pure client-server setups with predefined roles (violates assumption 2). For client-server setups it is better to use Akka I/O.

[!IMPORTANT] Using setups involving Network Address Translation, Load Balancers or Docker containers violates assumption 1, unless additional steps are taken in the network configuration to allow symmetric communication between involved systems. In such situations Akka can be configured to bind to a different

[!IMPORTANT] Using setups involving Network Address Translation, Load Balancers or Docker containers violates assumption 1, unless additional steps are taken in the network configuration to allow symmetric communication between involved systems. In such situations Akka can be configured to bind to a different network address than the one used for establishing connections between Akka.NET nodes.

Marking Points for Scaling Up with Routers

In addition to being able to run different parts of an actor system on different nodes of a cluster, it is also possible to scale up onto more cores by multiplying actor sub-trees which support parallelization (think for example a search engine processing different queries in parallel). The clones can then be routed to in different fashions, e.g. round-robin. The only thing necessary to achieve this is that the developer needs to declare a certain actor as `WithRouter`, then—in its stead—a router actor will be created which will spawn up a configurable number of children of the desired type and route to them in the configured fashion. Once such a router has been declared, its configuration can be freely overridden from the configuration file, including mixing it with the remote deployment of (some of) the children. Read more about this in [Routing](#).

Message Delivery Reliability

Akka.NET helps you build reliable applications which make use of multiple processor cores in one machine ("scaling up") or distributed across a computer network ("scaling out"). The key abstraction to make this work is that all interactions between your code units—actors—happen via message passing, which is why the precise semantics of how messages are passed between actors deserve their own chapter.

In order to give some context to the discussion below, consider an application which spans multiple network hosts. The basic mechanism for communication is the same whether sending to an actor on the local application or to a remote actor, but of course there will be observable differences in the latency of delivery (possibly also depending on the bandwidth of the network link and the message size) and the reliability. In case of a remote message send there are obviously more steps involved which means that more can go wrong. Another aspect is that local sending will just pass a reference to the message inside the same application, without any restrictions on the underlying object which is sent, whereas a remote transport will place a limit on the message size.

Writing your actors such that every interaction could possibly be remote is the safe, pessimistic bet. It means to only rely on those properties which are always guaranteed and which are discussed in detail below. This has of course some overhead in the actor's implementation. If you are willing to sacrifice full location transparency—for example in case of a group of closely collaborating actors—you can place them always on the same local application and enjoy stricter guarantees on message delivery. The details of this trade-off are discussed further below.

As a supplementary part we give a few pointers at how to build stronger reliability on top of the built-in ones. The chapter closes by discussing the role of the "Dead Letter Office".

The General Rules

These are the rules for message sends (i.e. the `Tell` method, which also underlies the `Ask` pattern):

- **at-most-once delivery**, i.e. no guaranteed delivery
- **message ordering per sender–receiver pair**

The first rule is typically found also in other actor implementations while the second is specific to Akka.NET.

Discussion: What does "at-most-once" mean?

When it comes to describing the semantics of a delivery mechanism, there are three basic categories:

- **at-most-once** delivery means that for each message handed to the mechanism, that message is delivered zero or one times; 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. The second one requires retries to counter transport losses, which means keeping state at the sending end and having an acknowledgement mechanism at the receiving end. The third is most expensive—and has consequently worst performance—because in addition to the second it requires state to be kept at the receiving end in order to filter out duplicate deliveries.

Discussion: Why No Guaranteed Delivery?

At the core of the problem lies the question what exactly this guarantee shall mean:

1. The message is sent out on the network?
2. The message is received by the other host?
3. The message is put into the target actor's mailbox?
4. The message is starting to be processed by the target actor?
5. The message is processed successfully by the target actor?

Each one of these have different challenges and costs, and it is obvious that there are conditions under which any message passing library would be unable to comply; think for example about configurable mailbox types and how a bounded mailbox would interact with the third point, or even what it would mean to decide upon the "successfully" part of point five.

Along those same lines goes the reasoning in [Nobody Needs Reliable Messaging](#). The only meaningful way for a sender to know whether an interaction was successful is by receiving a business-level acknowledgement message, which is not something Akka.NET could make up on its own (neither are we writing a "do what I mean" framework nor would you want us to).

Akka.NET embraces distributed computing and makes the fallibility of communication explicit through message passing, therefore it does not try to lie and emulate a leaky abstraction. This is a model that has been used with great success in Erlang and requires the users to design their applications around it. You can read more about this approach in the [Erlang documentation](#) (section 10.9 and 10.10), Akka.NET follows it closely.

Another angle on this issue is that by providing only basic guarantees those use cases which do not need stronger reliability do not pay the cost of their implementation; it is always possible to add stronger reliability on top of basic ones, but it is not possible to retro-actively remove reliability in order to gain more performance.

Discussion: Message Ordering

The rule more specifically is that *for a given pair of actors, messages sent from the first to the second will not be received out-of-order. The word directly emphasizes that this guarantee only applies when sending with the tell operator to the final destination, not when employing mediators or other message dissemination features (unless stated otherwise).

The guarantee is illustrated in the following:

Actor `A1` sends messages `M1`, `M2`, `M3` to `A2`.

Actor `A3` sends messages `M4`, `M5`, `M6` to `A2`.

This means that:

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

[!NOTE] It is important to note that Akka's guarantee applies to the order in which messages are enqueued into the recipient's mailbox. If the mailbox implementation does not respect FIFO order (e.g. a `PriorityMailbox`), then the order of processing by the actor can deviate from the enqueueing order.

Please note that this rule is **not transitive**:

Actor `A` sends message `M1` to actor `C`
Actor `A` then sends message `M2` to actor `B`
Actor `B` forwards message `M2` to actor `C`
Actor `C` may receive `M1` and `M2` in any order

Causal transitive ordering would imply that `M2` is never received before `M1` at actor `C` (though any of them might be lost). This ordering can be violated due to different message delivery latencies when `A`, `B` and `C` reside on different network hosts, see more below.

[!NOTE] Actor creation is treated as a message sent from the parent to the child, with the same semantics as discussed above. Sending a message to an actor in a way which could be reordered with this initial creation message means that the message might not arrive because the actor does not exist yet. An example where the message might arrive too early would be to create a remote-deployed actor `R1`, send its reference to another remote actor `R2` and have `R2` send a message to `R1`. An example of well-defined ordering is a parent which creates an actor and immediately sends a message to it.

Communication of failure

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 has their own mailboxes therefore the ordering of enqueue calls of a user and system message cannot guarantee the ordering of their dequeue times.

The Rules for In-App (Local) Message Sends

Be careful what you do with this section!

Relying on the stronger reliability in this section is not recommended since it will bind your application to local-only deployment: an application may have to be designed differently (as opposed to just employing some message exchange patterns local to some actors) in order to be fit for running on a cluster of machines. Our credo is "design once, deploy any way you wish", and to achieve this you should only rely on The [General Rules](#).

Reliability of Local Message Sends

The Akka.NET test suite relies on not losing messages in the local context (and for non-error condition tests also for remote deployment), meaning that we actually do apply the best effort to keep our tests stable. A local `Tell` operation can however fail for the same reasons as a normal method call can on the CLR:

- `StackOverflowException`
- `OutOfMemoryException`
- `other : SystemException`

In addition, local sends can fail in Akka-specific ways:

- if the mailbox does not accept the message (e.g. full `BoundedMailbox`)

- if the receiving actor fails while processing the message or is already terminated

While the first is clearly a matter of configuration the second deserves some thought: the sender of a message does not get feedback if there was an exception while processing, that notification goes to the supervisor instead. This is in general not distinguishable from a lost message for an outside observer.

Ordering of Local Message Sends

Assuming strict FIFO mailboxes the abovementioned caveat of non-transitivity of the message ordering guarantee is eliminated under certain conditions. As you will note, these are quite subtle as it stands, and it is even possible that future performance optimizations will invalidate this whole paragraph. The possibly non-exhaustive list of counter-indications is:

- Before receiving the first reply from a top-level actor, there is a lock which protects an internal interim queue, and this lock is not fair; the implication is that enqueue requests from different senders which arrive during the actor's construction (figuratively, the details are more involved) may be reordered depending on low-level thread scheduling. Since completely fair locks do not exist on the CLR this is unfixable.
- The same mechanism is used during the construction of a Router, more precisely the routed ActorRef, hence the same problem exists for actors deployed with Routers.
- As mentioned above, the problem occurs anywhere a lock is involved during enqueueing, which may also apply to custom mailboxes.

This list has been compiled carefully, but other problematic scenarios may have escaped our analysis.

How does Local Ordering relate to Network Ordering

The rule that for a given pair of actors, messages sent directly from the first to the second will not be received out-of-order holds for messages sent over the network with the TCP based Akka.NET remote transport protocol.

As explained in the previous section local message sends obey transitive causal ordering under certain conditions. This ordering can be violated due to different message delivery latencies. For example:

Actor A on node-1 sends message M1 to actor C on node-3

Actor A on node-1 then sends message M2 to actor B on node-2

Actor B on node-2 forwards message M2 to actor C on node-3

Actor C may receive M1 and M2 in any order

It might take longer time for M1 to "travel" to node-3 than it takes for M2 to "travel" to node-3 via node-2.

Higher-level abstractions

Based on a small and consistent tool set in Akka's core, Akka.NET also provides powerful, higher-level abstractions on top of it.

Messaging Patterns

As discussed above a straight-forward answer to the requirement of reliable delivery is an explicit ACK-RETRY protocol. In its simplest form this requires

- a way to identify individual messages to correlate message with acknowledgement
- a retry mechanism which will resend messages if not acknowledged in time

- a way for the receiver to detect and discard duplicates

The third becomes necessary by virtue of the acknowledgements not being guaranteed to arrive either. An ACK-RETRY protocol with business-level acknowledgements is supported by [At least once delivery](#) of the Akka.NET Persistence module. Duplicates can be detected by tracking the identifiers of messages sent via [At least once delivery](#). Another way of implementing the third part would be to make processing the messages idempotent on the level of the business logic.

Event Sourcing

Event sourcing (and sharding) is what makes large websites scale to billions of users, and the idea is quite simple: when a component (think actor) processes a command it will generate a list of events representing the effect of the command. These events are stored in addition to being applied to the component's state. The nice thing about this scheme is that events only ever are appended to the storage, nothing is ever mutated; this enables perfect replication and scaling of consumers of this event stream (i.e. other components may consume the event stream as a means to replicate the component's state on a different continent or to react to changes). If the component's state is lost—due to a machine failure or by being pushed out of a cache—it can easily be reconstructed by replaying the event stream (usually employing snapshots to speed up the process). [Event-sourcing](#) is supported by Akka.NET Persistence.

Mailbox with Explicit Acknowledgement

By implementing a custom mailbox type it is possible to retry message processing at the receiving actor's end in order to handle temporary failures. This pattern is mostly useful in the local communication context where delivery guarantees are otherwise sufficient to fulfill the application's requirements.

Dead Letters

Messages which cannot be delivered (and for which this can be ascertained) will be delivered to a synthetic actor called `/deadLetters`. This delivery happens on a best-effort basis; it may fail even within a single application in the local machine (e.g. during actor termination). Messages sent via unreliable network transports will be lost without turning up as dead letters.

What Should I Use Dead Letters For?

The main use of this facility is for debugging, especially if an actor send does not arrive consistently (where usually inspecting the dead letters will tell you that the sender or recipient was set wrong somewhere along the way). In order to be useful for this purpose it is good practice to avoid sending to `deadLetters` where possible, i.e. run your application with a suitable dead letter logger (see more below) from time to time and clean up the log output. This exercise—like all else—requires judicious application of common sense: it may well be that avoiding to send to a terminated actor complicates the sender's code more than is gained in debug output clarity.

The dead letter service follows the same rules with respect to delivery guarantees as all other message sends, hence it cannot be used to implement guaranteed delivery.

How do I Receive Dead Letters?

An actor can subscribe to class `Akka.Actor.DeadLetter` on the event stream, see [Event stream](#) for how to do that. The subscribed actor will then receive all dead letters published in the (local) system from that point onwards. Dead letters are not propagated over the network, if you want to collect them in one place you will have to subscribe one actor per network node and forward them manually. Also consider that dead letters are generated at that node which can determine that a send operation is failed, which for a remote send can be the local system (if no network connection can be established) or the remote one (if the actor you are sending to does not exist at that point in time).

Dead Letters Which are (Usually) not Worrisome

Every time an actor does not terminate by its own decision, there is a chance that some messages which it sends to itself are lost. There is one which happens quite easily in complex shutdown scenarios that is usually benign: seeing a `Akka.Dispatch.Terminate` message dropped means that two termination requests were given, but of course only one can succeed. In the same vein, you might see `Akka.Actor.Terminated` messages from children while stopping a hierarchy of actors turning up in dead letters if the parent is still watching the child when the parent terminates.

Akka.NET Configuration

Quoted from Akka.NET Bootcamp: Unit 2, Lesson 1 - "Using HOCON Configuration to Configure Akka.NET"

Akka.NET leverages a configuration format, called HOCON, to allow you to configure your Akka.NET applications with whatever level of granularity you want.

What is HOCON?

HOCON (Human-Optimized Config Object Notation) is a flexible and extensible configuration format. It will allow you to configure everything from Akka.NET's `IActorRefProvider` implementation, logging, network transports, and more commonly - how individual actors are deployed.

Values returned by HOCON are strongly typed (i.e. you can fetch out an `int`, a `Timespan`, etc).

What can I do with HOCON?

HOCON allows you to embed easily-readable configuration inside of the otherwise hard-to-read XML in App.config and Web.config. HOCON also lets you query configs by their section paths, and those sections are exposed strongly typed and parsed values you can use inside your applications.

HOCON also lets you nest and/or chain sections of configuration, creating layers of granularity and providing you a semantically namespaced config.

What is HOCON usually used for?

HOCON is commonly used for tuning logging settings, enabling special modules (such as `Akka.Remote`), or configuring deployments such as the `Dispatcher` or `Router` used for a particular actor.

For example, let's configure an `ActorSystem` with HOCON:

```
var config = ConfigurationFactory.ParseString(@"  
akka.remote.dot-netty.tcp {  
    transport-class = ""Akka.Remote.Transport.DotNetty.DotNettyTransport, Akka.Remote""  
    transport-protocol = tcp  
    port = 8091  
    hostname = ""127.0.0.1""  
}  
  
var system = ActorSystem.Create("MyActorSystem", config);
```

As you can see in that example, a HOCON `Config` object can be parsed from a `string` using the `ConfigurationFactory.ParseString` method. Once you have a `Config` object, you can then pass this to your `ActorSystem` inside the `ActorSystem.Create` method.

"Deployment"? What's that?

Deployment is a vague concept, but it's closely tied to HOCON. An actor is "deployed" when it is instantiated and put into service within the `ActorSystem` somewhere.

When an actor is instantiated within the `ActorSystem` it can be deployed in one of two places: inside the local process or in another process (this is what `Akka.Remote` does.)

When an actor is deployed by the `ActorSystem`, it has a range of configuration settings. These settings control a wide range of behavior options for the actor, such as: is this actor going to be a router? What `Dispatcher` will it use? What type of mailbox will it have? (More on these concepts in later lessons.)

We haven't gone over what all these options mean, but *the key thing to know for now is that the settings used by the `ActorSystem` to deploy an actor into service can be set within HOCON.*

This also means that you can change the behavior of actors dramatically (by changing these settings) without having to actually touch the actor code itself.

Flexible config FTW!

HOCON can be used inside `App.config` and `Web.config`

Parsing HOCON from a `string` is handy for small configuration sections, but what if you want to be able to take advantage of Configuration Transforms for `App.config` and `Web.config` and all of the other nice tools we have in the `System.Configuration` namespace?

As it turns out, you can use HOCON inside these configuration files too!

Here's an example of using HOCON inside `App.config`:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="akka" type="Akka.Configuration.Hocon.AkkaConfigurationSection, Akka" />
  </configSections>

  <akka>
    <hocon>
      <![CDATA[
        akka {
          # here we are configuring log levels
          log-config-on-start = off
          stdout-loglevel = INFO
          loglevel = ERROR
          # this config section will be referenced as akka.actor
          actor {
            provider = remote
            debug {
              receive = on
              autoreceive = on
              lifecycle = on
              event-stream = on
              unhandled = on
            }
          }
          # here we're configuring the Akka.Remote module
          remote {
            dot-netty.tcp {
              transport-class = "Akka.Remote.Transport.DotNettyTransport, Akka.Remote"
              #applied-adapters = []
              transport-protocol = tcp
              port = 8091
              hostname = "127.0.0.1"
            }
            log-remote-lifecycle-events = INFO
          }
        ]]>
      </hocon>
    </akka>
  </configuration>
```

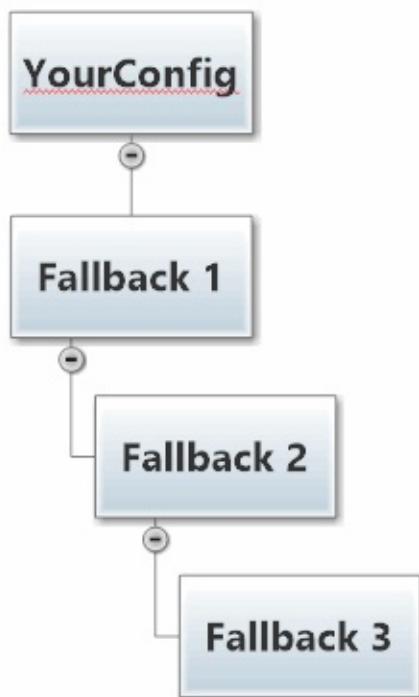
And then we can load this configuration section into our `ActorSystem` via the following code:

```
var system = ActorSystem.Create("MySystem"); //automatically loads App/Web.config
```

HOCON Configuration Supports Fallbacks

Although this isn't a concept we leverage explicitly in Unit 2, it's a powerful trait of the `Config` class that comes in handy in lots of production use cases.

HOCON supports the concept of "fallback" configurations - it's easiest to explain this concept visually.



To create something that looks like the diagram above, we have to create a `Config` object that has three fallbacks chained behind it using syntax like this:

```
var f0 = ConfigurationFactory.ParseString("a = bar");
var f1 = ConfigurationFactory.ParseString("b = biz");
var f2 = ConfigurationFactory.ParseString("c = baz");
var f3 = ConfigurationFactory.ParseString("a = foo");

var yourConfig = f0.WithFallback(f1)
    .WithFallback(f2)
    .WithFallback(f3);
```

If we request a value for a HOCON object with key "a", using the following code:

```
var a = yourConfig.GetString("a");
```

Then the internal HOCON engine will match the first HOCON file that contains a definition for key `a`. In this case, that is `f0`, which returns the value "bar".

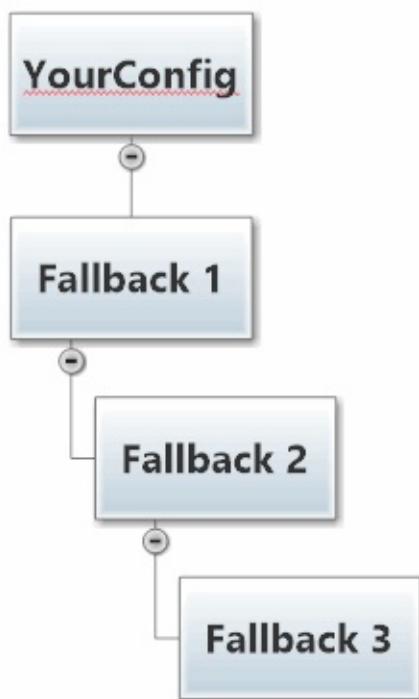
Why wasn't "foo" returned as the value for "a"?

The reason is because HOCON only searches through fallback `Config` objects if a match is NOT found earlier in the `Config` chain. If the top-level `Config` object has a match for `a`, then the fallbacks won't be searched. In this case, a match for `a` was found in `f0` so the `a=foo` in `f3` was never reached.

What happens when there is a HOCON key miss?

What happens if we run the following code, given that `c` isn't defined in `f0` or `f1`?

```
var c = yourConfig.GetString("c");
```



In this case `yourConfig` will fallback twice to `f2` and return "baz" as the value for key `c`.

ReceiveActor API

The Actor Model provides a higher level of abstraction for writing concurrent and distributed systems. It alleviates the developer from having to deal with explicit locking and thread management, making it easier to write correct concurrent and parallel systems. Actors were defined in the 1973 paper by Carl Hewitt but have been popularized by the Erlang language, and used for example at Ericsson with great success to build highly concurrent and reliable telecom systems.

Creating Actors

[!NOTE] Since Akka.NET enforces parental supervision every actor is supervised and (potentially) the supervisor of its children, it is advisable that you familiarize yourself with [Actor Systems](#) and [Supervision and Monitoring](#) and it may also help to read [Actor References, Paths and Addresses](#).

Defining an Actor class

In order to use the `Receive()` method inside an actor, the actor must inherit from `ReceiveActor`. Inside the constructor, add a call to `Receive<T>(Action<T> handler)` for every type of message you want to handle:

Here is an example:

```
public class MyActor : ReceiveActor
{
    private readonly ILoggingAdapter log = Context.GetLogger();

    public MyActor()
    {
        Receive<string>(message => {
            log.Info("Received String message: {0}", message);
            Sender.Tell(message);
        });
        Receive<SomeMessage>(message => {...});
    }
}
```

Props

`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

```
Props props1 = Props.Create(typeof(MyActor));
Props props2 = Props.Create(() => new MyActorWithArgs("arg"));
Props props3 = Props.Create<MyActor>();
Props props4 = Props.Create(typeof(MyActorWithArgs), "arg");
```

The second variant shows how to pass constructor arguments to the `Actor` being created, but it should only be used outside of actors as explained below.

Recommended Practices

It is a good idea to provide static factory methods on the `ReceiveActor` which help keeping the creation of suitable `Props` as close to the actor definition as possible.

```
public class DemoActor : ReceiveActor
{
    private readonly int _magicNumber;

    public DemoActor(int magicNumber)
    {
        _magicNumber = magicNumber;
        Receive<int>(x =>
        {
            Sender.Tell(x + _magicNumber);
        });
    }

    public static Props Props(int magicNumber)
    {
        return Akka.Actor.Props.Create(() => new DemoActor(magicNumber));
    }
}

system.ActorOf(DemoActor.Props(42), "demo");
```

Another good practice is to declare local messages (messages that are sent in process) within the Actor, which makes it easier to know what messages are generally being sent over the wire vs in process.:

```
public class DemoActor : UntypedActor
{
    protected override void OnReceive(object message)
    {
        switch (message)
        {
            case DemoActorLocalMessages.DemoActorLocalMessage1 msg1:
                // Handle message here...
                break;
            case DemoActorLocalMessages.DemoActorLocalMessage2 msg2:
                // Handle message here...
                break;
            default:
                break;
        }
    }

    class DemoActorLocalMessages
    {
        public class DemoActorLocalMessage1
        {
        }

        public class DemoActorLocalMessage2
        {
        }
    }
}
```

Creating Actors with Props

Actors are created by passing a `Props` instance into the `ActorOf` factory method which is available on `ActorSystem` and `ActorContext`.

```
// ActorSystem is a heavy object: create only one per application
ActorSystem system = ActorSystem.Create("MySystem");
```

```
IActorRef myActor = system.ActorOf<MyActor>("myactor");
```

Using the `ActorSystem` will create top-level actors, supervised by the actor system's provided guardian actor, while using an actor's context will create a child actor.

```
public class FirstActor : ReceiveActor
{
    IActorRef child = Context.ActorOf<MyActor>("myChild");
    // plus some behavior ...
}
```

It is recommended to create a hierarchy of children, grand-children and so on such that it fits the logical failure-handling structure of the application, see [Actor Systems](#).

The call to `ActorOf` returns an instance of `IActorRef`. This is a handle to the actor instance and the only way to interact with it. The `IActorRef` is immutable and has a one to one relationship with the `Actor` it represents. The `IActorRef` is also serializable and network-aware. This means that you can serialize it, send it over the wire and use it on a remote host and it will still be representing the same `Actor` on the original node, across the network.

The name parameter is optional, but you should preferably name your actors, since that is used in log messages and for identifying actors. The name must not be empty or start with `$$`, but it may contain URL encoded characters (eg. `%20` for a blank space). If the given name is already in use by another child to the same parent an `InvalidActorNameException` is thrown.

Actors are automatically started asynchronously when created.

Actor API

If the current actor behavior does not match a received message, it's recommended that you call the `unhandled` method, which by default publishes a new `Akka.Actor.UnhandledMessage(message, sender, recipient)` on the actor system's event stream (set configuration item `Unhandled` to `on` to have them converted into actual `Debug` messages).

In addition, it offers:

- `Self` reference to the `IActorRef` of the actor
- `Sender` reference sender Actor of the last received message, typically used as described in [Reply to messages](#).
- `SupervisorStrategy` user overridable definition the strategy to use for supervising child actors

This strategy is typically declared inside the actor in order to have access to the actor's internal state within the decider function: since failure is communicated as a message sent to the supervisor and processed like other messages (albeit outside of the normal behavior), all values and variables within the actor are available, as is the `Sender` reference (which will be the immediate child reporting the failure; if the original failure occurred within a distant descendant it is still reported one level up at a time).

- `Context` exposes contextual information for the actor and the current message, such as:
 - factory methods to create child actors (`ActorOf`)
 - system that the actor belongs to
 - parent supervisor
 - supervised children
 - lifecycle monitoring
 - hotswap behavior stack as described in [HotSwap](#)

The remaining visible methods are user-overridable life-cycle hooks which are described in the following:

```
public override void PreStart()
{
}

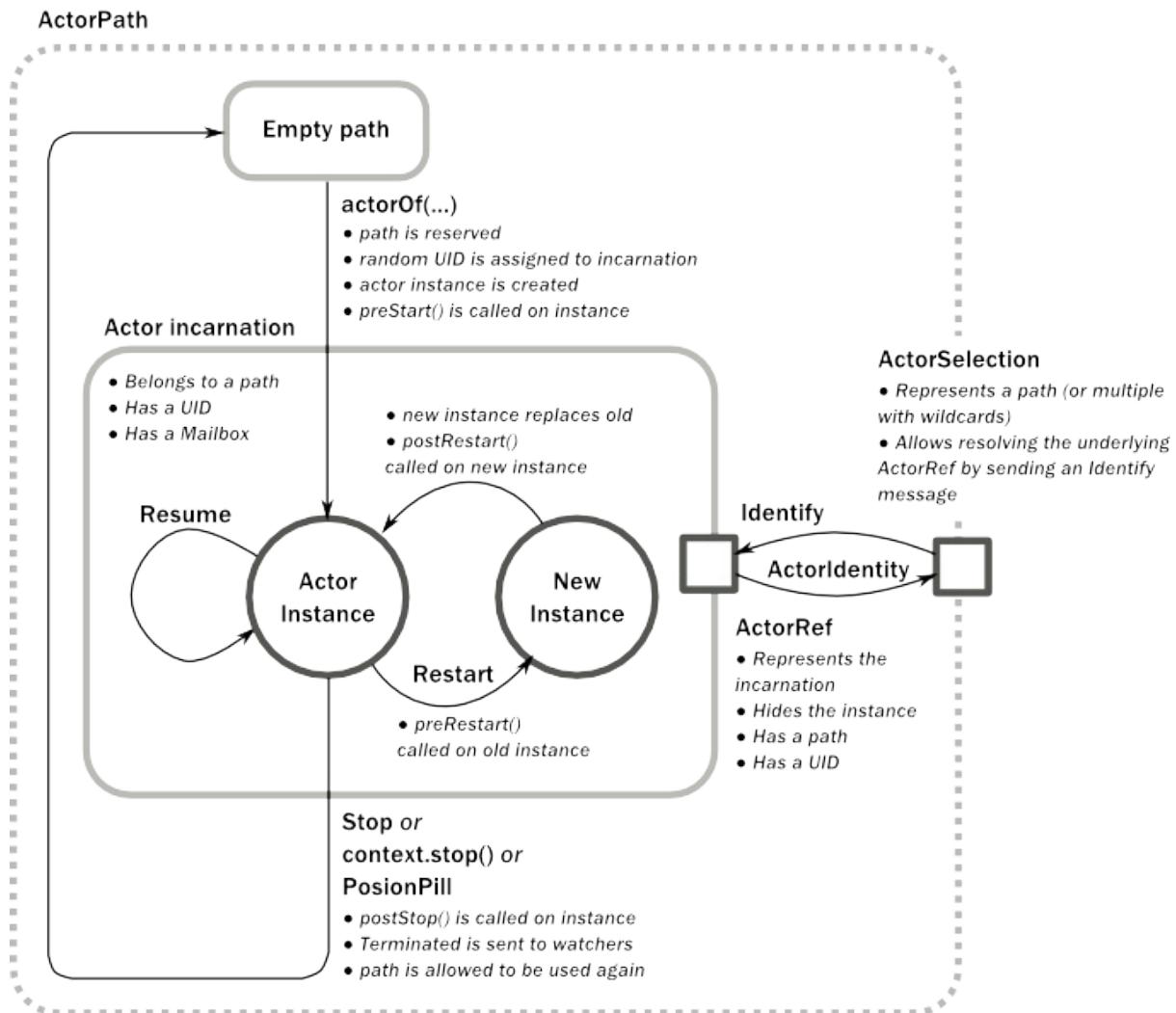
protected override void PreRestart(Exception reason, object message)
{
    foreach (IActorRef each in Context.GetChildren())
    {
        Context.Unwatch(each);
        Context.Stop(each);
    }
    PostStop();
}

protected override void PostRestart(Exception reason)
{
    PreStart();
}

protected override void PostStop()
{
}
```

The implementations shown above are the defaults provided by the `ReceiveActor` class.

Actor Lifecycle



A path in an actor system represents a "place" which might be occupied by a living actor. Initially (apart from system initialized actors) a path is empty. When `Actorof()` is called it assigns an incarnation of the actor described by the passed `Props` to the given path. An actor incarnation is identified by the path and a UID. A restart only swaps the Actor instance defined by the `Props` but the incarnation and hence the UID remains the same.

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 `IActorRef` 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 `IActorRef` of the old incarnation will not point to the new one.

`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 (see [Identifying Actors via Actor Selection](#)). This can also be done with the `resolveOne` method of the `Actorselection`, which returns a `Task` of the matching `IActorRef`.

Lifecycle Monitoring aka DeathWatch

In order to be notified when another actor terminates (i.e. stops permanently, not temporary failure and restart), an actor may register itself for reception of the `Terminated` message dispatched by the other actor upon termination (see [Stopping Actors](#)). This service is provided by the DeathWatch component of the actor system.

Registering a monitor is easy (see fourth line, the rest is for demonstrating the whole functionality):

```
public class WatchActor : ReceiveActor
{
    private IActorRef child = Context.ActorOf(Props.Empty, "child");
    private IActorRef lastSender = Context.System.DeadLetters;

    public WatchActor()
    {
        Context.Watch(child); // <-- this is the only call needed for registration

        Receive<string>(s => s.Equals("kill"), msg =>
        {
            Context.Stop(child);
            lastSender = Sender;
        });

        Receive<Terminated>(t => t.ActorRef.Equals(child), msg =>
        {
            lastSender.Tell("finished");
        });
    }
}
```

It should be noted that the `Terminated` message is generated independent of the order in which registration and termination occur. In particular, the watching actor will receive a `Terminated` message even if the watched actor has already been terminated at the time of registration.

Registering multiple times does not necessarily lead to multiple messages being generated, but there is no guarantee that only exactly one such message is received: if termination of the watched actor has generated and queued the message, and another registration is done before this message has been processed, then a second message will be queued, because registering for monitoring of an already terminated actor leads to the immediate generation of the `Terminated` message.

It is also possible to deregister from watching another actor's liveliness using `Context.Unwatch(target)`. This works even if the `Terminated` message has already been enqueued in the mailbox; after calling `unwatch` no `Terminated` message for that actor will be processed anymore.

Start Hook

Right after starting the actor, its `Prestart` method is invoked.

```
protected override void PreStart()
{
    child = Context.ActorOf(Props.Empty);
}
```

This method is called when the actor is first created. During restarts it is called by the default implementation of `PostRestart`, which means that by overriding that method you can choose whether the initialization code in this method is called only exactly once for this actor or for every restart. Initialization code which is part of the actor's constructor will always be called when an instance of the actor class is created, which happens at every restart.

Restart Hooks

All actors are supervised, i.e. linked to another actor with a fault handling strategy. Actors may be restarted in case an exception is thrown while processing a message (see [Supervision and Monitoring](#)). This restart involves the hooks mentioned above:

- The old actor is informed by calling `PreRestart` with the exception which caused the restart and the message which triggered that exception; the latter may be `None` if the restart was not caused by processing a message, e.g. when a supervisor does not trap the exception and is restarted in turn by its supervisor, or if an actor is restarted due to a sibling's failure. If the message is available, then that message's sender is also accessible in the usual way (i.e. by calling the `Sender` property). This method is the best place for cleaning up, preparing hand-over to the fresh actor instance, etc. By default it stops all children and calls `PostStop`.
- The initial factory from the `ActorOf` call is used to produce the fresh instance.
- The new actor's `PostRestart` method is invoked with the exception which caused the restart. By default the `PreStart` is called, just as in the normal start-up case.

An actor restart replaces only the actual actor object; the contents of the mailbox is unaffected by the restart, so processing of messages will resume after the `PostRestart` hook returns. The message that triggered the exception will not be received again. Any message sent to an actor while it is being restarted will be queued to its mailbox as usual.

[!WARNING] Be aware that the ordering of failure notifications relative to user messages is not deterministic. In particular, a parent might restart its child before it has processed the last messages sent by the child before the failure. See Discussion: [Message Ordering for details](#).

Stop Hook

After stopping an actor, its `PostStop` hook is called, which may be used e.g. for deregistering this actor from other services. This hook is guaranteed to run after message queuing has been disabled for this actor, i.e. messages sent to a stopped actor will be redirected to the `DeadLetters` of the `ActorSystem`.

Identifying Actors via Actor Selection

As described in Actor References, Paths and Addresses, each actor has a unique logical path, which is obtained by following the chain of actors from child to parent until reaching the root of the actor system, and it has a physical path, which may differ if the supervision chain includes any remote supervisors. These paths are used by the system to look up actors, e.g. when a remote message is received and the recipient is searched, but they are also useful more directly: actors may look up other actors by specifying absolute or relative paths—logical or physical—and receive back an `ActorSelection` with the result:

```
// will look up this absolute path
Context.ActorSelection("/user/serviceA/actor");

// will look up sibling beneath same supervisor
Context.ActorSelection("../joe");
```

[!NOTE] It is always preferable to communicate with other Actors using their `IActorRef` instead of relying upon `ActorSelection`. Exceptions are: sending messages using the At-Least-Once Delivery facility, initiating first contact with a remote system. In all other cases `ActorRefs` can be provided during Actor creation or initialization, passing them from parent to child or introducing Actors by sending their `ActorRefs` to other Actors within messages.

The supplied path is parsed as a `System.Uri`, which basically means that it is split on `/` into path elements. If the path starts with `/`, it is absolute and the look-up starts at the root guardian (which is the parent of `"/user"`); otherwise it starts at the current actor. If a path element equals `...`, the look-up will take a step "up" towards the supervisor of the currently traversed actor, otherwise it will step "down" to the named child. It should be noted that the `...` in actor paths here always means the logical structure, i.e. the supervisor.

The path elements of an actor selection may contain wildcard patterns allowing for broadcasting of messages to that section:

```
// will look all children to serviceB with names starting with worker
Context.ActorSelection("/user/serviceB/worker*");

// will look up all siblings beneath same supervisor
Context.ActorSelection("../*");
```

Messages can be sent via the `ActorSelection` and the path of the `ActorSelection` is looked up when delivering each message. If the selection does not match any actors the message will be dropped.

To acquire an `IActorRef` for an `ActorSelection` you need to send a message to the selection and use the `Sender` reference of the reply from the actor. There is a built-in `Identify` message that all Actors will understand and automatically reply to with a `ActorIdentity` message containing the `IActorRef`. This message is handled specially by the actors which are traversed in the sense that if a concrete name lookup fails (i.e. a non-wildcard path element does not correspond to a live actor) then a negative result is generated. Please note that this does not mean that delivery of that reply is guaranteed, it still is a normal message.

```
public class Follower : ReceiveActor
{
    private readonly IActorRef _probe;
    private string identifyId = "1";
    private IActorRef _another;

    public Follower(IActorRef probe)
    {
        _probe = probe;

        var selection = Context.ActorSelection("/user/another");
        selection.Tell(new Identify(identifyId), Self);

        Receive<ActorIdentity>(identity =>
        {
            if (identity.MessageId.Equals(identifyId))
            {
                var subject = identity.Subject;

                if (subject == null)
                {
                    Context.Stop(Self);
                }
                else
                {
                    _another = subject;
                    Context.Watch(_another);
                    _probe.Tell(subject, Self);
                }
            }
        });
    }

    Receive<Terminated>(t =>
    {
        if (t.ActorRef.Equals(_another))
        {
            Context.Stop(Self);
        }
    });
}
```

```

        });
    }
}
```

You can also acquire an `IActorRef` for an `ActorSelection` with the `ResolveOne` method of the `ActorSelection`. It returns a Task of the matching `IActorRef` if such an actor exists. It is completed with failure `akka.actor.ActorNotFound` if no such actor exists or the identification didn't complete within the supplied timeout.

Remote actor addresses may also be looked up, if `remoting` is enabled:

```
Context.ActorSelection("akka.tcp://app@otherhost:1234/user/serviceB");
```

Messages and Immutability

[!IMPORTANT] Messages can be any kind of object but have to be immutable. Akka can't enforce immutability (yet) so this has to be by convention.

Here is an example of an immutable message:

```

public class ImmutableMessage
{
    public ImmutableMessage(int sequenceNumber, List<string> values)
    {
        SequenceNumber = sequenceNumber;
        Values = values.AsReadOnly();
    }

    public int SequenceNumber { get; }
    public IReadOnlyCollection<string> Values { get; }
}
```

Send messages

Messages are sent to an Actor through one of the following methods.

- `Tell()` means `fire-and-forget`, e.g. send a message asynchronously and return immediately.
- `Ask()` sends a message asynchronously and returns a Future representing a possible reply.

Message ordering is guaranteed on a per-sender basis.

[!NOTE] There are performance implications of using `Ask` since something needs to keep track of when it times out, there needs to be something that bridges a `Task` into an `IActorRef` and it also needs to be reachable through remoting. So always prefer `Tell` for performance, and only `Ask` if you must.

In all these methods you have the option of passing along your own `IActorRef`. Make it a practice of doing so because it will allow the receiver actors to be able to respond to your message, since the `Sender` reference is sent along with the message.

Tell: Fire-forget

This is the preferred way of sending messages. No blocking waiting for a message. This gives the best concurrency and scalability characteristics.

```
// don't forget to think about who is the sender (2nd argument)
target.Tell(message, Self);
```

The sender reference is passed along with the message and available within the receiving actor via its `Sender` property while processing this message. Inside of an actor it is usually `self` who shall be the sender, but there can be cases where replies shall be routed to some other actor—e.g. the parent—in which the second argument to `Tell` would be a different one. Outside of an actor and if no reply is needed the second argument can be `null`; if a reply is needed outside of an actor you can use the ask-pattern described next.

Ask: Send-And-Receive-Future

The ask pattern involves actors as well as Tasks, hence it is offered as a use pattern rather than a method on `ActorRef`:

```
var tasks = new List<Task>();
tasks.Add(actorA.Ask("request", TimeSpan.FromSeconds(1)));
tasks.Add(actorB.Ask("another request", TimeSpan.FromSeconds(5)));

Task.WhenAll(tasks).PipeTo(actorC, Self);
```

This example demonstrates `Ask` together with the `Pipe` Pattern on tasks, because this is likely to be a common combination. Please note that all of the above is completely non-blocking and asynchronous: `Ask` produces a `Task`, two of which are awaited until both are completed, and when that happens, a new `Result` object is forwarded to another actor.

Using `Ask` will send a message to the receiving Actor as with `Tell`, and the receiving actor must reply with `Sender.Tell(reply, Self)` in order to complete the returned `Task` with a value. The `Ask` operation involves creating an internal actor for handling this reply, which needs to have a timeout after which it is destroyed in order not to leak resources; see more below.

[!WARNING] To complete the `Task` with an exception you need send a `Failure` message to the sender. This is not done automatically when an actor throws an exception while processing a message.

```
try
{
    var result = operation();
    Sender.Tell(result, Self);
}
catch (Exception e)
{
    Sender.Tell(new Failure { Exception = e }, Self);
}
```

If the actor does not complete the task, it will expire after the timeout period, specified as parameter to the `Ask` method, and the `Task` will be cancelled and throw a `TaskCancelledException`.

For more information on Tasks, check out the [MSDN documentation.aspx](#)).

[!WARNING] When using task callbacks inside actors, you need to carefully avoid closing over the containing actor's reference, i.e. do not call methods or access mutable state on the enclosing actor from within the callback. This would break the actor encapsulation and may introduce synchronization bugs and race conditions because the callback will be scheduled concurrently to the enclosing actor. Unfortunately there is not yet a way to detect these illegal accesses at compile time.

Forward message

You can forward a message from one actor to another. This means that the original sender address/reference is maintained even though the message is going through a 'mediator'. This can be useful when writing actors that work as routers, load-balancers, replicators etc. You need to pass along your context variable as well.

```
target.Forward(result, Context);
```

Receive messages

To receive a message you should create a `Receive` handler in a constructor.

```
Receive<string>(ms => Console.WriteLine("Received message: " + msg));
```

Handler priority

If more than one handler matches, the one that appears first is used while the others are ignored.

```
Receive<string>(s => Console.WriteLine("Received string: " + s));      //1
Receive<string>(s => Console.WriteLine("Also received string: " + s));    //2
Receive<object>(o => Console.WriteLine("Received object: " + o));        //3
```

Example

The actor receives a message of type string. Only the first handler is invoked, even though all three handlers can handle that message.

Using predicates

By specifying a predicate, you can choose which messages to handle.

```
Receive<string>(s => s.Length > 5, s => Console.WriteLine("Received string: " + s));
```

The handler above will only be invoked if the length of the string is greater than 5.

If the predicate do not match, the next matching handler will be used.

```
Receive<string>(s => s.Length > 5, s => Console.WriteLine("1: " + s));      //1
Receive<string>(s => s.Length > 2, s => Console.WriteLine("2: " + s));      //2
Receive<string>(s => Console.WriteLine("3: " + s));                          //3
```

Example

The actor receives the message "123456". Since the length of is 6, the predicate specified for the first handler will return true, and the first handler will be invoked resulting in "1: 123456" being written to the console.

[!NOTE] Note that even though the predicate for the second handler matches, and that the third handler matches all messages of type string only the first handler is invoked.

Example

If the actor receives the message "1234", then "2: 1234" will be written to the console.

Example

If the actor receives the message "12", then "3: 12" will be written on the console.

Predicates can be specified before the action handler or after. These two declarations are equivalent:

```
Receive<string>(s => s.Length > 5, s => Console.WriteLine("Received string: " + s));
Receive<string>(s => Console.WriteLine("Received string: " + s), s => s.Length > 5);
```

Unmatched messages

If the actor receives a message for which no handler matches, the unhandled message is published to the EventStream wrapped in an `UnhandledMessage`. To change this behavior override `Unhandled(object message)`

```
protected override void Unhandled(object message)
{
    //Do something with the message.
}
```

Another option is to add a handler last that matches all messages, using `ReceiveAny()`.

ReceiveAny

To catch messages of any type the `ReceiveAny(Action<object> handler)` overload can be specified.

```
Receive<string>(s => Console.WriteLine("Received string: " + s));
ReceiveAny(o => Console.WriteLine("Received object: " + o));
```

Since it handles everything, it must be specified last. Specifying handlers it after will cause an exception.

```
ReceiveAny(o => Console.WriteLine("Received object: " + o));
Receive<string>(s => Console.WriteLine("Received string: " + s)); //This will cause an exception
```

[!NOTE] Note that `Receive<object>(Action<object> handler)` behaves the same as `ReceiveAny()` as it catches all messages. These two are equivalent:

```
ReceiveAny(o => Console.WriteLine("Received object: " + o));
Receive<object>(o => Console.WriteLine("Received object: " + o));
```

Non generic overloads

`Receive` has non generic overloads:

```
Receive(typeof(string), obj => Console.WriteLine(obj.ToString()));
```

Predicates can go before or after the handler:

```
Receive(typeof(string), obj => ((string) obj).Length > 5, obj => Console.WriteLine(obj.ToString()));
Receive(typeof(string), obj => Console.WriteLine(obj.ToString()), obj => ((string) obj).Length > 5);
```

And the non generic Func

```
Receive(typeof(string), obj =>
{
    var s = (string) obj;
    if (s.Length > 5)
    {
        Console.WriteLine("1: " + s);
        return true;
    }
});
```

```
    return false;
});
```

Reply to messages

If you want to have a handle for replying to a message, you can use `Sender`, which gives you an `IActorRef`. You can reply by sending to that `IActorRef` with `Sender.Tell(replyMsg, Self)`. You can also store the `IActorRef` for replying later, or passing on to other actors. If there is no sender (a message was sent without an actor or task context) then the sender defaults to a 'dead-letter' actor ref.

```
Receive<string>(() =>
{
    var result = calculateResult();

    // do not forget the second argument!
    Sender.Tell(result, Self);
})
```

Receive timeout

The `IActorContext` `SetReceiveTimeout` defines the inactivity timeout after which the sending of a `ReceiveTimeout` message is triggered. When specified, the receive function should be able to handle an `Akka.Actor.ReceiveTimeout` message.

[!NOTE] Please note that the receive timeout might fire and enqueue the `ReceiveTimeout` message right after another message was enqueued; hence it is not guaranteed that upon reception of the receive timeout there must have been an idle period beforehand as configured via this method.

Once set, the receive timeout stays in effect (i.e. continues firing repeatedly after inactivity periods). Pass in `null` to `SetReceiveTimeout` to switch off this feature.

```
public class MyActor : ReceiveActor
{
    private ILoggingAdapter log = Context.GetLogger();

    public MyActor()
    {
        Receive<string>(s => s.Equals("Hello"), msg =>
        {
            Context.SetReceiveTimeout(TimeSpan.FromMilliseconds(100));
        });

        Receive<ReceiveTimeout>(msg =>
        {
            Context.SetReceiveTimeout(null);
            throw new Exception("Receive timed out");
            return;
        });
    }
}
```

Stopping actors

Actors are stopped by invoking the `Stop` method of a `ActorRefFactory`, i.e. `ActorContext` or `ActorSystem`. Typically the context is used for stopping child actors and the system for stopping top level actors. The actual termination of the actor is performed asynchronously, i.e. `stop` may return before the actor is stopped.

```
public class MyStoppingActor : ReceiveActor
{
    private IActorRef child;

    public MyStoppingActor()
    {
        Receive<string>(s => s.Equals("interrupt-child"), msg =>
        {
            Context.Stop(child);
        });

        Receive<string>(s => s.Equals("done"), msg =>
        {
            Context.Stop(Self);
        });
    }
}
```

Processing of the current message, if any, will continue before the actor is stopped, but additional messages in the mailbox will not be processed. By default these messages are sent to the `DeadLetters` of the `ActorSystem`, but that depends on the mailbox implementation.

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`, 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.

The `PostStop` hook is invoked after an actor is fully stopped. This enables cleaning up of resources:

```
protected override void PostStop()
{
    // clean up resources here ...
}
```

[!NOTE] Since stopping an actor is asynchronous, you cannot immediately reuse the name of the child you just stopped; this will result in an `InvalidActorNameException`. Instead, watch the terminating actor and create its replacement in response to the `Terminated` message which will eventually arrive.

PoisonPill

You can also send an actor the `Akka.Actor.PoisonPill` message, which will stop the actor when the message is processed. `PoisonPill` is enqueued as ordinary messages and will be handled after messages that were already queued in the mailbox.

Use it like this:

```
myActor.Tell(PoisonPill.Instance, Sender);
```

Graceful Stop

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

```
var manager = system.ActorOf<Manager>();

try
{
    await manager.GracefulStop(TimeSpan.FromMilliseconds(5), "shutdown");
    // the actor has been stopped
}
catch (TaskCanceledException)
{
    // the actor wasn't stopped within 5 seconds
}

...

public class Manager : ReceiveActor
{
    private IActorRef worker = Context.Watch(Context.ActorOf<Cruncher>("worker"));

    public Manager()
    {
        Receive<string>(s => s.Equals("job"), msg =>
        {
            worker.Tell("crunch");
        });

        Receive<Shutdown>(_ =>
        {
            worker.Tell(PoisonPill.Instance, Self);
            Context.Become(ShuttingDown);
        });
    }

    private void ShuttingDown(object message)
    {
        Receive<string>(s => s.Equals("job"), msg =>
        {
            Sender.Tell("service unavailable, shutting down", Self);
        });

        Receive<Shutdown>(_ =>
        {
            Context.Stop(Self);
        });
    }
}
```

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()`.

In the above example a `"shutdown"` message is sent to the target actor to initiate the process of stopping the actor. You can use `PoisonPill` for this, but then you have limited possibilities to perform interactions with other actors before stopping the target actor. Simple cleanup tasks can be handled in `PostStop`.

[!WARNING] Keep in mind that an actor stopping and its name being deregistered are separate events which happen asynchronously from each other. Therefore it may be that you will find the name still in use after `GracefulStop()` returned. In order to guarantee proper deregistration, only reuse names from within a supervisor you control and only in response to a `Terminated` message, i.e. not for top-level actors.

Become/Unbecome

Upgrade

Akka supports hotswapping the Actor's message loop (e.g. its implementation) at runtime. Use the `Context.Become` method from within the Actor. The hotswapped code is kept in a `stack` which can be pushed (replacing or adding at the top) and popped.

[!WARNING] Please note that the actor will revert to its original behavior when restarted by its Supervisor.

To hotswap the Actor using `Context.Become` :

```
public class HotSwapActor : ReceiveActor
{
    public HotSwapActor()
    {
        Receive<string>(s => s.Equals("foo"), msg =>
        {
            Become(Angry);
        });

        Receive<string>(s => s.Equals("bar"), msg =>
        {
            Become(Happy);
        });
    }

    private void Angry(object message)
    {
        Receive<string>(s => s.Equals("foo"), msg =>
        {
            Sender.Tell("I am already angry?");
        });

        Receive<string>(s => s.Equals("bar"), msg =>
        {
            Become(Happy);
        });
    }

    private void Happy(object message)
    {
        Receive<string>(s => s.Equals("foo"), msg =>
        {
            Sender.Tell("I am already happy :-)");
        });

        Receive<string>(s => s.Equals("bar"), msg =>
        {
            Become(Angry);
        });
    }
}
```

This variant of the `Become` method is useful for many different things, such as to implement a Finite State Machine (FSM). It will replace the current behavior (i.e. the top of the behavior stack), which means that you do not use `Unbecome`, instead always the next behavior is explicitly installed.

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).

```
public class Swapper : ReceiveActor
```

```

{
    public class Swap
    {
        public static Swap Instance = new Swap();
        private Swap() { }
    }

    private ILoggingAdapter log = Context.GetLogger();

    public Swapper()
    {
        Receive<Swap>(swap1 =>
        {
            log.Info("Hi");

            BecomeStacked(() =>
            {
                Receive<Swap>(swap2 =>
                {
                    log.Info("Ho");
                    UnbecomeStacked();
                });
            });
        });
    }
}

...

static void Main(string[] args)
{
    var system = ActorSystem.Create("MySystem");
    var swapper = system.ActorOf<Swapper>();

    swapper.Tell(Swapper.SWAP);
    swapper.Tell(Swapper.SWAP);
    swapper.Tell(Swapper.SWAP);
    swapper.Tell(Swapper.SWAP);
    swapper.Tell(Swapper.SWAP);
    swapper.Tell(Swapper.SWAP);

    Console.ReadLine();
}

```

Stash

The `IWithUnboundedStash` interface enables an actor to temporarily stash away messages that can not or should not be handled using the actor's current behavior. Upon changing the actor's message handler, i.e., right before invoking `Context.BecomeStacked()` or `Context.UnbecomeStacked()`, all stashed messages can be "unstashed", thereby prepending them to the actor's mailbox. This way, the stashed messages can be processed in the same order as they have been received originally. An actor that implements `IWithUnboundedStash` will automatically get a deque-based mailbox.

Here is an example of the `IWithUnboundedStash` interface in action:

```

public class ActorWithProtocol : ReceiveActor, IWithUnboundedStash
{
    public IStash Stash { get; set; }

    public ActorWithProtocol()
    {
        Receive<string>(s => s.Equals("open"), open =>
        {
            Stash.UnstashAll();
        });
    }
}

```

```

    BecomeStacked(() =>
    {
        Receive<string>(s => s.Equals("write"), write =>
        {
            // do writing...
        });

        Receive<string>(s => s.Equals("close"), write =>
        {
            Stash.UnstashAll();
            Context.UnbecomeStacked();
        });

        ReceiveAny(_ => Stash.Stash());
    });
};

ReceiveAny(_ => Stash.Stash());
}
}

```

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()`.

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 `IWithUnboundedStash` interface implementation of `PreRestart` will call `UnstashAll()`, which is usually the desired behavior.

Killing an Actor

You can kill an actor by sending a `Kill` message. This will cause the actor to throw a `ActorKilledException`, triggering a failure. The actor will suspend operation and its supervisor will be asked how to handle the failure, which may mean resuming the actor, restarting it or terminating it completely. See [What Supervision Means](#) for more information.

Use `Kill` like this:

```
// kill the 'victim' actor
victim.Tell(Akka.Actor.Kill.Instance, ActorRef.NoSender);
```

Actors and exceptions

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 and Monitoring). 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.

Initialization patterns

The rich lifecycle hooks of `Actors` provide a useful toolkit to implement various initialization patterns. During the lifetime of an `IActorRef`, an actor can potentially go through several restarts, where the old instance is replaced by a fresh one, invisibly to the outside observer who only sees the `IActorRef`.

One may think about the new instances as "incarnations". Initialization might be necessary for every incarnation of an actor, but sometimes one needs initialization to happen only at the birth of the first instance when the `IActorRef` is created. The following sections provide patterns for different initialization needs.

Initialization via constructor

Using the constructor for initialization has various benefits. First of all, it makes it possible to use readonly fields to store any state that does not change during the life of the actor instance, making the implementation of the actor more robust. The constructor is invoked for every incarnation of the actor, therefore the internals of the actor can always assume that proper initialization happened. This is also the drawback of this approach, as there are cases when one would like to avoid reinitializing internals on restart. For example, it is often useful to preserve child actors across restarts. The following section provides a pattern for this case.

Initialization via PreStart

The method `PreStart()` of an actor is only called once directly during the initialization of the first instance, that is, at creation of its `ActorRef`. In the case of restarts, `PreStart()` is called from `PostRestart()`, therefore if not overridden, `PreStart()` is called on every incarnation. However, overriding `PostRestart()` one can disable this behavior, and ensure that there is only one call to `PreStart()`.

One useful usage of this pattern is to disable creation of new `ActorRefs` for children during restarts. This can be achieved by overriding `PreRestart()`:

```
protected override void PreStart()
{
    // Initialize children here
}

// Overriding postRestart to disable the call to preStart() after restarts
protected override void PostRestart(Exception reason)
{
```

```

}

// The default implementation of PreRestart() stops all the children
// of the actor. To opt-out from stopping the children, we
// have to override PreRestart()
protected override void PreRestart(Exception reason, object message)
{
    // Keep the call to PostStop(), but no stopping of children
    PostStop();
}

```

Please note, that the child actors are *still restarted*, but no new `IActorRef` is created. One can recursively apply the same principles for the children, ensuring that their `PreStart()` method is called only at the creation of their refs.

For more information see [What Restarting Means](#).

Initialization via message passing

There are cases when it is impossible to pass all the information needed for actor initialization in the constructor, for example in the presence of circular dependencies. In this case the actor should listen for an initialization message, and use `Become()` or a finite state-machine state transition to encode the initialized and uninitialized states of the actor.

```

public class Service : ReceiveActor
{
    private string _initializeMe;

    public Service()
    {
        Receive<string>(s => s.Equals("init"), init =>
        {
            _initializeMe = "Up and running";

            Become(() =>
            {
                Receive<string>(s => s.Equals("U OK?") && _initializeMe != null, m =>
                {
                    Sender.Tell(_initializeMe, Self);
                });
            });
        });
    }
}

```

If the actor may receive messages before it has been initialized, a useful tool can be the `stash` to save messages until the initialization finishes, and replaying them after the actor became initialized.

[!WARNING] This pattern should be used with care, and applied only when none of the patterns above are applicable. One of the potential issues is that messages might be lost when sent to remote actors. Also, publishing an `IActorRef` in an uninitialized state might lead to the condition that it receives a user message before the initialization has been done.

UntypedActor API

The Actor Model provides a higher level of abstraction for writing concurrent and distributed systems. It alleviates the developer from having to deal with explicit locking and thread management, making it easier to write correct concurrent and parallel systems. Actors were defined in the 1973 paper by Carl Hewitt but have been popularized by the Erlang language, and used for example at Ericsson with great success to build highly concurrent and reliable telecom systems.

[!NOTE] UntypedActor API is recommended for C# 7 users.

Creating Actors

[!NOTE] Since Akka.NET enforces parental supervision every actor is supervised and (potentially) the supervisor of its children, it is advisable that you familiarize yourself with [Actor Systems](#) and [Supervision and Monitoring](#) and it may also help to read [Actor References, Paths and Addresses](#).

Defining an Actor class

Actors in C# are implemented by extending the `UntypedActor` class and and implementing the `OnReceive` method. This method takes the message as a parameter.

Here is an example:

```
public class MyActor : UntypedActor
{
    private ILoggingAdapter log = Context.GetLogger();

    protected override void OnReceive(object message)
    {
        switch (message)
        {
            case "test":
                log.Info("received test");
                break;
            default:
                log.Info("received unknown message");
                break;
        }
    }
}
```

Props

`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

```
Props props1 = Props.Create(typeof(MyActor));
Props props2 = Props.Create(() => new MyActorWithArgs("arg"));
Props props3 = Props.Create<MyActor>();
Props props4 = Props.Create(typeof(MyActorWithArgs), "arg");
```

The second variant shows how to pass constructor arguments to the `Actor` being created, but it should only be used outside of actors as explained below.

Recommended Practices

It is a good idea to provide static factory methods on the `ReceiveActor` which help keeping the creation of suitable `Props` as close to the actor definition as possible.

```
public class DemoActor : UntypedActor
{
    private readonly int _magicNumber;

    public DemoActor(int magicNumber)
    {
        _magicNumber = magicNumber;
    }

    protected override void OnReceive(object message)
    {
        if (message is int x)
        {
            Sender.Tell(x + _magicNumber);
        }
    }

    public static Props Props(int magicNumber)
    {
        return Akka.Actor.Props.Create(() => new DemoActor(magicNumber));
    }
}

system.ActorOf(DemoActor.Props(42), "demo");
```

Another good practice is to declare what messages an `Actor` can receive in the companion object of the `Actor`, which makes easier to know what it can receive:

```
public class DemoMessagesActor : UntypedActor
{
    public class Greeting
    {
        public Greeting(string from)
        {
            From = from;
        }

        public string From { get; }
    }

    public class Goodbye
    {
        public static Goodbye Instance = new Goodbye();

        private Goodbye() {}
    }

    private ILoggingAdapter log = Context.GetLogger();

    protected override void OnReceive(object message)
    {
        switch (message)
        {
            case Greeting greeting:
                Sender.Tell($"I was greeted by {greeting.From}", Self);
                break;
            case Goodbye goodbye:
                log.Info("Someone said goodbye to me.");
                break;
        }
    }
}
```

```

    }
}
```

Creating Actors with Props

Actors are created by passing a `Props` instance into the `ActorOf` factory method which is available on `ActorSystem` and `ActorContext`.

```
// ActorSystem is a heavy object: create only one per application
ActorSystem system = ActorSystem.Create("MySystem");
IActorRef myActor = system.ActorOf<MyActor>("myactor");
```

Using the `ActorSystem` will create top-level actors, supervised by the actor system's provided guardian actor, while using an actor's context will create a child actor.

```
public class FirstActor : UntypedActor
{
    IActorRef child = Context.ActorOf<MyActor>("myChild");
    // plus some behavior ...
}
```

It is recommended to create a hierarchy of children, grand-children and so on such that it fits the logical failure-handling structure of the application, see [Actor Systems](#).

The call to `ActorOf` returns an instance of `IActorRef`. This is a handle to the actor instance and the only way to interact with it. The `IActorRef` is immutable and has a one to one relationship with the `Actor` it represents. The `IActorRef` is also serializable and network-aware. This means that you can serialize it, send it over the wire and use it on a remote host and it will still be representing the same `Actor` on the original node, across the network.

The name parameter is optional, but you should preferably name your actors, since that is used in log messages and for identifying actors. The name must not be empty or start with `$$`, but it may contain URL encoded characters (eg. `%20` for a blank space). If the given name is already in use by another child to the same parent an `InvalidActorNameException` is thrown.

Actors are automatically started asynchronously when created.

Actor API

The `UntypedActor` class defines only one abstract method, the above mentioned `OnReceive(object message)`, which implements the behavior of the actor.

If the current actor behavior does not match a received message, it's recommended that you call the `Unhandled` method, which by default publishes a new `Akka.Actor.UnhandledMessage(message, sender, recipient)` on the actor system's event stream (set configuration item `UnHandled` to `on` to have them converted into actual `Debug` messages).

In addition, it offers:

- `Self` reference to the `IActorRef` of the actor
- `Sender` reference sender Actor of the last received message, typically used as described in [Reply to messages](#).
- `SupervisorStrategy` user overridable definition the strategy to use for supervising child actors

This strategy is typically declared inside the actor in order to have access to the actor's internal state within the decider function: since failure is communicated as a message sent to the supervisor and processed like other messages (albeit outside of the normal behavior), all values and variables within the actor are available, as is the

`Sender` reference (which will be the immediate child reporting the failure; if the original failure occurred within a distant descendant it is still reported one level up at a time).

- `Context` exposes contextual information for the actor and the current message, such as:
 - factory methods to create child actors (`ActorOf`)
 - system that the actor belongs to
 - parent supervisor
 - supervised children
 - lifecycle monitoring
 - hotswap behavior stack as described in [Become/Unbecome](#)

The remaining visible methods are user-overridable life-cycle hooks which are described in the following:

```
public override void PreStart()
{
}

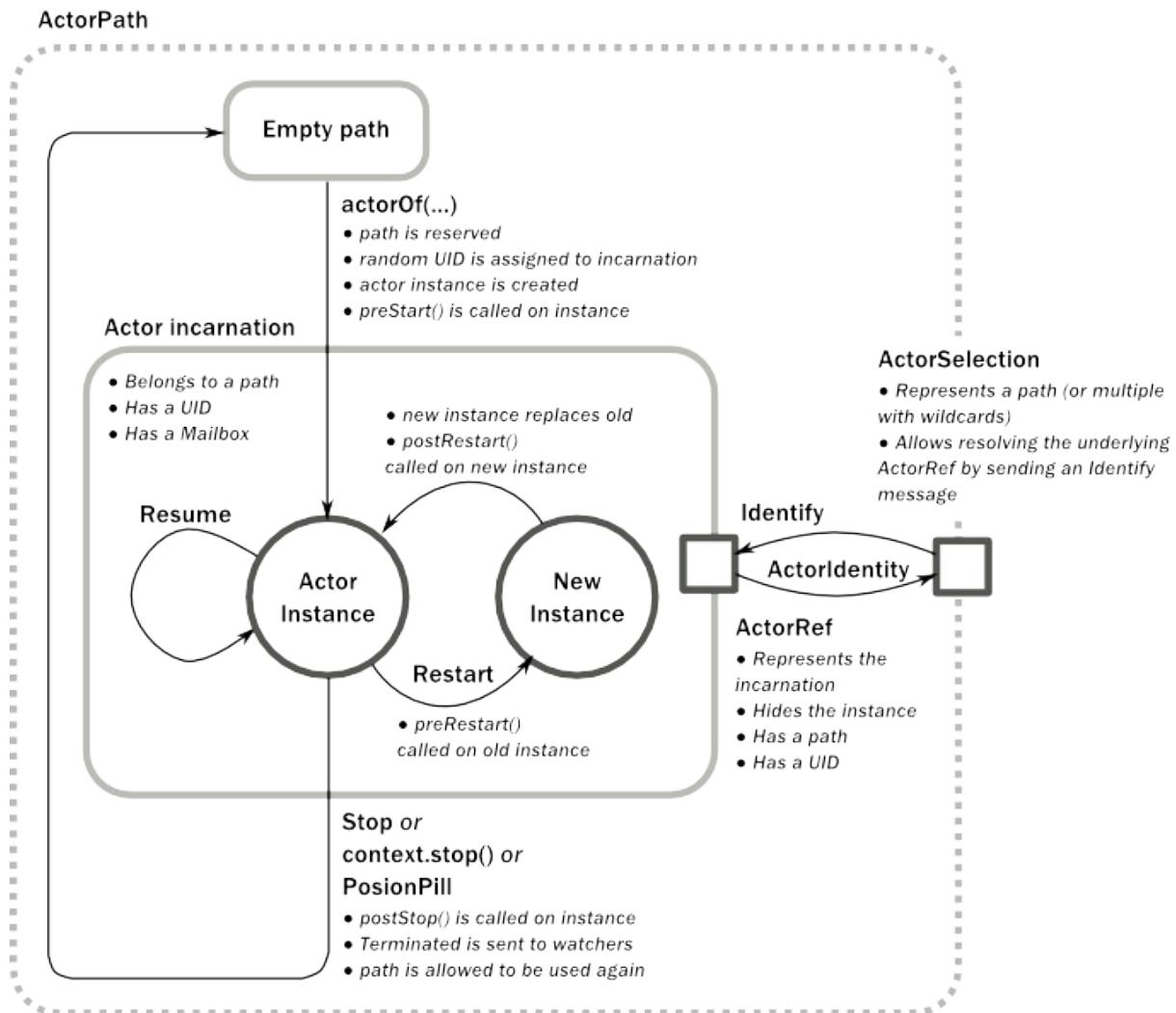
protected override void PreRestart(Exception reason, object message)
{
    foreach (IActorRef each in Context.GetChildren())
    {
        Context.Unwatch(each);
        Context.Stop(each);
    }
    PostStop();
}

protected override void PostRestart(Exception reason)
{
    PreStart();
}

protected override void PostStop()
{
}
```

The implementations shown above are the defaults provided by the `UntypedActor` class.

Actor Lifecycle



A path in an actor system represents a "place" which might be occupied by a living actor. Initially (apart from system initialized actors) a path is empty. When `Actorof()` is called it assigns an incarnation of the actor described by the passed `Props` to the given path. An actor incarnation is identified by the path and a UID. A restart only swaps the Actor instance defined by the `Props` but the incarnation and hence the UID remains the same.

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 `IActorRef` 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 `IActorRef` of the old incarnation will not point to the new one.

`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 (see [Identifying Actors via Actor Selection](#)). This can also be done with the `resolveOne` method of the `Actorselection`, which returns a `Task` of the matching `IActorRef`.

Lifecycle Monitoring aka DeathWatch

In order to be notified when another actor terminates (i.e. stops permanently, not temporary failure and restart), an actor may register itself for reception of the `Terminated` message dispatched by the other actor upon termination (see [Stopping Actors](#)). This service is provided by the DeathWatch component of the actor system.

Registering a monitor is easy (see fourth line, the rest is for demonstrating the whole functionality):

```
public class WatchActor : UntypedActor
{
    private IActorRef child = Context.ActorOf(Props.Empty, "child");
    private IActorRef lastSender = Context.System.DeadLetters;

    public WatchActor()
    {
        Context.Watch(child); // <- this is the only call needed for registration
    }

    protected override void OnReceive(object message)
    {
        switch (message)
        {
            case "kill":
                Context.Stop(child);
                lastSender = Sender;
                break;
            case Terminated t when t.ActorRef.Equals(child):
                lastSender.Tell("finished");
                break;
        }
    }
}
```

It should be noted that the `Terminated` message is generated independent of the order in which registration and termination occur. In particular, the watching actor will receive a `Terminated` message even if the watched actor has already been terminated at the time of registration.

Registering multiple times does not necessarily lead to multiple messages being generated, but there is no guarantee that only exactly one such message is received: if termination of the watched actor has generated and queued the message, and another registration is done before this message has been processed, then a second message will be queued, because registering for monitoring of an already terminated actor leads to the immediate generation of the `Terminated` message.

It is also possible to deregister from watching another actor's liveliness using `Context.Unwatch(target)`. This works even if the `Terminated` message has already been enqueued in the mailbox; after calling `unwatch` no `Terminated` message for that actor will be processed anymore.

Start Hook

Right after starting the actor, its `Prestart` method is invoked.

```
protected override void PreStart()
{
    child = Context.ActorOf(Props.Empty);
}
```

This method is called when the actor is first created. During restarts it is called by the default implementation of `PostRestart`, which means that by overriding that method you can choose whether the initialization code in this method is called only exactly once for this actor or for every restart. Initialization code which is part of the actor's

constructor will always be called when an instance of the actor class is created, which happens at every restart.

Restart Hooks

All actors are supervised, i.e. linked to another actor with a fault handling strategy. Actors may be restarted in case an exception is thrown while processing a message (see [Supervision and Monitoring](#)). This restart involves the hooks mentioned above:

- The old actor is informed by calling `PreRestart` with the exception which caused the restart and the message which triggered that exception; the latter may be `None` if the restart was not caused by processing a message, e.g. when a supervisor does not trap the exception and is restarted in turn by its supervisor, or if an actor is restarted due to a sibling's failure. If the message is available, then that message's sender is also accessible in the usual way (i.e. by calling the `sender` property). This method is the best place for cleaning up, preparing hand-over to the fresh actor instance, etc. By default it stops all children and calls `PostStop`.
- The initial factory from the `ActorOf` call is used to produce the fresh instance.
- The new actor's `PostRestart` method is invoked with the exception which caused the restart. By default the `PreStart` is called, just as in the normal start-up case.

An actor restart replaces only the actual actor object; the contents of the mailbox is unaffected by the restart, so processing of messages will resume after the `PostRestart` hook returns. The message that triggered the exception will not be received again. Any message sent to an actor while it is being restarted will be queued to its mailbox as usual.

[!WARNING] Be aware that the ordering of failure notifications relative to user messages is not deterministic. In particular, a parent might restart its child before it has processed the last messages sent by the child before the failure. See Discussion: [Message Ordering for details](#).

Stop Hook

After stopping an actor, its `PostStop` hook is called, which may be used e.g. for deregistering this actor from other services. This hook is guaranteed to run after message queuing has been disabled for this actor, i.e. messages sent to a stopped actor will be redirected to the `DeadLetters` of the `ActorSystem`.

Identifying Actors via Actor Selection

As described in [Actor References, Paths and Addresses](#), each actor has a unique logical path, which is obtained by following the chain of actors from child to parent until reaching the root of the actor system, and it has a physical path, which may differ if the supervision chain includes any remote supervisors. These paths are used by the system to look up actors, e.g. when a remote message is received and the recipient is searched, but they are also useful more directly: actors may look up other actors by specifying absolute or relative paths—logical or physical—and receive back an `ActorSelection` with the result:

```
// will look up this absolute path
Context.ActorSelection("/user/serviceA/actor");

// will look up sibling beneath same supervisor
Context.ActorSelection("../joe");
```

[!NOTE] It is always preferable to communicate with other Actors using their `IActorRef` instead of relying upon `ActorSelection`. Exceptions are: sending messages using the At-Least-Once Delivery facility, initiating first contact with a remote system. In all other cases `ActorRefs` can be provided during Actor creation or initialization, passing them from parent to child or introducing Actors by sending their `ActorRefs` to other Actors within messages.

The supplied path is parsed as a `System.Uri`, which basically means that it is split on `/` into path elements. If the path starts with `/`, it is absolute and the look-up starts at the root guardian (which is the parent of `"/user"`); otherwise it starts at the current actor. If a path element equals `...`, the look-up will take a step "up" towards the supervisor of the currently traversed actor, otherwise it will step "down" to the named child. It should be noted that the `...` in actor paths here always means the logical structure, i.e. the supervisor.

The path elements of an actor selection may contain wildcard patterns allowing for broadcasting of messages to that section:

```
// will look all children to serviceB with names starting with worker
Context.ActorSelection("/user/serviceB/worker*");

// will look up all siblings beneath same supervisor
Context.ActorSelection("../*");
```

Messages can be sent via the `ActorSelection` and the path of the `ActorSelection` is looked up when delivering each message. If the selection does not match any actors the message will be dropped.

To acquire an `IActorRef` for an `ActorSelection` you need to send a message to the selection and use the `Sender` reference of the reply from the actor. There is a built-in `Identify` message that all Actors will understand and automatically reply to with a `ActorIdentity` message containing the `IActorRef`. This message is handled specially by the actors which are traversed in the sense that if a concrete name lookup fails (i.e. a non-wildcard path element does not correspond to a live actor) then a negative result is generated. Please note that this does not mean that delivery of that reply is guaranteed, it still is a normal message.

[!code-csharpMain]

```
public class Follower : UntypedActor
{
    private string identifyId = "1";

    public Follower()
    {
        Context.ActorSelection("/user/another").Tell(new Identify(identifyId));
    }

    protected override void OnReceive(object message)
    {
        switch (message)
        {
            case ActorIdentity a when a.MessageId.Equals(identifyId) && a.Subject != null:
                Context.Watch(a.Subject);
                Context.Become(Active(a.Subject));
                break;
            case ActorIdentity a when a.MessageId.Equals(identifyId) && a.Subject == null:
                Context.Stop(Self);
                break;
        }
    }

    public UntypedReceive Active(IActorRef another)
    {
        return (message) =>
        {
            if (message is Terminated t && t.ActorRef.Equals(another))
            {
                Context.Stop(Self);
            }
        };
    }
}
```

You can also acquire an `IActorRef` for an `ActorSelection` with the `ResolveOne` method of the `ActorSelection`. It returns a `Task` of the matching `IActorRef` if such an actor exists. It is completed with failure `akka.actor.ActorNotFound` if no such actor exists or the identification didn't complete within the supplied timeout.

Remote actor addresses may also be looked up, if *remoting* is enabled:

```
Context.ActorSelection("akka.tcp://app@otherhost:1234/user/serviceB");
```

Messages and Immutability

[!IMPORTANT] Messages can be any kind of object but have to be immutable. Akka can't enforce immutability (yet) so this has to be by convention.

Here is an example of an immutable message:

```
public class ImmutableMessage
{
    public ImmutableMessage(int sequenceNumber, List<string> values)
    {
        SequenceNumber = sequenceNumber;
        Values = values.AsReadOnly();
    }

    public int SequenceNumber { get; }
    public IReadOnlyCollection<string> Values { get; }
}
```

Send messages

Messages are sent to an Actor through one of the following methods.

- `Tell()` means `fire-and-forget`, e.g. send a message asynchronously and return immediately.
- `Ask()` sends a message asynchronously and returns a `Future` representing a possible reply.

Message ordering is guaranteed on a per-sender basis.

[!NOTE] There are performance implications of using `Ask` since something needs to keep track of when it times out, there needs to be something that bridges a `Task` into an `IActorRef` and it also needs to be reachable through remoting. So always prefer `Tell` for performance, and only `Ask` if you must.

In all these methods you have the option of passing along your own `IActorRef`. Make it a practice of doing so because it will allow the receiver actors to be able to respond to your message, since the `Sender` reference is sent along with the message.

Tell: Fire-forget

This is the preferred way of sending messages. No blocking waiting for a message. This gives the best concurrency and scalability characteristics.

```
// don't forget to think about who is the sender (2nd argument)
target.Tell(message, Self);
```

The sender reference is passed along with the message and available within the receiving actor via its `Sender` property while processing this message. Inside of an actor it is usually `self` who shall be the sender, but there can be cases where replies shall be routed to some other actor—e.g. the parent—in which the second argument to `Tell` would be a different one. Outside of an actor and if no reply is needed the second argument can be `null`; if a reply is needed outside of an actor you can use the ask-pattern described next.

Ask: Send-And-Receive-Future

The ask pattern involves actors as well as Tasks, hence it is offered as a use pattern rather than a method on `ActorRef`:

```
var tasks = new List<Task>();
tasks.Add(actorA.Ask("request", TimeSpan.FromSeconds(1)));
tasks.Add(actorB.Ask("another request", TimeSpan.FromSeconds(5)));

Task.WhenAll(tasks).PipeTo(actorC, Self);
```

This example demonstrates `Ask` together with the `Pipe` Pattern on tasks, because this is likely to be a common combination. Please note that all of the above is completely non-blocking and asynchronous: `Ask` produces a `Task`, two of which are awaited until both are completed, and when that happens, a new `Result` object is forwarded to another actor.

Using `Ask` will send a message to the receiving Actor as with `Tell`, and the receiving actor must reply with `Sender.Tell(reply, Self)` in order to complete the returned `Task` with a value. The `Ask` operation involves creating an internal actor for handling this reply, which needs to have a timeout after which it is destroyed in order not to leak resources; see more below.

[!WARNING] To complete the `Task` with an exception you need send a `Failure` message to the sender. This is not done automatically when an actor throws an exception while processing a message.

```
try
{
    var result = operation();
    Sender.Tell(result, Self);
}
catch (Exception e)
{
    Sender.Tell(new Failure { Exception = e }, Self);
}
```

If the actor does not complete the task, it will expire after the timeout period, specified as parameter to the `Ask` method, and the `Task` will be cancelled and throw a `TaskCancelledException`.

For more information on Tasks, check out the [MSDN documentation.aspx](#)).

[!WARNING] When using task callbacks inside actors, you need to carefully avoid closing over the containing actor's reference, i.e. do not call methods or access mutable state on the enclosing actor from within the callback. This would break the actor encapsulation and may introduce synchronization bugs and race conditions because the callback will be scheduled concurrently to the enclosing actor. Unfortunately there is not yet a way to detect these illegal accesses at compile time.

Forward message

You can forward a message from one actor to another. This means that the original sender address/reference is maintained even though the message is going through a 'mediator'. This can be useful when writing actors that work as routers, load-balancers, replicators etc. You need to pass along your context variable as well.

```
target.Forward(result, Context);
```

Receive messages

When an actor receives a message it is passed into the `onReceive` method, this is an abstract method on the `UntypedActor` base class that needs to be defined.

Here is an example:

```
public class MyActor : UntypedActor
{
    private ILoggingAdapter log = Context.GetLogger();

    protected override void OnReceive(object message)
    {
        switch (message)
        {
            case "test":
                log.Info("received test");
                break;
            default:
                log.Info("received unknown message");
                break;
        }
    }
}
```

Reply to messages

If you want to have a handle for replying to a message, you can use `Sender`, which gives you an `IActorRef`. You can reply by sending to that `IActorRef` with `Sender.Tell(replyMsg, Self)`. You can also store the `IActorRef` for replying later, or passing on to other actors. If there is no sender (a message was sent without an actor or task context) then the sender defaults to a 'dead-letter' actor ref.

```
protected override void OnReceive(object message)
{
    var result = calculateResult();

    // do not forget the second argument!
    Sender.Tell(result, Self);
}
```

Receive timeout

The `IActorContext` `SetReceiveTimeout` defines the inactivity timeout after which the sending of a `ReceiveTimeout` message is triggered. When specified, the receive function should be able to handle an `Akka.Actor.ReceiveTimeout` message.

[!NOTE] Please note that the receive timeout might fire and enqueue the `ReceiveTimeout` message right after another message was enqueued; hence it is not guaranteed that upon reception of the receive timeout there must have been an idle period beforehand as configured via this method.

Once set, the receive timeout stays in effect (i.e. continues firing repeatedly after inactivity periods). Pass in `null` to `SetReceiveTimeout` to switch off this feature.

```
public class MyActor : UntypedActor
{
    private ILoggingAdapter log = Context.GetLogger();

    protected override void OnReceive(object message)
    {
        switch (message)
        {
            case "Hello":
                Context.SetReceiveTimeout(TimeSpan.FromMilliseconds(100));
                break;
            case ReceiveTimeout r:
                Context.SetReceiveTimeout(null);
                throw new Exception("Receive timed out");
        }
    }
}
```

Stopping actors

Actors are stopped by invoking the `Stop` method of a `ActorRefFactory`, i.e. `ActorContext` or `ActorSystem`. Typically the context is used for stopping child actors and the system for stopping top level actors. The actual termination of the actor is performed asynchronously, i.e. `stop` may return before the actor is stopped.

```
public class MyStoppingActor : UntypedActor
{
    private IActorRef child;

    protected override void OnReceive(object message)
    {
        switch (message)
        {
            case "interrupt-child":
                Context.Stop(child);
                break;
            case "done":
                Context.Stop(Self);
                break;
        }
    }
}
```

Processing of the current message, if any, will continue before the actor is stopped, but additional messages in the mailbox will not be processed. By default these messages are sent to the `DeadLetters` of the `ActorSystem`, but that depends on the mailbox implementation.

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`, 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.

The `PostStop` hook is invoked after an actor is fully stopped. This enables cleaning up of resources:

```
protected override void PostStop()
```

```
{
    // clean up resources here ...
}
```

[!NOTE] Since stopping an actor is asynchronous, you cannot immediately reuse the name of the child you just stopped; this will result in an `InvalidActorNameException`. Instead, watch the terminating actor and create its replacement in response to the `Terminated` message which will eventually arrive.

PoisonPill

You can also send an actor the `Akka.Actor.PoisonPill` message, which will stop the actor when the message is processed. `PoisonPill` is enqueued as ordinary messages and will be handled after messages that were already queued in the mailbox.

Use it like this:

```
myActor.Tell(PoisonPill.Instance, Sender);
```

Graceful Stop

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

```
var manager = system.ActorOf<Manager>();

try
{
    await manager.GracefulStop(TimeSpan.FromMilliseconds(5), "shutdown");
    // the actor has been stopped
}
catch (TaskCanceledException)
{
    // the actor wasn't stopped within 5 seconds
}

...
public class Manager : UntypedActor
{
    private IActorRef worker = Context.Watch(Context.ActorOf<Cruncher>("worker"));

    protected override void OnReceive(object message)
    {
        switch (message)
        {
            case "job":
                worker.Tell("crunch");
                break;
            case Shutdown s:
                worker.Tell(PoisonPill.Instance, Self);
                Context.Become(ShuttingDown);
                break;
        }
    }

    private void ShuttingDown(object message)
    {
        switch (message)
        {
            case "job":
                Sender.Tell("service unavailable, shutting down", Self);
                break;
        }
    }
}
```

```
        case Terminated t:
            Context.Stop(Self);
            break;
    }
}
```

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()`.

In the above example a "shutdown" message is sent to the target actor to initiate the process of stopping the actor. You can use `PoisonPill` for this, but then you have limited possibilities to perform interactions with other actors before stopping the target actor. Simple cleanup tasks can be handled in `PostStop`.

[!WARNING] Keep in mind that an actor stopping and its name being deregistered are separate events which happen asynchronously from each other. Therefore it may be that you will find the name still in use after `GracefulStop()` returned. In order to guarantee proper deregistration, only reuse names from within a supervisor you control and only in response to a `Terminated` message, i.e. not for top-level actors.

Become/Unbecome

Upgrade

Akka supports hotswapping the Actor's message loop (e.g. its implementation) at runtime. Use the `context.Become` method from within the Actor. The hotswapped code is kept in a `Stack` which can be pushed (replacing or adding at the top) and popped.

[!WARNING] Please note that the actor will revert to its original behavior when restarted by its Supervisor.

To hotswap the Actor using `Context.Become`:

```
public class HotSwapActor : UntypedActor
{
    protected override void OnReceive(object message)
    {
        switch (message)
        {
            case "foo":
                Become(Angry);
                break;
            case "bar":
                Become(Happy);
                break;
        }
    }

    private void Angry(object message)
    {
        switch (message)
        {
            case "foo":
                Sender.Tell("I am already angry?");
                break;
            case "bar":
                Become(Angry);
                break;
        }
    }

    private void Happy(object message)
    {
```

```

        switch (message)
    {
        case "foo":
            Sender.Tell("I am already happy :-)");
            break;
        case "bar":
            Become(Angry);
            break;
    }
}
}

```

This variant of the `Become` method is useful for many different things, such as to implement a Finite State Machine (FSM). It will replace the current behavior (i.e. the top of the behavior stack), which means that you do not use `Unbecome`, instead always the next behavior is explicitly installed.

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"4" 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).

```

public class Swapper : UntypedActor
{
    public class Swap
    {
        public static Swap Instance = new Swap();
        private Swap() { }
    }

    private ILoggingAdapter log = Context.GetLogger();

    protected override void OnReceive(object message)
    {
        switch (message)
        {
            case Swap s:
                log.Info("Hi");

                BecomeStacked((msg) =>
                {
                    if (msg is Swap)
                    {
                        log.Info("Ho");
                        UnbecomeStacked();
                    }
                });
                break;
        }
    }
}

static void Main(string[] args)
{
    var system = ActorSystem.Create("MySystem");
    var swapper = system.ActorOf<Swapper>();

    swapper.Tell(Swapper.Swap.Instance);
    swapper.Tell(Swapper.Swap.Instance);
    swapper.Tell(Swapper.Swap.Instance);
    swapper.Tell(Swapper.Swap.Instance);
    swapper.Tell(Swapper.Swap.Instance);
    swapper.Tell(Swapper.Swap.Instance);

    Console.ReadLine();
}

```

Stash

The `IWithUnboundedStash` interface enables an actor to temporarily stash away messages that can not or should not be handled using the actor's current behavior. Upon changing the actor's message handler, i.e., right before invoking `Context.BecomeStacked()` or `Context.UnbecomeStacked()`, all stashed messages can be "unstashed", thereby prepending them to the actor's mailbox. This way, the stashed messages can be processed in the same order as they have been received originally. An actor that implements `IWithUnboundedStash` will automatically get a deque-based mailbox.

Here is an example of the `IWithUnboundedStash` interface in action:

```
public class ActorWithProtocol : UntypedActor, IWithUnboundedStash
{
    public IStash Stash { get; set; }

    protected override void OnReceive(object message)
    {
        switch (message)
        {
            case "open":
                Stash.UnstashAll();
                BecomeStacked(msg =>
                {
                    switch (msg)
                    {
                        case "write":
                            // do writing...
                            break;
                        case "close":
                            Stash.UnstashAll();
                            Context.UnbecomeStacked();
                            break;
                        default:
                            Stash.Stash();
                            break;
                    }
                });
                break;
            default:
                Stash.Stash();
                break;
        }
    }
}
```

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()`.

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 `IWithUnboundedStash` interface implementation of `PreRestart` will call `UnstashAll()`, which is usually the desired behavior.

Killing an Actor

You can kill an actor by sending a `kill` message. This will cause the actor to throw a `ActorKilledException`, triggering a failure. The actor will suspend operation and its supervisor will be asked how to handle the failure, which may mean resuming the actor, restarting it or terminating it completely. See [What Supervision Means](#) for more information.

Use `kill` like this:

```
// kill the 'victim' actor
victim.Tell(Akka.Actor.Kill.Instance, ActorRef.NoSender);
```

Actors and exceptions

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 and Monitoring](#)). 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.

Initialization patterns

The rich lifecycle hooks of `Actors` provide a useful toolkit to implement various initialization patterns. During the lifetime of an `IActorRef`, an actor can potentially go through several restarts, where the old instance is replaced by a fresh one, invisibly to the outside observer who only sees the `IActorRef`.

One may think about the new instances as "incarnations". Initialization might be necessary for every incarnation of an actor, but sometimes one needs initialization to happen only at the birth of the first instance when the `IActorRef` is created. The following sections provide patterns for different initialization needs.

Initialization via constructor

Using the constructor for initialization has various benefits. First of all, it makes it possible to use readonly fields to store any state that does not change during the life of the actor instance, making the implementation of the actor more robust. The constructor is invoked for every incarnation of the actor, therefore the internals of the actor can always

assume that proper initialization happened. This is also the drawback of this approach, as there are cases when one would like to avoid reinitializing internals on restart. For example, it is often useful to preserve child actors across restarts. The following section provides a pattern for this case.

Initialization via PreStart

The method `PreStart()` of an actor is only called once directly during the initialization of the first instance, that is, at creation of its `ActorRef`. In the case of restarts, `PreStart()` is called from `PostRestart()`, therefore if not overridden, `PreStart()` is called on every incarnation. However, overriding `PostRestart()` one can disable this behavior, and ensure that there is only one call to `PreStart()`.

One useful usage of this pattern is to disable creation of new `ActorRefs` for children during restarts. This can be achieved by overriding `PreRestart()`:

```
protected override void PreStart()
{
    // Initialize children here
}

// Overriding postRestart to disable the call to preStart() after restarts
protected override void PostRestart(Exception reason)
{
}

// The default implementation of PreRestart() stops all the children
// of the actor. To opt-out from stopping the children, we
// have to override PreRestart()
protected override void PreRestart(Exception reason, object message)
{
    // Keep the call to PostStop(), but no stopping of children
    PostStop();
}
```

Please note, that the child actors are *still restarted*, but no new `IActorRef` is created. One can recursively apply the same principles for the children, ensuring that their `PreStart()` method is called only at the creation of their refs.

For more information see [What Restarting Means](#).

Initialization via message passing

There are cases when it is impossible to pass all the information needed for actor initialization in the constructor, for example in the presence of circular dependencies. In this case the actor should listen for an initialization message, and use `Become()` or a finite state-machine state transition to encode the initialized and uninitialized states of the actor.

```
public class Service : UntypedActor
{
    private string _initializeMe;

    protected override void OnReceive(object message)
    {
        switch (message)
        {
            case "init":
                _initializeMe = "Up and running";
                Become(m =>
                {
                    if (m is "U OK?" && _initializeMe != null)
                    {
                        Sender.Tell(_initializeMe, Self);
                    }
                });
        }
    }
}
```

```
        }
    });
    break;
}
}
```

If the actor may receive messages before it has been initialized, a useful tool can be the `stash` to save messages until the initialization finishes, and replaying them after the actor became initialized.

[!WARNING] This pattern should be used with care, and applied only when none of the patterns above are applicable. One of the potential issues is that messages might be lost when sent to remote actors. Also, publishing an `IActorRef` in an uninitialized state might lead to the condition that it receives a user message before the initialization has been done.

Routers

A *router* is a special type of actor whose job is to route messages to other actors called *routees*. Different routers use different *strategies* to route messages efficiently.

Routers can be used inside or outside of an actor, and you can manage the routees yourself or use a self contained router actor with configuration capabilities, and can also [resize dynamically](#) under load.

Akka.NET comes with several useful routers you can choose right out of the box, according to your application's needs. But it is also possible to create your own.

[!NOTE] In general, any message sent to a router will be forwarded to one of its routees, but there is one exception. The special [Broadcast Message](#) will be sent to all routees. See [Specially Handled Messages](#) section for details.

Deployment

Routers can be deployed in multiple ways, using code or configuration.

Code deployment

The example below shows how to deploy 5 workers using a round robin router:

```
var props = Props.Create<Worker>().WithRouter(new RoundRobinPool(5));
var actor = system.ActorOf(props, "worker");
```

It's important to understand that although you create the Props and add the router later, the deployment happens in reverse order, with the workers being added to the router. The above code can also be written as:

```
var props = new RoundRobinPool(5).Props(Props.Create<Worker>());
```

Configuration deployment

The same router may be defined using a [HOCON deployment configuration](#).

In order to do that, define a HOCON section with the path of the actor, and create the actor in that path using `FromConfig.Instance`.

```
akka.actor.deployment {
  /workers {
    router = round-robin-pool
    nr-of-instances = 5
  }
}
```

```
var props = Props.Create<Worker>().WithRouter(FromConfig.Instance);
var actor = system.ActorOf(props, "workers");
```

For router groups, the same idea applies with slightly different syntax:

```
akka.actor.deployment {
  /workers {
```

```

    router = round-robin-group
    routees.paths = ["/user/workers/w1", "/user/workers/w2", "/user/workers/w3"]
}
}

```

```

var props = Props.Create<Worker>().WithRouter(FromConfig.Instance);
var actor = system.ActorOf(props, "workers");

```

As you can see above, the advantage of using HOCON for routers is that you can change the deployment of routers easily without recompiling the code.

Pools vs. Groups

There are two types of routers:

- **Pools**

Router "Pools" are routers that create their own worker actors, that is; you provide the *number of instances* as a parameter to the router and the router will handle routee creation by itself.

- **Groups**

Sometimes, rather than having the router actor create its routees, it is desirable to create routees yourself and provide them to the router for its use. You can do this by passing the paths of the routees to the router's configuration. Messages will be sent with `ActorSelection` to these paths.

[!NOTE] Most routing strategies listed below are available in both types. Some of them may be available only in one type due to implementation requirements.

Supervision

Routers are implemented as actors, so a router is supervised by its parent, and they may supervise children.

Group routers use routees created somewhere else, it doesn't have children of its own. If a routee dies, a group router will have no knowledge of it.

Pool routers on the other hand create their own children. The router is therefore also the routee's supervisor.

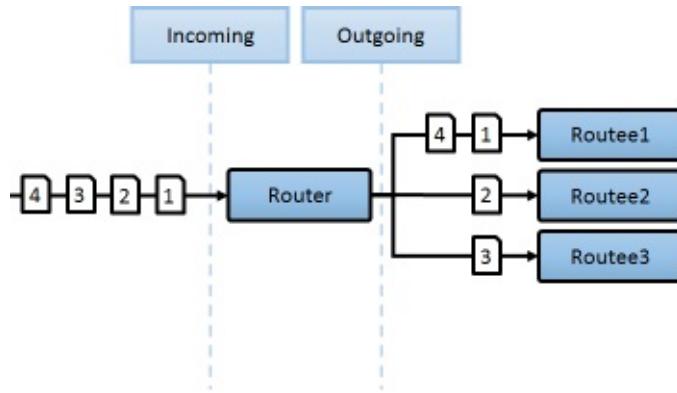
By default, pool routers use a custom strategy that only returns `Escalate` for all exceptions, the router supervising the failing worker will then escalate to its own parent, if the parent of the router decides to restart the router, all the pool workers will also be recreated as a result of this.

Routing Strategies

These are the routing strategies provided by Akka.NET out of the box.

RoundRobin

`RoundRobinPool` and `RoundRobinGroup` are routers that sends messages to routees in *round-robin* order. It's the simplest way to distribute messages to multiple worker actors, on a best-effort basis.



Usage:

RoundRobinPool defined in configuration:

```

akka.actor.deployment {
  /some-pool {
    router = round-robin-pool
    nr-of-instances = 5
  }
}

```

```
var router = system.ActorOf(Props.Create<Worker>().WithRouter(FromConfig.Instance), "some-pool");
```

RoundRobinPool defined in code:

```
var router = system.ActorOf(Props.Create<Worker>().WithRouter(new RoundRobinPool(5)), "some-pool");
```

RoundRobinGroup defined in configuration:

```

akka.actor.deployment {
  /some-group {
    router = round-robin-group
    routees.paths = ["/user/workers/w1", "/user/workers/w2", "/user/workers/w3"]
  }
}

```

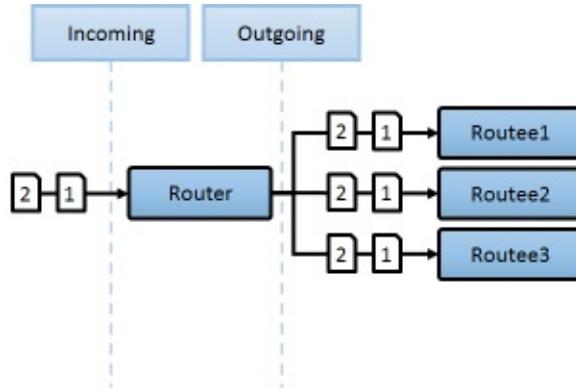
```
var router = system.ActorOf(Props.Empty.WithRouter(FromConfig.Instance), "some-group");
```

RoundRobinGroup defined in code:

```
var workers = new [] { "/user/workers/w1", "/user/workers/w3", "/user/workers/w3" }
var router = system.ActorOf(Props.Empty.WithRouter(new RoundRobinGroup(workers)), "some-group");
```

Broadcast

The `BroadcastPool` and `BroadcastGroup` routers will, as the name implies, broadcast any message to all of its routees.

**Usage:**

BroadcastPool defined in configuration:

```
akka.actor.deployment {
  /some-pool {
    router = broadcast-pool
    nr-of-instances = 5
  }
}
```

```
var router = system.ActorOf(Props.Create<Worker>().WithRouter(FromConfig.Instance), "some-pool");
```

BroadcastPool defined in code:

```
var router = system.ActorOf(Props.Create<Worker>().WithRouter(new BroadcastPool(5)), "some-pool");
```

BroadcastGroup defined in configuration:

```
akka.actor.deployment {
  /some-group {
    router = broadcast-group
    routees.paths = ["/user/a1", "/user/a2", "/user/a3"]
  }
}
```

```
var router = system.ActorOf(Props.Empty.WithRouter(FromConfig.Instance), "some-group");
```

BroadcastGroup defined in code:

```
var actors = new [] { "/user/a1", "/user/a2", "/user/a3" }
var router = system.ActorOf(Props.Empty.WithRouter(new BroadcastGroup(actors)), "some-group");
```

Random

The `RandomPool` and `RandomGroup` routers will forward messages to routees in random order.

Usage:

RandomPool defined in configuration:

```
akka.actor.deployment {
  /some-pool {
    router = random-pool
    nr-of-instances = 5
  }
}
```

```
var router = system.ActorOf(Props.Create<Worker>().WithRouter(FromConfig.Instance), "some-pool");
```

RandomPool defined in code:

```
var router = system.ActorOf(Props.Create<Worker>().WithRouter(new RandomPool(5)), "some-pool");
```

RandomGroup defined in configuration:

```
akka.actor.deployment {
  /some-group {
    router = random-group
    routees.paths = ["/user/workers/w1", "/user/workers/w2", "/user/workers/w3"]
  }
}
```

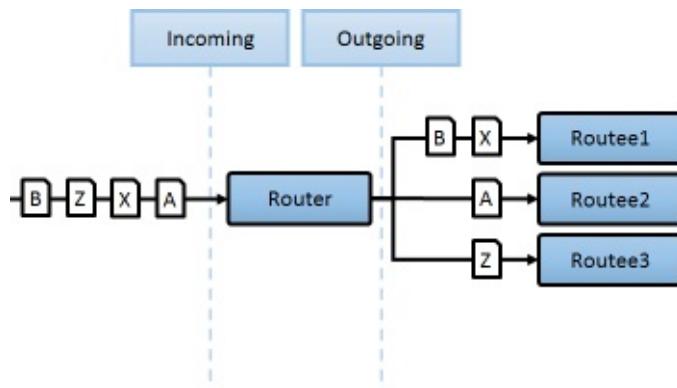
```
var router = system.ActorOf(Props.Empty.WithRouter(FromConfig.Instance), "some-group");
```

RandomGroup defined in code:

```
var workers = new [] { "/user/workers/w1", "/user/workers/w3", "/user/workers/w3" }
var router = system.ActorOf(Props.Empty.WithRouter(new RandomGroup(workers)), "some-group");
```

ConsistentHashing

The `ConsistentHashingPool` and `ConsistentHashingGroup` are routers that use a [consistent hashing algorithm](#) to select a routee to forward the message. The idea is that messages with the same key are forwarded to the same routee. Any .NET object can be used as a key, although it's usually a number, string or Guid.



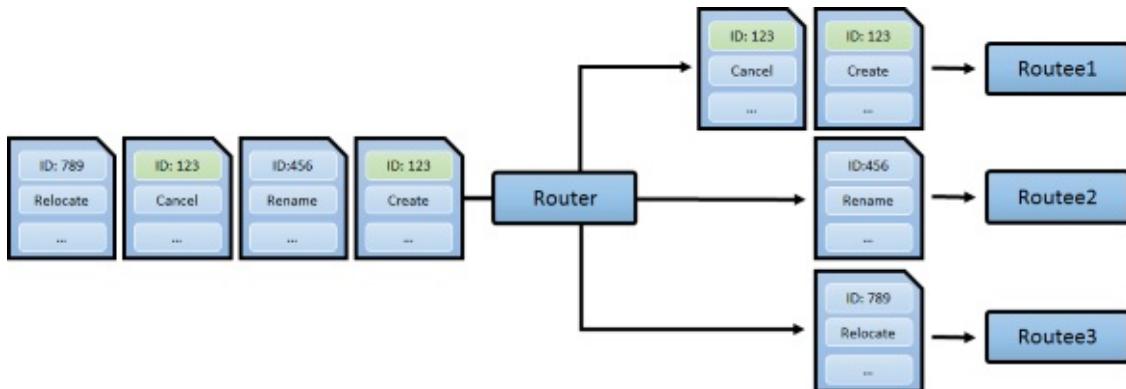
`ConsistentHash` can be very useful when dealing with **Commands** in the sense of [CQRS](#) or [\[Domain Driven Design\]](#).

For example, let's assume we have the following incoming sequence of "**Customer Commands**":



In this case we might want to group all messages based on "**Customer ID**" (ID in the diagram).

By using a `ConsistentHash` router we can now process multiple commands in parallel for different Customers, while still processing messages for each specific Customer in ordered sequence, and thus preventing us from getting race conditions with ourselves when applying each command on each customer entity.



There are 3 ways to define what data to use for the consistent hash key.

1. You can define a *hash mapping delegate* using the `WithHashMapper` method of the router to map incoming messages to their consistent hash key. This makes the decision transparent for the sender.

```
new ConsistentHashingPool(5).WithHashMapping(o =>
{
    if (o is IHasCustomKey)
        return ((IHasCustomKey)o).Key;

    return null;
});
```

2. The messages may implement `IConsistentHashable`. The key is part of the message and it's convenient to define it together with the message definition.

```
public class SomeMessage : IConsistentHashable
{
    public Guid GroupID { get; private set; }
    public object ConsistentHashKey { get { return GroupID; } }
}
```

3. The messages can be wrapped in a `ConsistentHashableEnvelope` to define what data to use for the consistent hash key. The sender knows the key to use.

```
public class SomeMessage
{
    public Guid GroupID { get; set; }

    var originalMsg = new SomeMessage { GroupID = Guid.NewGuid(); };
    var msg = new ConsistentHashableEnvelope(originalMsg, originalMsg.GroupID);
```

You may implement more than one hashing mechanism at the same time. Akka.NET will try them in the order above. That is, if the HashMapping method returns null, Akka.NET will check for the IConsistentHashable interface in the message (2 and 3 are technically the same).

Usage:

ConsistentHashingPool defined in configuration:

```
akka.actor.deployment {
  /some-pool {
    router = consistent-hashing-pool
    nr-of-instances = 5
    virtual-nodes-factor = 10
  }
}

var router = system.ActorOf(Props.Create<Worker>().WithRouter(FromConfig.Instance), "some-pool");
```

ConsistentHashingPool defined in code:

```
var router = system.ActorOf(Props.Create<Worker>().WithRouter(new ConsistentHashingPool(5)), "some-pool");
```

ConsistentHashingGroup defined in configuration:

```
akka.actor.deployment {
  /some-group {
    router = consistent-hashing-group
    routees.paths = ["/user/workers/w1", "/user/workers/w2", "/user/workers/w3"]
    virtual-nodes-factor = 10
  }
}

var router = system.ActorOf(Props.Empty.WithRouter(FromConfig.Instance), "some-group");
```

ConsistentHashingGroup defined in code:

```
var workers = new [] { "/user/workers/w1", "/user/workers/w3", "/user/workers/w3" }
var router = system.ActorOf(Props.Empty.WithRouter(new ConsistentHashingGroup(workers)), "some-group");
```

[!NOTE]

1. `virtual-nodes-factor` is the number of virtual nodes per routee that is used in the consistent hash node ring - if not defined, the default value is 10 and you shouldn't need to change it unless you understand how the algorithm works and know what you are doing.
2. It is possible to define this value in code using the `WithVirtualFactor(...)` method of the ConsistentHashingPool/Group object.

TailChopping

The `TailChoppingPool` and `TailChoppingGroup` routers send the message to a random routee, and if no response is received after a specified delay, send it to another randomly selected routee. It waits for the first reply from any of the routees, and forwards it back to the original sender. Other replies are discarded. If no reply is received after a specified interval, a timeout Failure is generated.

The goal of this router is to decrease latency by performing redundant queries to multiple routees, assuming that one of the other actors may still be faster to respond than the initial one.

This optimization was described nicely in a blog post by Peter Bailis: [Doing redundant work to speed up distributed queries](#).

Usage:

TailChoppingPool defined in configuration:

```
akka.actor.deployment {
  /some-pool {
    router = tail-chopping-pool
    nr-of-instances = 5
    within = 10s
    tail-chopping-router.interval = 20ms
  }
}
```

```
var router = system.ActorOf(Props.Create<Worker>().WithRouter(FromConfig.Instance), "some-pool");
```

TailChoppingPool defined in code:

```
var within = TimeSpan.FromSeconds(10);
var interval = TimeSpan.FromMilliseconds(20);
var router = system.ActorOf(Props.Create<Worker>().WithRouter(new TailChoppingPool(5, within, interval)), "some-pool");
```

TailChoppingGroup defined in configuration:

```
akka.actor.deployment {
  /some-group {
    router = tail-chopping-group
    routees.paths = ["/user/workers/w1", "/user/workers/w2", "/user/workers/w3"]
    within = 10s
    tail-chopping-router.interval = 20ms
  }
}
```

```
var router = system.ActorOf(Props.Empty.WithRouter(FromConfig.Instance), "some-group");
```

TailChoppingGroup defined in code:

```
var workers = new [] { "/user/workers/w1", "/user/workers/w3", "/user/workers/w3" }
var within = TimeSpan.FromSeconds(10);
var interval = TimeSpan.FromMilliseconds(20);
var router = system.ActorOf(Props.Empty.WithRouter(new TailChoppingGroup(workers, within, interval)), "some-group");
```

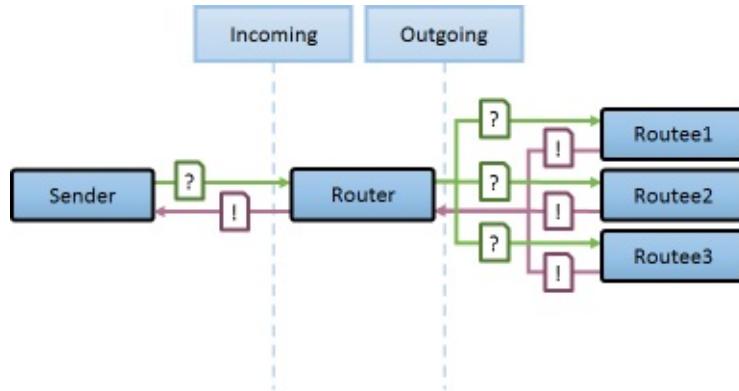
[!NOTE]

1. `within` is the time to wait for a reply from any routee before timing out
2. `tail-chopping-router.interval` is the interval between requests to the other routees

ScatterGatherFirstCompleted

The `ScatterGatherFirstCompletedPool` and `ScatterGatherFirstCompletedRouter` routers will broadcast the message to all routees and reply back to the original sender with the first reply it receives. All other replies are discarded. If no reply is received after a specified interval, a timeout Failure is generated.

This is useful in scenarios where you can accept any reply to a query and doesn't care who answers (eg. you may query multiple servers in a cluster just to know if any of them is online - if one answers, you may not care about the others).



Usage:

`ScatterGatherFirstCompletedPool` defined in configuration:

```

akka.actor.deployment {
  /some-pool {
    router = scatter-gather-pool
    nr-of-instances = 5
    within = 10s
  }
}
  
```

```

var router = system.ActorOf(Props.Create<Worker>().WithRouter(FromConfig.Instance), "some-pool");
  
```

`ScatterGatherFirstCompletedPool` defined in code:

```

var within = TimeSpan.FromSeconds(10);
var router = system.ActorOf(Props.Create<Worker>().WithRouter(new ScatterGatherFirstCompletedPool(5, within)),
  "some-pool");
  
```

`ScatterGatherFirstCompletedPool` defined in configuration:

```

akka.actor.deployment {
  /some-group {
    router = scatter-gather-group
    routees.paths = ["/user/workers/w1", "/user/workers/w2", "/user/workers/w3"]
    within = 10s
  }
}
  
```

```

var router = system.ActorOf(Props.Empty.WithRouter(FromConfig.Instance), "some-group");
  
```

`ScatterGatherFirstCompletedGroup` defined in code:

```

var workers = new [] { "/user/workers/w1", "/user/workers/w3", "/user/workers/w3" }
  
```

```
var within = TimeSpan.FromSeconds(10);
var router = system.ActorOf(Props.Empty.WithRouter(new ScatterGatherFirstCompletedGroup(workers, within)), "some-group");
```

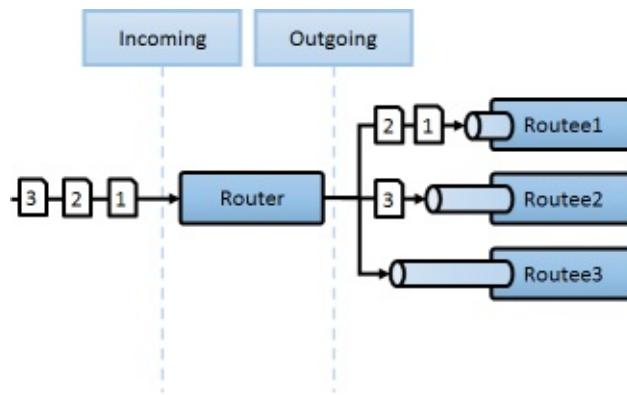
[!NOTE]

1. `within` is the time to wait for a reply from any routee before timing out

SmallestMailbox

The `SmallestMailboxPool` router will send the message to the routee with fewest messages in mailbox. The selection is done in this order:

1. Pick any idle routee (not processing message) with empty mailbox
2. Pick any routee with empty mailbox
3. Pick routee with fewest pending messages in mailbox
4. Pick any remote routee, remote actors are consider lowest priority, since their mailbox size is unknown



Usage:

SmallestMailboxPool defined in configuration:

```
akka.actor.deployment {
  /some-pool {
    router = smallest-mailbox-pool
    nr-of-instances = 5
  }
}
```

```
var router = system.ActorOf(Props.Create<Worker>().WithRouter(FromConfig.Instance), "some-pool");
```

SmallestMailboxPool defined in code:

```
var router = system.ActorOf(Props.Create<Worker>().WithRouter(new SmallestMailboxPool(5)), "some-pool");
```

Dynamically Resizable Pools

Routers pools can be dynamically resized to adjust the responsiveness of the system under load.

This is done by adding a `resizer` section to your router configuration:

```
akka.actor.deployment {
```

```
/my-router {
    router = round-robin-pool
    resizer {
        enabled = on
        lower-bound = 1
        upper-bound = 10
    }
}
```

You can also set a resizer in code when creating a router.

```
new RoundRobinPool(5, new DefaultResizer(1, 10))
```

These are settings you usually change in the resizer:

- `enabled` - Turns on or off the resizer. The default is `off`.
- `lower-bound` - The minimum number of routees that should remain active. The default is `1`.
- `upper-bound` - The maximum number of routees that should be created. The default is `10`.

The default resizer works by checking the pool size every X messages, and deciding to increase or decrease the pool accordingly. The following settings are used to fine-tune the resizer and are considered *good enough* for most cases, but can be changed if needed:

- `messages-per-resize` - The # of messages to route before checking if resize is needed. The default is `10`.
- `rampup-rate` - Percentage to increase the pool size. The default is `0.2`, meaning it will increase the pool size in 20% when resizing.
- `backoff-rate` - Percentage to decrease the pool size. The default is `0.1`, meaning it will decrease the pool size in 10% when resizing.
- `pressure-threshold` - A threshold used to decide if the pool should be increased. The default is `1`, meaning it will decide to increase the pool if all routees are busy and have at least 1 message in the mailbox.
 - `0` - the routee is busy and have no messages in the mailbox
 - `1` - the routee is busy and have at least 1 message waiting in the mailbox
 - `N` - the routee is busy and have N messages waiting in the mailbox (where $N > 1$)
- `backoff-threshold` - A threshold used to decide if the pool should be decreased. The default is `0.3`, meaning it will decide to decrease the pool if less than 30% of the routers are busy.

Specially Handled Messages

Most messages sent to router will be forwarded according to router's routing logic. However there are a few types of messages that have special behaviour.

Broadcast Messages

A `Broadcast` message can be used to send message to **all** routees of a router. When a router receives `Broadcast` message, it will broadcast that message's **payload** to all routees, no matter how that router normally handles its messages.

Here is an example of how to send a message to every routee of a router.

```
actorSystem.ActorOf(Props.Create<Worker>(), "worker1");
actorSystem.ActorOf(Props.Create<Worker>(), "worker2");
actorSystem.ActorOf(Props.Create<Worker>(), "worker3");

var workers = new[] { "/user/worker1", "/user/worker2", "/user/worker3" };
var router = actorSystem.ActorOf(Props.Empty.WithRouter(new RoundRobinGroup(workers)), "workers");
```

```
// this sends to individual worker
router.Tell("Hello, worker1");
router.Tell("Hello, worker2");
router.Tell("Hello, worker3");

// this sends to all workers
router.Tell(new Broadcast("Hello, workers"));
```

In this example, the router received the `Broadcast` message, extracted its payload (`Hello, workers`), and then dispatched it to all its routees. It is up to each routee actor to handle the payload.

PoisonPill Messages

When an actor received `PoisonPill` message, that actor will be stopped. (see [PoisonPill](#) for details).

For a router, which normally passes on messages to routees, the `PoisonPill` messages are processed **by the router only**. `PoisonPill` messages sent to a router will **not** be sent on to its routees.

However, a `PoisonPill` message sent to a router may still affect its routees, as it will stop the router which in turns stop children the router has created. Each child will process its current message and then stop. This could lead to some messages being unprocessed.

If you wish to stop a router and its routees, but you would like the routees to first process all the messages in their mailboxes, then you should send a `PoisonPill` message wrapped inside a `Broadcast` message so that each routee will receive the `PoisonPill` message.

[!NOTE] The above method will stop all routees, even if they are not created by the router. E.g. routees programatically provided to the router.

```
actorSystem.ActorOf(Props.Create<Worker>(), "worker1");
actorSystem.ActorOf(Props.Create<Worker>(), "worker2");
actorSystem.ActorOf(Props.Create<Worker>(), "worker3");

var workers = new[] { "/user/worker1", "/user/worker2", "/user/worker3" };
var router = actorSystem.ActorOf(Props.Empty.WithRouter(new RoundRobinGroup(workers)), "workers");

router.Tell("Hello, worker1");
router.Tell("Hello, worker2");
router.Tell("Hello, worker3");

// this sends PoisonPill message to all routees
router.Tell(new Broadcast(PoisonPill.Instance));
```

With the code shown above, each routee will receive a `PoisonPill` message. Each routee will continue to process its messages as normal, eventually processing the `PoisonPill`. This will cause the routee to stop. After all routees have stopped the router will itself be stopped automatically unless it is a dynamic router, e.g. using a resizer.

Kill Message

As with the `PoisonPill` messasge, there is a distinction between killing a router, which indirectly kills its children (who happen to be routees), and killing routees directly (some of whom may not be children.) To kill routees directly the router should be sent a Kill message wrapped in a `Broadcast` message.

See [Noisy on Purpose: Kill the Actor](#) for more details on how `Kill` message works.

Advanced

How Routing is Designed within Akka.NET

On the surface routers look like normal actors, but they are actually implemented differently. Routers are designed to be extremely efficient at receiving messages and passing them quickly on to routees.

A normal actor can be used for routing messages, but an actor's single-threaded processing can become a bottleneck. Routers can achieve much higher throughput with an optimization to the usual message-processing pipeline that allows concurrent routing. This is achieved by embedding routers' routing logic directly in their ActorRef rather than in the router actor. Messages sent to a router's ActorRef can be immediately routed to the routee, bypassing the single-threaded router actor entirely.

The cost to this is, of course, that the internals of routing code are more complicated than if routers were implemented with normal actors. Fortunately all of this complexity is invisible to consumers of the routing API. However, it is something to be aware of when implementing your own routers.

Router Logic

All routers implemented through *routing logic* classes (eg. RoundRobinRoutingLogic, TailChoppingRoutingLogic, etc). Pools and groups are implemented on top of these classes.

These classes are considered low-level and are exposed for extensibility purposes. They shouldn't be needed in normal applications. Pools and Groups are the recommended way to use routers.

Here is an example of how to use the routerlogic directly:

```
var routees = Enumerable
    .Range(1, 5)
    .Select(i => new ActorRefRoutee(system.ActorOf<Worker>("w" + i)))
    .ToArray();

var router = new Router(new RoundRobinRoutingLogic(), routees);

for (var i = 0; i < 10; i++)
    router.Route("msg #" + i, ActorRefs.NoSender);
```

Dispatchers

What Do Dispatchers Do?

Dispatchers are responsible for scheduling all code that runs inside the `ActorSystem`. Dispatchers are one of the most important parts of Akka.NET, as they control the throughput and time share for each of the actors, giving each one a fair share of resources.

By default, all actors share a single **Global Dispatcher**. Unless you change the configuration, this dispatcher uses the *.NET Thread Pool* behind the scenes, which is optimized for most common scenarios. **That means the default configuration should be good enough for most cases.**

Why should I use different dispatchers?

When messages arrive in the [actor's mailbox](#), the dispatcher schedules the delivery of messages in batches, and tries to deliver the entire batch before releasing the thread to another actor. While the default configuration is *good enough* for most scenarios, you may want to change ([through configuration](#)) how much time the scheduler should spend running each actor.

There are some other common reasons to select a different dispatcher. These reasons include (but are not limited to):

- isolating one or more actors to specific threads in order to:
 - ensure high-load actors don't starve the system by consuming too much cpu-time;
 - ensure important actors always have a dedicated thread to do their job;
 - create [bulkheads](#), ensuring problems created in one part of the system do not leak to others;
- allow actors to execute in a specific `SynchronizationContext`;

[!NOTE] Consider using custom dispatchers for special cases only. Correctly configuring dispatchers requires some understanding of how the framework works. Custom dispatchers *should not* be considered the default solution for performance problems. It's considered normal for complex applications to have one or a few custom dispatchers, it's not usual for most or all actors in a system to require a custom dispatcher configuration.

Dispatchers vs. Dispatcher Configurations

Throughout this documentation and most Akka literature available, the term *dispatcher* is used to refer to *dispatcher configurations*, but they are in fact different things.

- **Dispatchers** are low level components that are responsible for scheduling code execution in the system. These components are built into Akka.NET, there is a fixed number of them and you don't need to create or change them.
- **Dispatcher Configurations** are custom settings you can create to *make use of dispatchers* in specific ways. There are some built-in dispatcher configurations, and you can create as many as you need for your applications.

Therefore, when you read about "*creating a custom dispatcher*" it usually means "*using a custom configuration for one of the built-in dispatchers*".

Configuring Dispatchers

You can define a custom dispatcher configuration using a HOCON configuration section.

The example below creates a custom dispatcher called `my-dispatcher` that can be set in one or more actors during deployment:

```
my-dispatcher {
    type = Dispatcher
    throughput = 100
    throughput-deadline-time = 0ms
}
```

You can then set actor's dispatcher using the deployment configuration:

```
akka.actor.deployment {
    /my-actor {
        dispatcher = my-dispatcher
    }
}
```

Or you can also set it up in code:

```
system.ActorOf(Props.Create<MyActor>().WithDispatcher("my-dispatcher"), "my-actor");
```

Built-in Dispatcher Configurations

Some dispatcher configurations are available out-of-the-box for convenience. You can use them during actor deployment, [as described above](#).

- **default-dispatcher** - A configuration that uses the [ThreadPoolDispatcher](#). As the name says, this is the default dispatcher configuration used by the global dispatcher, and you don't need to define anything during deployment to use it.
- **task-dispatcher** - A configuration that uses the [TaskDispatcher](#).
- **default-fork-join-dispatcher** - A configuration that uses the [ForkJoinDispatcher](#).
- **synchronized-dispatcher** - A configuration that uses the [SynchronizedDispatcher](#).

Built-in Dispatchers

These are the underlying dispatchers built-in to Akka.NET:

ThreadPoolDispatcher

It schedules code to run in the [.NET Thread Pool](#), which is **good enough for most cases**.

The `type` used in the HOCON configuration for this dispatcher is just `Dispatcher`.

```
custom-dispatcher {
    type = Dispatcher
    throughput = 100
}
```

[!NOTE] While each configuration can have its own throughput settings, all dispatchers using this type will run in the same default .NET Thread Pool.

TaskDispatcher

The TaskDispatcher uses the [TPL](#) infrastructure to schedule code execution. This dispatcher is very similar to the Thread PoolDispatcher, but may be used in some rare scenarios where the thread pool isn't available.

```
custom-task-dispatcher {
    type = TaskDispatcher
    throughput = 100
}
```

PinnedDispatcher

The `PinnedDispatcher` uses a single dedicated thread to schedule code executions. Ideally, this dispatcher should be using sparingly.

```
custom-dedicated-dispatcher {
    type = PinnedDispatcher
}
```

ForkJoinDispatcher

The ForkJoinDispatcher uses a dedicated threadpool to schedule code execution. You can use this scheduler isolate some actors from the rest of the system. Each dispatcher configuration will have it's own thread pool.

This is the configuration for the [default-fork-join-dispatcher](#). You may use this as example for custom fork-join dispatchers.

```
default-fork-join-dispatcher {
    type = ForkJoinDispatcher
    throughput = 100
    dedicated-thread-pool {
        thread-count = 3
        deadlock-timeout = 3s
        threadtype = background
    }
}
```

- `thread-count` - The number of threads dedicated to this dispatcher.
- `deadlock-timeout` - The amount of time to wait before considering the thread as deadlocked. By default no timeout is set, meaning code can run in the threads for as long as they need. If you set a value, once the timeout is reached the thread will be aborted and a new threads will take it's place. Set this value carefully, as very low values may cause loss of work.
- `threadtype` - Must be `background` or `foreground`. This setting helps define [how .NET handles](#) the thread.

SynchronizedDispatcher

The `SynchronizedDispatcher` uses the [current SynchronizationContext](#) to schedule executions.

You may use this dispatcher to create actors that update UIs in a reactive manner. An application that displays real-time updates of stock prices may have a dedicated actor to update the UI controls directly for example.

[!NOTE] As a general rule, actors running in this dispatcher shouldn't do much work. Avoid doing any extra work that may be done by actors running in other pools.

This is the configuration for the [synchronized-dispatcher](#). You may use this as example for custom fork-join dispatchers.

```
synchronized-dispatcher {
```

```
type = "SynchronizedDispatcher"  
throughput = 10  
}
```

In order to use this dispatcher, you must create the actor from the synchronization context you want to run-it. For example:

```
private void Form1_Load(object sender, System.EventArgs e)  
{  
    system.ActorOf(Props.Create<UIWorker>().WithDispatcher("synchronized-dispatcher"), "ui-worker");  
}
```

Common Dispatcher Configuration

The following configuration keys are available for any dispatcher configuration:

- `type` - (Required) The type of dispatcher to be used: `Dispatcher` , `TaskDispatcher` , `PinnedDispatcher` , `ForkJoinDispatcher` or `SynchronizedDispatcher` .
- `throughput` - (Required) The maximum # of messages processed each time the actor is activated. Most dispatchers default to `100` .
- `throughput-deadline-time` - The maximum amount of time to process messages when the actor is activated, or `0` for no limit. The default is `0` .

[!NOTE] The throughput-deadline-time is used as a *best effort*, not as a *hard limit*. This means that if a message takes more time than the deadline allows, Akka.NET won't interrupt the process. Instead it will wait for it to finish before giving turn to the next actor.

Mailboxes

What Mailboxes Do?

In Akka.NET, Mailboxes hold messages that are destined for an actor. When you send a message to an actor, the message doesn't go directly to the actor, but goes to the actor's mailbox until the actor has time to process it.

A mailbox can be described as a queue of messages. Messages are usually then delivered from the mailbox to the actor one at a time in the order they were received, but there are implementations like the [Priority Mailbox](#) that can change the order of delivery.

Normally every actor has its own mailbox, but this is not a requirement. There are implementations of [routers](#) where all routees share a single mailbox.

Using a Mailbox

To make an actor use a specific mailbox, you can set it up one of the following locations:

1. In the actor's props

```
Props.Create<ActorType>().WithMailbox("my-custom-mailbox");
```

2. In the actor's deployment configuration

```
akka.actor.deployment {  
    /my-actor-path {  
        mailbox = my-custom-mailbox  
    }  
}
```

The `my-custom-mailbox` is a key that was setup using the [mailbox configuration](#).

Configuring Custom Mailboxes

In order to use a custom mailbox, it must be first configured with a key that the actor system can lookup. You can do this using a custom HOCON setting.

The example below will setup a mailbox key called `my-custom-mailbox` pointing to a custom mailbox implementation. Note that the configuration of the mailbox is outside the akka section.

```
akka { ... }  
my-custom-mailbox {  
    mailbox-type : "MyProject.CustomMailbox, MyProjectAssembly"  
}
```

Built-in Mailboxes

UnboundedMailbox

This is the default mailbox used by Akka.NET. It's a non-blocking unbounded mailbox, and should be good enough for most cases.

UnboundedPriorityMailbox

The unbounded mailbox priority mailbox is blocking mailbox that allows message prioritization, so that you can choose the order the actor should process messages that are already in the mailbox.

In order to use this mailbox, you need to extend the `UnboundedPriorityMailbox` class and provide an ordering logic. The value returned by the `PriorityGenerator` method will be used to order the message in the mailbox. Lower values will be delivered first. Delivery order for messages of equal priority is undefined.

```
public class IssueTrackerMailbox : UnboundedPriorityMailbox
{
    protected override int PriorityGenerator(object message)
    {
        var issue = message as Issue;

        if (issue != null)
        {
            if (issue.IsSecurityFlaw)
                return 0;

            if (issue.IsBug)
                return 1;
        }

        return 2;
    }
}
```

Once the class is implemented, you should set it up using the [instructions above](#).

[!WARNING] While we have updated the `UnboundedPriorityMailbox` to support Stashing. We don't recommend using it. Once you stash messages, they are no longer part of the prioritization process that your `PriorityMailbox` uses. Once you unstash all messages, they are fed to the Actor, in the order of stashing.

The Inbox

When writing code outside of actors which shall communicate with actors, the ask pattern can be a solution (see below), but there are two things it cannot do: receiving multiple replies (e.g. by subscribing an `IActorRef` to a notification service) and watching other actors' lifecycle. For these purposes there is the `Inbox` class:

```
var target = system.ActorOf(Props.Empty);
var inbox = Inbox.Create(system);

inbox.Send(target, "hello");

try
{
    inbox.Receive(TimeSpan.FromSeconds(1)).Equals("world");
}
catch (TimeoutException)
{
    // timeout
}
```

The send method wraps a normal `Tell` and supplies the internal actor's reference as the sender. This allows the reply to be received on the last line. Watching an actor is quite simple as well

```
using System.Diagnostics;
...
var inbox = Inbox.Create(system);
inbox.Watch(target);
target.Tell(PoisonPill.Instance, ActorRefs.NoSender);

try
{
    Debug.Assert(inbox.Receive(TimeSpan.FromSeconds(1)) is Terminated);
}
catch (TimeoutException)
{
    // timeout
}
```

Finite State Machine

A Simple Example

To demonstrate most of the features of the `FSM` class, consider an actor which shall receive and queue messages while they arrive in a burst and send them on after the burst ended or a flush request is received.

First, consider all of the below to use these import statements:

```
using Akka.Actor;
using Akka.Event;
```

The contract of our `Buncher` actor is that it accepts or produces the following messages:

```
// received events
public class SetTarget
{
    public SetTarget(IActorRef @ref)
    {
        Ref = @ref;
    }

    public IActorRef Ref { get; }
}

public class Queue
{
    public Queue(object obj)
    {
        Obj = obj;
    }

    public Object Obj { get; }
}

public class Flush { }

// send events
public class Batch
{
    public Batch(ImmutableList<object> obj)
    {
        Obj = obj;
    }

    public ImmutableList<object> Obj { get; }
}
```

`SetTarget` is needed for starting it up, setting the destination for the Batches to be passed on; `Queue` will add to the internal queue while `Flush` will mark the end of a burst.

```
public enum State
{
    Idle,
    Active
}

// data
public interface IData { }
```

```

public class Uninitialized : IData
{
    public static Uninitialized Instance = new Uninitialized();

    private Uninitialized() { }

}

public class Todo : IData
{
    public Todo(IActorRef target, ImmutableList<object> queue)
    {
        Target = target;
        Queue = queue;
    }

    public IActorRef Target { get; }

    public ImmutableList<object> Queue { get; }

    public Todo Copy(ImmutableList<object> queue)
    {
        return new Todo(Target, queue);
    }
}

```

The actor can be in two states: no message queued (aka `Idle`) or some message queued (aka `Active`). It will stay in the active state as long as messages keep arriving and no flush is requested. The internal state data of the actor is made up of the target actor reference to send the batches to and the actual queue of messages.

```

public class ExampleFSMActor : FSM<State, IData>
{
    private readonly ILoggingAdapter _log = Context.GetLogger();

    public ExampleFSMActor()
    {
        StartWith(State.Idle, Uninitialized.Instance);

        When(State.Idle, state =>
        {
            if (state.FsmEvent is SetTarget target && state.StateData is Uninitialized)
            {
                return Stay().Using(new Todo(target.Ref, ImmutableList<object>.Empty));
            }

            return null;
        });

        When(State.Active, state =>
        {
            if ((state.FsmEvent is Flush || state.FsmEvent is StateTimeout)
                && state.StateData is Todo t)
            {
                return GoTo(State.Idle).Using(t.Copy(ImmutableList<object>.Empty));
            }

            return null;
        }, TimeSpan.FromSeconds(1));

        WhenUnhandled(state =>
        {
            if (state.FsmEvent is Queue q && state.StateData is Todo t)
            {
                return GoTo(State.Active).Using(t.Copy(t.Queue.Add(q.Obj)));
            }
            else
            {

```

```

        _log.Warning("Received unhandled request {0} in state {1}/{2}", state.FsmEvent, StateName,
state.StateData);
        return Stay();
    }
});

OnTransition((initialState, nextState) =>
{
    if (initialState == State.Active && nextState == State.Idle)
    {
        if (StateData is Todo todo)
        {
            todo.Target.Tell(new Batch(todo.Queue));
        }
        else
        {
            // nothing to do
        }
    }
});
Initialize();
}
}

```

The basic strategy is to declare the actor, inherit from `FSM` class and specifying the possible states and data values as type parameters. Within the body of the actor a DSL is used for declaring the state machine:

- `StartWith` defines the initial state and initial data
- then there is one `When(<state>, () => {})` declaration per state to be handled
- finally starting it up using `initialize`, which performs the transition into the initial state and sets up timers (if required).

In this case, we start out in the `Idle` and `Uninitialized` state, where only the `SetTarget()` message is handled; `stay` prepares to end this event's processing for not leaving the current state, while the `using` modifier makes the `FSM` replace the internal state (which is `Uninitialized` at this point) with a fresh `Todo()` object containing the target actor reference. The `Active` state has a state timeout declared, which means that if no message is received for 1 second, a `FSM.StateTimeout` message will be generated. This has the same effect as receiving the `Flush` command in this case, namely to transition back into the `Idle` state and resetting the internal queue to the empty vector. But how do messages get queued? Since this shall work identically in both states, we make use of the fact that any event which is not handled by the `when()` block is passed to the `whenUnhandled()` block:

```

WhenUnhandled(state =>
{
    if (state.FsmEvent is Queue q && state.StateData is Todo t)
    {
        return GoTo(State.Active).Using(t.Copy(t.Queue.Add(q.Obj)));
    }
    else
    {
        _log.Warning("Received unhandled request {0} in state {1}/{2}", state.FsmEvent, StateName,
state.StateData);
        return Stay();
    }
});

```

The first case handled here is adding `Queue()` requests to the internal queue and going to the `Active` state (this does the obvious thing of staying in the `Active` state if already there), but only if the `FSM` data are not `Uninitialized` when the `Queue()` event is received. Otherwise—and in all other non-handled cases—the second case just logs a warning and does not change the internal state.

The only missing piece is where the `Batches` are actually sent to the target, for which we use the `OnTransition` mechanism: you can declare multiple such blocks and all of them will be tried for matching behavior in case a state transition occurs (i.e. only when the state actually changes).

```
OnTransition((initialState, nextState) =>
{
    if (initialState == State.Active && nextState == State.Idle)
    {
        if (StateData is Todo todo)
        {
            todo.Target.Tell(new Batch(todo.Queue));
        }
        else
        {
            // nothing to do
        }
    }
});
```

The transition callback is a function which takes as input a pair of states—the current and the next state. The `FSM` class includes a convenience extractor for these in form of an arrow operator, which conveniently reminds you of the direction of the state change which is being matched. During the state change, the old state data is available via `StateData` as shown, and the new state data would be available as `NextStateData`.

To verify that this buncher actually works, it is quite easy to write a test using the `Testing Actor Systems`.

```
public class ExampleFSMActorTests : TestKit
{
    [Fact]
    public void Simple_finite_state_machine_must_batch_correctly()
    {
        var buncher = Sys.ActorOf(Props.Create<ExampleFSMActor>());
        buncher.Tell(new SetTarget(TestActor));
        buncher.Tell(new Queue(42));
        buncher.Tell(new Queue(43));
        ExpectMsg<Batch>().Obj.Should().BeEquivalentTo(ImmutableList.Create(42, 43));
        buncher.Tell(new Queue(44));
        buncher.Tell(new Flush());
        buncher.Tell(new Queue(45));
        ExpectMsg<Batch>().Obj.Should().BeEquivalentTo(ImmutableList.Create(44));
        ExpectMsg<Batch>().Obj.Should().BeEquivalentTo(ImmutableList.Create(45));
    }

    [Fact]
    public void Simple_finite_state_machine_must_not_batch_if_uninitialized()
    {
        var buncher = Sys.ActorOf(Props.Create<ExampleFSMActor>());
        buncher.Tell(new Queue(42));
        ExpectNoMsg();
    }
}
```

Reference

The FSM class

The `FSM` class inherits directly from `ActorBase`, when you extend `FSM` you must be aware that an actor is actually created:

```
public class Buncher : FSM<State, IData>
```

```
{
    public ExampleFSMActor()
    {
        // fsm body ...
        Initialize();
    }
}
```

[!NOTE] The `fsm` class defines a receive method which handles internal messages and passes everything else through to the `fsm` logic (according to the current state). When overriding the receive method, keep in mind that e.g. state timeout handling depends on actually passing the messages through the `fsm` logic.

The `fsm` class takes two type parameters:

- the supertype of all state names, usually a sealed trait with case objects extending it,
- the type of the state data which are tracked by the `fsm` module itself.

[!NOTE] The state data together with the state name describe the internal state of the state machine; if you stick to this scheme and do not add mutable fields to the `fsm` class you have the advantage of making all changes of the internal state explicit in a few well-known places.

Defining States

A state is defined by one or more invocations of the method

```
When(<stateName>, <stateFunction>[, timeout = <timeout>]).
```

The given name must be an object which is type-compatible with the first type parameter given to the `fsm` class. This object is used as a hash key, so you must ensure that it properly implements `Equals` and `GetHashCode`; in particular it must not be mutable. The easiest fit for these requirements are case objects.

If the `timeout` parameter is given, then all transitions into this state, including staying, receive this timeout by default. Initiating the transition with an explicit timeout may be used to override this default, see [Initiating Transitions](#) for more information. The state timeout of any state may be changed during action processing with `SetStateTimeout(state, duration)`. This enables runtime configuration e.g. via external message.

The `stateFunction` argument is a `delegate State<TState, TData> StateFunction(Event<TData> fsmEvent)`.

```
When(State.Idle, state =>
{
    if (state.FsmEvent is SetTarget target && state.StateData is Uninitialized)
    {
        return Stay().Using(new Todo(target.Ref, ImmutableList<object>.Empty));
    }

    return null;
});

When(State.Active, state =>
{
    if ((state.FsmEvent is Flush || state.FsmEvent is StateTimeout)
        && state.StateData is Todo t)
    {
        return GoTo(State.Idle).Using(t.Copy(ImmutableList<object>.Empty));
    }

    return null;
}, TimeSpan.FromSeconds(1));
```

Defining the Initial State

Each `FSM` needs a starting point, which is declared using

```
StartWith(state, data[, timeout])
```

The optionally given `timeout` argument overrides any specification given for the desired initial state. If you want to cancel a default timeout, use `null`.

Unhandled Events

If a state doesn't handle a received event a warning is logged. If you want to do something else in this case you can specify that with `WhenUnhandled(stateFunction)`:

```
WhenUnhandled(state =>
{
    if (state.FsmEvent is Queue x)
    {
        _log.Info("Received unhandled event: " + x);
        return Stay();
    }
    else
    {
        _log.Warning("Received unknown event: " + state.FsmEvent);
        return Goto(new Error());
    }
});
```

Within this handler the state of the `FSM` may be queried using the `stateName` method.

[!IMPORTANT] This handler is not stacked, meaning that each invocation of `WhenUnhandled` replaces the previously installed handler.

Initiating Transitions

The result of any `stateFunction` must be a definition of the next state unless terminating the `FSM`, which is described in [Termination from Inside](#). The state definition can either be the current state, as described by the `stay` directive, or it is a different state as given by `Goto(state)`. The resulting object allows further qualification by way of the modifiers described in the following:

- `ForMax(duration)`. This modifier sets a state timeout on the next state. This means that a timer is started which upon expiry sends a `StateTimeout` message to the `FSM`. This timer is canceled upon reception of any other message in the meantime; you can rely on the fact that the `StateTimeout` message will not be processed after an intervening message. This modifier can also be used to override any default timeout which is specified for the target state. If you want to cancel the default timeout, use `null`.
- `Using(data)`. This modifier replaces the old state data with the new data given. If you follow the advice above, this is the only place where internal state data are ever modified.
- `Replies(msg)`. This modifier sends a reply to the currently processed message and otherwise does not modify the state transition.

All modifiers can be chained to achieve a nice and concise description:

```
When(State.SomeState, state => {
    return GoTo(new Processing())
        .Using(new Data())
        .ForMax(TimeSpan.FromSeconds(5))
        .Replies(new WillDo());
```

```
});
```

Monitoring Transitions

Transitions occur "between states" conceptually, which means after any actions you have put into the event handling block; this is obvious since the next state is only defined by the value returned by the event handling logic. You do not need to worry about the exact order with respect to setting the internal state variable, as everything within the `FSM` actor is running single-threaded anyway.

Internal Monitoring

Up to this point, the `FSM` DSL has been centered on states and events. The dual view is to describe it as a series of transitions. This is enabled by the method

```
OnTransition(handler)
```

which associates actions with a transition instead of with a state and event. The handler is a delegate `void TransitionHandler(TState initialState, TState nextState)` function which takes a pair of states as input; no resulting state is needed as it is not possible to modify the transition in progress.

```
OnTransition((initialState, nextState) =>
{
    if (initialState == State.Active && nextState == State.Idle)
    {
        SetTimer("timeout", new Tick(), TimeSpan.FromSeconds(1), repeat: true);
    }
    else if (initialState == State.Active)
    {
        CancelTimer("timeout");
    }
    else if (nextState == State.Idle)
    {
        _log.Info("entering Idle from " + initialState);
    }
});
```

The handlers registered with this method are stacked, so you can intersperse `onTransition` blocks with when blocks as suits your design. It should be noted, however, that all handlers will be invoked for each transition, not only the first matching one. This is designed specifically so you can put all transition handling for a certain aspect into one place without having to worry about earlier declarations shadowing later ones; the actions are still executed in declaration order, though.

[!NOTE] This kind of internal monitoring may be used to structure your FSM according to transitions, so that for example the cancellation of a timer upon leaving a certain state cannot be forgot when adding new target states.

External Monitoring

External actors may be registered to be notified of state transitions by sending a message `SubscribeTransitionCallBack(IActorRef)`. The named actor will be sent a `CurrentState(self, stateName)` message immediately and will receive `Transition(IActorRef, oldState, newState)` messages whenever a state change is triggered.

Please note that a state change includes the action of performing an `GoTo(s)`, while already being state S. In that case the monitoring actor will be notified with an `Transition(ref, s, s)` message. This may be useful if your FSM should react on all (also same-state) transitions. In case you'd rather not emit events for same-state transitions use `stay()` instead of `GoTo(s)`.

External monitors may be unregistered by sending `UnsubscribeTransitionCallBack(IActorRef)` to the `FSM` actor.

Stopping a listener without unregistering will not remove the listener from the subscription list; use `UnsubscribeTransitionCallback` before stopping the listener.

Timers

Besides state timeouts, `FSM` manages timers identified by String names. You may set a timer using

```
SetTimer(name, msg, interval, repeat);
```

where `msg` is the message object which will be sent after the duration interval has elapsed. If `repeat` is true, then the timer is scheduled at fixed rate given by the `interval` parameter. Any existing timer with the same name will automatically be canceled before adding the new timer.

Timers may be canceled using

```
CancelTimer(name);
```

which is guaranteed to work immediately, meaning that the scheduled message will not be processed after this call even if the timer already fired and queued it. The status of any timer may be inquired with

```
IsTimerActive(name);
```

These named timers complement state timeouts because they are not affected by intervening reception of other messages.

Termination from Inside

The `FSM` is stopped by specifying the result state as

```
Stop(reason, stateData);
```

The reason must be one of `Normal` (which is the default), `Shutdown` or `Failure(reason)`, and the second argument may be given to change the state data which is available during termination handling.

[!NOTE] It should be noted that stop does not abort the actions and stop the `FSM` immediately. The stop action must be returned from the event handler in the same way as a state transition.

```
When(State.Error, state =>
{
    if (state.FsmEvent == "stop")
    {
        return Stop();
    }

    return null;
});
```

You can use `OnTermination(handler)` to specify custom code that is executed when the `FSM` is stopped. The handler is a partial function which takes a `StopEvent(reason, stateName, stateData)` as argument:

```
OnTermination(termination =>
{
    switch (termination)
    {
        case StopEvent<State, IData> st when st.Reason is Normal:
            // ...
            break;
        case StopEvent<State, IData> st when st.Reason is Shutdown:
            // ...
            break;
        case StopEvent<State, IData> st when st.Reason is Failure:
            // ...
            break;
    }
});
```

As for the `WhenUnhandled` case, this handler is not stacked, so each invocation of `OnTermination` replaces the previously installed handler.

Termination from Outside

When an `IActorRef` associated to a `FSM` is stopped using the `stop` method, its `PostStop` hook will be executed. The default implementation by the `FSM` class is to execute the `onTermination` handler if that is prepared to handle a `StopEvent(Shutdown, ...)`.

[!WARNING] In case you override `PostStop` and want to have your `onTermination` handler called, do not forget to call `base.PostStop`.

Fault Tolerance

As explained in [Actor Systems](#) each actor is the supervisor of its children, and as such each actor defines fault handling supervisor strategy. This strategy cannot be changed afterwards as it is an integral part of the actor system's structure.

Fault Handling in Practice

First, let us look at a sample that illustrates one way to handle data store errors, which is a typical source of failure in real world applications. Of course it depends on the actual application what is possible to do when the data store is unavailable, but in this sample we use a best effort re-connect approach.

Creating a Supervisor Strategy

The following sections explain the fault handling mechanism and alternatives in more depth.

For the sake of demonstration let us consider the following strategy:

```
protected override SupervisorStrategy SupervisorStrategy()
{
    return new OneForOneStrategy(
        maxNrOfRetries: 10,
        withinTimeRange: TimeSpan.FromMinutes(1),
        localOnlyDecider: ex =>
    {
        switch (ex)
        {
            case ArithmeticException ae:
                return Directive.Resume;
            case NullReferenceException nre:
                return Directive.Restart;
            case ArgumentException are:
                return Directive.Stop;
            default:
                return Directive.Escalate;
        }
    });
}
```

I have chosen a few well-known exception types in order to demonstrate the application of the fault handling directives described in [Supervision and Monitoring](#). First off, it is a one-for-one strategy, meaning that each child is treated separately (an all-for-one strategy works very similarly, the only difference is that any decision is applied to all children of the supervisor, not only the failing one). There are limits set on the restart frequency, namely maximum 10 restarts per minute; each of these settings could be left out, which means that the respective limit does not apply, leaving the possibility to specify an absolute upper limit on the restarts or to make the restarts work infinitely. The child actor is stopped if the limit is exceeded.

This is the piece which maps child failure types to their corresponding directives.

[!NOTE] If the strategy is declared inside the supervising actor (as opposed to within a companion object) its decider has access to all internal state of the actor in a thread-safe fashion, including obtaining a reference to the currently failed child (available as the `sender` of the failure message).

Default Supervisor Strategy

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; and
- Any other type of `Exception` will restart the failing child 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.

You can combine your own strategy with the default strategy:

```
protected override SupervisorStrategy SupervisorStrategy()
{
    return new OneForOneStrategy(
        maxNrOfRetries: 10,
        withinTimeRange: TimeSpan.FromMinutes(1),
        localOnlyDecider: ex =>
    {
        if (ex is ArithmeticException)
        {
            return Directive.Resume;
        }

        return Akka.Actor.SupervisorStrategy.DefaultStrategy.Decider.Decide(ex);
    });
}
```

Stopping Supervisor Strategy

Closer to the Erlang way is the strategy to just stop children when they fail and then take corrective action in the supervisor when DeathWatch signals the loss of the child. This strategy is also provided pre-packaged as `SupervisorStrategy.StoppingStrategy` with an accompanying `StoppingSupervisorStrategy` configurator to be used when you want the `"/user"` guardian to apply it.

Logging of Actor Failures

By default the `SupervisorStrategy` logs failures unless they are escalated. Escalated failures are supposed to be handled, and potentially logged, at a level higher in the hierarchy.

You can mute the default logging of a `SupervisorStrategy` by setting `loggingEnabled` to `false` when instantiating it. Customized logging can be done inside the `Decider`. Note that the reference to the currently failed child is available as the `Sender` when the `SupervisorStrategy` is declared inside the supervising actor.

You may also customize the logging in your own `SupervisorStrategy` implementation by overriding the `logFailure` method.

Supervision of Top-Level Actors

Top-level actors means those which are created using `system.ActorOf()`, and they are children of the [User Guardian](#). There are no special rules applied in this case, the guardian simply applies the configured strategy.

Test Application

The following section shows the effects of the different directives in practice, wherefor a test setup is needed. First off, we need a suitable supervisor:

```

public class Supervisor : UntypedActor
{
    protected override SupervisorStrategy SupervisorStrategy()
    {
        return new OneForOneStrategy(
            maxNrOfRetries: 10,
            withinTimeRange: TimeSpan.FromMinutes(1),
            localOnlyDecider: ex =>
        {
            switch (ex)
            {
                case ArithmeticException ae:
                    return Directive.Resume;
                case NullReferenceException nre:
                    return Directive.Restart;
                case ArgumentException are:
                    return Directive.Stop;
                default:
                    return Directive.Escalate;
            }
        });
    }

    protected override void OnReceive(object message)
    {
        if (message is Props p)
        {
            var child = Context.ActorOf(p); // create child
            Sender.Tell(child); // send back reference to child actor
        }
    }
}

```

This supervisor will be used to create a child, with which we can experiment:

```

public class Child : UntypedActor
{
    private int state = 0;

    protected override void OnReceive(object message)
    {
        switch (message)
        {
            case Exception ex:
                throw ex;
                break;
            case int x:
                state = x;
                break;
            case "get":
                Sender.Tell(state);
                break;
        }
    }
}

```

The test is easier by using the utilities described in [Akka-Testkit](#).

Let us create actors:

```

var supervisor = system.ActorOf<Supervisor>("supervisor");

supervisor.Tell(Props.Create<Child>());
var child = ExpectMsg<IActorRef>(); // retrieve answer from TestKit's TestActor

```

The first test shall demonstrate the `Resume` directive, so we try it out by setting some non-initial state in the actor and have it fail:

```
child.Tell(42); // set state to 42
child.Tell("get");
ExpectMsg(42);

child.Tell(new ArithmeticException()); // crash it
child.Tell("get");
ExpectMsg(42);
```

As you can see the value 42 survives the fault handling directive because we're using the `Resume` directive, which does not cause the actor to restart. Now, if we change the failure to a more serious `NullReferenceException`, that will no longer be the case:

```
child.Tell(new NullReferenceException());
child.Tell("get");
ExpectMsg(0);
```

This is because the actor has restarted and the original `child` actor instance that was processing messages will be destroyed and replaced by a brand-new instance defined using the original `Props` passed to its parent.

And finally in case of the fatal `IllegalArgumentException` the child will be terminated by the supervisor:

```
Watch(child); // have testActor watch "child"
child.Tell(new ArgumentException()); // break it
ExpectMsg<Terminated>().ActorRef.Should().Be(child);
```

Up to now the supervisor was completely unaffected by the child's failure, because the directives set did handle it. In case of an `Exception`, this is not true anymore and the supervisor escalates the failure.

```
supervisor.Tell(Props.Create<Child>()); // create new child
var child2 = ExpectMsg<IActorRef>();
Watch(child2);
child2.Tell("get"); // verify it is alive
ExpectMsg(0);

child2.Tell(new Exception("CRASH"));
var message = ExpectMsg<Terminated>();
message.ActorRef.Should().Be(child2);
message.ExistenceConfirmed.Should().BeTrue();
```

The supervisor itself is supervised by the top-level actor provided by the `ActorSystem`, which has the default policy to restart in case of all `Exception` cases (with the notable exceptions of `ActorInitializationException` and `ActorKilledException`). Since the default directive in case of a restart is to kill all children, we expected our poor child not to survive this failure.

In case this is not desired (which depends on the use case), we need to use a different supervisor which overrides this behavior.

```
public class Supervisor2 : UntypedActor
{
    protected override SupervisorStrategy SupervisorStrategy()
    {
        return new OneForOneStrategy(
            maxNrOfRetries: 10,
            withinTimeRange: TimeSpan.FromMinutes(1),
            localOnlyDecider: ex =>
            {
```

```

        switch (ex)
    {
        case ArithmeticException ae:
            return Directive.Resume;
        case NullReferenceException nre:
            return Directive.Restart;
        case ArgumentException are:
            return Directive.Stop;
        default:
            return Directive.Escalate;
    }
});

protected override void PreRestart(Exception reason, object message)
{
}

protected override void OnReceive(object message)
{
    if (message is Props p)
    {
        var child = Context.ActorOf(p); // create child
        Sender.Tell(child); // send back reference to child actor
    }
}
}

```

With this parent, the child survives the escalated restart, as demonstrated in the last test:

```

var supervisor2 = system.ActorOf<Supervisor2>("supervisor2");

supervisor2.Tell(Props.Create<Child>());
var child3 = ExpectMsg<IActorRef>();

child3.Tell(23);
child3.Tell("get");
ExpectMsg(23);

child3.Tell(new Exception("CRASH"));
child3.Tell("get");
ExpectMsg(0);

```

Dependency Injection

If your actor has a constructor that takes parameters then those need to be part of the `Props` as well, as described above. But there are cases when a factory method must be used, for example when the actual constructor arguments are determined by a dependency injection framework.

The basic functionality is provided by a `DependencyResolver` class, that can create `Props` using the DI containerer.

```
// Create your DI container of preference
var someContainer = ... ;

// Create the actor system
var system = ActorSystem.Create("MySystem");

// Create the dependency resolver for the actor system
IDependencyResolver resolver = new XyzDependencyResolver(someContainer, system);
```

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

```
// Create the Props using the DI extension on your ActorSystem instance
var worker1Ref = system.ActorOf(system.DI().Props<TypedWorker>(), "Worker1");
var worker2Ref = system.ActorOf(system.DI().Props<TypedWorker>(), "Worker2");
```

Creating Child Actors using DI

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

```
// For example in the PreStart...
protected override void PreStart()
{
    var actorProps = Context.DI().Props<MyActor>()
        .WithRouter(/* options here */);

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

[!NOTE] There is currently still an extension method available for the actor Context. `Context.DI().ActorOf<>`. However this has been officially **deprecated** and will be removed in future versions.

Notes

[!WARNING] You might be tempted at times to use an `IndirectActorProducer` which always returns the same instance, e.g. by using a static field. This is not supported, as it goes against the meaning of an actor restart, which is described here: [What Restarting Means](#).

When using a dependency injection framework, there are a few things you have to keep in mind.

When scoping actor type dependencies using your DI container, only `TransientLifestyle` or `InstancePerDependency` like scopes are supported. This is due to the fact that Akka explicitly manages the lifecycle of its actors. So any scope which interferes with that is not supported.

This also means that when injecting dependencies into your actor, using a Singleton or Transient scope is fine. But having that dependency scoped per httpwebrequest for example won't work.

Techniques for dependency injection and integration with dependency injection frameworks are described in more depth in the [Using Akka with Dependency Injection](#) guideline.

Currently the following Akka.NET Dependency Injection plugins are available:

AutoFac

In order to use this plugin, install the Nuget package with `Install-Package Akka.DI.AutoFac`, then follow the instructions:

```
// Create and build your container
var builder = new Autofac.ContainerBuilder();
builder.RegisterType<WorkerService>().As<IWorkerService>();
builder.RegisterType<TypedWorker>();
var container = builder.Build();

// Create the ActorSystem and Dependency Resolver
var system = ActorSystem.Create("MySystem");
var propsResolver = new AutoFacDependencyResolver(container, system);
```

CastleWindsor

In order to use this plugin, install the Nuget package with `Install-Package Akka.DI.CastleWindsor`, then follow the instructions:

```
// Create and build your container
var container = new WindsorContainer();
container.Register(Component.For<IWorkerService>().ImplementedBy<WorkerService>());
container.Register(Component.For<TypedWorker>().Named("TypedWorker").LifestyleTransient());

// Create the ActorSystem and Dependency Resolver
var system = ActorSystem.Create("MySystem");
var propsResolver = new WindsorDependencyResolver(container, system);
```

Ninject

In order to use this plugin, install the Nuget package with `Install-Package Akka.DI.Ninject`, then follow the instructions:

```
// Create and build your container
var container = new Ninject.StandardKernel();
container.Bind<TypedWorker>().To(typeof(TypedWorker));
container.Bind<IWorkerService>().To(typeof(WorkerService));

// Create the ActorSystem and Dependency Resolver
var system = ActorSystem.Create("MySystem");
var propsResolver = new NinjectDependencyResolver(container, system);
```

Other frameworks

Support for additional dependency injection frameworks may be added in the future, but you can easily implement your own by implementing an [Actor Producer Extension](#).

Testing Actor Systems

As with any piece of software, automated tests are a very important part of the development cycle. The actor model presents a different view on how units of code are delimited and how they interact, which has an influence on how to perform tests.

Akka.NET comes with a dedicated module `Akka.TestKit` for supporting tests at different levels.

Asynchronous Testing: TestKit

Testkit allows you to test your actors in a controlled but realistic environment. The definition of the environment depends of course very much on the problem at hand and the level at which you intend to test, ranging from simple checks to full system tests.

The minimal setup consists of the test procedure, which provides the desired stimuli, the actor under test, and an actor receiving replies. Bigger systems replace the actor under test with a network of actors, apply stimuli at varying injection points and arrange results to be sent from different emission points, but the basic principle stays the same in that a single procedure drives the test.

The `TestKit` class contains a collection of tools which makes this common task easy.

[!code-csharpIntroSample]

The `TestKit` contains an actor named `TestActor` which is the entry point for messages to be examined with the various `ExpectMsg...` assertions detailed below. The `TestActor` may also be passed to other actors as usual, usually subscribing it as notification listener. There is a while set of examination methods, e.g. receiving all consecutive messages matching certain criteria, receiving a while sequence of fixed messages or classes, receiving nothing for some time, etc.

You can provide your own `ActorSystem` instance, or `Config` by overriding the `TestKit` constructor. The `ActorSystem` used by the `TestKit` is accessible via the `Sys` member.

Built-In Assertions

The above mentioned `ExpectMsg` is not the only method for formulating assertions concerning received messages. Here is the full list:

- `T ExpectMsg<T>(TimeSpan? duration = null, string hint)` The given message object must be received within the specified time; the object will be returned.
- `T ExpectMsgAnyOf<T>(params T[] messages)` An object must be received, and it must be equal to at least one of the passed reference objects; the received object will be returned.
- `IReadOnlyCollection<T> ExpectMsgAllOf<T>(TimeSpan max, params T[] messages)` A number of objects matching the size of the supplied object array must be received within the given time, and for each of the given objects there must exist at least one among the received ones which equals it. The full sequence of received objects is returned.
- `void ExpectNoMsg(TimeSpan duration)` No message must be received within the given time. This also fails if a message has been received before calling this method which has not been removed from the queue using one of the other methods.

- `T ExpectMsgFrom<T>(IActorRef sender, TimeSpan? duration = null, string hint = null)` Receive one message of the specified type from the test actor and assert that it equals the message and was sent by the specified sender
- `IReadOnlyCollection<object> ReceiveN(int numberofMessages, TimeSpan max)` `n` messages must be received within the given time; the received messages are returned.
- `object FishForMessage(Predicate<object> isMessage, TimeSpan? max, string)` Keep receiving messages as long as the time is not used up and the partial function matches and returns `false`. Returns the message received for which it returned `true` or throws an exception, which will include the provided hint for easier debugging.

In addition to message reception assertions there are also methods which help with messages flows:

- `object ReceiveOne(TimeSpan? max = null)` Receive one message from the internal queue of the `TestActor`. This method blocks the specified duration or until a message is received. If no message was received, `null` is returned.
- `IReadOnlyList<T> ReceiveWhile<T>(TimeSpan? max, TimeSpan? idle, Func<object, T> filter, int msgs = int.MaxValue)` Collect messages as long as
 - They are matching the provided filter
 - The given time interval is not used up
 - The next message is received within the idle timeout
 - The number of messages has not yet reached the maximum All collected messages are returned. The maximum duration defaults to the time remaining in the innermost enclosing `Within` block and the idle duration defaults to infinity (thereby disabling the idle timeout feature). The number of expected messages defaults to `Int.MaxValue`, which effectively disables this limit.
- `void AwaitCondition(Func<bool> conditionIsFulfilled, TimeSpan? max, TimeSpan? interval, string message = null)` Poll the given condition every `interval` until it returns `true` or the `max` duration is used up. The interval defaults to 100ms and the maximum defaults to the time remaining in the innermost enclosing `within` block.
- `void AwaitAssert(Action assertion, TimeSpan? duration = default(TimeSpan?), TimeSpan? interval = default(TimeSpan?))` Poll the given assert function every `interval` until it does not throw an exception or the `max` duration is used up. If the timeout expires the last exception is thrown. The interval defaults to 100ms and the maximum defaults to the time remaining in the innermost enclosing `within` block. The interval defaults to 100ms and the maximum defaults to the time remaining in the innermost enclosing `within` block.
- `void IgnoreMessages(Func<object, bool> shouldIgnoreMessage)` The internal `testActor` contains a partial function for ignoring messages: it will only enqueue messages which do not match the function or for which the function returns `false`. This feature is useful e.g. when testing a logging system, where you want to ignore regular messages and are only interesting in your specific ones.

Expecting Log Messages

Since an integration test does not allow to the internal processing of the participating actors, verifying expected exceptions cannot be done directly. Instead, use the logging system for this purpose: replacing the normal event handler with the `TestEventListener` and using an `EventFilter` allows assertions on log messages, including those which are generated by exceptions:

```
//TODO EVENTFILTER SAMPLE
```

If a number of occurrences is specific --as demonstrated above-- then `intercept` will block until that number of matching messages have been received or the timeout configured in `akka.test.filter-leeway` is used up (time starts counting after the passed-in block of code returns). In case of a timeout the test fails.

[!NOTE] By default the `TestKit` already loads the `TestEventListener` as a logger. Be aware that if you want to specify your own config. Use the `DefaultConfig` property to apply overrides.

Timing Assertions

Another important part of functional testing concerns timing: certain events must not happen immediately (like a timer), others need to happen before a deadline. Therefore, all examination methods accept an upper time limit within the positive or negative result must be obtained. Lower time limits need to be checked external to the examination, which is facilitated by a new construct for managing time constraints:

[!code-csharpWithinSample]

The block in `within` must complete after a `Duration` which is between `min` and `max`, where the former defaults to zero. The deadline calculated by adding the `max` parameter to the block's start time is implicitly available within the block to all examination methods, if you do not specify it, it is inherited from the innermost enclosing `within` block.

It should be noted that if the last message-receiving assertion of the block is `ExpectNoMsg` or `ReceiveWhile`, the final check of the `within` is skipped in order to avoid false positives due to wake-up latencies. This means that while individual contained assertions still use the maximum time bound, the overall block may take arbitrarily longer in this case.

```
var worker = ActorOf<Worker>();
Within(200.Milliseconds()) {
    worker.Tell("some work");
    ExpectMsg("Some Result");
    ExpectNoMsg(); //will block for the rest of the 200ms
    Thread.Sleep(300); //will NOT make this block fail
}
```

Accounting for Slow Test System

The tight timeouts you use during testing on your lightning-fast notebook will invariably lead to spurious test failures on your heavily loaded build server. To account for this situation, all maximum durations are internally scaled by a factor taken from the **Configuration**, `akka.test.timefactor`, which defaults to 1.

You can scale other durations with the same factor by using the `Dilated` method in `TestKit`.

//TODO DILATED EXAMPLE

Using Multiple Probe Actors

When the actors under test are supposed to send various messages to different destinations, it may be difficult distinguishing the message streams arriving at the `TestActor` when using the `TestKit` as shown until now. Another approach is to use it for creation of simple probe actors to be inserted in the message flows. The functionality is best explained using a small example:

[!code-csharpProbeSample]

This simple test verifies an equally simple Forwarder actor by injecting a probe as the forwarder's target. Another example would be two actors A and B which collaborate by A sending messages to B. In order to verify this message flow, a `TestProbe` could be inserted as target of A, using the forwarding capabilities or auto-pilot described below to include a real B in the test setup.

If you have many test probes, you can name them to get meaningful actor names in test logs and assertions:

[!code-csharpMultipleProbeSample]

Probes may also be equipped with custom assertions to make your test code even more concise and clear:

```
//TODO CUSTOM PROBE IMPL SAMPLE
```

You have complete flexibility here in mixing and matching the `TestKit` facilities with your own checks and choosing an intuitive name for it. In real life your code will probably be a bit more complicated than the example given above; just use the power!

[!WARNING] Any message send from a `TestProbe` to another actor which runs on the `CallingThreadDispatcher` runs the risk of dead-lock, if that other actor might also send to this probe. The implementation of `TestProbe.Watch` and `TestProbe.Unwatch` will also send a message to the wachee, which means that it is dangerous to try watching e.g. `TestActorRef` from a `TestProbe`.

Watching Other Actors from probes

A `TestProbe` can register itself for DeathWatch of any other actor:

```
var probe = CreateTestProbe();
probe.Watch(target);

target.Tell(PoisonPill.Instance);

var msg = probe.ExpectMsg<Terminated>();
Assert.Equal(msg.ActorRef, target);
```

Replying to Messages Received by Probes

The probes stores the sender of the last dequeued message (i.e. after its `ExpectMsg*` reception), which may be retrieved using the `GetLastSender()` method. This information can also implicitly be used for having the probe reply to the last received message:

[!code-csharpRepliesToProbeMessages]

Forwarding Messages Received by Probes

The probe can also forward a received message (i.e. after its `ExpectMsg*` reception), retaining the original sender:

[!code-csharpForwardingProbeMessages]

Auto-Pilot

Receiving messages in a queue for later inspection is nice, but in order to keep a test running and verify traces later you can also install an `AutoPilot` in the participating test probes (actually in any `TestKit`) which is invoked before enqueueing to the inspection queue. This code can be used to forward messages, e.g. in a chain `A --> Probe --> B`, as long as a certain protocol is obeyed.

[!code-csharpProbeAutopilot]

The `run` method must return the auto-pilot for the next message. There are multiple options here: You can return the `AutoPilot.NoAutoPilot` to stop the autopilot, or `AutoPilot.KeepRunning` to keep using the current `AutoPilot`. Obviously you can also chain a new `AutoPilot` instance to switch behaviors.

Caution about Timing Assertions

The behavior of `Within` blocks when using test probes might be perceived as counter-intuitive: you need to remember that the nicely scoped deadline as described **above** is local to each probe. Hence, probes do not react to each other's deadlines or to the deadline set in an enclosing `TestKit` instance.

Testing parent-child relationships

The parent of an actor is always the actor that created it. At times this leads to a coupling between the two that may not be straightforward to test. There are several approaches to improve testability of a child actor that needs to refer to its parent:

1. When creating a child, pass an explicit reference to its parent
2. Create the child with a `TestProbe` as parent
3. Create a fabricated parent when testing

Conversely, a parent's binding to its child can be lessened as follows:

1. When creating a parent, tell the parent how to create its child.

For example, the structure of the code you want to test may follow this pattern:

[!code-csharpParentStructure]

Introduce child to its parent

The first option is to avoid use of the `context.parent` function and create a child with a custom parent by passing an explicit reference to its parent instead.

[!code-csharpDependentChild]

Using a fabricated parent

If you prefer to avoid modifying the parent or child constructor you can create a fabricated parent in your test. This, however, does not enable you to test the parent actor in isolation.

[!code-csharpFabrikatedParent]

Externalize child making from the parent

Alternatively, you can tell the parent how to create its child. There are two ways to do this: by giving it a `Props` object or by giving it a function which takes care of creating the child actor:

[!code-csharpFabrikatedParent]

Creating the `Props` is straightforward and the function may look like this in your test code:

```
Func<IUntypedActorContext, IActorRef> maker = (ctx) => probe.Ref;
var parent = Sys.ActorOf(Props.Create<GenericDependentParent>(maker));
```

And like this in your application code:

```
Func<IUntypedActorContext, IActorRef> maker = (ctx) => ctx.ActorOf(Props.Create<Child>());
var parent = Sys.ActorOf(Props.Create<GenericDependentParent>(maker));
```

Which of these methods is the best depends on what is most important to test. The most generic option is to create the parent actor by passing it a function that is responsible for the Actor creation, but the fabricated parent is often sufficient.

Calling Thread Dispatcher

The `CallingThreadDispatcher` serves good purposes in unit testing, as described above, but originally it was conceived in order to allow contiguous stack traces to be generated in case of an error. As this special dispatcher runs everything which would normally be queued directly on the current thread, the full history of a message's processing chain is recorded on the call stack, so long as all intervening actors run on this dispatcher.

How to use it

Just set the dispatcher as you normally would

```
Sys.ActorOf(Props.Create<MyActor>().WithDispatcher(CallingThreadDispatcher.Id));
```

How it works

When receiving an invocation, the `CallingThreadDispatcher` checks whether the receiving actor is already active on the current thread. The simplest example for this situation is an actor which sends a message to itself. In this case, processing cannot continue immediately as that would violate the actor model, so the invocation is queued and will be processed when the active invocation on that actor finishes its processing; thus, it will be processed on the calling thread, but simply after the actor finishes its previous work. In the other case, the invocation is simply processed immediately on the current thread. Tasks scheduled via this dispatcher are also executed immediately.

This scheme makes the `CallingThreadDispatcher` work like a general purpose dispatcher for any actors which never block on external events.

In the presence of multiple threads it may happen that two invocations of an actor running on this dispatcher happen on two different threads at the same time. In this case, both will be processed directly on their respective threads, where both compete for the actor's lock and the loser has to wait. Thus, the actor model is left intact, but the price is loss of concurrency due to limited scheduling. In a sense this is equivalent to traditional mutex style concurrency.

The other remaining difficulty is correct handling of suspend and resume: when an actor is suspended, subsequent invocations will be queued in thread-local queues (the same ones used for queuing in the normal case). The call to resume, however, is done by one specific thread, and all other threads in the system will probably not be executing this specific actor, which leads to the problem that the thread-local queues cannot be emptied by their native threads. Hence, the thread calling resume will collect all currently queued invocations from all threads into its own queue and process them.

Benefits

To summarize, these are the features with the `CallingThreadDispatcher` has to offer:

- Deterministic execution of single-threaded tests while retaining nearly full actor semantics
- Full message processing history leading up to the point of failure in exception stack traces
- Exclusion of certain classes of dead-lock scenarios

Tracing Actor Invocations

The testing facilities described up to this point were aiming at formulating assertions about a system's behavior. If a test fails, it is usually your job to find the cause, fix it and verify the test again. This process is supported by debuggers as well as logging, where the Akka.NET offers the following options:

- Logging of exceptions thrown within Actor instances. This is always on; in contrast to the other logging mechanisms, this logs at *ERROR* level.
- Logging of special messages. Actors handle certain special messages automatically, e.g. `Kill`, `PoisonPill`, etc. Tracing of these message invocations is enabled by the setting `akka.actor.debug.autoreceive`, which enables

this on all actors.

- Logging of the actor lifecycle. Actor creation, start, restart, monitor start, monitor stop and stop may be traced by enabling the setting `akka.actor.debug.lifecycle`; this, too, is enabled uniformly on all actors.

All these messages are logged at `DEBUG` level. To summarize, you can enable full logging of actor activities using this configuration fragment:

```
akka {  
    loglevel = "DEBUG"  
    actor {  
        debug {  
            autoreceive = on  
            lifecycle = on  
        }  
    }  
}
```

Configuration

There are several configuration properties for the TestKit module, please refer to the [reference configuration](#)

TODO describe how to pass custom config

Synchronous Testing: `TestActorRef`

Testing the business logic inside `Actor` classes can be divided into two parts: first, each atomic operation must work in isolation, then sequences of incoming events must be processed correctly, even in the presence of some possible variability in the ordering of events. The former is the primary use case for single-threaded unit testing, while the latter can only be verified in integration tests.

Normally, the `IActorRef` shields the underlying `Actor` instance from the outside, the only communications channel is the actor's mailbox. This restriction is an impediment to unit testing, which led to the inception of the `TestActorRef`. This special type of reference is designed specifically for test purposes and allows access to the actor in two ways: either by obtaining a reference to the underlying actor instance, or by invoking or querying the actor's behaviour (receive). Each one warrants its own section below.

[!NOTE] It is highly recommended to stick to traditional behavioral testing (using messaging to ask the `Actor` to reply with the state you want to run assertions against), instead of using `TestActorRef` whenever possible.

Obtaining a Reference to an Actor

Having access to the actual `Actor` object allows application of all traditional unit testing techniques on the contained methods. Obtaining a reference is done like this:

```
var props = Props.Create<MyActor>();  
var myTestActor = new TestActorRef<MyActor>(Sys, props, null, "testA");  
MyActor myActor = myTestActor.UnderlyingActor;
```

Since `TestActorRef` is generic in the actor type it returns the underlying actor with its proper static type. From this point on you may bring any unit testing tool to bear on your actor as usual.

Testing Finite State Machines

If your actor under test is a `FSM`, you may use the special `TestFSMRef` which offers all features of a normal `TestActorRef` and in addition allows access to the internal state:

```
var fsm = new TestFSMRef<TestFsmActor, int, string>();

Assert.True(fsm.StateName == 1);
Assert.True(fsm.StateData == "");

fsm.Tell("go"); //being a TestActorRef, this runs on the CallingThreadDispatcher

Assert.True(fsm.StateName == 2);
Assert.True(fsm.StateData == "go");

fsm.SetState(1);
Assert.True(fsm.StateName == 1);

Assert.False(fsm.IsTimerActive("test"));
fsm.SetTimer("test", 12, 10.Milliseconds(), true);
Assert.True(fsm.IsTimerActive("test"));
fsm.CancelTimer("test");
Assert.False(fsm.IsTimerActive("test"));
```

All methods shown above directly access the FSM state without any synchronization; this is perfectly alright if the `CallingThreadDispatcher` is used and no other threads are involved, but it may lead to surprises if you were to actually exercise timer events, because those are executed on the `Scheduler` thread.

Testing the Actor's behavior

When the dispatcher invokes the processing behavior of an actor on a message, it actually calls apply on the current behavior registered for the actor. This starts out with the return value of the declared receive method, but it may also be changed using become and unbecome in response to external messages. All of this contributes to the overall actor behavior and it does not lend itself to easy testing on the `Actor` itself. Therefore the `TestActorRef` offers a different mode of operation to complement the `Actor` testing: it supports all operations also valid on normal `IActorRef`. Messages sent to the actor are processed synchronously on the current thread and answers may be sent back as usual. This trick is made possible by the `CallingThreadDispatcher` described below; this dispatcher is set implicitly for any actor instantiated into a `TestActorRef`.

```
var props = Props.Create<MyActor>();
var myTestActor = new TestActorRef<MyActor>(Sys, props, null, "testB");
Task<int> future = myTestActor.Ask<int>("say42", TimeSpan.FromMilliseconds(3000));
Assert.True(future.IsCompleted);
Assert.Equal(42, await future);
```

As the `TestActorRef` is a subclass of `LocalActorRef` with a few special extras, also aspects like supervision and restarting work properly, but beware that execution is only strictly synchronous as long as all actors involved use the `CallingThreadDispatcher`. As soon as you add elements which include more sophisticated scheduling you leave the realm of unit testing as you then need to think about asynchronicity again (in most cases the problem will be to wait until the desired effect had a chance to happen).

One more special aspect which is overridden for single-threaded tests is the `ReceiveTimeout`, as including that would entail asynchronous queuing of `ReceiveTimeout` messages, violating the synchronous contract.

The Way In-Between: Expecting Exceptions

If you want to test the actor behavior, including hotswapping, but without involving a dispatcher and without having the `TestActorRef` swallow any thrown exceptions, then there is another mode available for you: just use the `receive` method on `TestActorRef`, which will be forwarded to the underlying actor:

```
var props = Props.Create<MyActor>();
var myTestActor = new TestActorRef<MyActor>(Sys, props, null, "testB");
try
{
    myTestActor.Receive(new Exception("expected"));
}
catch (Exception e)
{
    Assert.Equal("expected", e.Message);
}
```

EventFilters

EventFilters are a tool use can use to scan and expect for LogEvents generated by your actors. Typically these are generated by custom calls on the `context.GetLogger()` object, when you log something. However DeadLetter messages and Exceptions ultimately also result in a `LogEvent` message being generated.

These are all things that can be intercepted, and asserted upon using the `EventFilter`. An example of how you can get a reference to the `EventFilter`

```
var filter = CreateEventFilter(Sys);

filter.DeadLetter<string>().ExpectOne(() =>
{
    //cause a message to be deadlettered here
});

filter.Custom(logEvent => logEvent is Error && (string)logEvent.Message == "whatever").ExpectOne(() =>
{
    Log.Error("whatever");
});

filter.Exception<MyException>().ExpectOne(() => Log.Error(new MyException(), "the message"));
```

Coordinated Shutdown

There's an `ActorSystem` extension called `CoordinatedShutdown` that will stop certain Akka.NET actors / services and execute tasks in a programmable order during shutdown.

The default phases and their orderings are defined in the default HOCON configuration as `akka.coordinated-shutdown.phases`, and they are defined below:

```
phases {

    # The first pre-defined phase that applications can add tasks to.
    # Note that more phases can be added in the application's
    # configuration by overriding this phase with an additional
    # depends-on.
    before-service-unbind {

    }

    # Stop accepting new incoming requests in for example HTTP.
    service-unbind {
        depends-on = [before-service-unbind]
    }

    # Wait for requests that are in progress to be completed.
    service-requests-done {
        depends-on = [service-unbind]
    }

    # Final shutdown of service endpoints.
    service-stop {
        depends-on = [service-requests-done]
    }

    # Phase for custom application tasks that are to be run
    # after service shutdown and before cluster shutdown.
    before-cluster-shutdown {
        depends-on = [service-stop]
    }

    # Graceful shutdown of the Cluster Sharding regions.
    cluster-sharding-shutdown-region {
        timeout = 10 s
        depends-on = [before-cluster-shutdown]
    }

    # Emit the leave command for the node that is shutting down.
    cluster-leave {
        depends-on = [cluster-sharding-shutdown-region]
    }

    # Shutdown cluster singletons
    cluster-exiting {
        timeout = 10 s
        depends-on = [cluster-leave]
    }

    # Wait until exiting has been completed
    cluster-exiting-done {
        depends-on = [cluster-exiting]
    }

    # Shutdown the cluster extension
    cluster-shutdown {
        depends-on = [cluster-exiting-done]
    }
}
```

```

# Phase for custom application tasks that are to be run
# after cluster shutdown and before ActorSystem termination.
before-actor-system-terminate {
    depends-on = [cluster-shutdown]
}

# Last phase. See terminate-actor-system and exit-jvm above.
# Don't add phases that depends on this phase because the
# dispatcher and scheduler of the ActorSystem have been shutdown.
actor-system-terminate {
    timeout = 10 s
    depends-on = [before-actor-system-terminate]
}
}

```

Custom Phases

As an end-user, you can register tasks to execute during any of these shutdown phases and add additional phases if you wish.

More phases can be added to an application by overriding the HOCON of an existing phase to include additional members in its `phase.depends-on` property. Here's an example where an additional phase might be executing before shutting down the cluster, for instance:

```

akka.coordinated-shutdown.phases.before-cluster-shutdown.depends-on = [service-stop, my-phase]
my-phase{
    timeout = 10s
    recover = on
}

```

For each phase, the following properties can be set:

1. `depends-on` - specifies which other phases have to run successfully first before this phase can begin;
2. `timeout` - specifies how long the tasks in this phase are allowed to run before timing out;
3. `recover` - if set to `off`, if any of the tasks in this phase throw an exception then the `CoordinatedShutdown` will be aborted and no further phases will be run. If set to `on` then any thrown errors are suppressed and the system will continue to execute the `CoordinatedShutdown`.

The default phases are defined in a linear order, but in practice the phases are ordered into a directed acyclic graph (DAG) using [Topological sort](#).

Registering Tasks to a Phase

For instance, if you're using [Akka.Cluster](#) it's commonplace to register application-specific cleanup tasks during the `cluster-leave` and `cluster-exiting` phases. Here's an example:

```

var coordShutdown = CoordinatedShutdown.Get(myActorSystem);
coordShutdown.AddTask(CoordinatedShutdown.PhaseClusterLeave, "cleanup-my-api", () =>
{
    return _myCustomSocketApi.CloseAsync().ContinueWith(tr => Done.Instance);
});

```

Each shutdown task added to a phase must specify a function that returns a value of type `Task<Done>`. This function will be invoked once the `CoordinatedShutdown.Run()` command is executed and the `Task<Done>` returned by the function will be completed in parallel with all other tasks executing in the given phase. Those tasks may complete in

any possible order; `CoordinatedShutdown` doesn't attempt to order the execution of tasks within a single phase. Once all of those tasks have completed OR if the phases timeout has expired, the next phase will begin.

Tasks should be registered as early as possible, preferably at system startup, in order to ensure that all registered tasks are run. If tasks are added after the `CoordinatedShutdown` have begun its run, it's possible that the newly registered tasks will not be executed.

Running `CoordinatedShutdown`

There are a few different ways to start the `CoordinatedShutdown` process.

If you wish to execute the `CoordinatedShutdown` yourself, you can simply call `CoordinatedShutdown.Run()`, which will return a `Task<Done>`.

```
CoordinatedShutdown.Get(myActorSystem).Run();
```

It's safe to call this method multiple times as the shutdown process will only be run once and will return the same completion task each time. The `Task<Done>` will complete once all phases have run successfully, or a phase with `recover = off` failed.

Automatic `ActorSystem` and Process Termination

By default, when the final phase of the `CoordinatedShutdown` executes the calling `ActorSystem` will be terminated. However, the CLR process will still be running even though the `ActorSystem` has been terminated.

If you'd like to automatically terminate the process running your `ActorSystem`, you can set the following HOCON value in your configuration:

```
akka.coordinated-shutdown.exit-clr = on
```

If this setting is enabled (it's disabled by default), you'll be able to shutdown the current running process automatically via an `Environment.Exit(0)` call made during the final phase of the `CoordinatedShutdown`.

`CoordinatedShutdown` and `Akka.Cluster`

If you're using `Akka.Cluster`, the `CoordinatedShutdown` will automatically register tasks for completing the following:

1. Gracefully leaving the cluster;
2. Gracefully handing over / terminating `ClusterSingleton` and `Cluster.Sharding` instances; and
3. Terminating the `Cluster` system itself.

By default, this graceful leave action will be triggered whenever the `CoordinatedShutdown.Run()` method is called. Conversely, calling `Cluster.Leave` on a cluster member will also cause the `CoordinatedShutdown` to run and will terminate the `ActorSystem` once the node has left the cluster.

By default, `CoordinatedShutdown.Run()` will also be executed if a node is removed via `Cluster.Down` (non-graceful exit), but this can be disabled by changing the following `Akka.Cluster` HOCON setting:

```
akka.run-coordinated-shutdown-when-down = off
```

Invoking `CoordinatedShutdown.Run()` on Process Exit

By default `CoordinatedShutdown.Run()` will be called whenever the current process attempts to exit (using the `AppDomain.ProcessExit` event hook) and this will give the `ActorSystem` and the underlying clustering tools an opportunity to cleanup gracefully before the process finishes exiting.

If you wish to disable this behavior, you can pass in the following HOCON configuration value:

```
akka.coordinated-shutdown.run-by-clr-shutdown-hook = off
```

Persistence

Akka.Persistence plugin enables stateful actors to persist their internal state so that it can be recovered when an actor is started, restarted after a CLR crash or by a supervisor, or migrated in a cluster. The key concept behind Akka persistence is that only changes to an actor's internal state are persisted but never its current state directly (except for optional snapshots). These changes are only ever appended to storage, nothing is ever mutated, which allows for very high transaction rates and efficient replication. Stateful actors are recovered by replaying stored changes to these actors from which they can rebuild internal state. This can be either the full history of changes or starting from a snapshot which can dramatically reduce recovery times. Akka persistence also provides point-to-point communication with at-least-once message delivery semantics.

Architecture

Akka.Persistence features are available through new set of actor base classes:

- `UntypedPersistentActor` - is a persistent, stateful actor. It is able to persist events to a journal and can react to them in a thread-safe manner. It can be used to implement both command as well as event sourced actors. When a persistent actor is started or restarted, journaled messages are replayed to that actor so that it can recover internal state from these messages.
- `PersistentView` is a persistent, stateful actor that receives journaled messages that have been written by another persistent actor. A view itself does not journal new messages, instead, it updates internal state only from a persistent actor's replicated message stream. Note: `PersistentView` is deprecated.
- `AtLeastOnceDeliveryActor` - is an actor which sends messages with at-least-once delivery semantics to destinations, also in case of sender and receiver CLR crashes.
- `AsyncWriteJournal` stores the sequence of messages sent to a persistent actor. An application can control which messages are journaled and which are received by the persistent actor without being journaled. Journal maintains `highestSequenceNr` that is increased on each message. The storage backend of a journal is pluggable. By default it uses an in-memory message stream and is NOT a persistent storage.
- `SnapshotStore` is used to persist snapshots of either persistent actor's or view's internal state. They can be used to reduce recovery times in case when a lot of events needs to be replayed for specific persistent actor. Storage backend of the snapshot store is pluggable. By default it uses local file system.

Receive Actors

- `ReceivePersistentActor` - receive actor version of `UntypedPersistentActor`.
- `AtLeastOnceDeliveryReceiveActor` - receive actor version of `AtLeastOnceDeliveryActor`.

Persistent Actors

Unlike the default `UntypedActor` class, `UntypedPersistentActor` and its derivatives requires the setup of a few more additional members:

- `PersistenceId` is a persistent actor's identifier that doesn't change across different actor incarnations. It's used to retrieve an event stream required by the persistent actor to recover its internal state.
- `OnRecover` is a method invoked during an actor's recovery cycle. Incoming objects may be user-defined events as well as system messages, for example `SnapshotOffer` which is used to deliver latest actor state saved in the snapshot store.
- `OnCommand` is an equivalent of the basic `Receive` method of default Akka.NET actors.

Persistent actors also offer a set of specialized members:

- `Persist` and `PersistAsync` methods can be used to send events to the event journal in order to store them inside. The second argument is a callback invoked when the journal confirms that events have been stored successfully.
- `DeferAsync` is used to perform various operations *after* events will be persisted and their callback handlers will be invoked. Unlike the persist methods, defer won't store an event in persistent storage. Defer method may NOT be invoked in case when the actor is restarted even though the journal will successfully persist events sent.
- `DeleteMessages` will order attached journal to remove part of its events. It can be only physical deletion, when the messages are removed physically from the journal.
- `LoadSnapshot` will send a request to the snapshot store to resend the current actor's snapshot.
- `SaveSnapshot` will send the current actor's internal state as a snapshot to be saved by the configured snapshot store.
- `DeleteSnapshot` and `DeleteSnapshots` methods may be used to specify snapshots to be removed from the snapshot store in cases where they are no longer needed.
- `OnReplaySuccess` is a virtual method which will be called when the recovery cycle ends successfully.
- `OnReplayFailure` is a virtual method which will be called when the recovery cycle fails unexpectedly from some reason.
- `IsRecovering` property determines if the current actor is performing a recovery cycle at the moment.
- `SnapshotSequenceNr` property may be used to determine the sequence number used for marking persisted events. This value changes in a monotonically increasing manner.

In case a manual recovery cycle initialization is necessary, it may be invoked by sending a `Recover` message to a persistent actor.

Event sourcing

The basic idea behind Event Sourcing is quite simple. A persistent actor receives a (non-persistent) command which is first validated if it can be applied to the current state. Here validation can mean anything from simple inspection of a command message's fields up to a conversation with several external services, for example. If validation succeeds, events are generated from the command, representing the effect of the command. These events are then persisted and, after successful persistence, used to change the actor's state. When the persistent actor needs to be recovered, only the persisted events are replayed of which we know that they can be successfully applied. In other words, events cannot fail when being replayed to a persistent actor, in contrast to commands. Event sourced actors may of course also process commands that do not change application state such as query commands for example.

Akka persistence supports event sourcing with the `UntypedPersistentActor` abstract class. An actor that extends this class uses the `persist` method to persist and handle events. The behavior of an `UntypedPersistentActor` is defined by implementing `OnRecover` and `OnCommand` methods. This is demonstrated in the following example.

[!code-csharpMain]

```

public class Cmd
{
    public Cmd(string data)
    {
        Data = data;
    }

    public string Data { get; }
}

public class Evt
{
    public Evt(string data)
    {
        Data = data;
    }

    public string Data { get; }
}

public class ExampleState
{
    private readonly ImmutableList<string> _events;

    public ExampleState(ImmutableList<string> events)
    {
        _events = events;
    }

    public ExampleState() : this(ImmutableList.Create<string>())
    {

    }

    public ExampleState Updated(Evt evt)
    {
        return new ExampleState(_events.Add(evt.Data));
    }

    public int Size => _events.Count;

    public override string ToString()
    {
        return string.Join(", ", _events.Reverse());
    }
}

```

```

    }

    public class PersistentActor : UntypedPersistentActor
    {
        private ExampleState _state = new ExampleState();

        private void UpdateState(Evt evt)
        {
            _state = _state.Updated(evt);
        }

        private int NumEvents => _state.Size;

        protected override void OnRecover(object message)
        {
            switch (message)
            {
                case Evt evt:
                    UpdateState(evt);
                    break;
                case SnapshotOffer snapshot when snapshot.Snapshot is ExampleState:
                    _state = (ExampleState)snapshot.Snapshot;
                    break;
            }
        }

        protected override void OnCommand(object message)
        {
            switch (message)
            {
                case Cmd cmd:
                    Persist(new Evt($"{cmd.Data}-{NumEvents}"), UpdateState);
                    Persist(new Evt($"{cmd.Data}-{NumEvents + 1}"), evt =>
                    {
                        UpdateState(evt);
                        Context.System.EventStream.Publish(evt);
                    });
                    break;
                case "snap":
                    SaveSnapshot(_state);
                    break;
                case "print":
                    Console.WriteLine(_state);
                    break;
            }
        }

        public override string PersistenceId { get; } = "sample-id-1";
    }
}

```

The example defines two data types, `Cmd` and `Evt` to represent commands and events, respectively. The state of the `PersistentActor` is a list of persisted event data contained in `ExampleState`.

The persistent actor's `OnRecover` method defines how state is updated during recovery by handling `Evt` and `SnapshotOffer` messages. The persistent actor's `OnCommand` method is a command handler. In this example, a command is handled by generating two events which are then persisted and handled. Events are persisted by calling `Persist` with an event (or a sequence of events) as first argument and an event handler as second argument.

The `Persist` method persists events asynchronously and the event handler is executed for successfully persisted events. Successfully persisted events are internally sent back to the persistent actor as individual messages that trigger event handler executions. An event handler may close over persistent actor state and mutate it. The sender of a persisted event is the sender of the corresponding command. This allows event handlers to reply to the sender of a command (not shown).

The main responsibility of an event handler is changing persistent actor state using event data and notifying others about successful state changes by publishing events.

When persisting events with `Persist` it is guaranteed that the persistent actor will not receive further commands between the `Persist` call and the execution(s) of the associated event handler. This also holds for multiple persist calls in context of a single command. Incoming messages are stashed until the `Persist` is completed.

If persistence of an event fails, `OnPersistFailure` will be invoked (logging the error by default), and the actor will unconditionally be stopped. If persistence of an event is rejected before it is stored, e.g. due to serialization error, `OnPersistRejected` will be invoked (logging a warning by default), and the actor continues with the next message.

[!NOTE] It's also possible to switch between different command handlers during normal processing and recovery with `Context.Become` and `Context.Unbecome`. To get the actor into the same state after recovery you need to take special care to perform the same state transitions with become and unbecome in the `OnRecover` method as you would have done in the command handler. Note that when using become from `OnRecover` it will still only use the `OnRecover` behavior when replaying the events. When replay is completed it will use the new behavior.

Identifiers

A persistent actor must have an identifier that doesn't change across different actor incarnations. The identifier must be defined with the `PersistenceId` method.

```
public override string PersistenceId { get; } = "my-stable-persistence-id";
```

[!NOTE] `PersistenceId` must be unique to a given entity in the journal (database table/keyspace). When replaying messages persisted to the journal, you query messages with a `PersistenceId`. So, if two different entities share the same `PersistenceId`, message-replaying behavior is corrupted.

Recovery

By default, a persistent actor is automatically recovered on start and on restart by replaying journaled messages. New messages sent to a persistent actor during recovery do not interfere with replayed messages. They are stashed and received by a persistent actor after recovery phase completes.

The number of concurrent recoveries of recoveries that can be in progress at the same time is limited to not overload the system and the backend data store. When exceeding the limit the actors will wait until other recoveries have been completed. This is configured by:

```
akka.persistence.max-concurrent-recoveries = 50
```

[!NOTE] Accessing the `Sender` for replayed messages will always result in a `DeadLetters` reference, as the original sender is presumed to be long gone. If you indeed have to notify an actor during recovery in the future, store its `ActorPath` explicitly in your persisted events.

Recovery customization

Applications may also customise how recovery is performed by returning a customised `Recovery` object in the recovery method of a `UntypedPersistentActor`.

To skip loading snapshots and replay all events you can use `SnapshotSelectionCriteria.None`. This can be useful if snapshot serialization format has changed in an incompatible way. It should typically not be used when events have been deleted.

```
public override Recovery Recovery => new Recovery(fromSnapshot: SnapshotSelectionCriteria.None)
```

Another example, which can be fun for experiments but probably not in a real application, is setting an upper bound to the replay which allows the actor to be replayed to a certain point "in the past" instead to its most up to date state. Note that after that it is a bad idea to persist new events because a later recovery will probably be confused by the new events that follow the events that were previously skipped.

```
public override Recovery Recovery => new Recovery(new SnapshotSelectionCriteria(457));
```

Recovery can be disabled by returning `Recovery.None` in the recovery property of a `UntypedPersistentActor`:

```
public override Recovery Recovery => Recovery.None;
```

Recovery status

A persistent actor can query its own recovery status via the methods

```
public bool IsRecovering { get; }
public bool IsRecoveryFinished { get; }
```

Sometimes there is a need for performing additional initialization when the recovery has completed before processing any other message sent to the persistent actor. The persistent actor will receive a special `RecoveryCompleted` message right after recovery and before any other received messages.

```
protected override void OnRecover(object message)
{
    if (message is RecoveryCompleted)
    {
        // perform init after recovery, before any other messages
    }
    else
    {
        // ...
    }
}

protected override void OnCommand(object message)
{
    // ...
}
```

The actor will always receive a `RecoveryCompleted` message, even if there are no events in the journal and the snapshot store is empty, or if it's a new persistent actor with a previously unused `PersistenceID`.

If there is a problem with recovering the state of the actor from the journal, `OnRecoveryFailure` is called (logging the error by default) and the actor will be stopped.

Internal stash

The persistent actor has a private stash for internally caching incoming messages during `Recovery` or the `Persist \ PersistAll` method persisting events. However You can use inherited stash or create one or more stashes if needed. The internal stash doesn't interfere with these stashes apart from user inherited `UnstashAll` method, which prepends all messages in the inherited stash to the internal stash instead of mailbox. Hence, If the message in the inherited stash need to be handled after the messages in the internal stash, you should call inherited unstash method.

You should be careful to not send more messages to a persistent actor than it can keep up with, otherwise the number of stashed messages will grow. It can be wise to protect against `OutOfMemoryException` by defining a maximum stash capacity in the mailbox configuration:

```
akka.actor.default-mailbox.stash-capacity = 10000
```

Note that the stash capacity is per actor. If you have many persistent actors, e.g. when using cluster sharding, you may need to define a small stash capacity to ensure that the total number of stashed messages in the system doesn't consume too much memory. Additionally, the persistent actor defines three strategies to handle failure when the internal stash capacity is exceeded. The default overflow strategy is the `ThrowOverflowExceptionStrategy`, which discards the current received message and throws a `StashOverflowException`, causing actor restart if the default supervision strategy is used. You can override the `InternalStashOverflowStrategy` property to return `DiscardToDeadLetterStrategy` or `ReplyToStrategy` for any "individual" persistent actor, or define the "default" for all persistent actors by providing FQCN, which must be a subclass of `stashOverflowStrategyConfigurator`, in the persistence configuration:

```
akka.persistence.internal-stash-overflow-strategy = "akka.persistence.ThrowExceptionConfigurator"
```

The `DiscardToDeadLetterStrategy` strategy also has a pre-packaged companion configurator `DiscardConfigurator`.

You can also query the default strategy via the Akka persistence extension singleton:

```
Context.System.DefaultInternalStashOverflowStrategy
```

[!NOTE] The bounded mailbox should be avoided in the persistent actor, because it may be discarding the messages come from Storage backends. You can use bounded stash instead of bounded mailbox.

Relaxed local consistency requirements and high throughput use-cases

If faced with relaxed local consistency requirements and high throughput demands sometimes `UntypedPersistentActor` and its `persist` may not be enough in terms of consuming incoming Commands at a high rate, because it has to wait until all Events related to a given Command are processed in order to start processing the next Command. While this abstraction is very useful for most cases, sometimes you may be faced with relaxed requirements about consistency – for example you may want to process commands as fast as you can, assuming that the Event will eventually be persisted and handled properly in the background, retroactively reacting to persistence failures if needed.

The `PersistAsync` method provides a tool for implementing high-throughput persistent actors. It will not stash incoming Commands while the Journal is still working on persisting and/or user code is executing event callbacks.

In the below example, the event callbacks may be called "at any time", even after the next Command has been processed. The ordering between events is still guaranteed ("evt-b-1" will be sent after "evt-a-2", which will be sent after "evt-a-1" etc.).

[!NOTE] In order to implement the pattern known as "command sourcing" simply `PersistAsync` all incoming messages right away and handle them in the callback.

[!code-csharpMain]

```
public class MyPersistentActor : UntypedPersistentActor
{
    public override string PersistenceId => "my-stable-persistence-id";

    protected override void OnRecover(object message)
    {
        // handle recovery here
    }

    protected override void OnCommand(object message)
    {
        if (message is string c)
        {
            Sender.Tell(c);
            Persist($"evt-{c}-1", e => Sender.Tell(e));
            Persist($"evt-{c}-2", e => Sender.Tell(e));
            DeferAsync($"evt-{c}-3", e => Sender.Tell(e));
        }
    }

    public static void MainApp()
    {
        var system = ActorSystem.Create("PersistAsync");
        var persistentActor = system.ActorOf<MyPersistentActor>();

        // usage
        persistentActor.Tell("a");
        persistentActor.Tell("b");

        // possible order of received messages:
        // a
        // b
        // evt-a-1
        // evt-a-2
        // evt-b-1
        // evt-b-2
    }
}
```

[!WARNING] The callback will not be invoked if the actor is restarted (or stopped) in between the call to `PersistAsync` and the journal has confirmed the write.

Deferring actions until preceding persist handlers have executed

Sometimes when working with `PersistAsync` or `Persist` you may find that it would be nice to define some actions in terms of happens-after the previous `PersistAsync` / `Persist` handlers have been invoked. `PersistentActor` provides an utility method called `DeferAsync`, which works similarly to `PersistAsync` yet does not persist the passed in event. It is recommended to use it for read operations, and actions which do not have corresponding events in your domain model.

Using this method is very similar to the `persist` family of methods, yet it does **not** persist the passed in event. It will be kept in memory and used when invoking the handler.

[!code-csharpMain]

```

public class MyPersistentActor : UntypedPersistentActor
{
    public override string PersistenceId => "my-stable-persistence-id";

    protected override void OnRecover(object message)
    {
        // handle recovery here
    }

    protected override void OnCommand(object message)
    {
        if (message is string c)
        {
            Sender.Tell(c);
            PersistAsync($"evt-{c}-1", e => Sender.Tell(e));
            PersistAsync($"evt-{c}-2", e => Sender.Tell(e));
            DeferAsync($"evt-{c}-3", e => Sender.Tell(e));
        }
    }
}

```

Notice that the `Sender` is safe to access in the handler callback, and will be pointing to the original sender of the command for which this `DeferAsync` handler was called.

The calling side will get the responses in this (guaranteed) order:

```

persistentActor.tell("a");
persistentActor.tell("b");

// order of received messages:
// a
// b
// evt-a-1
// evt-a-2
// evt-a-3
// evt-b-1
// evt-b-2
// evt-b-3

```

You can also call `DeferAsync` with `Persist`. [!code-csharpMain]

```

public class MyPersistentActor : UntypedPersistentActor
{
    public override string PersistenceId => "my-stable-persistence-id";

    protected override void OnRecover(object message)
    {
        // handle recovery here
    }

    protected override void OnCommand(object message)
    {
        if (message is string c)
        {
            Sender.Tell(c);
            Persist($"evt-{c}-1", e => Sender.Tell(e));
            Persist($"evt-{c}-2", e => Sender.Tell(e));
            DeferAsync($"evt-{c}-3", e => Sender.Tell(e));
        }
    }
}

```

[!WARNING] The callback will not be invoked if the actor is restarted (or stopped) in between the call to `DeferAsync` and the journal has processed and confirmed all preceding writes..

Nested persist calls

It is possible to call `Persist` and `PersistAsync` inside their respective callback blocks and they will properly retain both the thread safety (including the right value of `Sender`) as well as stashing guarantees.

In general it is encouraged to create command handlers which do not need to resort to nested event persisting, however there are situations where it may be useful. It is important to understand the ordering of callback execution in those situations, as well as their implication on the stashing behavior (that persist enforces). In the following example two persist calls are issued, and each of them issues another persist inside its callback:

[!code-csharpMain]

```
public class MyPersistentActor : UntypedPersistentActor
{
    public override string PersistenceId => "my-stable-persistence-id";

    protected override void OnRecover(object message)
    {
        // handle recovery here
    }

    protected override void OnCommand(object message)
    {
        if (message is string c)
        {
            Sender.Tell(c);

            Persist($"{message}-1-outer", outer1 =>
            {
                Sender.Tell(outer1, Self);
                Persist($"{c}-1-inner", inner1 => Sender.Tell(inner1));
            });

            Persist($"{message}-2-outer", outer2 =>
            {
                Sender.Tell(outer2, Self);
                Persist($"{c}-2-inner", inner2 => Sender.Tell(inner2));
            });
        }
    }
}
```

When sending two commands to this `UntypedPersistentActor`, the persist handlers will be executed in the following order:

[!code-csharpMain]

```
public static void MainApp()
{
    var system = ActorSystem.Create("NestedPersists");
    var persistentActor = system.ActorOf<MyPersistentActor>();

    persistentActor.Tell("a");
    persistentActor.Tell("b");

    // order of received messages:
    // a
    // a-outer-1
    // a-outer-2
```

```
// a-inner-1
// a-inner-2
// and only then process "b"
// b
// b-outer-1
// b-outer-2
// b-inner-1
// b-inner-2
}
```

First the "outer layer" of persist calls is issued and their callbacks are applied. After these have successfully completed, the inner callbacks will be invoked (once the events they are persisting have been confirmed to be persisted by the journal). Only after all these handlers have been successfully invoked will the next command be delivered to the persistent Actor. In other words, the stashing of incoming commands that is guaranteed by initially calling `Persist` on the outer layer is extended until all nested persist callbacks have been handled.

It is also possible to nest `PersistAsync` calls, using the same pattern:

[!code-csharp>Main]

```
"cs public class MyPersistentActor : UntypedPersistentActor { public override string PersistenceId => "my-stable-persistence-id";
```

```
protected override void OnRecover(object message)
{
    // handle recovery here
}

protected override void OnCommand(object message)
{
    if (message is string c)
    {
        Sender.Tell(c);

        PersistAsync($"{message}-1-outer", outer1 =>
    {
        Sender.Tell(outer1, Self);
        PersistAsync($"{c}-1-inner", inner1 => Sender.Tell(inner1));
    });

        PersistAsync($"{message}-2-outer", outer2 =>
    {
        Sender.Tell(outer2, Self);
        PersistAsync($"{c}-2-inner", inner2 => Sender.Tell(inner2));
    });
    }
}
}
```

In this case no stashing is happening, yet events are still persisted and callbacks are executed in the expected order:

```
[!code-csharp[Main](../../../../examples/DocsExamples/Persistence/PersistentActor/NestedPersistsAsync.cs?range=43-60)
]

```cs
public static void MainApp()
{
 var system = ActorSystem.Create("NestedPersistsAsync");
 var persistentActor = system.ActorOf<MyPersistentActor>();

 persistentActor.Tell("a");
 persistentActor.Tell("b");
```

```

 // order of received messages:
 // a
 // b
 // a-outer-1
 // a-outer-2
 // b-outer-1
 // b-outer-2
 // a-inner-1
 // a-inner-2
 // b-inner-1
 // b-inner-2

 // which can be seen as the following causal relationship:
 // a -> a-outer-1 -> a-outer-2 -> a-inner-1 -> a-inner-2
 // b -> b-outer-1 -> b-outer-2 -> b-inner-1 -> b-inner-2
}

```

While it is possible to nest mixed `Persist` and `PersistAsync` with keeping their respective semantics it is not a recommended practice, as it may lead to overly complex nesting.

[!WARNING] While it is possible to nest `Persist` calls within one another, it is not legal to call `Persist` from any other `Thread` than the Actors message processing `Thread`. For example, it is not legal to call `Persist` from `tasks!` Doing so will break the guarantees that the `Persist` methods aim to provide. Always call `Persist` and `PersistAsync` from within the Actor's receive block (or methods synchronously invoked from there).

## Failures

If persistence of an event fails, `OnPersistFailure` will be invoked (logging the error by default), and the actor will unconditionally be stopped.

The reason that it cannot resume when `Persist` fails is that it is unknown if the event was actually persisted or not, and therefore it is in an inconsistent state. Restarting on persistent failures will most likely fail anyway since the journal is probably unavailable. It is better to stop the actor and after a back-off timeout start it again. The `BackoffSupervisor` actor is provided to support such restarts.

```

protected override void PreStart()
{
 var childProps = Props.Create<PersistentActor>();
 var props = BackoffSupervisor.Props(
 Backoff.OnStop(
 childProps,
 "myActor",
 TimeSpan.FromSeconds(3),
 TimeSpan.FromSeconds(30),
 0.2));
 Context.ActorOf(props, name: "mySupervisor");
 base.PreStart();
}

```

If persistence of an event is rejected before it is stored, e.g. due to serialization error, `OnPersistRejected` will be invoked (logging a warning by default), and the actor continues with next message.

If there is a problem with recovering the state of the actor from the journal when the actor is started, `OnRecoveryFailure` is called (logging the error by default), and the actor will be stopped. Note that failure to load snapshot is also treated like this, but you can disable loading of snapshots if you for example know that serialization format has changed in an incompatible way, see [Recovery customization](#).

## Atomic writes

Each event is of course stored atomically, but it is also possible to store several events atomically by using the `PersistAll` or `PersistAllAsync` method. That means that all events passed to that method are stored or none of them are stored if there is an error.

The recovery of a persistent actor will therefore never be done partially with only a subset of events persisted by `PersistAll`.

Some journals may not support atomic writes of several events and they will then reject the `PersistAll` command, i.e. `OnPersistRejected` is called with an exception (typically `NotSupportedException`).

## Batch writes

In order to optimize throughput when using `PersistAsync`, a persistent actor internally batches events to be stored under high load before writing them to the journal (as a single batch). The batch size is dynamically determined by how many events are emitted during the time of a journal round-trip: after sending a batch to the journal no further batch can be sent before confirmation has been received that the previous batch has been written. Batch writes are never timer-based which keeps latencies at a minimum.

## Message deletion

It is possible to delete all messages (journalized by a single persistent actor) up to a specified sequence number; Persistent actors may call the `DeleteMessages` method to this end.

Deleting messages in event sourcing based applications is typically either not used at all, or used in conjunction with snapshotting, i.e. after a snapshot has been successfully stored, a `DeleteMessages` (`ToSequenceNr`) up until the sequence number of the data held by that snapshot can be issued to safely delete the previous events while still having access to the accumulated state during replays - by loading the snapshot.

[!WARNING] If you are using [Persistence Query](#), query results may be missing deleted messages in a journal, depending on how deletions are implemented in the journal plugin. Unless you use a plugin which still shows deleted messages in persistence query results, you have to design your application so that it is not affected by missing messages.

The result of the `DeleteMessages` request is signaled to the persistent actor with a `DeleteMessagesSuccess` message if the delete was successful or a `DeleteMessagesFailure` message if it failed.

Message deletion doesn't affect the highest sequence number of the journal, even if all messages were deleted from it after `DeleteMessages` invocation.

## Persistence status handling

Method	Success	Failure / Rejection	After failure handler invoked
Persist / PersistAsync	persist handler invoked	OnPersistFailure	Actor is stopped.
		OnPersistRejected	No automatic actions.
Recovery	RecoverySuccess	OnRecoveryFailure	Actor is stopped.
DeleteMessages	DeleteMessagesSuccess	DeleteMessagesFailure	No automatic actions.

The most important operations (Persist and Recovery) have failure handlers modelled as explicit callbacks which the user can override in the `UntypedPersistentActor`. The default implementations of these handlers emit a log message (error for persist/recovery failures, and warning for others), logging the failure cause and information about which message caused the failure.

For critical failures such as recovery or persisting events failing the persistent actor will be stopped after the failure handler is invoked. This is because if the underlying journal implementation is signalling persistence failures it is most likely either failing completely or overloaded and restarting right-away and trying to persist the event again will most likely not help the journal recover – as it would likely cause a Thundering herd problem, as many persistent actors would restart and try to persist their events again. Instead, using a `BackoffSupervisor` (as described in Failures) which implements an exponential-backoff strategy which allows for more breathing room for the journal to recover between restarts of the persistent actor.

[!NOTE] Journal implementations may choose to implement a retry mechanism, e.g. such that only after a write fails N number of times a persistence failure is signalled back to the user. In other words, once a journal returns a failure, it is considered fatal by Akka Persistence, and the persistent actor which caused the failure will be stopped. Check the documentation of the journal implementation you are using for details if/how it is using this technique.

## Safely shutting down persistent actors

Special care should be given when shutting down persistent actors from the outside. With normal Actors it is often acceptable to use the special `PoisonPill` message to signal to an Actor that it should stop itself once it receives this message – in fact this message is handled automatically by Akka, leaving the target actor no way to refuse stopping itself when given a poison pill.

This can be dangerous when used with `UntypedPersistentActor` due to the fact that incoming commands are stashed while the persistent actor is awaiting confirmation from the Journal that events have been written when `Persist` was used. Since the incoming commands will be drained from the Actor's mailbox and put into its internal stash while awaiting the confirmation (thus, before calling the persist handlers) the Actor may receive and (auto)handle the `PoisonPill` before it processes the other messages which have been put into its stash, causing a pre-mature shutdown of the Actor.

[!WARNING] Consider using explicit shut-down messages instead of `PoisonPill` when working with persistent actors.

The example below highlights how messages arrive in the Actor's mailbox and how they interact with its internal stashing mechanism when `Persist()` is used. Notice the early stop behavior that occurs when `PoisonPill` is used:

[!code-csharpMain]

```
public class Shutdown {}

public class SafePersistentActor : UntypedPersistentActor
{
 public override string PersistenceId => "safe-actor";

 protected override void OnRecover(object message)
 {
 // handle recovery here
 }

 protected override void OnCommand(object message)
 {
 if (message is string c)
 {
 Console.WriteLine(c);
 }
 }
}
```

```
Persist($"handle-{c}", param =>
{
 Console.WriteLine(param);
});

}

else if (message is Shutdown)
{
 Context.Stop(Self);
}

}
```

[!code-csharpMain]

```

public static void MainApp()
{
 var system = ActorSystem.Create("AvoidPoisonPill");
 var persistentActor = system.ActorOf<SafePersistentActor>();

 // UN-SAFE, due to PersistentActor's command stashing:
 persistentActor.Tell("a");
 persistentActor.Tell("b");
 persistentActor.Tell(PoisonPill.Instance);
 // order of received messages:
 // a
 // # b arrives at mailbox, stashing; internal-stash = [b]
 // # PoisonPill arrives at mailbox, stashing; internal-stash = [b, Shutdown]
 // PoisonPill is an AutoReceivedMessage, is handled automatically
 // !! stop !!
 // Actor is stopped without handling `b` nor the `a` handler!

 // SAFE:
 persistentActor.Tell("a");
 persistentActor.Tell("b");
 persistentActor.Tell(new Shutdown());
 // order of received messages:
 // a
 // # b arrives at mailbox, stashing; internal-stash = [b]
 // # Shutdown arrives at mailbox, stashing; internal-stash = [b, Shutdown]
 // handle-a
 // # unstashing; internal-stash = [Shutdown]
 // b
 // handle-b
 // # unstashing; internal-stash = []
 // Shutdown
 // -- stop --
}

```

## Replay filter

There could be cases where event streams are corrupted and multiple writers (i.e. multiple persistent actor instances) journaled different messages with the same sequence number. In such a case, you can configure how you filter replayed messages from multiple writers, upon recovery.

In your configuration, under the `akka.persistence.journal.xxx.replay-filter` section (where `xxx` is your journal plugin id), you can select the replay filter mode from one of the following values:

- repair-by-discard-old
  - fail
  - warn
  - off

For example, if you configure the replay filter for `sqlite` plugin, it looks like this:

```
The replay filter can detect a corrupt event stream by inspecting
sequence numbers and writerUuid when replaying events.
akka.persistence.journal.sqlite.replay-filter {
 # what the filter should do when detecting invalid events.
 # Supported values:
 # `repair-by-discard-old` : discard events from old writers,
 # warning is logged
 # `fail` : fail the replay, error is logged
 # `warn` : log warning but emit events untouched
 # `off` : disable this feature completely
 mode = repair-by-discard-old
}
```

# Persistent Views

[!WARNING] `PersistentView` is deprecated. Use `PersistenceQuery` when it will be ported. The corresponding query type is `EventsByPersistenceId`. There are several alternatives for connecting the `Source` to an actor corresponding to a previous `PersistentView` actor:

- `Sink.ActorRef` is simple, but has the disadvantage that there is no back-pressure signal from the destination actor, i.e. if the actor is not consuming the messages fast enough the mailbox of the actor will grow
- `MapAsync` combined with [Ask: Send-And-Receive-Future](#) is almost as simple with the advantage of back-pressure being propagated all the way
- `ActorSubscriber` in case you need more fine grained control

The consuming actor may be a plain `UntypedActor` or a `UntypedPersistentActor` if it needs to store its own state (e.g. `FromSequenceNr` `offset`).

While a persistent actor may be used to produce and persist events, views are used only to read internal state based on them. Like the persistent actor, a view has a `PersistenceId` to specify a collection of events to be resent to current view. This value should however be correlated with the `PersistentId` of an actor who is the producer of the events.

Other members:

- `viewId` property is a view unique identifier that doesn't change across different actor incarnations. It's useful in cases where there are multiple different views associated with a single persistent actor, but showing its state from a different perspectives.
- `IsAutoUpdate` property determines if the view will try to automatically update its state in specified time intervals. Without it, the view won't update its state until it receives an explicit `Update` message. This value can be set through configuration with `akka.persistence.view.auto-update` set to either `on` (by default) or `off`.
- `AutoUpdateInterval` specifies a time interval in which the view will be updating itself - only in cases where the `IsAutoUpdate` flag is on. This value can be set through configuration with `akka.persistence.view.auto-update-interval` key (5 seconds by default).
- `AutoUpdateReplayMax` property determines the maximum number of events to be replayed during a single `Update` cycle. This value can be set through configuration with `akka.persistence.view.auto-update-replay-max` key (by default it's `-1` - no limit).
- `LoadSnapshot` will send a request to the snapshot store to resend a current view's snapshot.
- `SaveSnapshot` will send the current view's internal state as a snapshot to be saved by the configured snapshot store.
- `DeleteSnapshot` and `DeleteSnapshots` methods may be used to specify snapshots to be removed from the snapshot store in cases where they are no longer needed.

The `PersistenceId` identifies the persistent actor from which the view receives journaled messages. It is not necessary that the referenced persistent actor is actually running. Views read messages from a persistent actor's journal directly. When a persistent actor is started later and begins to write new messages, by default the corresponding view is updated automatically.

It is possible to determine if a message was sent from the Journal or from another actor in user-land by calling the `IsPersistent` property. Having that said, very often you don't need this information at all and can simply apply the same logic to both cases (skip the if `IsPersistent` check).

## Updates

The default update interval of all persistent views of an actor system is configurable:

The default update interval of all persistent views of an actor system is configurable:

```
akka.persistence.view.auto-update-interval = 5s
```

`PersistentView` implementation classes may also override the `AutoUpdateInterval` method to return a custom update interval for a specific view class or view instance. Applications may also trigger additional updates at any time by sending a view an `Update` message.

```
IActorRef view = system.ActorOf<ViewActor>();
view.Tell(new Update(true));
```

If the `await` parameter is set to true, messages that follow the `Update` request are processed when the incremental message replay, triggered by that update request, completed. If set to false (default), messages following the update request may interleave with the replayed message stream.

Automated updates of all persistent views of an actor system can be turned off by configuration:

```
akka.persistence.view.auto-update = off
```

Implementation classes may override the configured default value by overriding the `autoUpdate` method. To limit the number of replayed messages per update request, applications can configure a custom `akka.persistence.view.auto-update-replay-max` value or override the `AutoUpdateReplayMax` property. The number of replayed messages for manual updates can be limited with the `replayMax` parameter of the `Update` message.

## Recovery

Initial recovery of persistent views works the very same way as for persistent actors (i.e. by sending a `Recover` message to self). The maximum number of replayed messages during initial recovery is determined by `AutoUpdateReplayMax`. Further possibilities to customize initial recovery are explained in section Recovery.

## Identifiers

A persistent view must have an identifier that doesn't change across different actor incarnations. The identifier must be defined with the `viewId` method.

The `viewId` must differ from the referenced `PersistenceId`, unless Snapshots of a view and its persistent actor should be shared (which is what applications usually do not want).

# Snapshots

Snapshots can dramatically reduce recovery times of persistent actors and views. The following discusses snapshots in context of persistent actors but this is also applicable to persistent views.

Persistent actors can save snapshots of internal state by calling the `SaveSnapshot` method. If saving of a snapshot succeeds, the persistent actor receives a `SaveSnapshotSuccess` message, otherwise a `SaveSnapshotFailure` message.

[!code-csharpMain]

```
public class MyPersistentActor : UntypedPersistentActor
{
 public override string PersistenceId => "my-stable-persistence-id";
 private const int SnapShotInterval = 1000;
 private object state = new object();

 protected override void OnRecover(object message)
 {
 // handle recovery here
 }

 protected override void OnCommand(object message)
 {
 if (message is SaveSnapshotSuccess s)
 {
 // ...
 }
 else if (message is SaveSnapshotFailure f)
 {
 // ...
 }
 else if (message is string cmd)
 {
 Persist($"evt-{cmd}", e =>
 {
 UpdateState(e);
 if (LastSequenceNr % SnapShotInterval == 0 && LastSequenceNr != 0)
 {
 SaveSnapshot(state);
 }
 });
 }
 }

 private void UpdateState(string e)
 {
 }
}
```

During recovery, the persistent actor is offered a previously saved snapshot via a `SnapshotOffer` message from which it can initialize internal state.

```
if (message is SnapshotOffer offeredSnapshot)
{
 state = offeredSnapshot.Snapshot;
}
else if (message is RecoveryCompleted)
{
 // ..
}
else
```

```
{
 // event
}
```

The replayed messages that follow the `SnapshotOffer` message, if any, are younger than the offered snapshot. They finally recover the persistent actor to its current (i.e. latest) state.

In general, a persistent actor is only offered a snapshot if that persistent actor has previously saved one or more snapshots and at least one of these snapshots matches the `SnapshotSelectionCriteria` that can be specified for recovery.

```
public override Recovery Recovery => new Recovery(fromSnapshot: new SnapshotSelectionCriteria(maxSequenceNr: 45
7L, maxTimeStamp: DateTime.UtcNow));
```

If not specified, they default to `SnapshotSelectionCriteria.Latest` which selects the latest (= youngest) snapshot. To disable snapshot-based recovery, applications should use `SnapshotSelectionCriteria.None`. A recovery where no saved snapshot matches the specified `SnapshotSelectionCriteria` will replay all journaled messages.

[!NOTE] In order to use snapshots, a default snapshot-store (`akka.persistence.snapshot-store.plugin`) must be configured, or the `UntypedPersistentActor` can pick a snapshot store explicitly by overriding `SnapshotPluginId`. Since it is acceptable for some applications to not use any snapshotting, it is legal to not configure a snapshot store. However, Akka will log a warning message when this situation is detected and then continue to operate until an actor tries to store a snapshot, at which point the operation will fail (by replying with an `SaveSnapshotFailure` for example). Note that `Cluster Sharding` is using snapshots, so if you use Cluster Sharding you need to define a snapshot store plugin.

## Snapshot deletion

A persistent actor can delete individual snapshots by calling the `DeleteSnapshot` method with the sequence number of when the snapshot was taken.

To bulk-delete a range of snapshots matching `SnapshotSelectionCriteria`, persistent actors should use the `DeleteSnapshots` method.

## Snapshot status handling

Saving or deleting snapshots can either succeed or fail – this information is reported back to the persistent actor via status messages as illustrated in the following table.

Method	Success	Failure message
SaveSnapshot	SaveSnapshotSuccess	SaveSnapshotFailure
DeleteSnapshot	DeleteSnapshotSuccess	DeleteSnapshotFailure
DeleteSnapshots	DeleteSnapshotsSuccess	DeleteSnapshotsFailure

If failure messages are left unhandled by the actor, a default warning log message will be logged for each incoming failure message. No default action is performed on the success messages, however you're free to handle them e.g. in order to delete an in memory representation of the snapshot, or in the case of failure to attempt save the snapshot again.

# At-Least-Once Delivery

To send messages with at-least-once delivery semantics to destinations you can mix-in `AtLeastOnceDelivery` class to your `PersistentActor` on the sending side. It takes care of re-sending messages when they have not been confirmed within a configurable timeout.

The state of the sending actor, including which messages have been sent that have not been confirmed by the recipient must be persistent so that it can survive a crash of the sending actor or CLR. The `AtLeastOnceDelivery` class does not persist anything by itself. It is your responsibility to persist the intent that a message is sent and that a confirmation has been received.

Members:

- `Deliver` method is used to send a message to another actor in `at-least-once` delivery semantics. A message sent this way must be confirmed by the other endpoint with the `ConfirmDelivery` method. Otherwise it will be resent again and again until the redelivery limit is reached.
- `GetDeliverySnapshot` and `SetDeliverySnapshot` methods are used as part of a delivery snapshotting strategy. They return/reset state of the current guaranteed delivery actor's unconfirmed messages. In order to save custom deliverer state inside a snapshot, a returned delivery snapshot should be included in that snapshot and reset in `ReceiveRecovery` method, when `SnapshotOffer` arrives.
- `RedeliveryBurstLimit` is a virtual property which determines the maximum number of unconfirmed messages to be send in each redelivery attempt. It may be useful in preventing message overflow scenarios. It may be overridden or configured inside HOCON configuration under `akka.persistence.at-least-once-delivery.redelivery-burst-limit` path (10 000 by default).
- `UnconfirmedDeliveryAttemptsToWarn` is a virtual property which determines how many unconfirmed deliveries may be sent before guaranteed delivery actor will send an `UnconfirmedWarning` message to itself. The count is reset after the actor's restart. It may be overridden or configured inside HOCON configuration under `akka.persistence.at-least-once-delivery.warn-after-number-of-unconfirmed-attempts` path (5 by default).
- `MaxUnconfirmedMessages` is a virtual property which determines the maximum number of unconfirmed deliveries to hold in memory. After this threshold is exceeded, any `Deliver` method will raise `MaxUnconfirmedMessagesExceeded`. It may be overridden or configured inside HOCON configuration under `akka.persistence.at-least-once-delivery.max-unconfirmed-messages` path (100 000 by default).
- `UnconfirmedCount` property shows the number of unconfirmed messages.

## Relationship between Deliver and ConfirmDelivery

To send messages to the destination path, use the `Deliver` method after you have persisted the intent to send the message.

The destination actor must send back a confirmation message. When the sending actor receives this confirmation message you should persist the fact that the message was delivered successfully and then call the `ConfirmDelivery` method.

If the persistent actor is not currently recovering, the deliver method will send the message to the destination actor. When recovering, messages will be buffered until they have been confirmed using `ConfirmDelivery`. Once recovery has completed, if there are outstanding messages that have not been confirmed (during the message replay), the persistent actor will resend these before sending any other messages.

Deliver requires a `deliveryMessageMapper` function to pass the provided `deliveryId` into the message so that the correlation between `Deliver` and `ConfirmDelivery` is possible. The `deliveryId` must do the round trip. Upon receipt of the message, the destination actor will send the same `deliveryId` wrapped in a confirmation message back to the

sender. The sender will then use it to call the `ConfirmDelivery` method to complete the delivery routine.

[!code-csharpMain]

```
public class Msg
{
 public Msg(long deliveryId, string message)
 {
 DeliveryId = deliveryId;
 Message = message;
 }

 public long DeliveryId { get; }

 public string Message { get; }
}

public class Confirm
{
 public Confirm(long deliveryId)
 {
 DeliveryId = deliveryId;
 }

 public long DeliveryId { get; }
}

public interface IEvent
{

}

public class MsgSent : IEvent
{
 public MsgSent(string message)
 {
 Message = message;
 }

 public string Message { get; }
}

public class MsgConfirmed : IEvent
{
 public MsgConfirmed(long deliveryId)
 {
 DeliveryId = deliveryId;
 }

 public long DeliveryId { get; }
}
```

[!code-csharpMain]

```
public class ExampleAtLeastOnceDeliveryReceiveActor : AtLeastOnceDeliveryReceiveActor
{
 private readonly IActorRef _destinationActor = Context.ActorOf<ExampleDestinationAtLeastOnceDeliveryRe
ceiveActor>();

 public ExampleAtLeastOnceDeliveryReceiveActor()
 {
 Recover<MsgSent>(msgSent => Handler(msgSent));
 Recover<MsgConfirmed>(msgConfirmed => Handler(msgConfirmed));

 Command<string>(str =>
 {
 Persist(new MsgSent(str), Handler);
 });
 }
}
```

```

 });

 Command<Confirm>(confirm =>
 {
 Persist(new MsgConfirmed(confirm.DeliveryId), Handler);
 });
}

private void Handler(MsgSent msgSent)
{
 Deliver(_destinationActor.Path, l => new Msg(l, msgSent.Message));
}

private void Handler(MsgConfirmed msgConfirmed)
{
 ConfirmDelivery(msgConfirmed.DeliveryId);
}

public override string PersistenceId { get; } = "persistence-id";
}

public class ExampleDestinationAtLeastOnceDeliveryReceiveActor : ReceiveActor
{
 public ExampleDestinationAtLeastOnceDeliveryReceiveActor()
 {
 Receive<Msg>(msg =>
 {
 Sender.Tell(new Confirm(msg.DeliveryId), Self);
 });
 }
}
}

```

The `deliveryId` generated by the persistence module is a strictly monotonically increasing sequence number without gaps. The same sequence is used for all destinations of the actor, i.e. when sending to multiple destinations the destinations will see gaps in the sequence. It is not possible to use custom `deliveryId`. However, you can send a custom correlation identifier in the message to the destination. You must then retain a mapping between the internal `deliveryId` (passed into the `deliveryMessageMapper` function) and your custom correlation id (passed into the message). You can do this by storing such mapping in a `Map(CorrelationId -> DeliveryId)` from which you can retrieve the `deliveryId` to be passed into the `ConfirmDelivery` method once the receiver of your message has replied with your custom correlation id.

The `AtLeastOnceDeliveryReceiveActor` class has a state consisting of unconfirmed messages and a sequence number. It does not store this state itself. You must persist events corresponding to the `Deliver` and `ConfirmDelivery` invocations from your `PersistentActor` so that the state can be restored by calling the same methods during the recovery phase of the `PersistentActor`. Sometimes these events can be derived from other business level events, and sometimes you must create separate events. During recovery, calls to `deliver` will not send out messages, those will be sent later if no matching `ConfirmDelivery` will have been performed.

Support for snapshots is provided by `GetDeliverySnapshot` and `SetDeliverySnapshot`. The `AtLeastOnceDeliverySnapshot` contains the full delivery state, including unconfirmed messages. If you need a custom snapshot for other parts of the actor state you must also include the `AtLeastOnceDeliverySnapshot`. It is serialized using protobuf with the ordinary Akka serialization mechanism. It is easiest to include the bytes of the `AtLeastOnceDeliverySnapshot` as a blob in your custom snapshot.

The interval between redelivery attempts is defined by the `RedeliverInterval` method. The default value can be configured with the `akka.persistence.at-least-once-delivery.redeliver-interval` configuration key. The method can be overridden by implementation classes to return non-default values.

The maximum number of messages that will be sent at each redelivery burst is defined by the `RedeliveryBurstLimit` method (burst frequency is half of the redelivery interval). If there's a lot of unconfirmed messages (e.g. if the destination is not available for a long time), this helps to prevent an overwhelming amount of messages to be sent at once. The default value can be configured with the `akka.persistence.at-least-once-delivery.redelivery-burst-limit` configuration key. The method can be overridden by implementation classes to return non-default values.

After a number of delivery attempts a `UnconfirmedWarning` message will be sent to self. The re-sending will still continue, but you can choose to call `ConfirmDelivery` to cancel the re-sending. The number of delivery attempts before emitting the warning is defined by the `WarnAfterNumberOfUnconfirmedAttempts` property. The default value can be configured with the `akka.persistence.at-least-once-delivery.warn-after-number-of-unconfirmed-attempts` configuration key. The method can be overridden by implementation classes to return non-default values.

The `AtLeastOnceDeliveryReceiveActor` class holds messages in memory until their successful delivery has been confirmed. The maximum number of unconfirmed messages that the actor is allowed to hold in memory is defined by the `MaxUnconfirmedMessages` method. If this limit is exceeded the deliver method will not accept more messages and it will throw `MaxUnconfirmedMessagesExceededException`. The default value can be configured with the `akka.persistence.at-least-once-delivery.max-unconfirmed-messages` configuration key. The method can be overridden by implementation classes to return non-default values.

# Persistence FSM

`PersistentFSM` handles the incoming messages in an FSM like fashion. Its internal state is persisted as a sequence of changes, later referred to as domain events. Relationship between incoming messages, FSM's states and transitions, persistence of domain events is defined by a DSL.

## A Simple Example

To demonstrate the features of the `PersistentFSM` class, consider an actor which represents a Web store customer. The contract of our "`WebStoreCustomerFSMActor`" is that it accepts the following commands:

[!code-csharpWebStoreCustomerFSMActor.cs]

```
public interface ICommand { }

public class AddItem : ICommand
{
 public AddItem(Item item)
 {
 Item = item;
 }

 public Item Item { get; set; }
}

public class Buy : ICommand
{
 public static Buy Instance { get; } = new Buy();
 private Buy() { }
}

public class Leave : ICommand
{
 public static Leave Instance { get; } = new Leave();
 private Leave() { }
}

public class GetCurrentCart : ICommand
{
 public static GetCurrentCart Instance { get; } = new GetCurrentCart();
 private GetCurrentCart() { }
}
```

`AddItem` sent when the customer adds an item to a shopping cart `Buy` - when the customer finishes the purchase  
`Leave` - when the customer leaves the store without purchasing anything `GetCurrentCart` allows to query the current state of customer's shopping cart

The customer can be in one of the following states: [!code-csharpWebStoreCustomerFSMActor.cs]

```
public interface IUserState : Akka.Persistence.Fsm.PersistentFSM.IFsmState { }

public class LookingAround : IUserState
{
 public static LookingAround Instance { get; } = new LookingAround();
 private LookingAround() { }
 public string Identifier => "Looking Around";
}

public class Shopping : IUserState
```

```
{
 public static Shopping Instance { get; } = new Shopping();
 private Shopping() { }
 public string Identifier => "Shopping";
}

public class Inactive : IUserState
{
 public static Inactive Instance { get; } = new Inactive();
 private Inactive() { }
 public string Identifier => "Inactive";
}

public class Paid : IUserState
{
 public static Paid Instance { get; } = new Paid();
 private Paid() { }
 public string Identifier => "Paid";
}
```

LookingAround customer is browsing the site, but hasn't added anything to the shopping cart Shopping customer has recently added items to the shopping cart Inactive customer has items in the shopping cart, but hasn't added anything recently Paid customer has purchased the items

[!NOTE] PersistentFSM states must inherit from trait PersistentFSM.IFsmState and implement the string Identifier property. This is required in order to simplify the serialization of FSM states. String identifiers should be unique!

Customer's actions are "recorded" as a sequence of "domain events" which are persisted. Those events are replayed on an actor's start in order to restore the latest customer's state:

[!code-csharp WebStoreCustomerFSMActor.cs]

```
public interface IDomainEvent { }

public class ItemAdded : IDomainEvent
{
 public ItemAdded(Item item)
 {
 Item = item;
 }

 public Item Item { get; set; }
}

public class OrderExecuted : IDomainEvent
{
 public static OrderExecuted Instance { get; } = new OrderExecuted();
 private OrderExecuted() { }
}

public class OrderDiscarded : IDomainEvent
{
 public static OrderDiscarded Instance { get; } = new OrderDiscarded();
 private OrderDiscarded() { }
}
```

Customer state data represents the items in a customer's shopping cart:

[!code-csharp WebStoreCustomerFSMActor.cs]

```
public class Item
{
 public Item(string id, string name, double price)
```

```

 {
 Id = id;
 Name = name;
 Price = price;
 }

 public string Id { get; }

 public string Name { get; }

 public double Price { get; }
}

public interface IShoppingCart
{
 IShoppingCart AddItem(Item item);
 IShoppingCart Empty();
}

public class EmptyShoppingCart : IShoppingCart
{
 public IShoppingCart AddItem(Item item)
 {
 return new NonEmptyShoppingCart(ImmutableList.Create(item));
 }

 public IShoppingCart Empty()
 {
 return this;
 }
}

public class NonEmptyShoppingCart : IShoppingCart
{
 public NonEmptyShoppingCart(ImmutableList<Item> items)
 {
 Items = items;
 }

 public IShoppingCart AddItem(Item item)
 {
 return new NonEmptyShoppingCart(Items.Add(item));
 }

 public IShoppingCart Empty()
 {
 return new EmptyShoppingCart();
 }

 public ImmutableList<Item> Items { get; }
}

```

Side-effects:

[!code-csharpWebStoreCustomerFSMActor.cs]

```

public interface IReportEvent { }

public class PurchaseWasMade : IReportEvent
{
 public PurchaseWasMade(IEnumerable<Item> items)
 {
 Items = items;
 }

 public IEnumerable<Item> Items { get; }
}

```

```

public class ShoppingCardDiscarded : IReportEvent
{
 public static ShoppingCardDiscarded Instance { get; } = new ShoppingCardDiscarded();
 private ShoppingCardDiscarded() { }
}

```

Here is how everything is wired together: [!code-csharp WebStoreCustomerFSMActor.cs]

```

StartWith(LookingAround.Instance, new EmptyShoppingCart());

When(LookingAround.Instance, (evt, state) =>
{
 if (evt.FsmEvent is AddItem addItem)
 {
 return GoTo(Shopping.Instance)
 .Applying(new ItemAdded(addItem.Item))
 .ForMax(TimeSpan.FromSeconds(1));
 }
 else if (evt.FsmEvent is GetCurrentCart)
 {
 return Stay().Replying(evt.StateData);
 }

 return Stay();
});

When(Shopping.Instance, (evt, state) =>
{
 if (evt.FsmEvent is AddItem addItem)
 {
 return Stay()
 .Applying(new ItemAdded(addItem.Item))
 .ForMax(TimeSpan.FromSeconds(1));
 }
 else if (evt.FsmEvent is Buy)
 {
 return GoTo(Paid.Instance).Applying(OrderExecuted.Instance)
 .AndThen(cart =>
 {
 if (cart is NonEmptyShoppingCart nonShoppingCart)
 {
 reportActor.Tell(new PurchaseWasMade(nonShoppingCart.Items));
 SaveStateSnapshot();
 }
 else if (cart is EmptyShoppingCart)
 {
 SaveStateSnapshot();
 }
 });
 }
 else if (evt.FsmEvent is Leave)
 {
 return Stop().Applying(OrderDiscarded.Instance)
 .AndThen(cart =>
 {
 reportActor.Tell(ShoppingCardDiscarded.Instance);
 SaveStateSnapshot();
 });
 }
 else if (evt.FsmEvent is GetCurrentCart)
 {
 return Stay().Replying(evt.StateData);
 }
 else if (evt.FsmEvent is FSMBase.StateTimeout)
 {
 return GoTo(Inactive.Instance).ForMax(TimeSpan.FromSeconds(2));
 }
}

```

```

 }

 return Stay();
 });

 When(Inactive.Instance, (evt, state) =>
 {
 if (evt.FsmEvent is AddItem addItem)
 {
 return GoTo(Shopping.Instance)
 .Applying(new ItemAdded(addItem.Item))
 .ForMax(TimeSpan.FromSeconds(1));
 }
 else if (evt.FsmEvent is FSMBase.StateTimeout)
 {
 return Stop()
 .Applying(OrderDiscarded.Instance)
 .AndThen(cart => reportActor.Tell(ShoppingCardDiscarded.Instance));
 }

 return Stay();
 });

 When(Paid.Instance, (evt, state) =>
 {
 if (evt.FsmEvent is Leave)
 {
 return Stop();
 }
 else if (evt.FsmEvent is GetCurrentCart)
 {
 return Stay().Replying(evt.StateData);
 }

 return Stay();
 });
}

```

[!NOTE] State data can only be modified directly on initialization. Later it's modified only as a result of applying domain events. Override the `ApplyEvent` method to define how state data is affected by domain events, see the example below

#### [!code-csharp]WebStoreCustomerFSMActor.cs

```

protected override IShoppingCart ApplyEvent(IDomainEvent evt, IShoppingCart cartBeforeEvent)
{
 switch (evt)
 {
 case ItemAdded itemAdded: return cartBeforeEvent.AddItem(itemAdded.Item);
 case OrderExecuted _: return cartBeforeEvent;
 case OrderDiscarded _: return cartBeforeEvent.Empty();
 default: return cartBeforeEvent;
 }
}

```

`AndThen` can be used to define actions which will be executed following event's persistence - convenient for "side effects" like sending a message or logging. Notice that actions defined in `andThen` block are not executed on recovery:

```

GoTo(Paid.Instance).Applying(OrderExecuted.Instance).AndThen(cart =>
{
 if (cart is NonEmptyShoppingCart nonShoppingCart)
 {
 reportActor.Tell(new PurchaseWasMade(nonShoppingCart.Items));
 }
}

```

```
});
```

A snapshot of state data can be persisted by calling the `SaveStateSnapshot()` method:

```
Stop().Applying(OrderDiscarded.Instance).AndThen(cart =>
{
 reportActor.Tell(ShoppingCardDiscarded.Instance);
 SaveStateSnapshot();
});
```

On recovery state data is initialized according to the latest available snapshot, then the remaining domain events are replayed, triggering the `ApplyEvent` method.

## Periodical snapshot by snapshot-after

You can enable periodical `SaveStateSnapshot()` calls in `PersistentFSM` if you turn the following flag on in `reference.conf`

```
akka.persistence.fsm.snapshot-after = 1000
```

this means `SaveStateSnapshot()` is called after the sequence number reaches multiple of 1000.

[!NOTE] `SaveStateSnapshot()` might not be called exactly at sequence numbers being multiple of the `snapshot-after` configuration value. This is because `PersistentFSM` works in a sort of "batch" mode when processing and persisting events, and `SaveStateSnapshot()` is called only at the end of the "batch". For example, if you set `akka.persistence.fsm.snapshot-after = 1000`, it is possible that `saveStateSnapshot()` is called at `lastSequenceNr = 1005, 2003, ...`. A single batch might persist state transition, also there could be multiple domain events to be persisted if you pass them to `Applying` method in the `PersistentFSM` DSL.

# Storage plugins

## Journals

Journal is a specialized type of actor which exposes an API to handle incoming events and store them in backend storage. By default Akka.Persistence uses a `MemoryJournal` which stores all events in memory and therefore it's not persistent storage. A custom journal configuration path may be specified inside `akka.persistence.journal.plugin` path and by default it requires two keys set: `class` and `plugin-dispatcher`. Example configuration:

[!code-jsonMain]

## Snapshot store

Snapshot store is a specialized type of actor which exposes an API to handle incoming snapshot-related requests and is able to save snapshots in some backend storage. By default Akka.Persistence uses a `LocalSnapshotStore`, which uses a local file system as storage. A custom snapshot store configuration path may be specified inside `akka.persistence.snapshot-store.plugin` path and by default it requires two keys set: `class` and `plugin-dispatcher`.

Example configuration:

[!code-jsonMain]

# Custom serialization

Serialization of snapshots and payloads of Persistent messages is configurable with Akka's Serialization infrastructure. For example, if an application wants to serialize

- payloads of type MyPayload with a custom MyPayloadSerializer and
- snapshots of type MySnapshot with a custom MySnapshotSerializer

it must add

```
akka.actor {
 serializers {
 my-payload = "docs.persistence.MyPayloadSerializer"
 my-snapshot = "docs.persistence.MySnapshotSerializer"
 }
 serialization-bindings {
 "docs.persistence.MyPayload" = my-payload
 "docs.persistence.MySnapshot" = my-snapshot
 }
}
```

to the application configuration. If not specified, a default serializer is used.

# Persistence Query

Akka persistence query complements Persistence by providing a universal asynchronous stream based query interface that various journal plugins can implement in order to expose their query capabilities.

The most typical use case of persistence query is implementing the so-called query side (also known as "read side") in the popular CQRS architecture pattern - in which the writing side of the application (e.g. implemented using akka persistence) is completely separated from the "query side". Akka Persistence Query itself is not directly the query side of an application, however it can help to migrate data from the write side to the query side database. In very simple scenarios Persistence Query may be powerful enough to fulfill the query needs of your app, however we highly recommend (in the spirit of CQRS) of splitting up the write/read sides into separate datastores as the need arises.

## Design overview

Akka Persistence Query is purposely designed to be a very loosely specified API. This is in order to keep the provided APIs general enough for each journal implementation to be able to expose its best features, e.g. a SQL journal can use complex SQL queries or if a journal is able to subscribe to a live event stream this should also be possible to expose the same API - a typed stream of events.

**Each read journal must explicitly document which types of queries it supports.** Refer to your journal's plugins documentation for details on which queries and semantics it supports.

While Akka Persistence Query does not provide actual implementations of ReadJournals, it defines a number of pre-defined query types for the most common query scenarios, that most journals are likely to implement (however they are not required to).

## Read Journals

In order to issue queries one has to first obtain an instance of a ReadJournal. Read journals are implemented as Community plugins, each targeting a specific datastore (for example Cassandra or ADO.NET databases). For example, given a library that provides a akka.persistence.query.my-read-journal obtaining the related journal is as simple as:

```
var actorSystem = ActorSystem.Create("query");

// obtain read journal by plugin id
var readJournal = PersistenceQuery.Get(actorSystem)
 .ReadJournalFor<SqlReadJournal>("akka.persistence.query.my-read-journal");

// issue query to journal
Source<EventEnvelope, NotUsed> source = readJournal
 .EventsByPersistenceId("user-1337", 0, long.MaxValue);

// materialize stream, consuming events
var mat = ActorMaterializer.Create(actorSystem);
source.RunWithEach(envelope =>
{
 Console.WriteLine($"event {envelope}");
}, mat);
```

Journal implementers are encouraged to put this identifier in a variable known to the user, such that one can access it via `ReadJournalFor<SqlReadJournal>(SqlReadJournal.Identifier)`, however this is not enforced.

Read journal implementations are available as Community plugins

Read journal implementations are available as Community plugins.

## Predefined queries

Akka persistence query comes with a number of query interfaces built in and suggests Journal implementors to implement them according to the semantics described below. It is important to notice that while these query types are very common a journal is not required to implement all of them - for example because in a given journal such query would be significantly inefficient.

[!NOTE] Refer to the documentation of the ReadJournal plugin you are using for a specific list of supported query types. For example, Journal plugins should document their stream completion strategies.

The predefined queries are:

### AllPersistenceIdsQuery and CurrentPersistenceIdsQuery

`AllPersistenceIds` is used for retrieving all `PersistenceIds` of all persistent actors.

```
var queries = PersistenceQuery.Get(actorSystem)
 .ReadJournalFor<SqlReadJournal>("akka.persistence.query.my-read-journal");

var mat = ActorMaterializer.Create(actorSystem);
Source<string, NotUsed> src = queries.AllPersistenceIds();
```

The returned event stream is unordered and you can expect different order for multiple executions of the query.

The stream is not completed when it reaches the end of the currently used `PersistenceIds`, but it continues to push new `PersistenceIds` when new persistent actors are created. Corresponding query that is completed when it reaches the end of the currently used `PersistenceIds` is provided by `CurrentPersistenceIds`.

The write journal is notifying the query side as soon as new `PersistenceIds` are created and there is no periodic polling or batching involved in this query.

The stream is completed with failure if there is a failure in executing the query in the backend journal.

### EventsByPersistenceIdQuery and CurrentEventsByPersistenceIdQuery

`EventsByPersistenceId` is used for retrieving events for a specific `PersistentActor` identified by `PersistenceId`.

```
var queries = PersistenceQuery.Get(actorSystem)
 .ReadJournalFor<SqlReadJournal>("akka.persistence.query.my-read-journal");

var mat = ActorMaterializer.Create(actorSystem);
var src = queries.EventsByPersistenceId("some-persistence-id", 0L, long.MaxValue);
Source<object, NotUsed> events = src.Select(c => c.Event);
```

You can retrieve a subset of all events by specifying `FromSequenceNr` and `ToSequenceNr` or use `0L` and `long.MaxValue` respectively to retrieve all events. Note that the corresponding sequence number of each event is provided in the `EventEnvelope`, which makes it possible to resume the stream at a later point from a given sequence number.

The returned event stream is ordered by sequence number, i.e. the same order as the `PersistentActor` persisted the events. The same prefix of stream elements (in same order) are returned for multiple executions of the query, except for when events have been deleted.

The stream is not completed when it reaches the end of the currently stored events, but it continues to push new events when new events are persisted. Corresponding query that is completed when it reaches the end of the currently stored events is provided by `CurrentEventsByPersistenceId`.

The write journal is notifying the query side as soon as events are persisted, but for efficiency reasons the query side retrieves the events in batches that sometimes can be delayed up to the configured `refresh-interval` or given `RefreshInterval` hint.

The stream is completed with failure if there is a failure in executing the query in the backend journal.

### EventsByTag and CurrentEventsByTag

`EventsByTag` allows querying events regardless of which `PersistenceId` they are associated with. This query is hard to implement in some journals or may need some additional preparation of the used data store to be executed efficiently. The goal of this query is to allow querying for all events which are "tagged" with a specific tag. That includes the use case to query all domain events of an Aggregate Root type. Please refer to your read journal plugin's documentation to find out if and how it is supported.

Some journals may support tagging of events via an Event Adapters that wraps the events in a `Akka.Persistence.Journal.Tagged` with the given tags. The journal may support other ways of doing tagging - again, how exactly this is implemented depends on the used journal. Here is an example of such a tagging event adapter:

```
public class MyTaggingEventAdapter : IWriteEventAdapter
{
 private ImmutableHashSet<string> colors = ImmutableHashSet.Create("green", "black", "blue");

 public string Manifest(object evt)
 {
 return string.Empty;
 }

 public object ToJournal(object evt)
 {
 var str = evt as string;
 if (str != null)
 {
 var tags = colors.Aggregate(
 ImmutableHashSet<string>.Empty,
 (acc, c) => str.Equals(c) ? acc.Add(c) : acc);
 if (tags.IsEmpty)
 return evt;
 else
 return new Tagged(evt, tags);
 }
 else
 {
 return evt;
 }
 }
}
```

[!NOTE] A very important thing to keep in mind when using queries spanning multiple `PersistenceIds`, such as `EventsByTag` is that the order of events at which the events appear in the stream rarely is guaranteed (or stable between materializations).

Journals may choose to opt for strict ordering of the events, and should then document explicitly what kind of ordering guarantee they provide - for example "ordered by timestamp ascending, independently of `PersistenceId`" is easy to achieve on relational databases, yet may be hard to implement efficiently on plain key-value datastores.

In the example below we query all events which have been tagged (we assume this was performed by the write-side using an EventAdapter, or that the journal is smart enough that it can figure out what we mean by this tag - for example if the journal stored the events as json it may try to find those with the field tag set to this value etc.).

```
// assuming journal is able to work with numeric offsets we can:
Source<EventEnvelope, NotUsed> blueThings = readJournal.EventsByTag("blue", 0L);
```

```
// find top 10 blue things:
Task<ImmutableHashSet<object>> top10BlueThings = blueThings
 .Select(c => c.Event)
 .Take(10) // cancels the query stream after pulling 10 elements
 .RunAggregate(
 ImmutableHashSet<object>.Empty,
 (acc, c) => acc.Add(c),
 mat);

// start another query, from the known offset
var furtherBlueThings = readJournal.EventsByTag("blue", offset: 10);
```

As you can see, we can use all the usual stream combinator available from Akka Streams on the resulting query stream, including for example taking the first 10 and cancelling the stream. It is worth pointing out that the built-in `EventsByTag` query has an optionally supported offset parameter (of type Long) which the journals can use to implement resumable-streams. For example a journal may be able to use a WHERE clause to begin the read starting from a specific row, or in a datastore that is able to order events by insertion time it could treat the Long as a timestamp and select only older events.

If your usage does not require a live stream, you can use the `CurrentEventsByTag` query.

## Materialized values of queries

Journals are able to provide additional information related to a query by exposing materialized values, which are a feature of Akka Streams that allows to expose additional values at stream materialization time.

More advanced query journals may use this technique to expose information about the character of the materialized stream, for example if it's finite or infinite, strictly ordered or not ordered at all. The materialized value type is defined as the second type parameter of the returned `Source`, which allows journals to provide users with their specialised query object, as demonstrated in the sample below:

```
public class RichEvent
{
 public RichEvent(ISet<string> tags, object payload)
 {
 Tags = tags;
 Payload = payload;
 }

 public ISet<string> Tags { get; }
 public object Payload { get; }
}

public class QueryMetadata
{
 public QueryMetadata(bool deterministicOrder, bool infinite)
 {
 DeterministicOrder = deterministicOrder;
 Infinite = infinite;
 }

 public bool DeterministicOrder { get; }
 public bool Infinite { get; }
}

public Source<RichEvent, QueryMetadata> ByTagsWithMeta(ISet<string> tags) { }

var query = readJournal.ByTagsWithMeta(ImmutableHashSet.Create("red", "blue"));
query
```

```

 .MapMaterializedValue(meta =>
{
 Console.WriteLine(
 $"The query is: ordered deterministically: {meta.DeterministicOrder}, infinite: {meta.Infinite}");
 return meta;
})
.Select(evt =>
{
 Console.WriteLine($"Event payload: {evt.Payload}");
 return evt;
})
.RunWith(Sink.Ignore<RichEvent>(), mat);

```

## Performance and denormalization

When building systems using Event sourcing and CQRS ([Command & Query Responsibility Segregation](#)) techniques it is tremendously important to realise that the write-side has completely different needs from the read-side, and separating those concerns into datastores that are optimised for either side makes it possible to offer the best experience for the write and read sides independently.

For example, in a bidding system it is important to "take the write" and respond to the bidder that we have accepted the bid as soon as possible, which means that write-throughput is of highest importance for the write-side – often this means that data stores which are able to scale to accommodate these requirements have a less expressive query side.

On the other hand the same application may have some complex statistics view or we may have analysts working with the data to figure out best bidding strategies and trends – this often requires some kind of expressive query capabilities like for example SQL or writing Spark jobs to analyse the data. Therefore the data stored in the write-side needs to be projected into the other read-optimised datastore.

[!NOTE] When referring to Materialized Views in Akka Persistence think of it as "some persistent storage of the result of a Query". In other words, it means that the view is created once, in order to be afterwards queried multiple times, as in this format it may be more efficient or interesting to query it (instead of the source events directly).

## Materialize view to Reactive Streams compatible datastore

If the read datastore exposes a Reactive Streams interface then implementing a simple projection is as simple as, using the read-journal and feeding it into the databases driver interface, for example like so:

```

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

var readJournal =
 PersistenceQuery.Get(system).ReadJournalFor<MyReadJournal>(JournalId)
ISubscriber<IImmutableList<object>> dbBatchWriter =
 new ReactiveStreamsCompatibleDBDriver.BatchWriter();

// Using an example (Reactive Streams) Database driver
readJournal
 .EventsByPersistenceId("user-1337")
 .Select(envelope => envelope.Event)
 .Select(ConvertToReadSideTypes) // convert to datatype
 .Grouped(20) // batch inserts into groups of 20
 .RunWith(Sink.FromSubscriber(dbBatchWriter), mat); // write batches to read-side database

```

## Materialize view using SelectAsync

If the target database does not provide a reactive streams Subscriber that can perform writes, you may have to implement the write logic using plain functions or Actors instead.

In case your write logic is state-less and you just need to convert the events from one data type to another before writing into the alternative datastore, then the projection is as simple as:

```
public class ExampleStore
{
 public Task<object> Save(object evt)
 {
 return Task.FromResult(evt);
 }
}
```

```
var store = new ExampleStore();

readJournal
 .EventsByTag("bid", 0L)
 .SelectAsync(1, e => store.Save(e))
 .RunWith(Sink.Ignore<object>(), mat);
```

## Resumable projections

Sometimes you may need to implement "resumable" projections, that will not start from the beginning of time each time when run. In this case you will need to store the sequence number (or offset) of the processed event and use it the next time this projection is started. This pattern is not built-in, however is rather simple to implement yourself.

The example below additionally highlights how you would use Actors to implement the write side, in case you need to do some complex logic that would be best handled inside an Actor before persisting the event into the other datastore:

```
var timeout = new Timeout(TimeSpan.FromSeconds(3));

var bidProjection = new MyResumableProjection("bid");

var writerProps = Props.Create(typeof(TheOneWhoWritesToQueryJournal), "bid");
var writer = system.ActorOf(writerProps, "bid-projection-writer");

readJournal
 .EventsByTag("bid", bidProjection.LatestOffset ?? 0L)
 .SelectAsync(8, envelope => writer.Ask(envelope.Event, timeout).ContinueWith(t => envelope.Offset, TaskContinuationOptions.OnlyOnRanToCompletion))
 .SelectAsync(1, offset => bidProjection.saveProgress(offset))
 .RunWith(Sink.Ignore<object>(), mat);

public class TheOneWhoWritesToQueryJournal(id: String) : ActorBase
{
 public TheOneWhoWritesToQueryJournal(string id) {}

 private DummyStore _store = new DummyStore();

 private ComplexState _state = ComplexState();

 protected override bool Receive(object message) {
 _state = UpdateState(_state, message);
 if (_state.IsReadyToSave())
 _store.Save(new Record(_state));
 return true;
 }

 private ComplexState UpdateState(ComplexState state, object msg)
 {
 // some complicated aggregation logic here ...
 }
}
```

```
 return state;
}
}
```

## Configuration

Configuration settings can be defined in the configuration section with the absolute path corresponding to the identifier, which is `Akka.Persistence.Query.Journal.Sqlite` for the default `SqlReadJournal.Identifier`.

It can be configured with the following properties:

[!code-jsonMain]

# Introduction

## Motivation

The way we consume services from the internet today includes many instances of streaming data, both downloading from a service as well as uploading to it or peer-to-peer data transfers. Regarding data as a stream of elements instead of in its entirety is very useful because it matches the way computers send and receive them (for example via TCP), but it is often also a necessity because data sets frequently become too large to be handled as a whole. We spread computations or analyses over large clusters and call it "big data", where the whole principle of processing them is by feeding those data sequentially --as a stream-- through some CPUs.

Actors can be seen as dealing with streams as well: they send and receive series of messages in order to transfer knowledge (or data) from one place to another. We have found it tedious and error-prone to implement all the proper measures in order to achieve stable streaming between actors, since in addition to sending and receiving we also need to take care to not overflow any buffers or mailboxes in the process. Another pitfall is that Actor messages can be lost and must be retransmitted in that case lest the stream have holes on the receiving side. When dealing with streams of elements of a fixed given type, Actors also do not currently offer good static guarantees that no wiring errors are made: type-safety could be improved in this case.

For these reasons it was decided to bundle up a solution to these problems as an Akka Streams API. The purpose is to offer an intuitive and safe way to formulate stream processing setups such that we can then execute them efficiently and with bounded resource usage -- no more OutOfMemoryErrors. In order to achieve this our streams need to be able to limit the buffering that they employ, they need to be able to slow down producers if the consumers cannot keep up. This feature is called back-pressure and is at the core of the [Reactive Streams](#) initiative of which Akka is a founding member. For you this means that the hard problem of propagating and reacting to back-pressure has been incorporated in the design of Akka Streams already, so you have one less thing to worry about; it also means that Akka Streams interoperate seamlessly with all other Reactive Streams implementations (where Reactive Streams interfaces define the interoperability SPI while implementations like Akka Streams offer a nice user API).

## Relationship with Reactive Streams

The Akka Streams API is completely decoupled from the Reactive Streams interfaces. While Akka Streams focus on the formulation of transformations on data streams the scope of Reactive Streams is just to define a common mechanism of how to move data across an asynchronous boundary without losses, buffering or resource exhaustion.

The relationship between these two is that the Akka Streams API is geared towards end-users while the Akka Streams implementation uses the Reactive Streams interfaces internally to pass data between the different processing stages. For this reason you will not find any resemblance between the Reactive Streams interfaces and the Akka Streams API. This is in line with the expectations of the Reactive Streams project, whose primary purpose is to define interfaces such that different streaming implementation can interoperate; it is not the purpose of Reactive Streams to describe an end-user API.

## How to read these docs

Stream processing is a different paradigm to the Actor Model or to Task composition, therefore it may take some careful study of this subject until you feel familiar with the tools and techniques. The documentation is here to help and for best results we recommend the following approach:

- Read the [Quick Start Guide](#) to get a feel for how streams look like and what they can do.

- The top-down learners may want to peruse the [Design Principles behind Akka Streams](#) at this point.
- The bottom-up learners may feel more at home rummaging through the [Streams Cookbook](#).
- For a complete overview of the built-in processing stages you can look at the table in [Overview of built-in stages and their semantics](#).
- The other sections can be read sequentially or as needed during the previous steps, each digging deeper into specific topics

# Streams Quickstart Guide

A stream usually begins at a source, so this is also how we start an Akka Stream. Before we create one, we import the full complement of streaming tools:

```
using Akka.Streams;
using Akka.Streams.Dsl;
```

Now we will start with a rather simple source, emitting the integers 1 to 100;

```
Source<T, NotUsed> source = Source.From(Enumerable.Range(1, 100))
```

The `Source` type is parametrized with two types: the first one is the type of element that this source emits and the second one may signal that running the source produces some auxiliary value (e.g. a network source may provide information about the bound port or the peer's address). Where no auxiliary information is produced, the type `NotUsed` is used -- and a simple range of integers surely falls into this category.

Having created this source means that we have a description of how to emit the first 100 natural numbers, but this source is not yet active. In order to get those numbers out we have to run it:

```
source.RunForEach(i => Console.WriteLine(i.ToString()), materializer)
```

This line will complement the source with a consumer function--in this example we simply print out the numbers to the console--and pass this little stream setup to an Actor that runs it. This activation is signaled by having "run" be part of the method name; there are other methods that run Akka Streams, and they all follow this pattern.

You may wonder where the Actor gets created that runs the stream, and you are probably also asking yourself what this materializer means. In order to get this value we first need to create an Actor system:

```
using (var system = ActorSystem.Create("system"))
using (var materializer = system.Materializer())
```

There are other ways to create a materializer, e.g. from an `ActorContext` when using streams from within Actors. The `Materializer` is a factory for stream execution engines, it is the thing that makes streams run --you don't need to worry about any of the details just now apart from that you need one for calling any of the run methods on a `Source`.

The nice thing about Akka Streams is that the `Source` is just a description of what you want to run, and like an architect's blueprint it can be reused, incorporated into a larger design. We may choose to transform the source of integers and write it to a file instead:

```
var factorials = source.Scan(new BigInteger(1), (acc, next) => acc * next);
var result =
 factorials
 .Select(num => ByteString.FromString($"{num}\n"))
 .RunWith(FileIOToFile(new FileInfo("factorials.txt")), materializer);
```

First we use the `scan` combinator to run a computation over the whole stream: starting with the number 1 (`BigInteger(1)`) we multiple by each of the incoming numbers, one after the other; the `scan` operation emits the initial value and then every calculation result. This yields the series of factorial numbers which we stash away as a Source for later reuse --it is important to keep in mind that nothing is actually computed yet, this is just a description of what we want to have computed once we run the stream. Then we convert the resulting series of numbers into a

stream of `ByteString` objects describing lines in a text file. This stream is then run by attaching a file as the receiver of the data. In the terminology of Akka Streams this is called a `sink`. `IOResult` is a type that IO operations return in Akka Streams in order to tell you how many bytes or elements were consumed and whether the stream terminated normally or exceptionally.

## Reusable Pieces

One of the nice parts of Akka Stream --and something that other stream libraries do not offer-- is that not only sources can be reused like blueprints, all other elements can be as well. We can take the file-writing `sink`, prepend the processing steps necessary to get the `ByteString` elements from incoming string and package that up as a reusable piece as well. Since the language for writing these streams always flows from left to right (just like plain English), we need a starting point that is like a source but with an "open" input. In Akka Streams this is called a `Flow`:

```
public static Sink<string, Task<IOResult>> LineSink(string filename) {
 return Flow.Create<string>()
 .Select(s => ByteString.FromString($"{s}\n"))
 .ToMaterialized(FileIO.ToFile(new FileInfo(filename)), Keep.Right);
}
```

Starting from a flow of strings we convert each to `ByteString` and then feed to the already known file-writing `sink`. The resulting blueprint is a `Sink<string, Task<IOResult>>`, which means that it accepts strings as its input and when materialized it will create auxiliary information of type `Task<IOResult>` (when chaining operations on a `Source` or `Flow` the type of the auxiliary information --called the "materialized value"-- is given by the leftmost starting point; since we want to retain what the `FileIO.ToFile` sink has to offer, we need to say `Keep.Right`).

We can use the new and shiny `Sink` we just created by attaching it to our factorials source --after a small adaptation to turn the numbers into strings:

```
factorials.Select(_ => _.ToString()).RunWith(LineSink("factorial2.txt"), materializer);
```

## Time-Based Processing

Before we start looking at a more involved example we explore the streaming nature of what Akka Streams can do. Starting from the `factorials` source we transform the stream by zipping it together with another stream, represented by a `Source` that emits the number 0 to 100: the first number emitted by the `factorials` source is the factorial of zero, the second is the factorial of one, and so on. We combine these two by forming strings like "3! = 6".

```
await factorials
 .ZipWith(Source.From(Enumerable.Range(0, 100)), (num, idx) => $"{idx}! = {num}")
 .Throttle(1, TimeSpan.FromSeconds(1), 1, ThrottleMode.Shaping)
 .RunForeach(Console.WriteLine, materializer);
```

All operations so far have been time-independent and could have been performed in the same fashion on strict collections of elements. The next line demonstrates that we are in fact dealing with streams that can flow at a certain speed: we use the `throttle` combinator to slow down the stream to 1 element per second (the second 1 in the argument list is the maximum size of a burst that we want to allow--passing 1 means that the first element gets through immediately and the second then has to wait for one second and so on).

If you run this program you will see one line printed per second. One aspect that is not immediately visible deserves mention, though: if you try and set the streams to produce a billion numbers each then you will notice that your environment does not crash with an `OutOfMemoryError`, even though you will also notice that running the streams happens in the background, asynchronously (this is the reason for the auxiliary information to be provided as a

Task ). The secret that makes this work is that Akka Streams implicitly implement pervasive flow control, all combinators respect back-pressure. This allows the throttle combinator to signal to all its upstream sources of data that it can only accept elements at a certain rate--when the incoming rate is higher than one per second the throttle combinator will assert back-pressure upstream.

This is basically all there is to Akka Streams in a nutshell--glossing over the fact that there are dozens of sources and sinks and many more stream transformation combinators to choose from, see also [Overview of built-in stages and their semantics](#).

# Reactive Tweets

A typical use case for stream processing is consuming a live stream of data that we want to extract or aggregate some other data from. In this example we'll consider consuming a stream of tweets and extracting information from them.

We will also consider the problem inherent to all non-blocking streaming solutions: "*What if the subscriber is too slow to consume the live stream of data?*". Traditionally the solution is often to buffer the elements, but this can—and usually will—cause eventual buffer overflows and instability of such systems. Instead Akka Streams depend on internal backpressure signals that allow to control what should happen in such scenarios.

[!NOTE] You can find a example implementation [here](#), it's using `Tweetinvi` to call the Twitter STREAM API. Due to the fact that Tweetinvi doesn't implement the Reactive Streams specifications, we push the tweets into the stream via the `IActorRef` that is materialized from the following Source `Source.ActorRef<ITweet>(100, OverflowStrategy.DropHead);`.

## Transforming and consuming simple streams

The example application we will be looking at is a simple Twitter feed stream from which we'll want to extract certain information, like for example the number of tweets a user has posted.

In order to prepare our environment by creating an `ActorSystem` and `ActorMaterializer`, which will be responsible for materializing and running the streams we are about to create:

```
using (var sys = ActorSystem.Create("Reactive-Tweets"))
{
 using (var mat = sys.Materializer())
 {
 }
}
```

The `ActorMaterializer` can optionally take `ActorMaterializerSettings` which can be used to define materialization properties, such as default buffer sizes (see also [Buffers for asynchronous stages](#)), the dispatcher to be used by the pipeline etc. These can be overridden with `WithAttributes` on `Flow`, `Source`, `Sink` and `IGraph`.

Let's assume we have a stream of tweets readily available. In Akka this is expressed as a `Source[Out, M]`:

```
Source<ITweet, NotUsed> tweetSource;
```

Streams always start flowing from a `Source[Out, M1]` then can continue through `Flow[In, Out, M2]` elements or more advanced graph elements to finally be consumed by a `Sink[In, M3]` (ignore the type parameters `M1`, `M2` and `M3` for now, they are not relevant to the types of the elements produced/consumed by these classes – they are "materialized types", which we'll talk about [below](#)).

The operations should look familiar to anyone who has used the .Net Collections library, however they operate on streams and not collections of data (which is a very important distinction, as some operations only make sense in streaming and vice versa):

```
Source<string, NotUsed> formattedRetweets = tweetSource
 .Where(tweet => tweet.IsRetweet)
 .Select(FormatTweet);
```

Finally in order to `materialize` and run the stream computation we need to attach the Flow to a `sink` that will get the Flow running. The simplest way to do this is to call `RunWith(sink, mat)` on a `source`. For convenience a number of common Sinks are predefined and collected as methods on the `Sink companion class`. For now let's simply print the formatted retweets:

```
formattedRetweets.RunWith(Sink.ForEach<string>(Console.WriteLine), mat);
```

or by using the shorthand version (which are defined only for the most popular Sinks such as `Sink.Aggregate` and `Sink.Foreach`):

```
formattedRetweets.RunForeach(Console.WriteLine, mat);
```

Materializing and running a stream always requires a `Materializer` to be passed in like this: `.Run(materializer)`

```
using (var sys = ActorSystem.Create("Reactive-Tweets"))
{
 using (var mat = sys.Materializer())
 {
 Source<string, NotUsed> formattedRetweets = tweetSource
 .Where(tweet => tweet.IsRetweet)
 .Select(FormatTweet);

 formattedRetweets.RunForeach(Console.WriteLine, mat);
 }
}
```

## Flattening sequences in streams

In the previous section we were working on 1:1 relationships of elements which is the most common case, but sometimes we might want to map from one element to a number of elements and receive a "flattened" stream, similarly like `SelectMany` works on .Net Collections. In order to get a flattened stream of hashtags from our stream of tweets we can use the `SelectMany` combinator:

```
Source<IHashtagEntity, NotUsed> hashTags = tweetSource.SelectMany(tweet => tweet.Hashtags);
```

## Broadcasting a stream

Now let's say we want to persist all hashtags, as well as all author names from this one live stream. For example we'd like to write all author handles into one file, and all hashtags into another file on disk. This means we have to split the source stream into two streams which will handle the writing to these different files.

Elements that can be used to form such "fan-out" (or "fan-in") structures are referred to as "junctions" in Akka Streams. One of these that we'll be using in this example is called `Broadcast`, and it simply emits elements from its input port to all of its output ports.

Akka Streams intentionally separate the linear stream structures (Flows) from the non-linear, branching ones (Graphs) in order to offer the most convenient API for both of these cases. Graphs can express arbitrarily complex stream setups at the expense of not reading as familiarly as collection transformations.

Graphs are constructed using `GraphDSL` like this:

```
Sink<IUser, NotUsed> writeAuthors = null;
Sink<IHashtagEntity, NotUsed> writeHashtags = null;
```

```

var g = RunnableGraph.FromGraph(GraphDsl.Create(b =>
{
 var broadcast = b.Add(new Broadcast<ITweet>(2));
 b.From(tweetSource).To(broadcast.In);
 b.From(broadcast.Out(0))
 .Via(Flow.Create<ITweet>().Select(tweet => tweet.CreatedBy))
 .To(writeAuthors);
 b.From(broadcast.Out(1))
 .Via(Flow.Create<ITweet>().SelectMany(tweet => tweet.Hashtags))
 .To(writeHashtags);

 return ClosedShape.Instance;
}));

g.Run(mat);

```

As you can see, inside the `GraphDSL` we use an implicit graph builder `b` to mutably construct the graph using the `from`, `via` or `to` "edge operator".

`GraphDSL.Create` returns a `IGraph`, in this example a `IGraph<ClosedShape, NotUsed>` where `ClosedShape` means that it is a *fully connected graph* or "closed" - there are no unconnected inputs or outputs. Since it is closed it is possible to transform the graph into a `IRunnableGraph` using `RunnableGraph.FromGraph`. The runnable graph can then be `Run()` to materialize a stream out of it.

Both `IGraph` and `IRunnableGraph` are *immutable, thread-safe, and freely shareable*.

A graph can also have one of several other shapes, with one or more unconnected ports. Having unconnected ports expresses a graph that is a *partial graph*. Concepts around composing and nesting graphs in large structures are explained in detail in [Modularity, Composition and Hierarchy](#). It is also possible to wrap complex computation graphs as Flows, Sinks or Sources, which will be explained in detail in [Constructing Sources, Sinks and Flows from Partial Graphs](#).

## Back-pressure in action

One of the main advantages of Akka Streams is that they *always* propagate back-pressure information from stream Sinks (Subscribers) to their Sources (Publishers). It is not an optional feature, and is enabled at all times. To learn more about the back-pressure protocol used by Akka Streams and all other Reactive Streams compatible implementations read [Back-pressure explained](#).

A typical problem applications (not using Akka Streams) like this often face is that they are unable to process the incoming data fast enough, either temporarily or by design, and will start buffering incoming data until there's no more space to buffer, resulting in either `outOfMemoryError`'s or other severe degradations of service responsiveness. With Akka Streams buffering can and must be handled explicitly. For example, if we are only interested in the "*most recent tweets, with a buffer of 10 elements*" this can be expressed using the `Buffer` element:

```

tweetSource
 .Buffer(10, OverflowStrategy.DropHead)
 .Select(SlowComputation)
 .RunWith(Sink.Ignore<ComputationResult>(), mat);

```

The `Buffer` element takes an explicit and required `overflowStrategy`, which defines how the buffer should react when it receives another element while it is full. Strategies provided include dropping the oldest element (`DropHead`), dropping the entire buffer, signalling errors etc. Be sure to pick and choose the strategy that fits your use case best.

## Materialized value

So far we've been only processing data using Flows and consuming it into some kind of external Sink - be it by printing values or storing them in some external system. However sometimes we may be interested in some value that can be obtained from the materialized processing pipeline. For example, we want to know how many tweets we have processed. While this question is not as obvious to give an answer to in case of an infinite stream of tweets (one way to answer this question in a streaming setting would be to create a stream of counts described as "*up until now, we've processed N tweets*"), but in general it is possible to deal with finite streams and come up with a nice result such as a total count of elements.

```
var count = Flow.Create<ITweet>().Select(_ => 1);

var sumSink = Sink.Aggregate<int, int>(0, (agg, i) => agg + i);

var counterGraph = tweetSource.Via(count).ToMaterialized(sumSink, Keep.Right);

var sum = counterGraph.Run(mat).Result;
```

First we prepare a reusable `Flow` that will change each incoming tweet into an integer of value `1`. We'll use this in order to combine those with a `Sink.Aggregate` that will sum all `Int` elements of the stream and make its result available as a `Task<Int>`. Next we connect the `tweets` stream to `count` with `via`. Finally we connect the `Flow` to the previously prepared `Sink` using `ToMaterialized`.

Remember those mysterious `Mat` type parameters on `Source<Out, Mat>`, `Flow<In, Out, Mat>` and `Sink<In, Mat>?` They represent the type of values these processing parts return when materialized. When you chain these together, you can explicitly combine their materialized values. In our example we used the `Keep.Right` predefined function, which tells the implementation to only care about the materialized type of the stage currently appended to the right. The materialized type of `sumSink` is `Task<Int>` and because of using `Keep.Right`, the resulting `IRunnableGraph` has also a type parameter of `Task<Int>`.

This step does *not* yet materialize the processing pipeline, it merely prepares the description of the `Flow`, which is now connected to a `Sink`, and therefore can be `Run()`, as indicated by its type: `IRunnableGraph<Task<Int>>`. Next we call `Run(mat)` to materialize and run the `Flow`. The value returned by calling `Run()` on a `IRunnableGraph<T>` is of type `T`. In our case this type is `Task<Int>` which, when completed, will contain the total length of our `tweets` stream. In case of the stream failing, this future would complete with a `Failure`.

A `IRunnableGraph` may be reused and materialized multiple times, because it is just the "blueprint" of the stream. This means that if we materialize a stream, for example one that consumes a live stream of tweets within a minute, the materialized values for those two materializations will be different, as illustrated by this example:

First, let's write such an element counter using `Sink.Aggregate` and see how the types look like:

```
var sumSink = Sink.Aggregate<int, int>(0, (agg, i) => agg + i);

var counterRunnableGraph = tweetsInMinuteFromNow
 .Where(tweet => tweet.IsRetweet)
 .Select(_ => 1)
 .ToMaterialized(sumSink, Keep.Right);

// materialize the stream once in the morning
var morningTweetsCount = counterGraph.Run(mat);
// and once in the evening, reusing the flow
var eveningTweetsCount = counterGraph.Run(mat);
```

Many elements in Akka Streams provide materialized values which can be used for obtaining either results of computation or steering these elements which will be discussed in detail in [Stream Materialization](#). Summing up this section, now we know what happens behind the scenes when we run this one-liner, which is equivalent to the multi line version above:

```
var sum = tweetSource.Select(_ => 1).RunWith(sumSink, mat);
```

[!NOTE] `RunWith()` is a convenience method that automatically ignores the materialized value of any other stages except those appended by the `RunWith()` itself. In the above example it translates to using `Keep.Right` as the combiner for materialized values.

# Design Principles behind Akka Streams

It took quite a while until we were reasonably happy with the look and feel of the API and the architecture of the implementation, and while being guided by intuition the design phase was very much exploratory research. This section details the findings and codifies them into a set of principles that have emerged during the process.

[!NOTE] As detailed in the introduction keep in mind that the Akka Streams API is completely decoupled from the Reactive Streams interfaces which are just an implementation detail for how to pass stream data between individual processing stages.

## What shall users of Akka Streams expect?

Akka.NET is built upon a conscious decision to offer APIs that are minimal and consistent --as opposed to easy or intuitive. The credo is that we favour explicitness over magic, and if we provide a feature then it must work always, no exceptions. Another way to say this is that we minimize the number of rules a user has to learn instead of trying to keep the rules close to what we think users might expect.

From this follows that the principles implemented by Akka Streams are:

- all features are explicit in the API, no magic
- supreme compositionality: combined pieces retain the function of each part
- exhaustive model of the domain of distributed bounded stream processing

This means that we provide all the tools necessary to express any stream processing topology, that we model all the essential aspects of this domain (back-pressure, buffering, transformations, failure recovery, etc.) and that whatever the user builds is reusable in a larger context.

## Akka Streams does not send dropped stream elements to the dead letter office

One important consequence of offering only features that can be relied upon is the restriction that Akka Streams cannot ensure that all objects sent through a processing topology will be processed. Elements can be dropped for a number of reason:

- plan user code can consume one element in a `Select(...)` stage and produce an entirely different one as its result
- Common stream operators drop elements intentionally, e.g., `take/drop/where/conflate/buffer`
- stream failure will tear down the stream without waiting for processing to finish, all elements that are in flight will be discarded
- stream cancellation will propagate upstream (e.g. from a `take` operator) leading to upstream processing steps being terminated without having processed all of their inputs

This means that sending CLR objects into a stream that needs to be cleaned up will require the user to ensure that this happens outside of the Akka Streams facilities (e.g. by cleaning them up after a time-out or when their results are observed on the stream output, or by other means like finalizers etc.)

## Resulting implementation Constraints

Compositionality entails re-usability of partial stream topologies, which led us to the lifted approach of describing data flows as (partial) graphs that can act as composite sources, flows (a.k.a. pipes) and sinks of data. These building blocks shall then be freely shareable, with the ability to combine them freely to form larger graphs. The representation of these pieces must therefore be an immutable blueprint that is materialized in an explicit step in order to start the

stream processing. The resulting stream processing engine is then also immutable in the sense of having a fixed topology that is prescribed by the blueprint. Dynamic networks need to be modelled by explicitly using the Reactive Streams interfaces for plugging different engines together.

The process of materialization will often create specific objects that are useful to interact with the processing engine once it is running, for example for shutting it down or for extracting metrics. This means that the materialization function produces a result termed the *materialized value of a graph*.

## Interoperation with other Reactive Streams implementations

Akka Streams fully implement the `Reactive Streams` specification and interoperate with all other conformant implementations. We chose to completely separate the Reactive Streams interfaces from the user-level API because we regard them to be an SPI that is not targeted at endusers. In order to obtain a `Publisher` or `Subscriber` from an Akka Stream topology, a corresponding `Sink.AsPublisher` or `Source.AsSubscriber` element must be used.

All stream Processors produced by the default materialization of Akka Streams are restricted to having a single `Subscriber`, additional Subscribers will be rejected. The reason for this is that the stream topologies described using our DSL never require fan-out behavior from the Publisher sides of the elements, all fan-out is done using explicit elements like `Broadcast<T>`.

This means that `Sink.AsPublisher<T>(true)` (for enabling fan-out support) must be used where broadcast behavior is needed for interoperation with other Reactive Streams implementations.

## What shall users of streaming libraries expect?

We expect libraries to be built on top of Akka Streams. In order to allow users to profit from the principles that are described for Akka Streams above, the following rules are established:

- libraries shall provide their users with reusable pieces, i.e expose factories that return graphs, allowing full compositionality
- libraries may optionally and additionally provide facilities that consume and materialize graphs

The reasoning behind the first rule is that compositionality would be destroyed if different libraries only accepted graphs and expected to materialize them: using two of these together would be impossible because materialization can only happen once. As a consequence, the functionality of a library must be expressed such that materialization can be done by the user, outside of the library's control.

The second rule allows a library to additionally provide nice sugar for the common case.

[!NOTE] One important consequence of this is that a reusable flow description cannot be bound to "live" resources, any connection to or allocation of such resources must be deferred until materialization time. Examples of "live" resources are already existing TCP connections, a multicast Publisher, etc.; a TickSource does not fall into this category if its timer is created only upon materialization (as is the case for our implementation). Exceptions from this need to be well-justified and carefully documented.

## Resulting Implementation Constraints

Akka Streams must enable a library to express any stream processing utility in terms of immutable blueprints. The most common building blocks are

- Source: something with exactly one output stream
- Sink: something with exactly one input stream

- Flow: something with exactly one input and one output stream
- BidiFlow: something with exactly two input streams and two output streams that conceptually behave like two Flows of opposite direction
- Graph: a packaged stream processing topology that exposes a certain set of input and output ports, characterized by an object of type `Shape`.

[!NOTE] A source that emits a stream of streams is still just a normal Source, the kind of elements that are produced does not play a role in the static stream topology that is being expressed.

## The difference between Error and Failure

The starting point for this discussion is the definition given by the [Reactive Manifesto](#). Translated to streams this means that an error is accessible within the stream as a normal data element, while a failure means that the stream itself has failed and is collapsing. In concrete terms, on the Reactive Streams interface level data elements (including errors) are signalled via `onNext` while failures raise the `onError` signal.

[!NOTE] Unfortunately the method name for signalling *failure* to a Subscriber is called `onError` for historical reasons. Always keep in mind that the Reactive Streams interfaces (Publisher/Subscription/Subscriber) are modeling the low-level infrastructure for passing streams between execution units, and errors on this level are precisely the failures that we are talking about on the higher level that is modelled by Akka Streams.

There is only limited support for treating `onError` in Akka Streams compared to the operators that are available for the transformation of data elements, which is intentional in the spirit of the previous paragraph. Since `onError` signals that the stream is collapsing, its ordering semantics are not the same as for stream completion: transformation stages of any kind will just collapse with the stream, possibly still holding elements in implicit or explicit buffers. This means that data elements emitted before a failure can still be lost if the `onError` overtakes them.

The ability for failures to propagate faster than data elements is essential for tearing down streams that are back-pressed --especially since back-pressure can be the failure mode (e.g. by tripping upstream buffers which then abort because they cannot do anything else; or if a dead-lock occurred).

## The semantics of stream recovery

A recovery element (i.e. any transformation that absorbs an `onError` signal and turns that into possibly more data elements followed normal stream completion) acts as a bulkhead that confines a stream collapse to a given region of the stream topology. Within the collapsed region buffered elements may be lost, but the outside is not affected by the failure.

This works in the same fashion as a `try-catch` expression: it marks a region in which exceptions are caught, but the exact amount of code that was skipped within this region in case of a failure might not be known precisely--the placement of statements matters.

# Basics and working with Flows

## Core concepts

Akka Streams is a library to process and transfer a sequence of elements using bounded buffer space. This latter property is what we refer to as *boundedness* and it is the defining feature of Akka Streams. Translated to everyday terms it is possible to express a chain (or as we see later, graphs) of processing entities, each executing independently (and possibly concurrently) from the others while only buffering a limited number of elements at any given time. This property of bounded buffers is one of the differences from the actor model, where each actor usually has an unbounded, or a bounded, but dropping mailbox. Akka Stream processing entities have bounded "mailboxes" that do not drop.

Before we move on, let's define some basic terminology which will be used throughout the entire documentation:

### **Stream**

An active process that involves moving and transforming data.

### **Element**

An element is the processing unit of streams. All operations transform and transfer elements from upstream to downstream. Buffer sizes are always expressed as number of elements independently from the actual size of the elements.

### **Back-pressure**

A means of flow-control, a way for consumers of data to notify a producer about their current availability, effectively slowing down the upstream producer to match their consumption speeds. In the context of Akka Streams back-pressure is always understood as *non-blocking* and *asynchronous*.

### **Non-Blocking**

Means that a certain operation does not hinder the progress of the calling thread, even if it takes long time to finish the requested operation.

### **Graph**

A description of a stream processing topology, defining the pathways through which elements shall flow when the stream is running.

### **Processing Stage**

The common name for all building blocks that build up a `Graph`. Examples of a processing stage would be operations like `Select()`, `Where()`, custom `GraphStage`'s and graph junctions like `Merge` or `Broadcast`. For the full list of built-in processing stages see [stages overview](#)

When we talk about *asynchronous*, *non-blocking* *backpressure* we mean that the processing stages available in Akka Streams will not use blocking calls but asynchronous message passing to exchange messages between each other, and they will use asynchronous means to slow down a fast producer, without blocking its thread. This is a thread-pool friendly design, since entities that need to wait (a fast producer waiting on a slow consumer) will not block the thread but can hand it back for further use to an underlying thread-pool.

## Defining and running streams

Linear processing pipelines can be expressed in Akka Streams using the following core abstractions:

**Source**

A processing stage with *exactly one output*, emitting data elements whenever downstream processing stages are ready to receive them.

**Sink**

A processing stage with *exactly one input*, requesting and accepting data elements possibly slowing down the upstream producer of elements

**Flow**

A processing stage which has *exactly one input and output*, which connects its up- and downstreams by transforming the data elements flowing through it.

**RunnableGraph**

A Flow that has both ends "attached" to a `source` and `sink` respectively, and is ready to be `run()`.

It is possible to attach a `Flow` to a `Source` resulting in a composite source, and it is also possible to prepend a `Flow` to a `Sink` to get a new sink. After a stream is properly terminated by having both a source and a sink, it will be represented by the `RunnableGraph` type, indicating that it is ready to be executed.

It is important to remember that even after constructing the `RunnableGraph` by connecting all the source, sink and different processing stages, no data will flow through it until it is materialized. Materialization is the process of allocating all resources needed to run the computation described by a Graph (in Akka Streams this will often involve starting up Actors). Thanks to Flows being simply a description of the processing pipeline they are *immutable, thread-safe, and freely shareable*, which means that it is for example safe to share and send them between actors, to have one actor prepare the work, and then have it be materialized at some completely different place in the code.

```
var source = Source.From(Enumerable.Range(1, 10));
var sink = Sink.Aggregate<int, int>(0, (agg, i) => agg + i);

// connect the Source to the Sink, obtaining a RunnableGraph
var runnable = source.ToMaterialized(sink, Keep.Right);

// materialize the flow and get the value of the AggregateSink
Task<int> sum = runnable.Run(materializer);
```

After running (materializing) the `RunnableGraph[T]` we get back the materialized value of type T. Every stream processing stage can produce a materialized value, and it is the responsibility of the user to combine them to a new type. In the above example we used `ToMaterialized` to indicate that we want to transform the materialized value of the source and sink, and we used the convenience function `Keep.Right` to say that we are only interested in the materialized value of the sink.

In our example the `AggregateSink` materializes a value of type `Task<int>` which will represent the result of the aggregating process over the stream. In general, a stream can expose multiple materialized values, but it is quite common to be interested in only the value of the Source or the Sink in the stream. For this reason there is a convenience method called `RunWith()` available for `Sink`, `Source` or `Flow` requiring, respectively, a supplied `Source` (in order to run a `Sink`), a `Sink` (in order to run a `Source`) or both a `Source` and a `Sink` (in order to run a `Flow`, since it has neither attached yet).

```
var source = Source.From(Enumerable.Range(1, 10));
var sink = Sink.Aggregate<int, int>(0, (agg, i) => agg + i);

// materialize the flow, getting the Sinks materialized value
Task<int> sum = source.RunWith(sink, materializer);
```

It is worth pointing out that since processing stages are *immutable*, connecting them returns a new processing stage, instead of modifying the existing instance, so while constructing long flows, remember to assign the new value to a variable or run it:

```
var source = Source.From(Enumerable.Range(1, 10));
source.Select(_ => 0); // has no effect on source, since it's immutable
source.RunWith(Sink.Aggregate<int,int>(0, (agg, i) => agg + i), materializer); // 55

var zeroes = source.Select(_ => 0); // returns new Source<Int>, with Select() appended
zeroes.RunWith(Sink.Aggregate<int,int>(0, (agg, i) => agg + i), materializer); // 0
```

[!NOTE] By default Akka Streams elements support **exactly one** downstream processing stage. Making fan-out (supporting multiple downstream processing stages) an explicit opt-in feature allows default stream elements to be less complex and more efficient. Also it allows for greater flexibility on *how exactly* to handle the multicast scenarios, by providing named fan-out elements such as broadcast (signals all down-stream elements) or balance (signals one of available down-stream elements).

In the above example we used the `RunWith` method, which both materializes the stream and returns the materialized value of the given sink or source.

Since a stream can be materialized multiple times, the materialized value will also be calculated new for each such materialization, usually leading to different values being returned each time. In the example below we create two running materialized instance of the stream that we described in the `runnable` variable, and both materializations give us a different `Task` from the map even though we used the same `sink` to refer to the task:

```
// connect the Source to the Sink, obtaining a RunnableGraph
var sink = Sink.Aggregate<int, int>(0, (sum, i) => sum + i);
var runnable = Source.From(Enumerable.Range(1, 10))
 .ToMaterialized(sink, Keep.Right);

// get the materialized value of the AggregateSink
var sum1 = runnable.Run(materializer);
var sum2 = runnable.Run(materializer);

// sum1 and sum2 are different Tasks!
```

## Defining sources, sinks and flows

The objects `Source` and `Sink` define various ways to create sources and sinks of elements. The following examples show some of the most useful constructs (refer to the API documentation for more details):

```
// Create a source from an Iterable
Source.From(new List<int> {1, 2, 3});

// Create a source from a Task
Source.FromTask(Task.FromResult("Hello Streams!"));

// Create a source from a single element
Source.Single("only one element")

// an empty source
Source.Empty<int>();

// Sink that aggregates over the stream and returns a Task
// of the final result as its materialized value
Sink.Aggregate<int, int>(0, (sum, i) => sum + i);

// Sink that returns a Task as its materialized value,
// containing the first element of the stream
Sink.First<int>();
```

```
// A Sink that consumes a stream without doing anything with the elements
Sink.Ignore<int>();

// A Sink that executes a side-effecting call for every element of the stream
Sink.ForEach<string>(Console.WriteLine);
```

There are various ways to wire up different parts of a stream, the following examples show some of the available options:

```
// Explicitly creating and wiring up a Source, Sink and Flow
Source.From(Enumerable.Range(1, 6))
 .Via(Flow.Create<int>().Select(x => x*2))
 .To(Sink.ForEach<int>(x => Console.WriteLine(x.ToString())));

// Starting from a Source
var source = Source.From(Enumerable.Range(1, 6)).Select(x => x * 2);
source.To(Sink.ForEach<int>(x => Console.WriteLine(x.ToString())));

// Starting from a Sink
var sink = Flow.Create<int>()
 .Select(x => x*2)
 .To(Sink.ForEach<int>(x => Console.WriteLine(x.ToString())));
Source.From(Enumerable.Range(1, 6)).To(sink);

// Broadcast to a sink inline
var sink = Sink.ForEach<int>(x => Console.WriteLine(x.ToString()))
 .MapMaterializedValue(_ => NotUsed.Instance);
var otherSink = Flow.Create<int>().AlsoTo(sink).To(Sink.Ignore<int>());
Source.From(Enumerable.Range(1, 6)).To(otherSink);
```

## Illegal stream elements

In accordance to the Reactive Streams specification ([Rule 2.13] (<https://github.com/reactive-streams/reactive-streams-jvm#2.13>)) Akka Streams do not allow `null` to be passed through the stream as an element. In case you want to model the concept of absence of a value we recommend using `Akka.Streams.Util.Option<T>` or `Akka.Util.Either<TA, TB>`.

## Back-pressure explained

Akka Streams implement an asynchronous non-blocking back-pressure protocol standardised by the [Reactive Streams](#) specification, which Akka is a founding member of.

The user of the library does not have to write any explicit back-pressure handling code — it is built in and dealt with automatically by all of the provided Akka Streams processing stages. It is possible however to add explicit buffer stages with overflow strategies that can influence the behavior of the stream. This is especially important in complex processing graphs which may even contain loops (which *must* be treated with very special care, as explained in [Graph cycles, liveness and deadlocks](#)).

The back pressure protocol is defined in terms of the number of elements a downstream `subscriber` is able to receive and buffer, referred to as `demand`. The source of data, referred to as `Publisher` in Reactive Streams terminology and implemented as `Source` in Akka Streams, guarantees that it will never emit more elements than the received total demand for any given `Subscriber`.

[!NOTE] The Reactive Streams specification defines its protocol in terms of `Publisher` and `Subscriber`. These types are **not** meant to be user facing API, instead they serve as the low level building blocks for different Reactive Streams implementations. Akka Streams implements these concepts as `Source`, `Flow` (referred to

as `Processor` in Reactive Streams) and `sink` without exposing the Reactive Streams interfaces directly. > If you need to integrate with other Reactive Stream libraries read [Integrating with Reactive Streams](#).

The mode in which Reactive Streams back-pressure works can be colloquially described as "dynamic push / pull mode", since it will switch between push and pull based back-pressure models depending on the downstream being able to cope with the upstream production rate or not.

To illustrate this further let us consider both problem situations and how the back-pressure protocol handles them:

## Slow Publisher, fast Subscriber

This is the happy case of course – we do not need to slow down the Publisher in this case. However signalling rates are rarely constant and could change at any point in time, suddenly ending up in a situation where the Subscriber is now slower than the Publisher. In order to safeguard from these situations, the back-pressure protocol must still be enabled during such situations, however we do not want to pay a high penalty for this safety net being enabled.

The Reactive Streams protocol solves this by asynchronously signalling from the Subscriber to the Publisher `Request(int n)` signals. The protocol guarantees that the Publisher will never signal *more* elements than the signalled demand. Since the Subscriber however is currently faster, it will be signalling these Request messages at a higher rate (and possibly also batching together the demand - requesting multiple elements in one Request signal). This means that the Publisher should not ever have to wait (be back-pressured) with publishing its incoming elements.

As we can see, in this scenario we effectively operate in so called push-mode since the Publisher can continue producing elements as fast as it can, since the pending demand will be recovered just-in-time while it is emitting elements.

## Fast Publisher, slow Subscriber

This is the case when back-pressuring the `Publisher` is required, because the `Subscriber` is not able to cope with the rate at which its upstream would like to emit data elements.

Since the `Publisher` is not allowed to signal more elements than the pending demand signalled by the `Subscriber`, it will have to abide to this back-pressure by applying one of the below strategies:

- not generate elements, if it is able to control their production rate,
- try buffering the elements in a *bounded* manner until more demand is signalled,
- drop elements until more demand is signalled,
- tear down the stream if unable to apply any of the above strategies.

As we can see, this scenario effectively means that the `Subscriber` will *pull* the elements from the Publisher – this mode of operation is referred to as pull-based back-pressure.

## Stream Materialization

When constructing flows and graphs in Akka Streams think of them as preparing a blueprint, an execution plan. Stream materialization is the process of taking a stream description (the graph) and allocating all the necessary resources it needs in order to run. In the case of Akka Streams this often means starting up Actors which power the processing, but is not restricted to that—it could also mean opening files or socket connections etc.—depending on what the stream needs.

Materialization is triggered at so called "terminal operations". Most notably this includes the various forms of the `Run()` and `RunWith()` methods defined on `Source` and `Flow` elements as well as a small number of special syntactic sugars for running with well-known sinks, such as `RunForeach(e1 => ...)` (being an alias to `RunWith(Sink.Foreach(e1 => ...))`).

Materialization is currently performed synchronously on the materializing thread. The actual stream processing is handled by actors started up during the streams materialization, which will be running on the thread pools they have been configured to run on - which defaults to the dispatcher set in `MaterializationSettings` while constructing the `ActorMaterializer`.

[!NOTE] Reusing *instances* of linear computation stages (Source, Sink, Flow) inside composite Graphs is legal, yet will materialize that stage multiple times.

## Operator Fusion

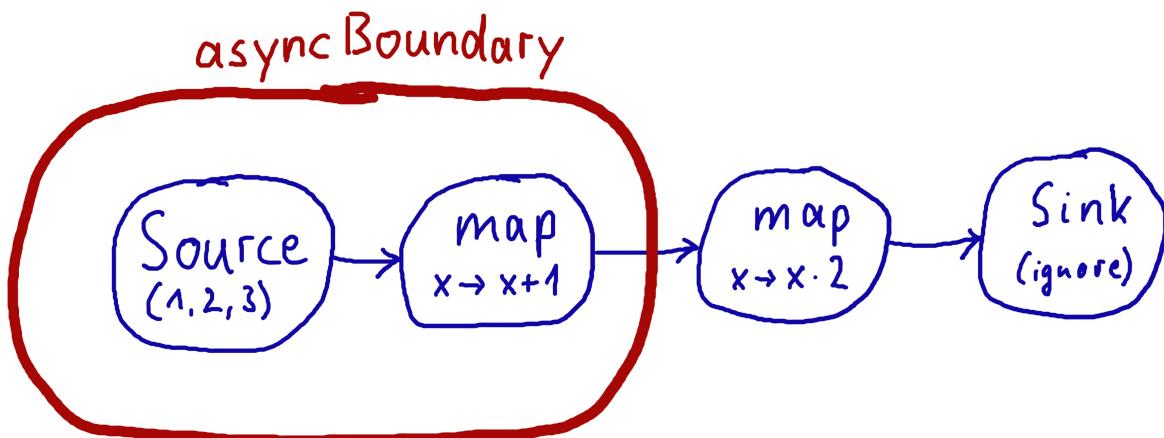
By default Akka Streams will fuse the stream operators. This means that the processing steps of a flow or stream graph can be executed within the same Actor and has two consequences:

- passing elements from one processing stage to the next is a lot faster between fused stages due to avoiding the asynchronous messaging overhead
- fused stream processing stages does not run in parallel to each other, meaning that only up to one CPU core is used for each fused part

To allow for parallel processing you will have to insert asynchronous boundaries manually into your flows and graphs by way of adding `Attributes.AsyncBoundary` using the method `Async` on `Source`, `Sink` and `Flow` to pieces that shall communicate with the rest of the graph in an asynchronous fashion.

```
Source.From(new[] {1, 2, 3})
 .Select(x => x + 1)
 .Async()
 .Select(x => x*2)
 .To(Sink.Ignore<int>());
```

In this example we create two regions within the flow which will be executed in one Actor each—assuming that adding and multiplying integers is an extremely costly operation this will lead to a performance gain since two CPUs can work on the tasks in parallel. It is important to note that asynchronous boundaries are not singular places within a flow where elements are passed asynchronously (as in other streaming libraries), but instead attributes always work by adding information to the flow graph that has been constructed up to this point:



This means that everything that is inside the red bubble will be executed by one actor and everything outside of it by another. This scheme can be applied successively, always having one such boundary enclose the previous ones plus all processing stages that have been added since them.

The new fusing behavior can be disabled by setting the configuration parameter `akka.stream.materializer.auto-fusing=off`. In that case you can still manually fuse those graphs which shall run on less Actors. With the exception of the `SslTlsStage` and the `GroupBy` operator all built-in processing stages can be fused.

## Combining materialized values

Since every processing stage in Akka Streams can provide a materialized value after being materialized, it is necessary to somehow express how these values should be composed to a final value when we plug these stages together. For this, many combinator methods have variants that take an additional argument, a function, that will be used to combine the resulting values. Some examples of using these combiners are illustrated in the example below.

```
// An source that can be signalled explicitly from the outside
Source<int, TaskCompletionSource<int>> source = Source.Maybe<int>();

// A flow that internally throttles elements to 1 / second, and returns a Cancellable
// which can be used to shut down the stream
Flow<int, int, ICancelable> flow = throttler;

// A sink that returns the first element of a stream in the returned Task
Sink<int, Task<int>> sink = Sink.First<int>();

// By default, the materialized value of the leftmost stage is preserved
IRunnableGraph<TaskCompletionSource<int>> r1 = source.Via(flow).To(sink);

// Simple selection of materialized values by using Keep.right
IRunnableGraph<ICancelable> r2 = source.ViaMaterialized(flow, Keep.Right).To(sink);
IRunnableGraph<Task<int>> r3 = source.Via(flow).ToMaterialized(sink, Keep.Right);

// Using RunWith will always give the materialized values of the stages added
// by RunWith itself
Task<int> r4 = source.Via(flow).RunWith(sink, materializer);
TaskCompletionSource<int> r5 = flow.To(sink).RunWith(source, materializer);
Tuple<TaskCompletionSource<int>, Task<int>> r6 = flow.RunWith(source, sink, materializer);

// Using more complex combinations
IRunnableGraph<Tuple<TaskCompletionSource<int>, ICancelable>> r7 =
 source.ViaMaterialized(flow, Keep.Both).To(sink);

IRunnableGraph<Tuple<TaskCompletionSource<int>, Task<int>>> r8 =
 source.Via(flow).ToMaterialized(sink, Keep.Both);

IRunnableGraph<Tuple<Tuple<TaskCompletionSource<int>, ICancelable>, Task<int>>> r9 =
 source.ViaMaterialized(flow, Keep.Both).ToMaterialized(sink, Keep.Both);

IRunnableGraph<Tuple<ICancelable, Task<int>>> r10 =
 source.ViaMaterialized(flow, Keep.Right).ToMaterialized(sink, Keep.Both);

// It is also possible to map over the materialized values. In r9 we had a
// doubly nested pair, but we want to flatten it out
IRunnableGraph<Tuple<TaskCompletionSource<int>, ICancelable, Task<int>>> r11 =
 r9.MapMaterializedValue(t => Tuple.Create(t.Item1.Item1, t.Item1.Item2, t.Item2));

// Now we can get the resulting materialized values
var tuple = r11.Run(materializer);
var completion = tuple.Item1;
var cancellable = tuple.Item2;
var task = tuple.Item3;

// Type inference works as expected
completion.SetResult(1);
cancellable.Cancel();
int plus3 = task.Result + 3;

// The result of r11 can be also achieved by using the Graph API
RunnableGraph<Tuple<TaskCompletionSource<int>, ICancelable, Task<int>>> r12 =
 RunnableGraph.FromGraph(GraphDsl.Create(source, flow, sink,
 Tuple.Create,
 (builder, src, f, dst) =>
 {
 builder.From(src).Via(f).To(dst);
 return ClosedShape.Instance;
```

```
});
```

[!NOTE] In Graphs it is possible to access the materialized value from inside the stream processing graph. For details see [Accessing the materialized value inside the Graph](#).

## Source pre-materialization

There are situations in which you require a `Source` materialized value **before** the `Source` gets hooked up to the rest of the graph. This is particularly useful in the case of "materialized value powered" `Source`s, like `Source.Queue`, `Source.ActorRef` or `Source.Maybe`.

By using the `PreMaterialize` operator on a `Source`, you can obtain its materialized value and another `Source`. The latter can be used to consume messages from the original `Source`. Note that this can be materialized multiple times.

[!code-csharpFlowDocTests.cs]

```
public class FlowDocTests : TestKit
{
 [Fact]
 public void Source_prematerialization()
 {
 #region source-prematerialization

 var matPoweredSource =
 Source.ActorRef<string>(bufferSize: 100, overflowStrategy: OverflowStrategy.Fail);

 Tuple<IActorRef, Source<string, NotUsed>> materialized = matPoweredSource.PreMaterialize(Sys.Materializer());

 var actorRef = materialized.Item1;
 var source = materialized.Item2;

 actorRef.Tell("hit");

 // pass source around for materialization
 source.RunWith(Sink.ForEach<string>(Console.WriteLine), Sys.Materializer());

 #endregion
 }
}
```

## Stream ordering

In Akka Streams almost all computation stages *preserve input order* of elements. This means that if inputs  $\{IA_1, IA_2, \dots, IA_n\}$  "cause" outputs  $\{OA_1, OA_2, \dots, OA_k\}$  and inputs  $\{IB_1, IB_2, \dots, IB_m\}$  "cause" outputs  $\{OB_1, OB_2, \dots, OB_l\}$  and all of  $IA_i$  happened before all  $IB_j$  then  $OA_i$  happens before  $OB_j$ .

This property is even upheld by async operations such as `SelectAsync`, however an unordered version exists called `SelectAsyncUnordered` which does not preserve this ordering.

However, in the case of Junctions which handle multiple input streams (e.g. `Merge`) the output order is, in general, *not defined* for elements arriving on different input ports. That is a merge-like operation may emit  $A_i$  before emitting  $B_i$ , and it is up to its internal logic to decide the order of emitted elements. Specialized elements such as `Zip` however *do guarantee* their outputs order, as each output element depends on all upstream elements having been signalled already – thus the ordering in the case of zipping is defined by this property.

If you find yourself in need of fine grained control over order of emitted elements in fan-in scenarios consider using `MergePreferred`, `MergePrioritized` or `GraphStage` – which gives you full control over how the merge is performed.

# Working with Graphs

In Akka Streams computation graphs are not expressed using a fluent DSL like linear computations are, instead they are written in a more graph-resembling DSL which aims to make translating graph drawings (e.g. from notes taken from design discussions, or illustrations in protocol specifications) to and from code simpler. In this section we'll dive into the multiple ways of constructing and re-using graphs, as well as explain common pitfalls and how to avoid them.

Graphs are needed whenever you want to perform any kind of fan-in ("multiple inputs") or fan-out ("multiple outputs") operations. Considering linear Flows to be like roads, we can picture graph operations as junctions: multiple flows being connected at a single point. Some graph operations which are common enough and fit the linear style of Flows, such as `Concat` (which concatenates two streams, such that the second one is consumed after the first one has completed), may have shorthand methods defined on `Flow` or `Source` themselves, however you should keep in mind that those are also implemented as graph junctions.

## Constructing Graphs

Graphs are built from simple Flows which serve as the linear connections within the graphs as well as junctions which serve as fan-in and fan-out points for Flows. Thanks to the junctions having meaningful types based on their behaviour and making them explicit elements these elements should be rather straightforward to use.

Akka Streams currently provide these junctions (for a detailed list see [Overview of built-in stages and their semantics](#)):

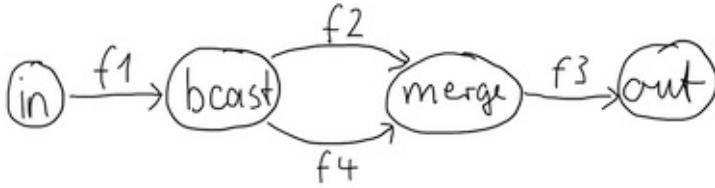
- **Fan-out**

- `Broadcast<T>` – (*1 input, N outputs*) given an input element emits to each output
- `Balance<T>` – (*1 input, N outputs*) given an input element emits to one of its output ports
- `UnzipWith<In, A, B, ...>` – (*1 input, N outputs*) takes a function of 1 input that given a value for each input emits N output elements (where  $N \leq 20$ )
- `UnZip<A, B>` – (*1 input, 2 outputs*) splits a stream of `(A, B)` tuples into two streams, one of type `A` and one of type `B`

- **Fan-in**

- `Merge<In>` – (*N inputs, 1 output*) picks randomly from inputs pushing them one by one to its output
- `MergePreferred<In>` – like `Merge` but if elements are available on `preferred` port, it picks from it, otherwise randomly from `others`
- `MergePrioritized<In>` – like `Merge` but if elements are available on all input ports, it picks from them randomly based on their `priority`
- `ZipWith<A, B, ..., out>` – (*N inputs, 1 output*) which takes a function of  $N$  inputs that given a value for each input emits 1 output element
- `Zip<A, B>` – (*2 inputs, 1 output*) is a `ZipWith` specialised to zipping input streams of `A` and `B` into an `(A, B)` tuple stream
- `Concat<A>` – (*2 inputs, 1 output*) concatenates two streams (first consume one, then the second one)

One of the goals of the GraphDSL DSL is to look similar to how one would draw a graph on a whiteboard, so that it is simple to translate a design from whiteboard to code and be able to relate those two. Let's illustrate this by translating the below hand drawn graph into Akka Streams:



Such graph is simple to translate to the Graph DSL since each linear element corresponds to a `Flow`, and each circle corresponds to either a `Junction` or a `Source` or `Sink` if it is beginning or ending a `Flow`. Junctions must always be created with defined type parameters.

```

var g = RunnableGraph.FromGraph(GraphDsl.Create(builder =>
{
 var source = Source.From(Enumerable.Range(1, 10));
 var sink = Sink.Ignore<int>().MapMaterializedValue(_ => NotUsed.Instance);

 var broadcast = builder.Add(new Broadcast<int>(2));
 var merge = builder.Add(new Merge<int>(2));

 var f1 = Flow.Create<int>().Select(x => x + 10);
 var f2 = Flow.Create<int>().Select(x => x + 10);
 var f3 = Flow.Create<int>().Select(x => x + 10);
 var f4 = Flow.Create<int>().Select(x => x + 10);

 builder.From(source).Via(f1).Via(broadcast).Via(f2).Via(merge).Via(f3).To(sink);
 builder.From(broadcast).Via(f4).To(merge);

 return ClosedShape.Instance;
}));
```

[!NOTE] Junction reference equality defines *graph node equality* (i.e. the same merge *instance* used in a GraphDSL refers to the same location in the resulting graph).

By looking at the snippets above, it should be apparent that the `GraphDSL.Builder` object is *mutable*. The reason for this design choice is to enable simpler creation of complex graphs, which may even contain cycles. Once the GraphDSL has been constructed though, the `GraphDSL` instance is *immutable, thread-safe, and freely shareable*. The same is true of all graph pieces—sources, sinks, and flows—once they are constructed. This means that you can safely re-use one given Flow or junction in multiple places in a processing graph.

We have seen examples of such re-use already above: the merge and broadcast junctions were imported into the graph using `builder.Add(...)`, an operation that will make a copy of the blueprint that is passed to it and return the inlets and outlets of the resulting copy so that they can be wired up. Another alternative is to pass existing graphs—of any shape—into the factory method that produces a new graph. The difference between these approaches is that importing using `builder.Add(...)` ignores the materialized value of the imported graph while importing via the factory method allows its inclusion; for more details see [Stream Materialization](#).

In the example below we prepare a graph that consists of two parallel streams, in which we re-use the same instance of `Flow`, yet it will properly be materialized as two connections between the corresponding Sources and Sinks:

```

var topHeadSink = Sink.First<int>();
var bottomHeadSink = Sink.First<int>();
var sharedDoubler = Flow.Create<int>().Select(x => x*2);

RunnableGraph.FromGraph(GraphDsl.Create(topHeadSink, bottomHeadSink, Keep.Both,
(builder, topHs, bottomHs) =>
{
 var broadcast = builder.Add(new Broadcast<int>(2));
 var source = Source.Single(1).MapMaterializedValue<Tuple<Task<int>, Task<int>>(_ => null);

 builder.From(source).To(broadcast.In);
```

```

 builder.From(broadcast.Out(0)).Via(sharedDoubler).To(topHs.Inlet);
 builder.From(broadcast.Out(1)).Via(sharedDoubler).To(bottomHs.Inlet);

 return ClosedShape.Instance;
 });
}

```

## Constructing and combining Partial Graphs

Sometimes it is not possible (or needed) to construct the entire computation graph in one place, but instead construct all of its different phases in different places and in the end connect them all into a complete graph and run it.

This can be achieved by returning a different `shape` than `ClosedShape`, for example `FlowShape(in, out)`, from the function given to `GraphDSL.Create`. See [Predefined shapes](#) for a list of such predefined shapes. Making a `Graph` a `RunnableGraph` requires all ports to be connected, and if they are not it will throw an exception at construction time, which helps to avoid simple wiring errors while working with graphs. A partial graph however allows you to return the set of yet to be connected ports from the code block that performs the internal wiring.

Let's imagine we want to provide users with a specialized element that given 3 inputs will pick the greatest int value of each zipped triple. We'll want to expose 3 input ports (unconnected sources) and one output port (unconnected sink).

```

var pickMaxOfThree = GraphDSL.Create(b =>
{
 var zip1 = b.Add(ZipWith.Apply<int, int, int>(Math.Max));
 var zip2 = b.Add(ZipWith.Apply<int, int, int>(Math.Max));
 b.From(zip1.Out).To(zip2.In0);

 return new UniformFanInShape<int, int>(zip2.Out, zip1.In0, zip1.In1, zip2.In1);
});

var resultSink = Sink.First<int>();

var g = RunnableGraph.FromGraph(GraphDSL.Create(resultSink, (b, sink) =>
{
 // importing the partial graph will return its shape (inlets & outlets)
 var pm3 = b.Add(pickMaxOfThree);
 var s1 = Source.Single(1).MapMaterializedValue<Task<int>>(_ => null);
 var s2 = Source.Single(2).MapMaterializedValue<Task<int>>(_ => null);
 var s3 = Source.Single(3).MapMaterializedValue<Task<int>>(_ => null);

 b.From(s1).To(pm3.In(0));
 b.From(s2).To(pm3.In(1));
 b.From(s3).To(pm3.In(2));

 b.From(pm3.Out).To(sink.Inlet);

 return ClosedShape.Instance;
}));

var max = g.Run(materializer);
max.Wait(TimeSpan.FromSeconds(3)).Should().BeTrue();
max.Result.Should().Be(3);

```

As you can see, first we construct the partial graph that contains all the zipping and comparing of stream elements. This partial graph will have three inputs and one output, wherefore we use the `UniformFanInShape`. Then we import it (all of its nodes and connections) explicitly into the closed graph built in the second step in which all the undefined elements are rewired to real sources and sinks. The graph can then be run and yields the expected result.

[!WARNING] Please note that `GraphDSL` is not able to provide compile time type-safety about whether or not all elements have been properly connected—this validation is performed as a runtime check during the graph's instantiation. A partial graph also verifies that all ports are either connected or part of the returned `Shape`.

## Constructing Sources, Sinks and Flows from Partial Graphs

Instead of treating a partial graph as simply a collection of flows and junctions which may not yet all be connected it is sometimes useful to expose such a complex graph as a simpler structure, such as a `Source`, `Sink` or `Flow`.

In fact, these concepts can be easily expressed as special cases of a partially connected graph:

- `Source` is a partial graph with *exactly one* output, that is it returns a `SourceShape`.
- `Sink` is a partial graph with *exactly one* input, that is it returns a `SinkShape`.
- `Flow` is a partial graph with *exactly one* input and *exactly one* output, that is it returns a `FlowShape`.

Being able to hide complex graphs inside of simple elements such as `Sink` / `Source` / `Flow` enables you to easily create one complex element and from there on treat it as simple compound stage for linear computations.

In order to create a `Source` from a graph the method `Source.FromGraph` is used, to use it we must have a `IGraph<SourceShape, T>`. This is constructed using `GraphDSL.Create` and returning a `SourceShape` from the function passed in. The single outlet must be provided to the `SourceShape.Of` method and will become "the sink that must be attached before this `Source` can run".

Refer to the example below, in which we create a `Source` that zips together two numbers, to see this graph construction in action:

```
var pairs = Source.FromGraph(GraphDSL.Create(b =>
{
 // prepare graph elements
 var zip = b.Add(new Zip<int, int>());
 Func<Source<int, Task<Tuple<int, int>>> ints = () =>
 Source.From(Enumerable.Range(1, int.MaxValue))
 .MapMaterializedValue<Task<Tuple<int, int>>(_ => null);

 // connect the graph
 b.From(ints().Where(x => x % 2 != 0)).To(zip.In0);
 b.From(ints().Where(x => x % 2 == 0)).To(zip.In1);

 // expose port
 return new SourceShape<Tuple<int, int>>(zip.out);
}));

var firstPair = pairs.RunWith(Sink.First<Tuple<int, int>>(), materializer);
```

Similarly the same can be done for a `Sink<T>`, using `SinkShape.of` in which case the provided value must be an `Inlet<T>`. For defining a `Flow<T>` we need to expose both an inlet and an outlet:

```
var pairUpWithToString = Flow.FromGraph(
 GraphDSL.Create(b =>
{
 // prepare graph elements
 var broadcast = b.Add(new Broadcast<int>(2));
 var zip = b.Add(new Zip<int, string>());

 // connect the graph
 b.From(broadcast.out(0)).Via(Flow.Create<int>().Select(x => x)).To(zip.In0);
 b.From(broadcast.out(1)).Via(Flow.Create<int>().Select(x => x.ToString())).To(zip.In1);

 // expose ports
 return new FlowShape<int, Tuple<int, string>>(broadcast.in, zip.out);
}));
```

```
});

pairUpWithToString.RunWith(Source.From(new[] {1}), Sink.First<Tuple<int, string>>(), materializer);
```

## Combining Sources and Sinks with simplified API

There is a simplified API you can use to combine sources and sinks with junctions like: `Broadcast<T>` , `Merge<In>` and `Concat<A>` without the need for using the Graph DSL. The `combine` method takes care of constructing the necessary graph underneath. In following example we combine two sources into one (fan-in):

```
var sourceOne = Source.Single(1);
var sourceTwo = Source.Single(2);
var merged = Source.Combine(sourceOne, sourceTwo, i => new Merge<int, int>(i));

var mergedResult = merged.RunWith(Sink.Aggregate<int, int>(0, (agg, i) => agg + i), materializer);
```

The same can be done for a `Sink<T>` but in this case it will be fan-out:

```
var sendRemotely = Sink.ActorRef<int>(actorRef, "Done");
var localProcessing = Sink.ForEach<int>(_ => { /* do something usefull */ })
 .MapMaterializedValue(_ => NotUsed.Instance);

var sink = Sink.Combine(i => new Broadcast<int>(i), sendRemotely, localProcessing);

Source.From(new[] {0, 1, 2}).RunWith(sink, materializer);
```

## Building reusable Graph components

It is possible to build reusable, encapsulated components of arbitrary input and output ports using the graph DSL.

As an example, we will build a graph junction that represents a pool of workers, where a worker is expressed as a `Flow<I, O, _>`, i.e. a simple transformation of jobs of type `I` to results of type `O` (as you have seen already, this flow can actually contain a complex graph inside). Our reusable worker pool junction will not preserve the order of the incoming jobs (they are assumed to have a proper ID field) and it will use a `Balance` junction to schedule jobs to available workers. On top of this, our junction will feature a "fast lane", a dedicated port where jobs of higher priority can be sent.

Altogether, our junction will have two input ports of type `I` (for the normal and priority jobs) and an output port of type `O`. To represent this interface, we need to define a custom `Shape`. The following lines show how to do that.

```
public class PriorityWorkerPoolShape<TIn, Tout> : Shape
{
 public PriorityWorkerPoolShape(Inlet<TIn> jobsIn, Inlet<TIn> priorityJobsIn, Outlet<TOut> resultsOut)
 {
 JobsIn = jobsIn;
 PriorityJobsIn = priorityJobsIn;
 ResultsOut = resultsOut;

 Inlets = ImmutableList.Create<Inlet>(jobsIn, priorityJobsIn);
 Outlets = ImmutableList.Create<Outlet>(resultsOut);
 }

 public override ImmutableList<Inlet> Inlets { get; }

 public override ImmutableList<Outlet> Outlets { get; }

 public Inlet<TIn> JobsIn { get; }
```

```

public Inlet<TIn> PriorityJobsIn { get; }

public Outlet<TOut> ResultsOut { get; }

public override Shape DeepCopy()
{
 return new PriorityWorkerPoolShape<TIn, TOut>((Inlet<TIn>)JobsIn.CarbonCopy(),
 (Inlet<TIn>)PriorityJobsIn.CarbonCopy(), (Outlet<TOut>)ResultsOut.CarbonCopy());
}

public override Shape CopyFromPorts(ImmutableArray<Inlet> inlets, ImmutableArray<Outlet> outlets)
{
 if (inlets.Length != Inlets.Length)
 throw new ArgumentException(
 $"Inlets have the wrong length, expected {Inlets.Length} found {inlets.Length}", nameof(inlets));
 if (outlets.Length != Outlets.Length)
 throw new ArgumentException(
 $"Outlets have the wrong length, expected {Outlets.Length} found {outlets.Length}", nameof(outlets));

 // This is why order matters when overriding inlets and outlets.
 return new PriorityWorkerPoolShape<TIn, TOut>((Inlet<TIn>)inlets[0], (Inlet<TIn>)inlets[1],
 (Outlet<TOut>)outlets[0]);
}
}

```

## Predefined shapes

In general a custom `Shape` needs to be able to provide all its input and output ports, be able to copy itself, and also be able to create a new instance from given ports. There are some predefined shapes provided to avoid unnecessary boilerplate:

- `SourceShape`, `SinkShape`, `FlowShape` for simpler shapes,
- `UniformFanInShape` and `UniformFanOutShape` for junctions with multiple input (or output) ports of the same type,
- `FanInShape1`, `FanInShape2`, ..., `FanOutShape1`, `FanOutShape2`, ... for junctions with multiple input (or output) ports of different types.

Since our shape has two input ports and one output port, we can just use the `FanInShape` DSL to define our custom shape:

```

public class PriorityWorkerPoolShape2<TIn, TOut> : FanInShape<TOut>
{
 public PriorityWorkerPoolShape2(IInit init = null)
 : base(init ?? new InitName("PriorityWorkerPool"))
 {

 }

 protected override FanInShape<TOut> Construct(IInit init)
 => new PriorityWorkerPoolShape2<TIn, TOut>(init);

 public Inlet<TIn> JobsIn { get; } = new Inlet<TIn>("JobsIn");

 public Inlet<TIn> PriorityJobsIn { get; } = new Inlet<TIn>("priorityJobsIn");

 // Outlet[Out] with name "out" is automatically created
}

```

Now that we have a `Shape` we can wire up a Graph that represents our worker pool. First, we will merge incoming normal and priority jobs using `MergePreferred`, then we will send the jobs to a `Balance` junction which will fan-out to a configurable number of workers (flows), finally we merge all these results together and send them out through our only output port. This is expressed by the following code:

```
public static class PriorityWorkerPool
{
 public static IGraph<PriorityWorkerPoolShape<TIn, TOut>, NotUsed> Create<TIn, TOut>(
 Flow<TIn, TOut, NotUsed> worker, int workerCount)
 {
 return GraphDsl.Create(b =>
 {
 var priorityMerge = b.Add(new MergePreferred<TIn>(1));
 var balance = b.Add(new Balance<TIn>(workerCount));
 var resultsMerge = b.Add(new Merge<TOut>(workerCount));

 // After merging priority and ordinary jobs, we feed them to the balancer
 b.From(priorityMerge).To(balance);

 // Wire up each of the outputs of the balancer to a worker flow
 // then merge them back
 for (var i = 0; i < workerCount; i++)
 b.From(balance.Out(i)).Via(worker).To(resultsMerge.In(i));

 // We now expose the input ports of the priorityMerge and the output
 // of the resultsMerge as our PriorityWorkerPool ports
 // -- all neatly wrapped in our domain specific Shape
 return new PriorityWorkerPoolShape<TIn, TOut>(jobsIn: priorityMerge.In(0),
 priorityJobsIn: priorityMerge.Preferred, resultsOut: resultsMerge.Out);
 });
 }
}
```

All we need to do now is to use our custom junction in a graph. The following code simulates some simple workers and jobs using plain strings and prints out the results. Actually we used *two* instances of our worker pool junction using `Add()` twice.

```
var worker1 = Flow.Create<string>().Select(s => "step 1 " + s);
var worker2 = Flow.Create<string>().Select(s => "step 2 " + s);

RunnableGraph<GraphDsl> CreateGraph()
{
 Func<string, Source<string, NotUsed>> createSource = desc =>
 Source.From(Enumerable.Range(1, 100))
 .Select(s => desc + s);

 var priorityPool1 = b.Add(PriorityWorkerPool.Create(worker1, 4));
 var priorityPool2 = b.Add(PriorityWorkerPool.Create(worker2, 2));

 b.From(createSource("job: ")).To(priorityPool1.JobsIn);
 b.From(createSource("priority job: ")).To(priorityPool1.PriorityJobsIn);

 b.From(priorityPool1.ResultsOut).To(priorityPool2.JobsIn);
 b.From(createSource("one-step, priority : ")).To(priorityPool2.PriorityJobsIn);

 var sink = Sink.ForEach<string>(Console.WriteLine).MapMaterializedValue(_ => NotUsed.Instance);
 b.From(priorityPool2.ResultsOut).To(sink);
 return ClosedShape.Instance;
}).Run(materializer);
```

## Bidirectional Flows

A graph topology that is often useful is that of two flows going in opposite directions. Take for example a codec stage that serializes outgoing messages and deserializes incoming octet streams. Another such stage could add a framing protocol that attaches a length header to outgoing data and parses incoming frames back into the original octet stream chunks. These two stages are meant to be composed, applying one atop the other as part of a protocol stack. For this purpose exists the special type `BidiFlow` which is a graph that has exactly two open inlets and two open outlets. The corresponding shape is called `BidiShape` and is defined like this:

```
/**
 * A bidirectional flow of elements that consequently has two inputs and two
 * outputs, arranged like this:
 *
 * +-----+
 * In1 ->| | -> Out1
 * | bidi |
 * Out2 <-| | <- In2
 * +-----+
 */
public sealed class BidiShape<TIn1, TOut1, TIn2, TOut2> : Shape
{
 public BidiShape(Inlet<TIn1> in1, Outlet<TOut1> out1, Inlet<TIn2> in2, Outlet<TOut2> out2)
 {
 }

 // implementation details elided ...
}
```

A bidirectional flow is defined just like a unidirectional `Flow` as demonstrated for the codec mentioned above:

```
public interface IMessage { }

public struct Ping : IMessage
{
 public Ping(int id)
 {
 Id = id;
 }

 public int Id { get; }
}

public struct Pong : IMessage
{
 public Pong(int id)
 {
 Id = id;
 }

 public int Id { get; }
}

public static ByteString ToBytes(IMessage message)
{
 // implementation details elided ...
}

public static IMessage FromBytes(ByteString bytes)
{
 // implementation details elided ...
}

var codecVerbose =
 BidiFlow.FromGraph(GraphDsl.Create(b =>
 {
```

```

 // construct and add the top flow, going outbound
 var outbound = b.Add(Flow.Create<IMessage>().Select(ToBytes));
 // construct and add the bottom flow, going inbound
 var inbound = b.Add(Flow.Create<ByteString>().Select(FromBytes));
 // fuse them together into a BidiShape
 return BidiShape.FromFlows(outbound, inbound);
});

// this is the same as the above
var codec = BidiFlow.FromFunction<IMessage, ByteString, ByteString, IMESSAGE>(ToBytes, FromBytes);

```

The first version resembles the partial graph constructor, while for the simple case of a functional 1:1 transformation there is a concise convenience method as shown on the last line. The implementation of the two functions is not difficult either:

```

public static ByteString ToBytes(IMessage message)
{
 var order = ByteOrder.LittleEndian;

 var ping = message as Ping;
 if (ping != null)
 return new ByteStringBuilder().PutByte(1).PutInt(ping.Id, order).Result();

 var pong = message as Pong;
 if (pong != null)
 return new ByteStringBuilder().PutByte(2).PutInt(pong.Id, order).Result();

 throw new ArgumentException("Message is neither Pong nor Ping", nameof(message));
}

public static IMessage FromBytes(ByteString bytes)
{
 var order = ByteOrder.LittleEndian;
 var it = bytes.Iterator();
 var b = it.GetByte();

 if(b == 1)
 return new Ping(it.GetInt(order));
 if(b == 2)
 return new Pong(it.GetInt(order));

 throw new SystemException($"Parse error: expected 1|2 got {b}");
}

```

In this way you could easily integrate any other serialization library that turns an object into a sequence of bytes.

The other stage that we talked about is a little more involved since reversing a framing protocol means that any received chunk of bytes may correspond to zero or more messages. This is best implemented using a `GraphStage` (see also [Custom processing with GraphStage](#)).

```

public static ByteString AddLengthHeader(ByteString bytes, ByteOrder order)
 => new ByteStringBuilder().PutInt(bytes.Count, order).Append(bytes).Result();

public class FrameParser : GraphStage<FlowShape<ByteString, ByteString>>
{
 private sealed class Logic : GraphStageLogic
 {
 private readonly FrameParser _parser;
 // this holds the received but not yet parsed bytes
 private ByteString _stash;
 // this holds the current message length or -1 if at a boundary
 private int _needed = -1;

 public Logic(FrameParser parser) : base(parser.Shape)
 }
}

```

```

 {
 _parser = parser;
 _stash = ByteString.Empty;

 SetHandler(parser.Out, onPull: () =>
 {
 if (IsClosed(parser.In))
 Run();
 else
 Pull(parser.In);
 });

 SetHandler(parser.In, onPush: () =>
 {
 var bytes = Grab(parser.In);
 _stash += bytes;
 Run();
 }, onUpstreamFinish: () =>
 {
 if(_stash.IsEmpty)
 CompleteStage();
 // wait with completion and let Run() complete when the
 // rest of the stash has been sent downstream
 });
 }

 private void Run()
 {
 if (_needed == -1)
 {
 // are we at a boundary? then figure out next length
 if (_stash.Count < 4)
 {
 if (IsClosed(_parser.In))
 CompleteStage();
 else
 Pull(_parser.In);
 }
 else
 {
 _needed = _stashIterator.GetInt(_parser._order);
 _stash = _stash.Drop(4);
 Run(); // cycle back to possibly already emit the next chunk
 }
 }
 else if (_stash.Count < _needed)
 {
 // we are in the middle of a message, need more bytes,
 // or have to stop if input closed
 if (IsClosed(_parser.In))
 CompleteStage();
 else
 Pull(_parser.In);
 }
 else
 {
 // we have enough to emit at least one message, so do it
 var emit = _stash.Take(_needed);
 _stash = _stash.Drop(_needed);
 _needed = -1;
 Push(_parser.Out, emit);
 }
 }
}

private readonly ByteOrder _order;

public FrameParser(ByteOrder order)
{

```

```

 _order = order;
 Shape = new FlowShape<ByteString, ByteString>(In, Out);
 }

 public Inlet<ByteString> In { get; } = new Inlet<ByteString>("FrameParser.in");

 public Outlet<ByteString> Out { get; } = new Outlet<ByteString>("FrameParser.out");

 public override FlowShape<ByteString, ByteString> Shape { get; }

 protected override GraphStageLogic CreateLogic(Attributes inheritedAttributes) => new Logic(this);
}

var framing =
 BidiFlow.FromGraph(
 GraphDsl.Create(b =>
 {
 var order = ByteOrder.LittleEndian;

 var outbound = b.Add(Flow.Create<ByteString>().Select(bytes => AddLengthHeader(bytes, order)));
 var inbound = b.Add(Flow.Create<ByteString>().Via(new FrameParser(order)));

 return BidiShape.FromFlows(outbound, inbound);
 }));

```

With these implementations we can build a protocol stack and test it:

```

/* construct protocol stack
 * +-----+
 * | stack |
 * |
 * | +-----+ +-----+ |
 * ~> 0~o | ~> | 0~o ~>
 * Message | codec | ByteString | framing | ByteString
 * <- 0~o | <- | 0~o <-
 * | +-----+ +-----+ |
 * +-----+-----+
 */

var stack = codec.Atop(framing);

// test it by plugging it into its own inverse and closing the right end
var pingpong = Flow.Create<IMessage>().Collect(message =>
{
 var ping = message as Ping;
 return ping != null
 ? new Pong(ping.Id) as IMessage
 : null;
});
var flow = stack.Atop(stack.Reversed()).Join(pingpong);
var result =
 Source.From(Enumerable.Range(0, 10))
 .Select(i => new Ping(i) as IMessage)
 .Via(flow)
 .Limit(20)
 .RunWith(Sink.Seq<IMessage>(), materializer);

result.Wait(TimeSpan.FromSeconds(1)).Should().BeTrue();
result.Result.ShouldAllBeEquivalentTo(Enumerable.Range(0, 10));

```

This example demonstrates how `BidiFlow` subgraphs can be hooked together and also turned around with the `.Reversed` method. The test simulates both parties of a network communication protocol without actually having to open a network connection—the flows can just be connected directly.

## Accessing the materialized value inside the Graph

In certain cases it might be necessary to feed back the materialized value of a Graph (partial, closed or backing a Source, Sink, Flow or BidiFlow). This is possible by using `builder.MaterializedValue` which gives an `outlet` that can be used in the graph as an ordinary source or outlet, and which will eventually emit the materialized value. If the materialized value is needed at more than one place, it is possible to call `MaterializedValue` any number of times to acquire the necessary number of outlets.

```
var aggregateFlow = Flow.FromGraph(GraphDsl.Create(Sink.Aggregate<int, int>(0, (sum, i) => sum + i), (b, aggregate) =>
{
 var outlet = b.From(b.MaterializedValue)
 .Via(Flow.Create<Task<int>>().SelectAsync(4, x => x))
 .Out;
 return new FlowShape<int, int>(aggregate.Inlet, outlet);
}));
```

Be careful not to introduce a cycle where the materialized value actually contributes to the materialized value. The following example demonstrates a case where the materialized `Task` of a aggregate is fed back to the aggregate itself.

```
var cyclicAggregate = Source.FromGraph(GraphDsl.Create(Sink.Aggregate<int, int>(0, (sum, i) => sum + i),
 (b, aggregate) =>
{
 // - Aggregate cannot complete until its upstream SelectAsync completes
 // - SelectAsync cannot complete until the materialized Task produced by
 // Aggregate completes
 // As a result this Source will never emit anything, and its materialized
 // Task will never complete
 var flow = Flow.Create<Task<int>>().SelectAsync(4, x => x);
 b.From(b.MaterializedValue).Via(flow).To(aggregate);
 return new SourceShape<int>(b.From(b.MaterializedValue).Via(flow).Out);
}));
```

## Graph cycles, liveness and deadlocks

Cycles in bounded stream topologies need special considerations to avoid potential deadlocks and other liveness issues. This section shows several examples of problems that can arise from the presence of feedback arcs in stream processing graphs.

The first example demonstrates a graph that contains a naïve cycle. The graph takes elements from the source, prints them, then broadcasts those elements to a consumer (we just used `Sink.Ignore` for now) and to a feedback arc that is merged back into the main stream via a `Merge` junction.

[!NOTE] The graph DSL allows the connection methods to be reversed, which is particularly handy when writing cycles—as we will see there are cases where this is very helpful.

```
// WARNING! The graph below deadlocks!
RunnableGraph.FromGraph(GraphDsl.Create(b =>
{
 var merge = b.Add(new Merge<int>(2));
 var broadcast = b.Add(new Broadcast<int>(2));
 var print = Flow.Create<int>().Select(s =>
 {
 Console.WriteLine(s);
 return s;
 });
 broadcast.To(merge);
 print.To(merge);
 merge.To(broadcast);
 merge.To(print);
});
```

```

 var sink = Sink.Ignore<int>().MapMaterializedValue(_ => NotUsed.Instance);
 b.From(source).Via(merge).Via(print).Via(broadcast).To(sink);
 b.To(merge).From(broadcast);

 return ClosedShape.Instance;
})));

```

Running this we observe that after a few numbers have been printed, no more elements are logged to the console - all processing stops after some time. After some investigation we observe that:

- through merging from `source` we increase the number of elements flowing in the cycle
- by broadcasting back to the cycle we do not decrease the number of elements in the cycle

Since Akka Streams (and Reactive Streams in general) guarantee bounded processing (see the "Buffering" section for more details) it means that only a bounded number of elements are buffered over any time span. Since our cycle gains more and more elements, eventually all of its internal buffers become full, backpressuring `source` forever. To be able to process more elements from `source` elements would need to leave the cycle somehow.

If we modify our feedback loop by replacing the `Merge` junction with a `MergePreferred` we can avoid the deadlock. `MergePreferred` is unfair as it always tries to consume from a preferred input port if there are elements available before trying the other lower priority input ports. Since we feed back through the preferred port it is always guaranteed that the elements in the cycles can flow.

```

// WARNING! The graph below stops consuming from "source" after a few steps
RunnableGraph.FromGraph(GraphDsl.Create(b =>
{
 var merge = b.Add(new MergePreferred<int>(1));
 var broadcast = b.Add(new Broadcast<int>(2));
 var print = Flow.Create<int>().Select(s =>
 {
 Console.WriteLine(s);
 return s;
 });

 var sink = Sink.Ignore<int>().MapMaterializedValue(_ => NotUsed.Instance);
 b.From(source).Via(merge).Via(print).Via(broadcast).To(sink);
 b.To(merge.Preferred).From(broadcast);

 return ClosedShape.Instance;
}));
```

If we run the example we see that the same sequence of numbers are printed over and over again, but the processing does not stop. Hence, we avoided the deadlock, but `source` is still back-pressed forever, because buffer space is never recovered: the only action we see is the circulation of a couple of initial elements from `source`.

[NOTE] What we see here is that in certain cases we need to choose between boundedness and liveness. Our first example would not deadlock if there would be an infinite buffer in the loop, or vice versa, if the elements in the cycle would be balanced (as many elements are removed as many are injected) then there would be no deadlock.

To make our cycle both live (not deadlocking) and fair we can introduce a dropping element on the feedback arc. In this case we chose the `Buffer()` operation giving it a dropping strategy `OverflowStrategy.DropHead`.

```

RunnableGraph.FromGraph(GraphDsl.Create(b =>
{
 var merge = b.Add(new Merge<int>(2));
 var broadcast = b.Add(new Broadcast<int>(2));
 var print = Flow.Create<int>().Select(s =>
 {
 Console.WriteLine(s);
 return s;
 });

 var sink = Sink.Ignore<int>().MapMaterializedValue(_ => NotUsed.Instance);
 b.From(source).Via(merge).Via(print).Via(broadcast).To(sink);
 b.To(merge).From(broadcast);
```

```

 });
 var buffer = Flow.Create<int>().Buffer(10, OverflowStrategy.DropHead);

 var sink = Sink.Ignore<int>().MapMaterializedValue(_ => NotUsed.Instance);
 b.From(source).Via(merge).Via(print).Via(broadcast).To(sink);
 b.To(merge).Via(buffer).From(broadcast);

 return ClosedShape.Instance;
});
}

```

If we run this example we see that

- The flow of elements does not stop, there are always elements printed
- We see that some of the numbers are printed several times over time (due to the feedback loop) but on average the numbers are increasing in the long term

This example highlights that one solution to avoid deadlocks in the presence of potentially unbalanced cycles (cycles where the number of circulating elements are unbounded) is to drop elements. An alternative would be to define a larger buffer with `overflowStrategy.Fail` which would fail the stream instead of deadlocking it after all buffer space has been consumed.

As we discovered in the previous examples, the core problem was the unbalanced nature of the feedback loop. We circumvented this issue by adding a dropping element, but now we want to build a cycle that is balanced from the beginning instead. To achieve this we modify our first graph by replacing the `Merge` junction with a `zipWith`. Since `zipWith` takes one element from `source` and from the feedback arc to inject one element into the cycle, we maintain the balance of elements.

```

// WARNING! The graph below never processes any elements
RunnableGraph.FromGraph(GraphDsl.Create(b =>
{
 var zip = b.Add(ZipWith.Apply<int, int, int>(Keep.Right));
 var broadcast = b.Add(new Broadcast<int>(2));
 var print = Flow.Create<int>().Select(s =>
 {
 Console.WriteLine(s);
 return s;
 });

 var sink = Sink.Ignore<int>().MapMaterializedValue(_ => NotUsed.Instance);

 b.From(source).To(zip.In0);
 b.From(zip.Out).Via(print).Via(broadcast).To(sink);
 b.To(zip.In1).From(broadcast);

 return ClosedShape.Instance;
}));

```

Still, when we try to run the example it turns out that no element is printed at all! After some investigation we realize that:

- In order to get the first element from `source` into the cycle we need an already existing element in the cycle
- In order to get an initial element in the cycle we need an element from `source`

These two conditions are a typical "chicken-and-egg" problem. The solution is to inject an initial element into the cycle that is independent from `source`. We do this by using a `concat` junction on the backwards arc that injects a single element using `Source.Single`.

[!WARNING] This approach isn't working in the current version of Akka.NET (1.3.0), the graph is still not printing any elements.

```
RunnableGraph.FromGraph(GraphDsl.Create(b =>
```

```
{
 var zip = b.Add(ZipWith.Apply<int, int, int>(Keep.Right));
 var broadcast = b.Add(new Broadcast<int>(2));
 var concat = b.Add(new Concat<int, int>());
 var start = Source.Single(0);
 var print = Flow.Create<int>().Select(s =>
 {
 Console.WriteLine(s);
 return s;
 });
 var sink = Sink.Ignore<int>().MapMaterializedValue(_ => NotUsed.Instance);

 b.From(source).To(zip.In0);
 b.From(zip.Out).Via(print).Via(broadcast).To(sink);

 b.To(zip.In1).Via(concat).From(start);
 b.To(concat).From(broadcast);

 return ClosedShape.Instance;
});
```

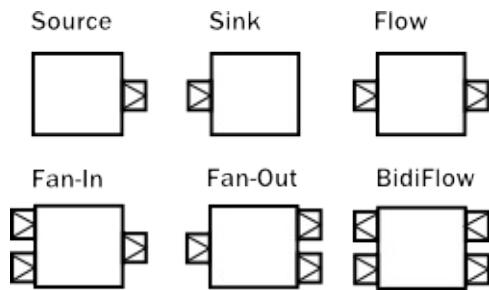
When we run the above example we see that processing starts and never stops. The important takeaway from this example is that balanced cycles often need an initial "kick-off" element to be injected into the cycle.

# Modularity, Composition and Hierarchy

Akka Streams provide a uniform model of stream processing graphs, which allows flexible composition of reusable components. In this chapter we show how these look like from the conceptual and API perspective, demonstrating the modularity aspects of the library.

## Basics of composition and modularity

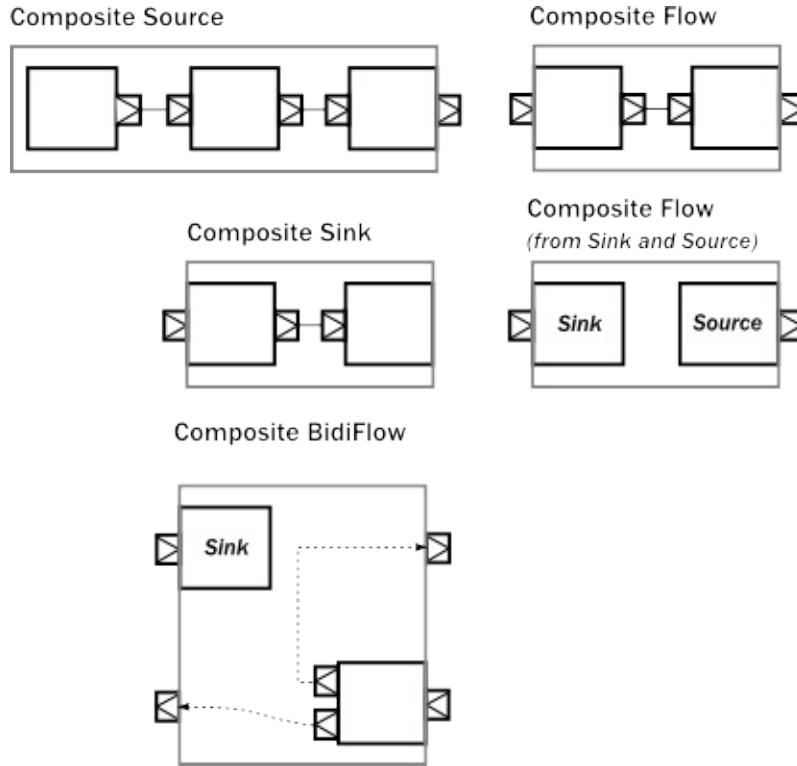
Every processing stage used in Akka Streams can be imagined as a "box" with input and output ports where elements to be processed arrive and leave the stage. In this view, a `Source` is nothing else than a "box" with a single output port, or, a `BidiFlow` is a "box" with exactly two input and two output ports. In the figure below we illustrate the most common used stages viewed as "boxes".



The *linear* stages are `Source`, `Sink` and `Flow`, as these can be used to compose strict chains of processing stages. Fan-in and Fan-out stages have usually multiple input or multiple output ports, therefore they allow to build more complex graph layouts, not just chains. `BidiFlow` stages are usually useful in IO related tasks, where there are input and output channels to be handled. Due to the specific shape of `BidiFlow` it is easy to stack them on top of each other to build a layered protocol for example. The `TLS` support in Akka is for example implemented as a `BidiFlow`.

These reusable components already allow the creation of complex processing networks. What we have seen so far does not implement modularity though. It is desirable for example to package up a larger graph entity into a reusable component which hides its internals only exposing the ports that are meant to the users of the module to interact with. One good example is the `Http` server component, which is encoded internally as a `BidiFlow` which interfaces with the client TCP connection using an input-output port pair accepting and sending `ByteString`s, while its upper ports emit and receive `HttpRequest` and `HttpResponse` instances.

The following figure demonstrates various composite stages, that contain various other type of stages internally, but hiding them behind a *shape* that looks like a `Source`, `Flow`, etc.

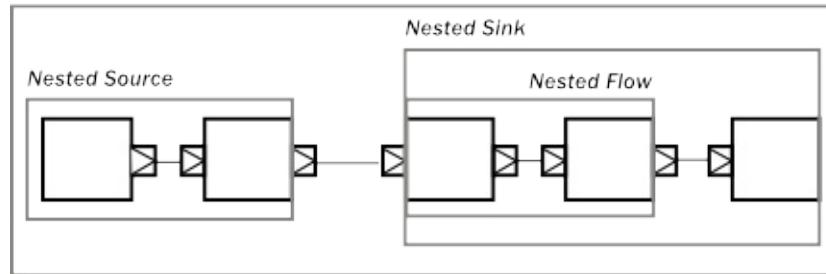


One interesting example above is a `Flow` which is composed of a disconnected `sink` and `source`. This can be achieved by using the `FromSinkAndSource()` constructor method on `Flow` which takes the two parts as parameters.

The example `BidiFlow` demonstrates that internally a module can be of arbitrary complexity, and the exposed ports can be wired in flexible ways. The only constraint is that all the ports of enclosed modules must be either connected to each other, or exposed as interface ports, and the number of such ports needs to match the requirement of the shape, for example a `source` allows only one exposed output port, the rest of the internal ports must be properly connected.

These mechanics allow arbitrary nesting of modules. For example the following figure demonstrates a `RunnableGraph` that is built from a composite `Source` and a composite `Sink` (which in turn contains a composite `Flow`).

#### RunnableGraph



The above diagram contains one more shape that we have not seen yet, which is called `RunnableGraph`. It turns out, that if we wire all exposed ports together, so that no more open ports remain, we get a module that is *closed*. This is what the `RunnableGraph` class represents. This is the shape that a `Materializer` can take and turn into a network of running entities that perform the task described. In fact, a `RunnableGraph` is a module itself, and (maybe somewhat surprisingly) it can be used as part of larger graphs. It is rarely useful to embed a closed graph shape in a larger graph (since it becomes an isolated island as there are no open port for communication with the rest of the graph), but this demonstrates the uniform underlying model.

If we try to build a code snippet that corresponds to the above diagram, our first try might look like this:

```
Source.Single(0)
 .Select(x => x + 1)
```

```
.Where(x => x != 0)
.Select(x => x - 2)
.To(Sink.Aggregate<int, int>(0, (sum, x) => sum + x));
```

It is clear however that there is no nesting present in our first attempt, since the library cannot figure out where we intended to put composite module boundaries, it is our responsibility to do that. If we are using the DSL provided by the `Flow`, `Source`, `Sink` classes then nesting can be achieved by calling one of the methods `WithAttributes()` or `Named()` (where the latter is just a shorthand for adding a name attribute).

The following code demonstrates how to achieve the desired nesting:

```
var nestedSource =
 Source.Single(0) // An atomic source
 .Select(x => x + 1) // an atomic processing stage
 .Named("nestedSource"); // wraps up the current Source and gives it a name

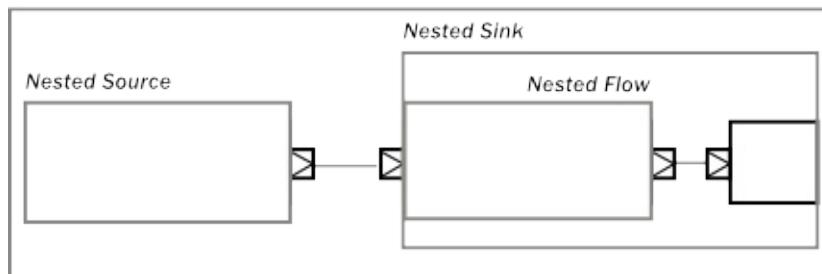
var nestedFlow =
 Flow.Create<int>().Where(x => x != 0) // an atomic processing stage
 .Select(x => x - 2) // another atomic processing stage
 .Named("nestedFlow"); // wraps up the Flow, and gives it a name

var nestedSink = nestedFlow
 .To(Sink.Aggregate<int, int>(0, (sum, x) => sum + x)) // wire an atomic sink to the nestedFlow
 .Named("nestedSink"); // wrap it up

// Create a RunnableGraph
var runnableGraph = nestedSource.To(nestedSink);
```

Once we have hidden the internals of our components, they act like any other built-in component of similar shape. If we hide some of the internals of our composites, the result looks just like if any other predefined component has been used:

#### RunnableGraph



If we look at usage of built-in components, and our custom components, there is no difference in usage as the code snippet below demonstrates.

```
// Create a RunnableGraph
var runnableGraph = nestedSource.To(nestedSink);

// Usage is uniform, no matter if modules are composite or atomic
var runnableGraph2 = Source.Single(0).To(Sink.Aggregate<int, int>(0, (sum, x) => sum + x));
```

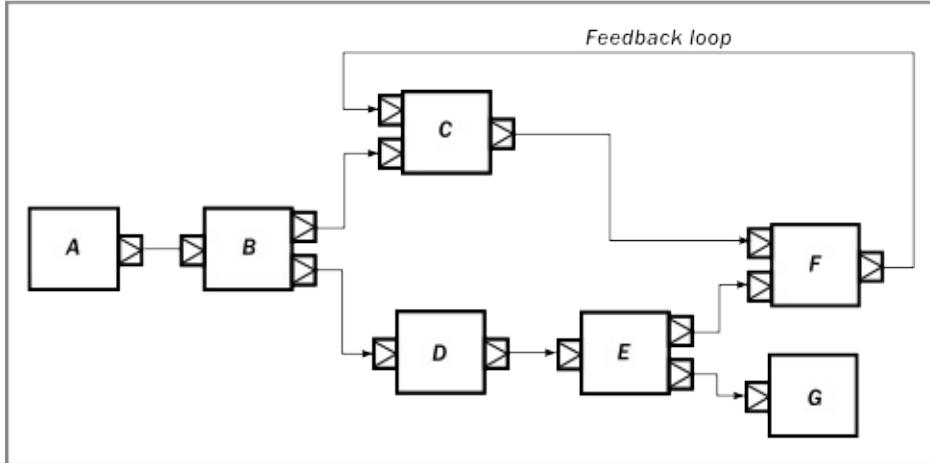
## Composing complex systems

In the previous section we explored the possibility of composition, and hierarchy, but we stayed away from non-linear, generalized graph components. There is nothing in Akka Streams though that enforces that stream processing layouts can only be linear. The DSL for `Source` and friends is optimized for creating such linear chains, as they are the most

common in practice. There is a more advanced DSL for building complex graphs, that can be used if more flexibility is needed. We will see that the difference between the two DSLs is only on the surface: the concepts they operate on are uniform across all DSLs and fit together nicely.

As a first example, let's look at a more complex layout:

### RunnableGraph



The diagram shows a `RunnableGraph` (remember, if there are no unwired ports, the graph is closed, and therefore can be materialized) that encapsulates a non-trivial stream processing network. It contains fan-in, fan-out stages, directed and non-directed cycles. The `Runnable()` method of the `GraphDSL` object allows the creation of a general, closed, and runnable graph. For example the network on the diagram can be realized like this:

```
RunnableGraph.FromGraph(GraphDsl.Create(builder =>
{
 var a = builder.Add(Source.Single(0)).Outlet;
 var b = builder.Add(new Broadcast<int>(2));
 var c = builder.Add(new Merge<int>(2));
 var d = builder.Add(Flow.Create<int>().Select(x => x + 1));
 var e = builder.Add(new Balance<int>(2));
 var f = builder.Add(new Merge<int>(2));
 var g = builder.Add(Sink.ForEach<int>(Console.WriteLine)).Inlet;

 builder.To(c).From(f);
 builder.From(a).Via(b).Via(c).To(f);
 builder.From(b).Via(d).Via(e).To(f);
 builder.From(e).To(g);

 return ClosedShape.Instance;
}));
```

In the code above we used the implicit port numbering feature (to make the graph more readable and similar to the diagram) and we imported `Source`s, `Sink`s and `Flow`s explicitly. It is possible to refer to the ports explicitly, and it is not necessary to import our linear stages via `Add()`, so another version might look like this:

```
RunnableGraph.FromGraph(GraphDsl.Create(builder =>
{
 var b = builder.Add(new Broadcast<int>(2));
 var c = builder.Add(new Merge<int>(2));
 var e = builder.Add(new Balance<int>(2));
 var f = builder.Add(new Merge<int>(2));

 builder.To(c.In(0)).From(f.Out);

 builder.From(Source.Single(0)).To(b.In);
 builder.From(b.Out(0)).To(c.In(1));
 builder.From(c.Out).To(f.In(0));
```

```

builder.From(b.Out(1)).Via(Flow.Create<int>().Select(x => x + 1)).To(e.In);
builder.From(e.Out(0)).To(f.In(1));

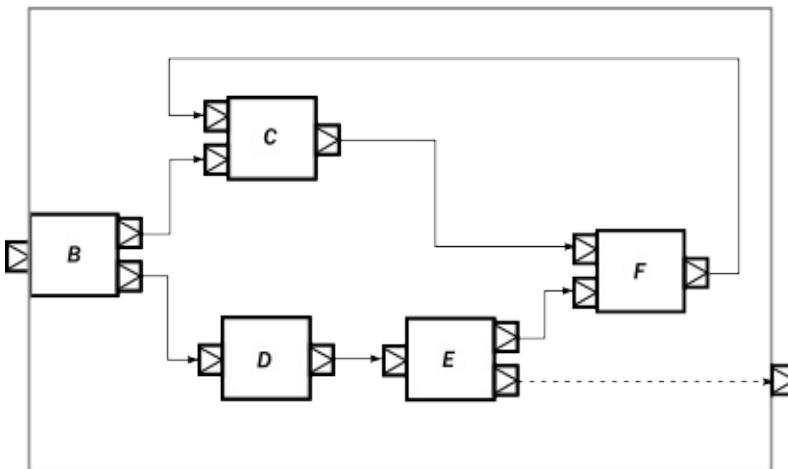
var sink = Sink.ForEach<int>(Console.WriteLine)
 .MapMaterializedValue(_ => NotUsed.Instance);
builder.From(e.Out(1)).To(sink);

return ClosedShape.Instance;
});

```

Similar to the case in the first section, so far we have not considered modularity. We created a complex graph, but the layout is flat, not modularized. We will modify our example, and create a reusable component with the graph DSL. The way to do it is to use the `Create()` factory method on `GraphDSL`. If we remove the sources and sinks from the previous example, what remains is a partial graph:

PartialGraph



We can recreate a similar graph in code, using the DSL in a similar way than before:

```

var partial = GraphDsl.Create(builder =>
{
 var b = builder.Add(new Broadcast<int>(2));
 var c = builder.Add(new Merge<int>(2));
 var e = builder.Add(new Balance<int>(2));
 var f = builder.Add(new Merge<int>(2));

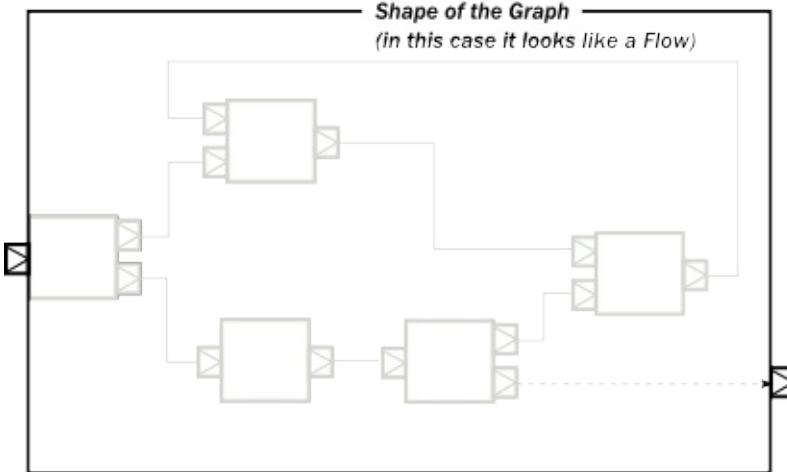
 builder.To(c).From(f);
 builder.From(b).Via(c).To(f);
 builder.From(b).Via(Flow.Create<int>().Select(x => x + 1)).Via(e).To(f);

 return new FlowShape<int, int>(b.In, e.out(1));
}).Named("partial");

```

The only new addition is the return value of the builder block, which is a `shape`. All graphs (including `Source`, `BidiFlow`, etc) have a `shape`, which encodes the *typed* ports of the module. In our example there is exactly one input and output port left, so we can declare it to have a `FlowShape` by returning an instance of it. While it is possible to create new `shape` types, it is usually recommended to use one of the matching built-in ones.

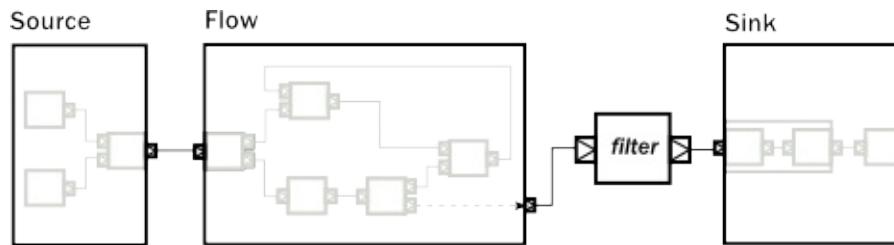
The resulting graph is already a properly wrapped module, so there is no need to call `Named()` to encapsulate the graph, but it is a good practice to give names to modules to help debugging.

**PartialGraph**

Since our partial graph has the right shape, it can be already used in the simpler, linear DSL:

```
Source.Single(0).Via(partial).To(Sink.Ignore<int>());
```

It is not possible to use it as a `Flow` yet, though (i.e. we cannot call `.Where()` on it), but `Flow` has a `FromGraph()` method that just adds the DSL to a `FlowShape`. There are similar methods on `Source`, `Sink` and `Bidishape`, so it is easy to get back to the simpler DSL if a graph has the right shape. For convenience, it is also possible to skip the partial graph creation, and use one of the convenience creator methods. To demonstrate this, we will create the following graph:



The code version of the above closed graph might look like this:

```
// Convert the partial graph of FlowShape to a Flow to get
// access to the fluid DSL (for example to be able to call .Where())
var flow = Flow.FromGraph(partial);

// Simple way to create a graph backed Source
var source = Source.FromGraph(GraphDsl.Create(b =>
{
 var merge = b.Add(new Merge<int>(2));

 b.From(Source.Single(0)).To(merge);
 b.From(Source.From(new[] {2, 3, 4})).To(merge);

 // Exposing exactly one output port
 return new SourceShape<int>(merge.out);
}));

// Building a Sink with a nested Flow, using the fluid DSL
var nestedFlow = Flow.Create<int>().Select(x => x*2).Skip(10).Named("nestedFlow");
var sink = nestedFlow.To(Sink.First<int>());

// Putting all together
var closed = source.Via(flow.Where(x => x > 1)).To(sink);
```

[!NOTE] All graph builder sections check if the resulting graph has all ports connected except the exposed ones and will throw an exception if this is violated.

We are still in debt of demonstrating that `RunnableGraph` is a component just like any other, which can be embedded in graphs. In the following snippet we embed one closed graph in another:

```
var closed1 = Source.Single(0).To(Sink.ForEach<int>(Console.WriteLine));
var closed2 = RunnableGraph.FromGraph(GraphDsl.Create(b =>
{
 var embeddedClosed = b.Add(closed1);
 // ...
 return embeddedClosed;
}));
```

The type of the imported module indicates that the imported module has a `ClosedShape`, and so we are not able to wire it to anything else inside the enclosing closed graph. Nevertheless, this "island" is embedded properly, and will be materialized just like any other module that is part of the graph.

As we have demonstrated, the two DSLs are fully interoperable, as they encode a similar nested structure of "boxes with ports", it is only the DSLs that differ to be as much powerful as possible on the given abstraction level. It is possible to embed complex graphs in the fluid DSL, and it is just as easy to import and embed a `Flow`, etc, in a larger, complex structure.

We have also seen, that every module has a `Shape` (for example a `Sink` has a `SinkShape`) independently which DSL was used to create it. This uniform representation enables the rich compositability of various stream processing entities in a convenient way.

## Materialized values

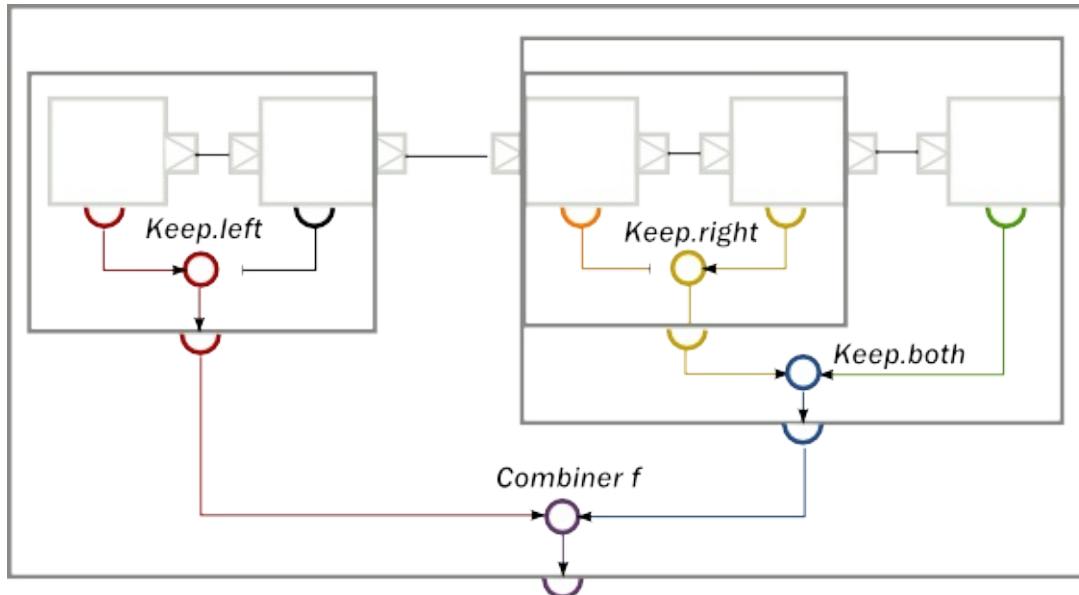
After realizing that `RunnableGraph` is nothing more than a module with no unused ports (it is an island), it becomes clear that after materialization the only way to communicate with the running stream processing logic is via some side-channel. This side channel is represented as a *materialized value*. The situation is similar to `Actor`'s, where the `Props` instance describes the actor logic, but it is the call to `ActorOf()` that creates an actually running actor, and returns an `IActorRef` that can be used to communicate with the running actor itself. Since the `Props` can be reused, each call will return a different reference.

When it comes to streams, each materialization creates a new running network corresponding to the blueprint that was encoded in the provided `RunnableGraph`. To be able to interact with the running network, each materialization needs to return a different object that provides the necessary interaction capabilities. In other words, the `RunnableGraph` can be seen as a factory, which creates:

- a network of running processing entities, inaccessible from the outside
- a materialized value, optionally providing a controlled interaction capability with the network

Unlike actors though, each of the processing stages might provide a materialized value, so when we compose multiple stages or modules, we need to combine the materialized value as well (there are default rules which make this easier, for example `To()` and `Via()` takes care of the most common case of taking the materialized value to the left. See [Combining materialized values](#) for details). We demonstrate how this works by a code example and a diagram which graphically demonstrates what is happening.

The propagation of the individual materialized values from the enclosed modules towards the top will look like this:



To implement the above, first, we create a composite `Source`, where the enclosed `Source` have a materialized type of `Task<NotUsed>`. By using the combiner function `Keep.Left`, the resulting materialized type is of the nested module (indicated by the color `red` on the diagram):

```

// Materializes to TaskCompletionSource<int> (red)
var source = Source.Maybe<int>();

// Materializes to NotUsed (black)
var flow = Flow.Create<int>().Take(100);

// Materializes to TaskCompletionSource<int> (red)
var nestedSource = source.ViaMaterialized(flow, Keep.Left).Named("nestedSource");

```

Next, we create a composite `Flow` from two smaller components. Here, the second enclosed `Flow` has a materialized type of `Task<OutgoingConnection>`, and we propagate this to the parent by using `Keep.Right` as the combiner function (indicated by the color `yellow` on the diagram):

```

// Materializes to NotUsed (orange)
var flow1 = Flow.Create<int>().Select(x => ByteString.FromString(x.ToString()));

// Materializes to Task<OutgoingConnection> (yellow)
var flow2 = Sys.TcpStream().OutgoingConnection("localhost", 8080);

// Materializes to Task<OutgoingConnection> (yellow)
var nestedFlow = flow1.ViaMaterialized(flow2, Keep.Right).Named("nestedFlow");

```

As a third step, we create a composite `Sink`, using our `nestedFlow` as a building block. In this snippet, both the enclosed `Flow` and the folding `Sink` has a materialized value that is interesting for us, so we use `Keep.Both` to get a `Tuple` of them as the materialized type of `nestedSink` (indicated by the color `blue` on the diagram)

```

// Materializes to Task<String> (green)
var sink = Sink.Aggregate<ByteString, string>("", (agg, s) => agg + s.DecodeString());

// Materializes to (Task<OutgoingConnection>, Task<String>) (blue)
var nestedSink = nestedFlow.ToMaterialized(sink, Keep.Both);

```

As the last example, we wire together `nestedSource` and `nestedSink` and we use a custom combiner function to create a yet another materialized type of the resulting `RunnableGraph`. This combiner function just ignores the `Task[Sink]` part, and wraps the other two values in a custom case class `MyClass` (indicated by color *purple* on the diagram):

```
public sealed class MyClass
{
 private readonly TaskCompletionSource<int> _completion;
 private readonly Tcp.OutgoingConnection _connection;

 public MyClass(TaskCompletionSource<int> completion, Tcp.OutgoingConnection connection)
 {
 _completion = completion;
 _connection = connection;
 }

 public void Close() => _completion.SetResult(1);
}

// Materializes to Task<MyClass> (purple)
var runnableGraph = nestedSource.ToMaterialized(nestedSink, (completion, rest) =>
{
 var connectionTask = rest.Item1;
 return connectionTask.ContinueWith(task => new MyClass(completion, task.Result));
});
```

[!NOTE] The nested structure in the above example is not necessary for combining the materialized values, it just demonstrates how the two features work together. See [Combining materialized values](#) for further examples of combining materialized values without nesting and hierarchy involved.

## Attributes

We have seen that we can use `Named()` to introduce a nesting level in the fluid DSL (and also explicit nesting by using `Create()` from `:class: GraphDSL`). Apart from having the effect of adding a nesting level, `Named()` is actually a shorthand for calling `WithAttributes(Attributes.CreateName("someName"))`. Attributes provide a way to fine-tune certain aspects of the materialized running entity. For example buffer sizes for asynchronous stages can be controlled via attributes ([see](#)). When it comes to hierachic composition, attributes are inherited by nested modules, unless they override them with a custom value.

The code below, a modification of an earlier example sets the `InputBuffer` attribute on certain modules, but not on others:

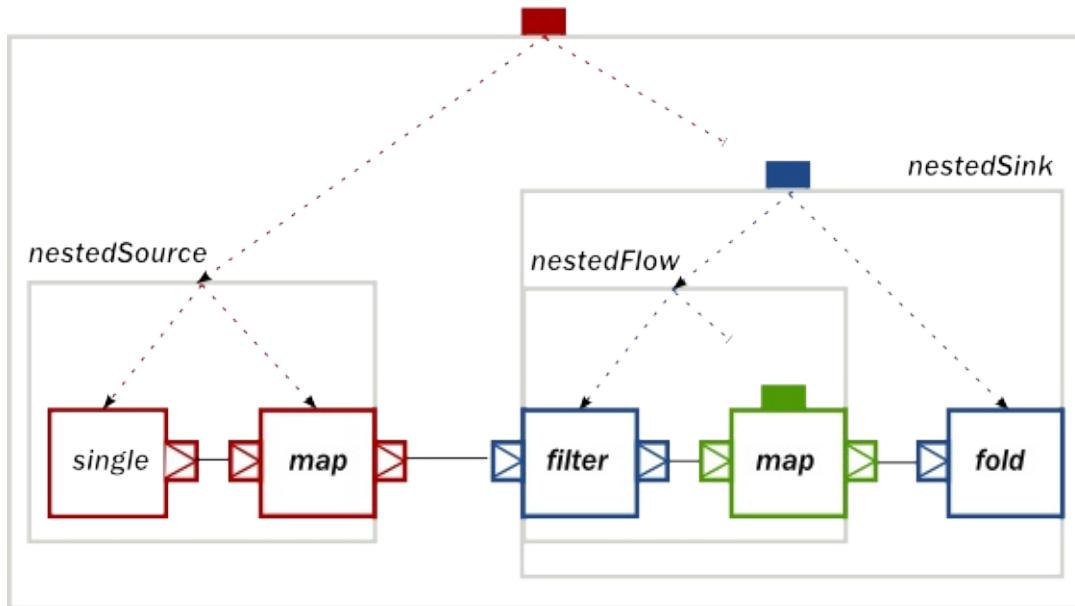
```
var nestedSource = Source.Single(0)
 .Select(x => x + 1)
 .Named("nestedSource"); // Wrap, no inputBuffer set

var nestedFlow =
 Flow.Create<int>()
 .Where(x => x != 0)
 .Via(Flow.Create<int>().Select(x => x - 2).WithAttributes(Attributes.CreateInputBuffer(4, 4)))
 // override
 .Named("nestedFlow"); // Wrap, no inputBuffer set

var nestedSink = nestedFlow
 .To(Sink.Aggregate<int, int>(0, (sum, i) => sum + i)) // wire an atomic sink to the nestedFlow
 .WithAttributes(Attributes.CreateName("nestedSink").And(Attributes.CreateInputBuffer(3, 3))); // override
```

The effect is, that each module inherits the `InputBuffer` attribute from its enclosing parent, unless it has the same attribute explicitly set. `nestedSource` gets the default attributes from the materializer itself. `nestedSink` on the other hand has this attribute set, so it will be used by all nested modules. `nestedFlow` will inherit from `nestedSink` except the `Select` stage which has again an explicitly provided attribute overriding the inherited one.

### Materializer Defaults and top level Attributes



This diagram illustrates the inheritance process for the example code (representing the materializer default attributes as the color *red*, the attributes set on `nestedSink` as *blue* and the attributes set on `nestedFlow` as *green*).

# Buffers and working with rate

When upstream and downstream rates differ, especially when the throughput has spikes, it can be useful to introduce buffers in a stream. In this chapter we cover how buffers are used in Akka Streams.

## Buffers for asynchronous stages

In this section we will discuss internal buffers that are introduced as an optimization when using asynchronous stages. To run a stage asynchronously it has to be marked explicitly as such using the `.async` method. Being run asynchronously means that a stage, after handing out an element to its downstream consumer is able to immediately process the next message. To demonstrate what we mean by this, let's take a look at the following example:

```
var source = Source.From(Enumerable.Range(1, 100))
 .Select(i => { Console.WriteLine($"A: {i}"); return i; }).Async()
 .Select(i => { Console.WriteLine($"B: {i}"); return i; }).Async()
 .Select(i => { Console.WriteLine($"C: {i}"); return i; }).Async()
 .RunWith(Sink.Ignore<int>(), materializer);
```

Running the above example, one of the possible outputs looks like this:

```
A: 2
B: 1
A: 3
B: 2
C: 1
B: 3
C: 2
C: 3
```

Note that the order is *not* `A:1, B:1, C:1, A:2, B:2, C:2`, which would correspond to the normal fused synchronous execution model of flows where an element completely passes through the processing pipeline before the next element enters the flow. The next element is processed by an asynchronous stage as soon as it is emitted the previous one.

While pipelining in general increases throughput, in practice there is a cost of passing an element through the asynchronous (and therefore thread crossing) boundary which is significant. To amortize this cost Akka Streams uses a windowed, batching backpressure strategy internally. It is windowed because as opposed to a Stop-And-Wait protocol multiple elements might be "in-flight" concurrently with requests for elements. It is also batching because a new element is not immediately requested once an element has been drained from the window-buffer but multiple elements are requested after multiple elements have been drained. This batching strategy reduces the communication cost of propagating the backpressure signal through the asynchronous boundary.

While this internal protocol is mostly invisible to the user (apart from its throughput increasing effects) there are situations when these details get exposed. In all of our previous examples we always assumed that the rate of the processing chain is strictly coordinated through the backpressure signal causing all stages to process no faster than the throughput of the connected chain. There are tools in Akka Streams however that enable the rates of different segments of a processing chain to be "detached" or to define the maximum throughput of the stream through external timing sources. These situations are exactly those where the internal batching buffering strategy suddenly becomes non-transparent.

## Internal buffers and their effect

As we have explained, for performance reasons Akka Streams introduces a buffer for every asynchronous processing stage. The purpose of these buffers is solely optimization, in fact the size of 1 would be the most natural choice if there would be no need for throughput improvements. Therefore it is recommended to keep these buffer sizes small, and increase them only to a level suitable for the throughput requirements of the application. Default buffer sizes can be set through configuration:

```
akka.stream.materializer.max-input-buffer-size = 16
```

Alternatively they can be set by passing a `ActorMaterializerSettings` to the materializer:

```
var materializer = ActorMaterializer.Create(system,
 ActorMaterializerSettings.Create(system).WithInputBuffer(64, 64));
```

If the buffer size needs to be set for segments of a Flow only, it is possible by defining a separate Flow with these attributes:

```
var section = Flow.Create<int>()
 .Select(_ => _ * 2)
 .Async()
 .WithAttributes(Attributes.CreateInputBuffer(1, 1)); // the buffer size of this map is 1

var flow = Flow.FromGraph(section)
 .Via(Flow.Create<int>().Select(_ => _ / 2))
 .Async(); // the buffer size of this map is the default
```

Here is an example of a code that demonstrate some of the issues caused by internal buffers:

```
RunnableGraph.FromGraph(GraphDsl.Create(b => {
 // this is the asynchronous stage in this graph
 var zipper = b.Add(ZipWith.Apply<Tick, int, int>((tick, count) => count).Async());

 var s = b.Add(Source.Tick(TimeSpan.FromSeconds(3), TimeSpan.FromSeconds(3), new Tick()));
 b.From(s).To(zipper.In0);

 var s2 = b.Add(Source.Tick(TimeSpan.FromSeconds(1), TimeSpan.FromSeconds(1), "message!"))
 .ConflateWithSeed(seed => 1, (count, _) => count + 1);

 b.From(s2).To(zipper.In1);

 b.From(zipper.Out).To(Sink.ForEach<int>(i => Console.WriteLine($"test: {i}"))
 .MapMaterializedValue(_ => NotUsed.Instance));

 return ClosedShape.Instance;
}));
```

Running the above example one would expect the number 3 to be printed in every 3 seconds (the `conflateWithSeed` step here is configured so that it counts the number of elements received before the downstream `ZipWith` consumes them). What is being printed is different though, we will see the number 1. The reason for this is the internal buffer which is by default 16 elements large, and prefetches elements before the `ZipWith` starts consuming them. It is possible to fix this issue by changing the buffer size of `ZipWith` (or the whole graph) to 1. We will still see a leading 1 though which is caused by an initial prefetch of the `ZipWith` element.

[!NOTE] In general, when time or rate driven processing stages exhibit strange behavior, one of the first solutions to try should be to decrease the input buffer of the affected elements to 1.

## Buffers in Akka Streams

In this section we will discuss explicit user defined buffers that are part of the domain logic of the stream processing pipeline of an application.

The example below will ensure that 1000 jobs (but not more) are dequeued from an external (imaginary) system and stored locally in memory -- relieving the external system:

```
// Getting a stream of jobs from an imaginary external system as a Source
Source<Job, NotUsed> jobs = inboundJobsConnector()
 .buffer(1000, OverflowStrategy.backpressure);
```

The next example will also queue up 1000 jobs locally, but if there are more jobs waiting in the imaginary external systems, it makes space for the new element by dropping one element from the tail of the buffer. Dropping from the tail is a very common strategy but it must be noted that this will drop the youngest waiting job. If some "fairness" is desired in the sense that we want to be nice to jobs that has been waiting for long, then this option can be useful.

```
jobs.buffer(1000, OverflowStrategy.dropTail)
```

Instead of dropping the youngest element from the tail of the buffer a new element can be dropped without enqueueing it to the buffer at all.

```
jobs.buffer(1000, OverflowStrategy.dropNew)
```

Here is another example with a queue of 1000 jobs, but it makes space for the new element by dropping one element from the head of the buffer. This is the oldest waiting job. This is the preferred strategy if jobs are expected to be resent if not processed in a certain period. The oldest element will be retransmitted soon, (in fact a retransmitted duplicate might be already in the queue!) so it makes sense to drop it first.

```
jobs.buffer(1000, OverflowStrategy.dropHead)
```

Compared to the dropping strategies above, dropBuffer drops all the 1000 jobs it has enqueued once the buffer gets full. This aggressive strategy is useful when dropping jobs is preferred to delaying jobs.

```
jobs.buffer(1000, OverflowStrategy.dropBuffer)
```

If our imaginary external job provider is a client using our API, we might want to enforce that the client cannot have more than 1000 queued jobs otherwise we consider it flooding and terminate the connection. This is easily achievable by the error strategy which simply fails the stream once the buffer gets full.

```
jobs.buffer(1000, OverflowStrategy.fail)
```

## Rate transformation

### Understanding conflate

When a fast producer can not be informed to slow down by backpressure or some other signal, conflate might be useful to combine elements from a producer until a demand signal comes from a consumer.

Below is an example snippet that summarizes fast stream of elements to a standard deviation, mean and count of elements that have arrived while the stats have been calculated.

```
var statsFlow = Flow.Create<double>()
```

```

 .ConflateWithSeed(_ => ImmutableList.Create(_), (agg, acc) => agg.Add(acc))
 .Select(s => {
 var u = s.Sum()/s.Count();

 var se = s.Select(x => Math.Pow(x - u, 2));
 var s = Math.Sqrt(se.Sum() / se.Count());
 return new { s, u, size=s.Count()};
 });
}

```

This example demonstrates that such flow's rate is decoupled. The element rate at the start of the flow can be much higher than the element rate at the end of the flow.

Another possible use of `conflate` is to not consider all elements for summary when producer starts getting too fast. Example below demonstrates how `conflate` can be used to implement random drop of elements when consumer is not able to keep up with the producer.

```

var p = 0.01;
var sampleFlow = Flow.Create<double>()
 .ConflateWithSeed(x => ImmutableList.Create(x), (agg, d) => {
 if (ThreadLocalRandom.Current.NextDouble() < p)
 agg.Add(d);
 return agg;
 }).Concat(identity);

```

## Understanding expand

`Expand` helps to deal with slow producers which are unable to keep up with the demand coming from consumers. `Expand` allows to extrapolate a value to be sent as an element to a consumer.

As a simple use of `expand` here is a flow that sends the same element to consumer when producer does not send any new elements.

```

var lastFlow = Flow.Create<int>()
 .Expand(_ => Enumerable.Repeat(_, int.MaxValue).GetEnumerator());

```

`Expand` also allows to keep some state between demand requests from the downstream. Leveraging this, here is a flow that tracks and reports a drift between fast consumer and slow producer.

```

var driftFlow = Flow.Create<int>()
 .Expand(i => Enumerable.Repeat(0, int.MaxValue).Select(n => new { i, n}).GetEnumerator());

```

Note that all of the elements coming from upstream will go through `expand` at least once. This means that the output of this flow is going to report a drift of zero if producer is fast enough, or a larger drift otherwise.

# Custom stream processing

While the processing vocabulary of Akka Streams is quite rich (see the [Streams Cookbook](#) for examples) it is sometimes necessary to define new transformation stages either because some functionality is missing from the stock operations, or for performance reasons. In this part we show how to build custom processing stages and graph junctions of various kinds.

[!NOTE] A custom graph stage should not be the first tool you reach for, defining graphs using flows and the graph DSL is in general easier and does to a larger extent protect you from mistakes that might be easy to make with a custom `GraphStage`

## Custom processing with `GraphStage`

The `GraphStage` abstraction can be used to create arbitrary graph processing stages with any number of input or output ports. It is a counterpart of the `GraphDSL.Create()` method which creates new stream processing stages by composing others. Where `GraphStage` differs is that it creates a stage that is itself not divisible into smaller ones, and allows state to be maintained inside it in a safe way.

As a first motivating example, we will build a new `Source` that will simply emit numbers from 1 until it is cancelled. To start, we need to define the "interface" of our stage, which is called shape in Akka Streams terminology (this is explained in more detail in the section [Modularity, Composition and Hierarchy](#)). This is how this looks like:

```
using Akka.Streams.Stage;

class NumbersSource : GraphStage<SourceShape<int>>
{
 //this is where the actual (possibly statefull) logic will live
 private sealed class Logic : GraphStageLogic
 {
 public Logic(NumbersSource source) : base(source.Shape)
 {
 }
 }

 // Define the (sole) output port of this stage
 public Outlet<int> Out { get; } = new Outlet<int>("NumbersSource");

 // Define the shape of this stage, which is SourceShape with the port we defined above
 public override SourceShape<int> Shape => new SourceShape<int>(Out);

 //this is where the actual logic will be created
 protected override GraphStageLogic CreateLogic(Attributes inheritedAttributes) => new Logic(this);
}
```

As you see, in itself the `GraphStage` only defines the ports of this stage and a shape that contains the ports. It also has, a currently unimplemented method called `CreateLogic`. If you recall, stages are reusable in multiple materializations, each resulting in a different executing entity. In the case of `GraphStage` the actual running logic is modelled as an instance of a `GraphStageLogic` which will be created by the materializer by calling the `CreateLogic` method. In other words, all we need to do is to create a suitable logic that will emit the numbers we want.

[!NOTE] It is very important to keep the `GraphStage` object itself immutable and reusable. All mutable state needs to be confined to the `GraphStageLogic` that is created for every materialization.

In order to emit from a `source` in a backpressured stream one needs first to have demand from downstream. To receive the necessary events one needs to register a callback with the output port (`outlet`). This callback will receive events related to the lifecycle of the port. In our case we need to override `onPull` which indicates that we are free to emit a single element. There is another callback, `onDownstreamFinish` which is called if the downstream cancelled. Since the default behavior of that callback is to stop the stage, we don't need to override it. In the `onPull` callback we will simply emit the next number. This is how it looks like in the end:

```
using Akka.Streams.Stage;

class NumbersSource : GraphStage<SourceShape<int>>
{
 //this is where the actual (possibly statefull) logic will live
 private sealed class Logic : GraphStageLogic
 {
 // All state MUST be inside the GraphStageLogic,
 // never inside the enclosing GraphStage.
 // This state is safe to access and modify from all the
 // callbacks that are provided by GraphStageLogic and the
 // registered handlers.
 private int _counter = 1;

 public Logic(NumbersSource source) : base(source.Shape)
 {
 SetHandler(source.out, onPull: () => Push(source.out, _counter++));
 }
 }

 // Define the (sole) output port of this stage
 public Outlet<int> Out { get; } = new Outlet<int>("NumbersSource");

 // Define the shape of this stage, which is SourceShape with the port we defined above
 public override SourceShape<int> Shape => new SourceShape<int>(Out);

 //this is where the actual logic will be created
 protected override GraphStageLogic CreateLogic(Attributes inheritedAttributes) => new Logic(this);
}
```

Instances of the above `GraphStage` are subclasses of `Graph<SourceShape<int>, NotUsed>` which means that they are already usable in many situations, but do not provide the DSL methods we usually have for other `source`s. In order to convert this `Graph` to a proper `Source` we need to wrap it using `Source.FromGraph` (see [Modularity, Composition and Hierarchy](#) for more details about graphs and DSLs). Now we can use the source as any other built-in one:

```
// A GraphStage is a proper Graph, just like what GraphDSL.Create would return
var sourceGraph = new NumbersSource();

// Create a Source from the Graph to access the DSL
var mySource = Source.FromGraph(sourceGraph);

// Returns 55
var result1Task = mySource.Take(10).RunAggregate(0, (sum, next) => sum + next, materializer);

// Returns 5050
var result2Task = mySource.Take(100).RunAggregate(0, (sum, next) => sum + next, materializer);
```

## Port states, InHandler and OutHandler

In order to interact with a port (`Inlet` or `Outlet`) of the stage we need to be able to receive events and generate new events belonging to the port. From the `GraphStageLogic` the following operations are available on an output port:

- `Push(out, elem)` pushes an element to the output port. Only possible after the port has been pulled by downstream.

- `Complete(out)` closes the output port normally.
- `Fail(out, exception)` closes the port with a failure signal.

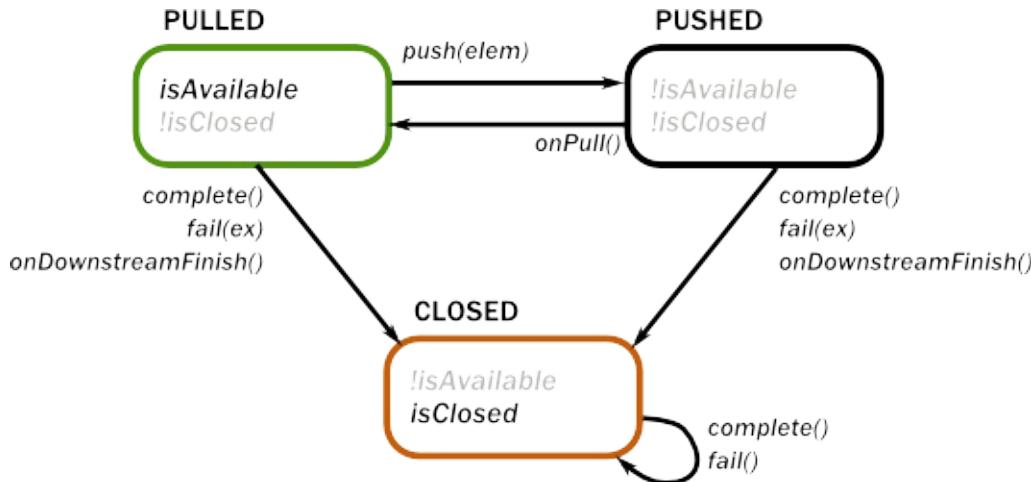
The events corresponding to an *output* port can be received in an `Action` registered to the output port using `setHandler(out, action)`. This handler has two callbacks:

- `onPull` is called when the output port is ready to emit the next element, `Push(out, elem)` is now allowed to be called on this port.
- `onDownstreamFinish` is called once the downstream has cancelled and no longer allows messages to be pushed to it. No more `onPull` will arrive after this event. If not overridden this will default to stopping the stage.

Also, there are two query methods available for output ports:

- `IsAvailable(out)` returns true if the port can be pushed
- `IsClosed(out)` returns true if the port is closed. At this point the port can not be pushed and will not be pulled anymore.

The relationship of the above operations, events and queries are summarized in the state machine below. Green shows the initial state while orange indicates the end state. If an operation is not listed for a state, then it is invalid to call it while the port is in that state. If an event is not listed for a state, then that event cannot happen in that state.



The following operations are available for *input* ports:

- `Pull(in)` requests a new element from an input port. This is only possible after the port has been pushed by upstream.
- `Grab(in)` acquires the element that has been received during an `onPush`. It cannot be called again until the port is pushed again by the upstream.
- `Cancel(in)` closes the input port.

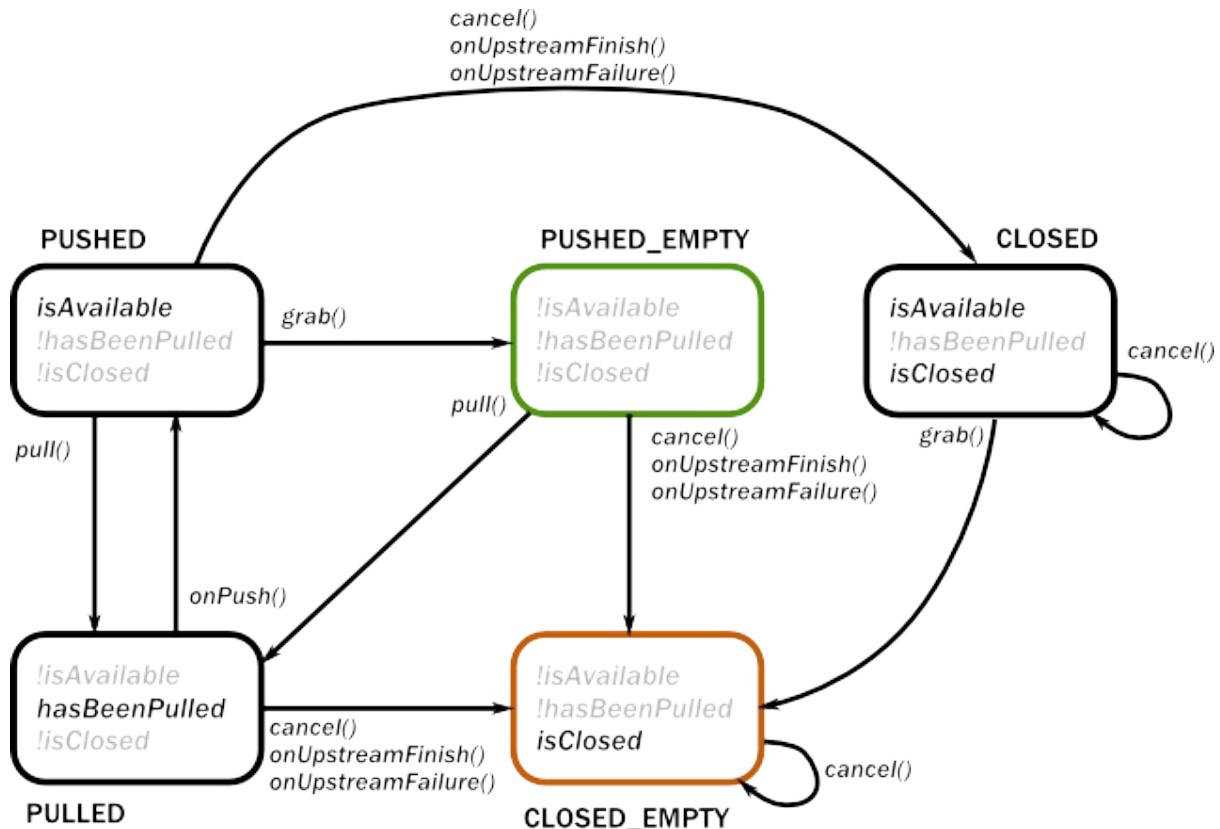
The events corresponding to an *input* port can be received in an `Action` registered to the input port using `setHandler(in, action)`. This handler has three callbacks:

- `onPush` is called when the output port has now a new element. Now it is possible to acquire this element using `Grab(in)` and/or call `Pull(in)` on the port to request the next element. It is not mandatory to grab the element, but if it is pulled while the element has not been grabbed it will drop the buffered element.
- `onUpstreamFinish` is called once the upstream has completed and no longer can be pulled for new elements. No more `onPush` will arrive after this event. If not overridden this will default to stopping the stage.
- `onUpstreamFailure` is called if the upstream failed with an exception and no longer can be pulled for new elements. No more `onPush` will arrive after this event. If not overridden this will default to failing the stage.

Also, there are three query methods available for input ports:

- `IsAvailable(in)` returns true if the port can be grabbed.
- `HasBeenPulled(in)` returns true if the port has been already pulled. Calling `Pull(in)` in this state is illegal.
- `IsClosed(in)` returns true if the port is closed. At this point the port can not be pulled and will not be pushed anymore.

The relationship of the above operations, events and queries are summarized in the state machine below. Green shows the initial state while orange indicates the end state. If an operation is not listed for a state, then it is invalid to call it while the port is in that state. If an event is not listed for a state, then that event cannot happen in that state.



Finally, there are two methods available for convenience to complete the stage and all of its ports:

- `CompleteStage()` is equivalent to closing all output ports and cancelling all input ports.
- `FailStage(exception)` is equivalent to failing all output ports and cancelling all input ports.

In some cases it is inconvenient and error prone to react on the regular state machine events with the signal based API described above. For those cases there is a API which allows for a more declarative sequencing of actions which will greatly simplify some use cases at the cost of some extra allocations. The difference between the two APIs could be described as that the first one is signal driven from the outside, while this API is more active and drives its surroundings.

The operations of this part of the `:class: GraphStage` API are:

- `Emit(out, elem)` and `EmitMultiple(out, Enumerable(elem1, elem2))` replaces the `OutHandler` with a handler that emits one or more elements when there is demand, and then reinstalls the current handlers
- `Read(in, andThen, onClose)` and `ReadMany(in, n, andThen, onClose)` replaces the `InHandler` with a handler that reads one or more elements as they are pushed and allows the handler to react once the requested number of elements has been read.
- `AbortEmitting(out)` and `AbortReading(in)` which will cancel an ongoing emit or read

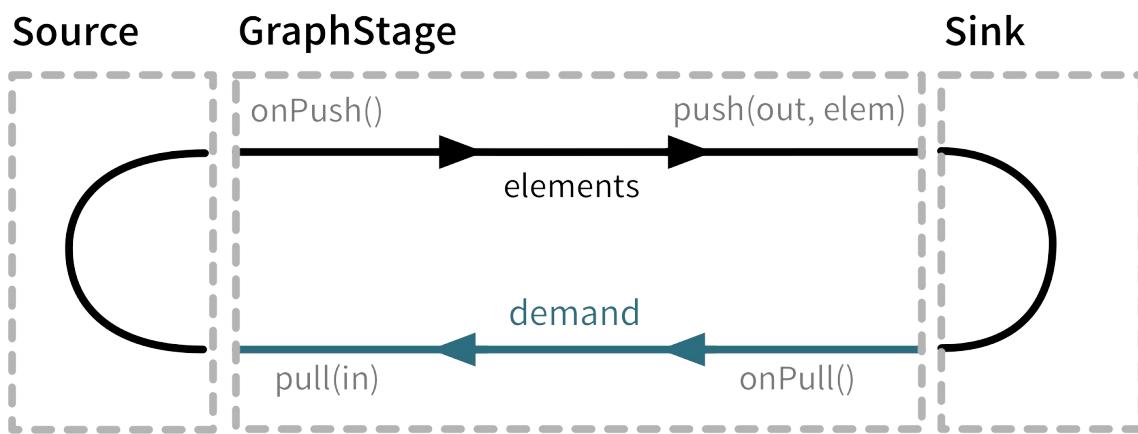
Note that since the above methods are implemented by temporarily replacing the handlers of the stage you should never call `SetHandler` while they are running `Emit` or `Read` as that interferes with how they are implemented. The following methods are safe to call after invoking `Emit` and `Read` (and will lead to actually running the operation when those are done): `Complete(out)`, `CompleteStage()`, `Emit`, `EmitMultiple`, `AbortEmitting()` and `AbortReading()`

An example of how this API simplifies a stage can be found below in the second version of the Duplicator.

## Custom linear processing stages using GraphStage

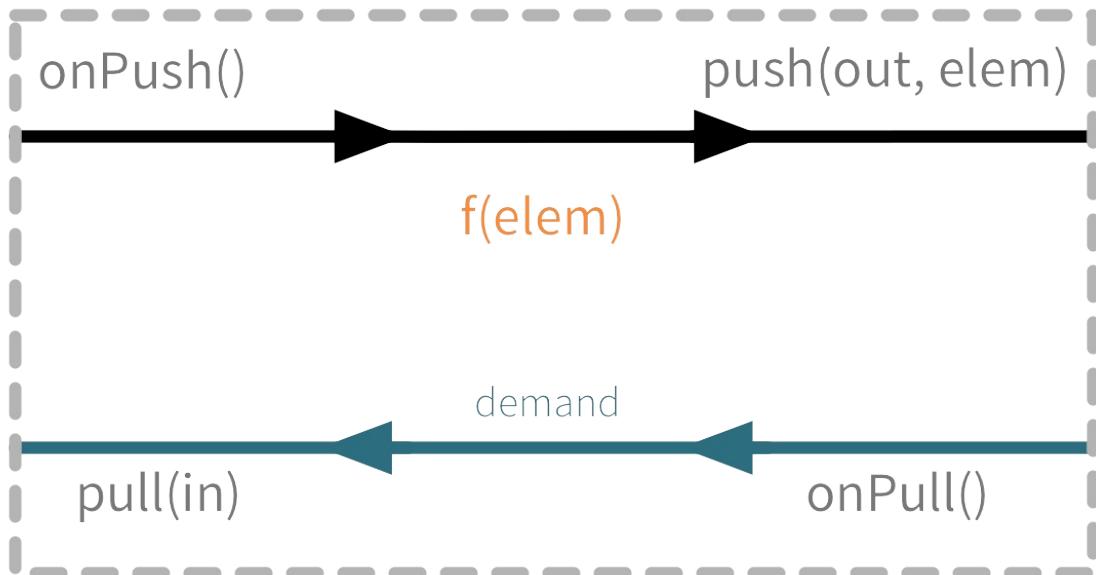
Graph stages allows for custom linear processing stages through letting them have one input and one output and using `FlowShape` as their shape.

Such a stage can be illustrated as a box with two flows as it is seen in the illustration below. Demand flowing upstream leading to elements flowing downstream.



To illustrate these concepts we create a small `GraphStage` that implements the `Map` transformation.

## Map



Map calls `Push(out)` from the `onPush` handler and it also calls `Pull()` from the `onPull` handler resulting in the conceptual wiring above, and fully expressed in code below:

```
class Map<TIn, TOut> : GraphStage<FlowShape<TIn, TOut>>
{
 private sealed class Logic : GraphStageLogic
 {
 public Logic(Map<TIn, TOut> map) : base(map.Shape)
 {
 SetHandler(map.In, onPush: () => Push(map.Out, map._func(Grab(map.In))));
 SetHandler(map.Out, onPull: ()=> Pull(map.In));
 }
 }

 private readonly Func<TIn, TOut> _func;

 public Map(Func<TIn, TOut> func)
 {
 _func = func;
 Shape = new FlowShape<TIn, TOut>(In, Out);
 }

 public Inlet<TIn> In { get; } = new Inlet<TIn>("Map.in");

 public Outlet<TOut> Out { get; } = new Outlet<TOut>("Map.out");

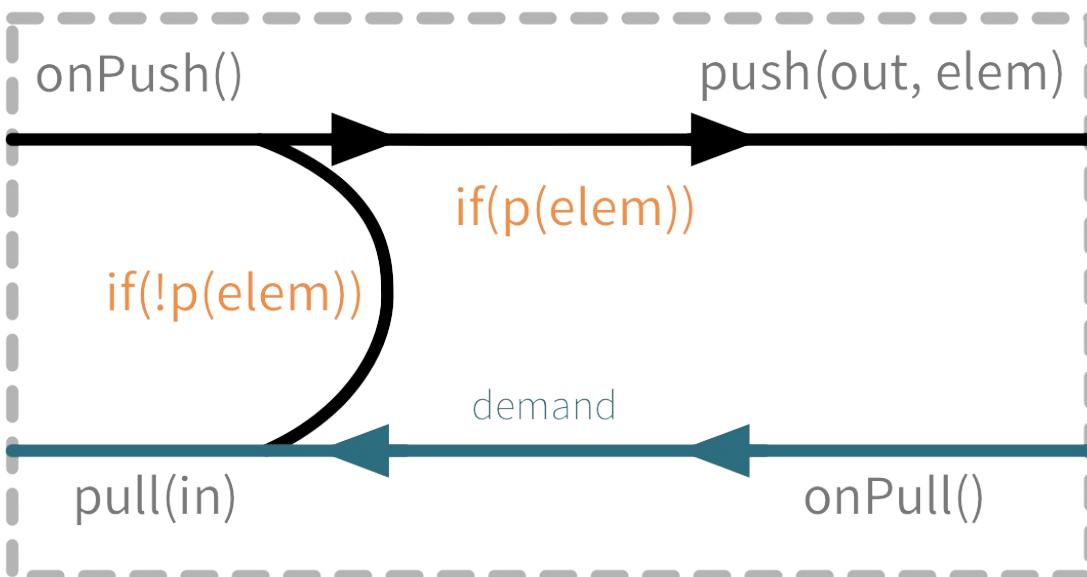
 public override FlowShape<TIn, TOut> Shape { get; }

 protected override GraphStageLogic CreateLogic(Attributes inheritedAttributes) => new Logic(this);
}
```

Map is a typical example of a one-to-one transformation of a stream where demand is passed along upstream elements passed on downstream.

To demonstrate a many-to-one stage we will implement filter. The conceptual wiring of `Filter` looks like this:

## Filter



As we see above, if the given predicate matches the current element we are propagating it downwards, otherwise we return the "ball" to our upstream so that we get the new element. This is achieved by modifying the map example by adding a conditional in the `onPush` handler and decide between a `Pull(in)` or `Push(out)` call (and of course not having a mapping `f` function).

```
class Filter<T> : GraphStage<FlowShape<T, T>>
{
 private sealed class Logic : GraphStageLogic
 {
 public Logic(Filter<T> filter) : base(filter.Shape)
 {
 SetHandler(filter.In, onPush: () =>
 {
 var element = Grab(filter.In);
 if(filter._predicate(element))
 Push(filter.Out, element);
 else
 Pull(filter.In);
 });

 SetHandler(filter.Out, onPull: ()=> Pull(filter.In));
 }
 }

 private readonly Predicate<T> _predicate;

 public Filter(Predicate<T> predicate)
 {
 _predicate = predicate;
 Shape = new FlowShape<T, T>(In, Out);
 }

 public Inlet<T> In { get; } = new Inlet<T>("Filter.in");

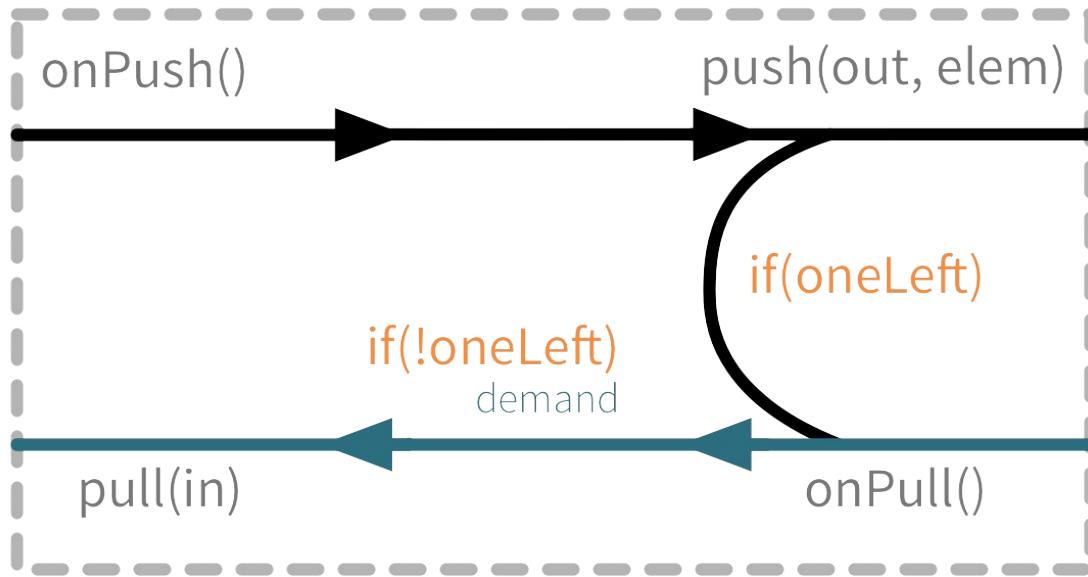
 public Outlet<T> Out { get; } = new Outlet<T>("Filter.out");

 public override FlowShape<T, T> Shape { get; }

 protected override GraphStageLogic CreateLogic(Attributes inheritedAttributes) => new Logic(this);
}
```

To complete the picture we define a one-to-many transformation as the next step. We chose a straightforward example stage that emits every upstream element twice downstream. The conceptual wiring of this stage looks like this:

# Duplicate



This is a stage that has state: an option with the last element it has seen indicating if it has duplicated this last element already or not. We must also make sure to emit the extra element if the upstream completes.

```

class Duplicator<T> : GraphStage<FlowShape<T, T>>
{
 private sealed class Logic : GraphStageLogic
 {
 Option<T> _lastElement = Option<T>.None;

 public Logic(Duplicator<T> duplicator) : base(duplicator.Shape)
 {
 SetHandler(duplicator.In,
 onPush: () =>
 {
 var element = Grab(duplicator.In);
 _lastElement = element;
 Push(duplicator.Out, element);
 },
 onUpstreamFinish: () =>
 {
 if (_lastElement.HasValue)
 Emit(duplicator.Out, _lastElement.Value);

 Complete(duplicator.Out);
 });
 SetHandler(duplicator.Out, onPull: () =>
 {
 if (_lastElement.HasValue)
 {
 Push(duplicator.Out, _lastElement.Value);
 _lastElement = Option<T>.None;
 }
 else
 Pull(duplicator.In);
 });
 }
 }
}

```

```

public Duplicator(Predicate<T> predicate)
{
 Shape = new FlowShape<T, T>(In, Out);
}

public Inlet<T> In { get; } = new Inlet<T>("Duplicator.in");

public Outlet<T> Out { get; } = new Outlet<T>("Duplicator.out");

public override FlowShape<T, T> Shape { get; }

protected override GraphStageLogic CreateLogic(Attributes inheritedAttributes) => new Logic(this);
}

```

In this case a pull from downstream might be consumed by the stage itself rather than passed along upstream as the stage might contain an element it wants to push. Note that we also need to handle the case where the upstream closes while the stage still has elements it wants to push downstream. This is done by overriding `onUpstreamFinish` in the `InHandler` and provide custom logic that should happen when the upstream has been finished.

This example can be simplified by replacing the usage of a mutable state with calls to `EmitMultiple` which will replace the handlers, emit each of multiple elements and then reinstate the original handlers:

```

class Duplicator<T> : GraphStage<FlowShape<T, T>>
{
 private sealed class Logic : GraphStageLogic
 {
 public Logic(Duplicator<T> duplicator) : base(duplicator.Shape)
 {

 SetHandler(duplicator.In, onPush: () =>
 {
 var element = Grab(duplicator.In);
 // this will temporarily suspend this handler until the two elems
 // are emitted and then reinstates it
 EmitMultiple(duplicator.Out, new[] {element, element});
 });

 SetHandler(duplicator.Out, onPull: () => Pull(duplicator.In));
 }
 }

 public Duplicator(Predicate<T> predicate)
 {
 Shape = new FlowShape<T, T>(In, Out);
 }

 public Inlet<T> In { get; } = new Inlet<T>("Duplicator.in");

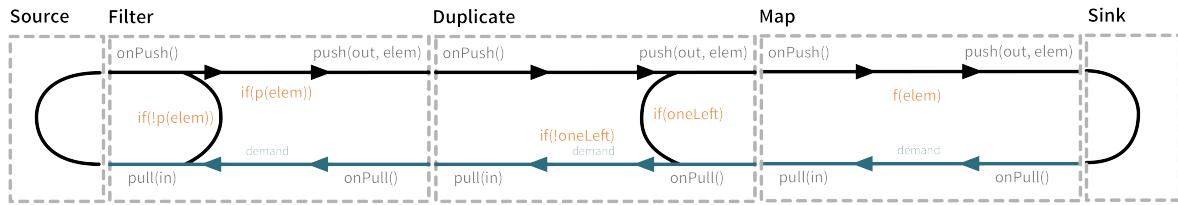
 public Outlet<T> Out { get; } = new Outlet<T>("Duplicator.out");

 public override FlowShape<T, T> Shape { get; }

 protected override GraphStageLogic CreateLogic(Attributes inheritedAttributes) => new Logic(this);
}

```

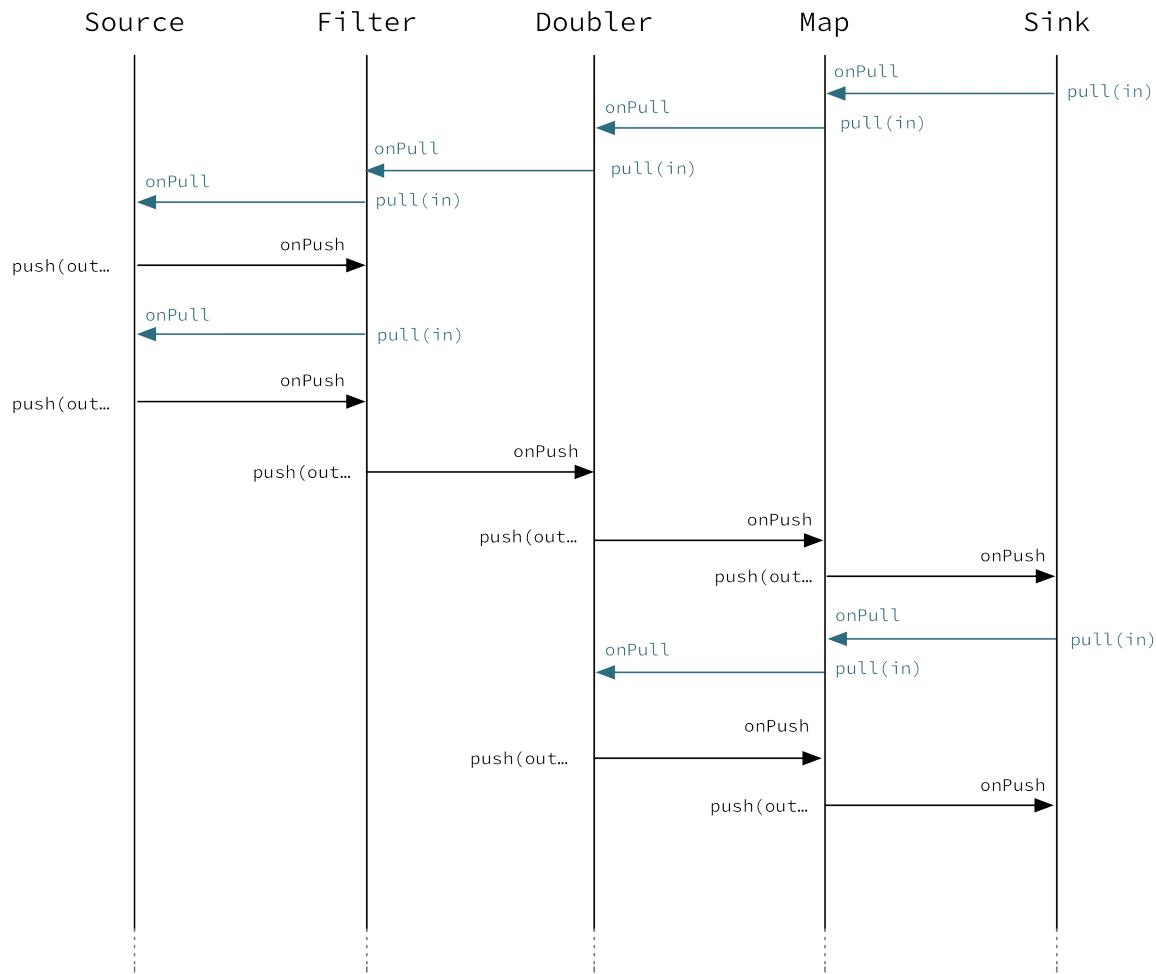
Finally, to demonstrate all of the stages above, we put them together into a processing chain, which conceptually would correspond to the following structure:



In code this is only a few lines, using the `via` use our custom stages in a stream:

```
var resultTask = Source.From(new [] {1,2,3,4,5})
 .Via(new Filter<int>(n => n%2 ==0))
 .Via(new Duplicator<int>())
 .Via(new Map<int, int>(n=>n/2))
 .RunAggregate(0, (sum, next) => sum + next, materializer);
```

If we attempt to draw the sequence of events, it shows that there is one "event token" in circulation in a potential chain of stages, just like our conceptual "railroad tracks" representation predicts.



## Completion

Completion handling usually (but not exclusively) comes into the picture when processing stages need to emit a few more elements after their upstream source has been completed. We have seen an example of this in our first `Duplicator` implementation where the last element needs to be doubled even after the upstream neighbor stage has

been completed. This can be done by overriding the `onUpstreamFinish` callback in `SetHandler(in, action)`.

Stages by default automatically stop once all of their ports (input and output) have been closed externally or internally. It is possible to opt out from this behavior by invoking `SetKeepGoing(true)` (which is not supported from the stage's constructor and usually done in `Prestart`). In this case the stage **must** be explicitly closed by calling `CompleteStage()` or `FailStage(exception)`. This feature carries the risk of leaking streams and actors, therefore it should be used with care.

## Logging inside GraphStages

Logging debug or other important information in your stages is often a very good idea, especially when developing more advanced stages which may need to be debugged at some point.

The `Log` property is provided to enable you to easily obtain a `LoggingAdapter` inside of a `GraphStage` as long as the `Materializer` you're using is able to provide you with a logger. In that sense, it serves a very similar purpose as `ActorLogging` does for Actors.

[!NOTE] Please note that you can always simply use a logging library directly inside a Stage. Make sure to use an asynchronous appender however, to not accidentally block the stage when writing to files etc.

The stage gets access to the `Log` property which it can safely use from any `GraphStage` callbacks:

```
private sealed class RandomLettersSource : GraphStage<SourceShape<string>>
{
 #region internal classes

 private sealed class Logic : GraphStageLogic
 {
 public Logic(RandomLettersSource stage) : base(stage.Shape)
 {
 SetHandler(stage.Out, onPull: () =>
 {
 var c = NextChar(); // ASCII lower case letters

 Log.Debug($"Randomly generated: {c}");

 Push(stage.Out, c.ToString());
 });
 }

 private static char NextChar() => (char) ThreadLocalRandom.Current.Next('a', 'z'1);
 }

 #endregion

 public RandomLettersSource()
 {
 Shape = new SourceShape<string>(Out);
 }

 private Outlet<string> Out { get; } = new Outlet<string>("RandomLettersSource.out");

 public override SourceShape<string> Shape { get; }

 protected override GraphStageLogic CreateLogic(Attributes inheritedAttributes) => new Logic(this);
}

[Fact]
public void A_GraphStageLogic_must_support_logging_in_custom_graphstage()
{
 const int n = 10;
 EventFilter.Debug(start: "Randomly generated").Expect(n, () =>
 {
}
```

```

 Source.FromGraph(new RandomLettersSource())
 .Take(n)
 .RunWith(Sink.Ignore<string>(), Materializer)
 .Wait(TimeSpan.FromSeconds(3));
 });
}

```

[!NOTE] **SPI Note:** If you're implementing a Materializer, you can add this ability to your materializer by implementing `IMaterializerLoggingProvider` in your `Materializer`.

## Using timers

It is possible to use timers in `GraphStages` by using `TimerGraphStageLogic` as the base class for the returned logic. Timers can be scheduled by calling one of `ScheduleOnce(key, delay)`, `SchedulePeriodically(key, period)` or `SchedulePeriodicallyWithInitialDelay(key, delay, period)` and passing an object as a key for that timer (can be any object, for example a String). The `onTimer(key)` method needs to be overridden and it will be called once the timer of key fires. It is possible to cancel a timer using `CancelTimer(key)` and check the status of a timer with `IsTimerActive(key)`. Timers will be automatically cleaned up when the stage completes.

Timers can not be scheduled from the constructor of the logic, but it is possible to schedule them from the `PreStart()` lifecycle hook.

In this sample the stage toggles between open and closed, where open means no elements are passed through. The stage starts out as closed but as soon as an element is pushed downstream the gate becomes open for a duration of time during which it will consume and drop upstream messages:

```

class TimedGate<T> : GraphStage<FlowShape<T, T>>
{
 private readonly TimeSpan _silencePeriod;

 private sealed class Logic : TimerGraphStageLogic
 {
 private bool _open;

 public Logic(TimedGate<T> gate) : base(gate.Shape)
 {
 SetHandler(gate.In, onPush: () =>
 {
 var element = Grab(gate.In);
 if (_open)
 Pull(gate.In);
 else
 {
 Push(gate.Out, element);
 _open = true;
 ScheduleOnce("Close", gate._silencePeriod);
 }
 });
 SetHandler(gate.Out, onPull: () => Pull(gate.In));
 }

 protected internal override void OnTimer(object timerKey) => _open = false;
 }

 public TimedGate(TimeSpan silencePeriod)
 {
 _silencePeriod = silencePeriod;
 Shape = new FlowShape<T, T>(In, Out);
 }

 public Inlet<T> In { get; } = new Inlet<T>("TimedGate.in");
}

```

```

 public Outlet<T> Out { get; } = new Outlet<T>("TimedGate.out");

 public override FlowShape<T, T> Shape { get; }

 protected override GraphStageLogic CreateLogic(Attributes inheritedAttributes) => new Logic(this);
}

```

## Using asynchronous side-channels

In order to receive asynchronous events that are not arriving as stream elements (for example a completion of a task or a callback from a 3rd party API) one must acquire a `AsyncCallback` by calling `GetAsyncCallback()` from the stage logic. The method `GetAsyncCallback` takes as a parameter a callback that will be called once the asynchronous event fires. It is important to **not call the callback directly**, instead, the external API must `invoke` the returned `Action`. The execution engine will take care of calling the provided callback in a thread-safe way. The callback can safely access the state of the `GraphStageLogic` implementation.

Sharing the `AsyncCallback` from the constructor risks race conditions, therefore it is recommended to use the `PreStart()` lifecycle hook instead.

This example shows an asynchronous side channel graph stage that starts dropping elements when a future completes:

```

class KillSwitch<T> : GraphStage<FlowShape<T, T>>
{
 private sealed class Logic : GraphStageLogic
 {
 private readonly KillSwitch<T> _killSwitch;

 public Logic(KillSwitch<T> killSwitch) : base(killSwitch.Shape)
 {
 _killSwitch = killSwitch;

 SetHandler(killSwitch.In, onPush: () => Push(killSwitch.Out, Grab(killSwitch.In)));
 SetHandler(killSwitch.Out, onPull: () => Pull(killSwitch.In));
 }

 public override void PreStart()
 {
 var callback = GetAsyncCallback(CompleteStage);
 _killSwitch._killSwitch.ContinueWith(_ => callback());
 }
 }

 private readonly Task _killSwitch;

 public KillSwitch(Task killSwitch)
 {
 _killSwitch = killSwitch;
 Shape = new FlowShape<T, T>(In, Out);
 }

 public Inlet<T> In { get; } = new Inlet<T>("KillSwitch.in");

 public Outlet<T> Out { get; } = new Outlet<T>("KillSwitch.out");

 public override FlowShape<T, T> Shape { get; }

 protected override GraphStageLogic CreateLogic(Attributes inheritedAttributes) => new Logic(this);
}

```

## Integration with actors

**This section is a stub and will be extended in the next release This is an experimental feature\***

It is possible to acquire an `ActorRef` that can be addressed from the outside of the stage, similarly how `AsyncCallback` allows injecting asynchronous events into a stage logic. This reference can be obtained by calling

`GetStageActorRef(receive)` passing in a function that takes a `Tuple` of the sender `IActorRef` and the received message. This reference can be used to watch other actors by calling its `watch(ref)` or `unwatch(ref)` methods. The reference can be also watched by external actors. The current limitations of this `IActorRef` are:

- they are not location transparent, they cannot be accessed via remoting.
- they cannot be returned as materialized values.
- they cannot be accessed from the constructor of the `GraphStageLogic`, but they can be accessed from the `PreStart()` method.

## Custom materialized values

Custom stages can return materialized values instead of `NotUsed` by inheriting from `GraphStageWithMaterializedValue` instead of the simpler `GraphStage`. The difference is that in this case the method

`CreateLogicAndMaterializedValue(inheritedAttributes)` needs to be overridden, and in addition to the stage logic the materialized value must be provided

[!WARNING] There is no built-in synchronization of accessing this value from both of the thread where the logic runs and the thread that got hold of the materialized value. It is the responsibility of the programmer to add the necessary (non-blocking) synchronization and visibility guarantees to this shared object.

In this sample the materialized value is a task containing the first element to go through the stream:

```
class FirstValue<T> : GraphStageWithMaterializedValue<FlowShape<T, T>, Task<T>>
{
 private sealed class Logic : GraphStageLogic
 {
 public Logic(FirstValue<T> first, TaskCompletionSource<T> completion) : base(first.Shape)
 {
 SetHandler(first.In, onPush: () =>
 {
 var element = Grab(first.In);
 completion.SetResult(element);
 Push(first.Out, element);

 // replace handler with one just forwarding
 SetHandler(first.In, onPush: () => Push(first.Out, Grab(first.In)));
 });

 SetHandler(first.Out, onPull: () => Pull(first.In));
 }
 }

 public FirstValue()
 {
 Shape = new FlowShape<T, T>(In, Out);
 }

 public Inlet<T> In { get; } = new Inlet<T>("FirstValue.in");

 public Outlet<T> Out { get; } = new Outlet<T>("FirstValue.out");

 public override FlowShape<T, T> Shape { get; }

 public override ILogicAndMaterializedValue<Task<T>> CreateLogicAndMaterializedValue(Attributes inheritedAttributes)
 {
 var completion = new TaskCompletionSource<T>();
 var logic = new Logic(this, completion);
 return logic;
 }
}
```

```
 return new LogicAndMaterializedValue<Task<T>>(logic, completion.Task);
 }
}
```

## Using attributes to affect the behavior of a stage

This section is a stub and will be extended in the next release

Stages can access the `Attributes` object created by the materializer. This contains all the applied (inherited) attributes applying to the stage, ordered from least specific (outermost) towards the most specific (innermost) attribute. It is the responsibility of the stage to decide how to reconcile this inheritance chain to a final effective decision.

See [Modularity, Composition and Hierarchy](#) for an explanation on how attributes work.

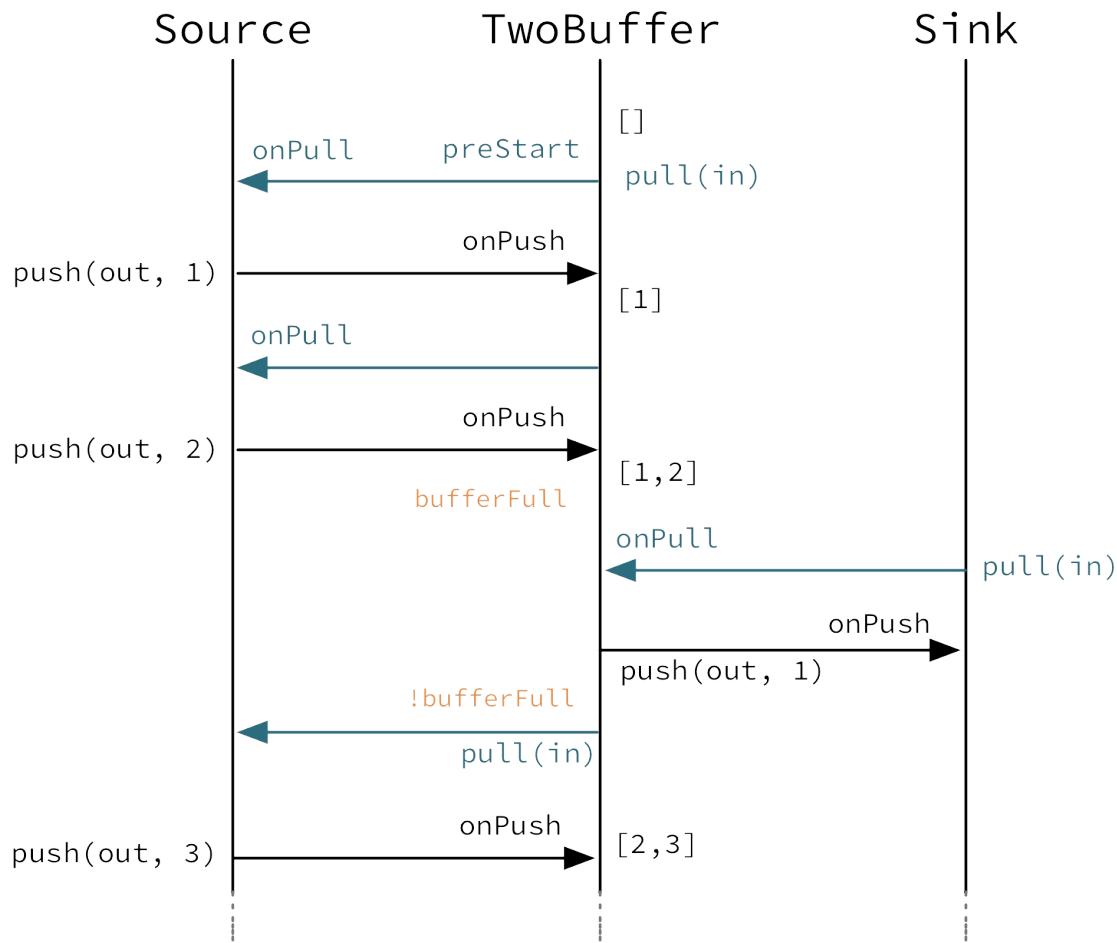
## Rate decoupled graph stages

Sometimes it is desirable to decouple the rate of the upstream and downstream of a stage, synchronizing only when needed.

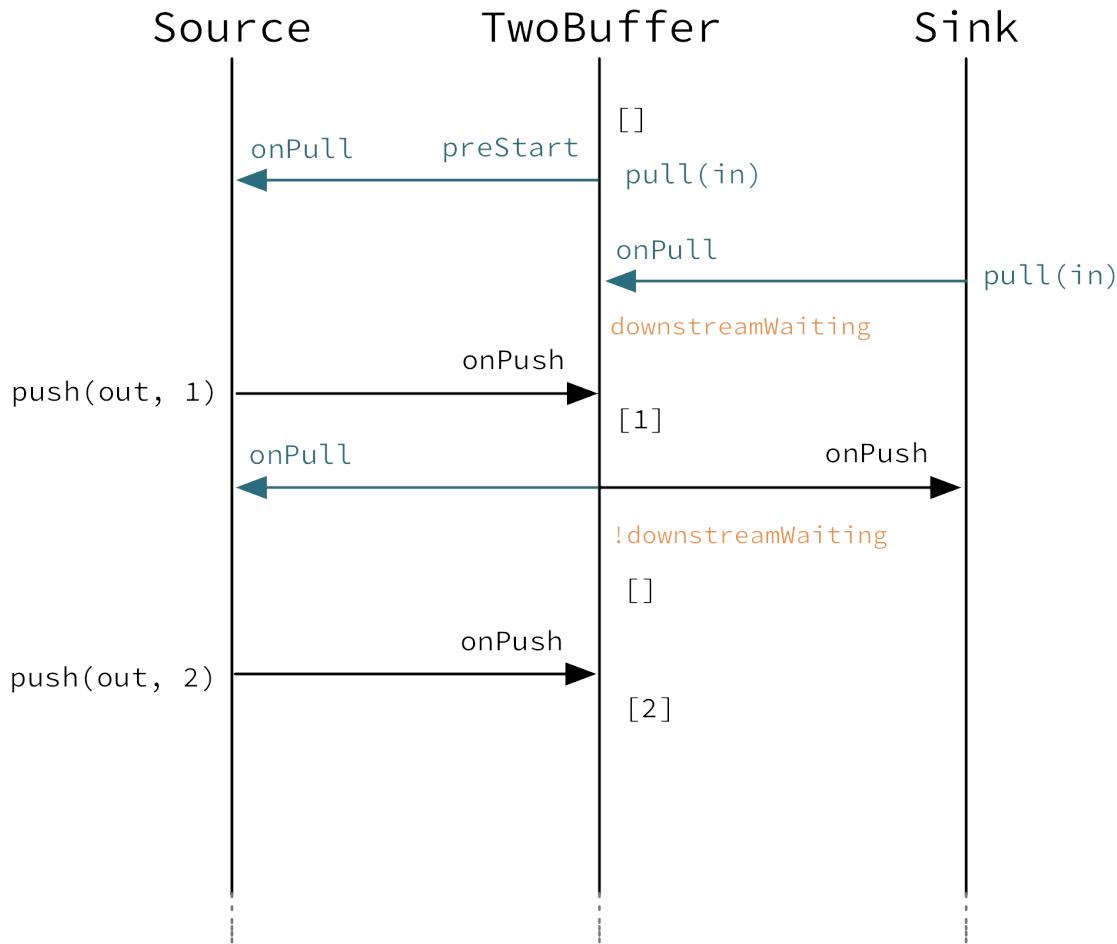
This is achieved in the model by representing a `GraphStage` as a *boundary* between two regions where the demand sent upstream is decoupled from the demand that arrives from downstream. One immediate consequence of this difference is that an `onPush` call does not always lead to calling `Push` and an `onPull` call does not always lead to calling `Pull`.

One of the important use-case for this is to build buffer-like entities, that allow independent progress of upstream and downstream stages when the buffer is not full or empty, and slowing down the appropriate side if the buffer becomes empty or full.

The next diagram illustrates the event sequence for a buffer with capacity of two elements in a setting where the downstream demand is slow to start and the buffer will fill up with upstream elements before any demand is seen from downstream.



Another scenario would be where the demand from downstream starts coming in before any element is pushed into the buffer stage.



The first difference we can notice is that our `TwoBuffer` stage is automatically pulling its upstream on initialization. The buffer has demand for up to two elements without any downstream demand.

The following code example demonstrates a buffer class corresponding to the message sequence chart above.

```

class TwoBuffer<T> : GraphStage<FlowShape<T, T>>
{
 private sealed class Logic : GraphStageLogic
 {
 private readonly TwoBuffer<T> _buffer;
 private readonly Queue<T> _queue;
 private bool _downstreamWaiting = false;

 public Logic(TwoBuffer<T> buffer) : base(buffer.Shape)
 {
 _buffer = buffer;
 _queue = new Queue<T>();

 SetHandler(buffer.In, OnPush, OnUpstreamFinish);
 SetHandler(buffer.Out, OnPull);
 }

 private bool BufferFull => _queue.Count == 2;

 private void OnPush()
 {
 var element = Grab(_buffer.In);
 _queue.Enqueue(element);
 if (_downstreamWaiting)

```

```

 {
 _downstreamWaiting = false;
 var bufferedElement = _queue.Dequeue();
 Push(_buffer.Out, bufferedElement);
 }
 if(!BufferFull)
 Pull(_buffer.In);
 }

 private void OnUpstreamFinish()
 {
 if (_queue.Count != 0)
 {
 // emit the rest if possible
 EmitMultiple(_buffer.Out, _queue);
 }

 CompleteStage();
 }

 private void OnPull()
 {
 if (_queue.Count == 0)
 _downstreamWaiting = true;
 else
 {
 var element = _queue.Dequeue();
 Push(_buffer.Out, element);
 }

 if(!BufferFull && !HasBeenPulled(_buffer.In))
 Pull(_buffer.In);
 }
}

public TwoBuffer()
{
 Shape = new FlowShape<T, T>(In, Out);
}

public Inlet<T> In { get; } = new Inlet<T>("TwoBuffer.in");

public Outlet<T> Out { get; } = new Outlet<T>("TwoBuffer.out");

public override FlowShape<T, T> Shape { get; }

protected override GraphStageLogic CreateLogic(Attributes inheritedAttributes) => new Logic(this);
}

```

## Thread safety of custom processing stages

All of the above custom stages (linear or graph) provide a few simple guarantees that implementors can rely on.

- The callbacks exposed by all of these classes are never called concurrently.
- The state encapsulated by these classes can be safely modified from the provided callbacks, without any further synchronization.

In essence, the above guarantees are similar to what `Actor`'s `provide`, if one thinks of the state of a custom stage as state of an actor, and the callbacks as the `receive` block of the actor.

[!WARNING] It is **not** safe to access the state of any custom stage outside of the callbacks that it provides, just like it is unsafe to access the state of an actor from the outside. This means that Future callbacks should not close over internal state of custom stages because such access can be concurrent with the provided callbacks,

leading to undefined behavior.

## Resources and the stage lifecycle

If a stage manages a resource with a lifecycle, for example objects that need to be shutdown when they are not used anymore it is important to make sure this will happen in all circumstances when the stage shuts down.

Cleaning up resources should be done in `GraphStageLogic.PostStop` and not in the `InHandler` and `outHandler` callbacks. The reason for this is that when the stage itself completes or is failed there is no signal from the upstreams for the downstreams. Even for stages that do not complete or fail in this manner, this can happen when the `Materializer` is shutdown or the `ActorSystem` is terminated while a stream is still running, what is called an "abrupt termination".

## Extending Flow Combinators with Custom Operators

The most general way of extending any `Source`, `Flow` or `SubFlow` (e.g. from `GroupBy`) is demonstrated above: create a graph of flow-shape like the `Filter` example given above and use the `.Via(...)` combinator to integrate it into your stream topology. This works with all `IFlow` sub-types, including the ports that you connect with the graph DSL.

Advanced .Net users may wonder whether it is possible to write extension methods that enrich `IFlow` to allow nicer syntax. The short answer is that .Net does not support this in a fully generic fashion, the problem is that it is impossible to abstract over the kind of stream that is being extended because `Source`, `Flow` and `SubFlow` differ in the number and kind of their type parameters.

A lot simpler is the task of just adding an extension method to `Source` and `Flow` as shown below:

```
public static class Extensions
{
 public static Flow<TIn, TOut, TMat> Filter<TIn, TOut, TMat>(this Flow<TIn, TOut, TMat> flow,
 Predicate<TOut> predicate) => flow.Via(new Filter<TOut>(predicate));

 public static Source<TOut, TMat> Filter<TOut, TMat>(this Source<TOut, TMat> source,
 Predicate<TOut> predicate) => source.Via(new Filter<TOut>(predicate));
}
```

you can then replace the `.via` call from above with the extension method:

```
var resultTask = Source.From(new [] {1,2,3,4,5})
 .Filter(n => n % 2 == 0)
 .Via(new Duplicator<int>())
 .Via(new Map<int, int>(n=>n/2))
 .RunAggregate(0, (sum, next) => sum + next, materializer);
```

If you try to write this for `SubFlow`, though, you will run into the same issue as when trying to unify the two solutions above, only on a higher level (the type constructors needed for that unification would have rank two, meaning that some of their type arguments are type constructors themselves when trying to extend the solution shown in the linked sketch the author encountered such a density of compiler StackOverflowErrors and IDE failures that he gave up).

# Integration

## Integrating with Actors

For piping the elements of a stream as messages to an ordinary actor you can use `Ask` in a `SelectAsync` or use `Sink.ActorRefWithAck`.

Messages can be sent to a stream with `Source.Queue` or via the `IActorRef` that is materialized by `Source.ActorRef`.

### SelectAsync + Ask

A nice way to delegate some processing of elements in a stream to an actor is to use `Ask` in `SelectAsync`. The back-pressure of the stream is maintained by the `Task` of the `Ask` and the mailbox of the actor will not be filled with more messages than the given `parallelism` of the `SelectAsync` stage.

```
var words = Source.From(new [] { "hello", "hi" });
words
 .SelectAsync(5, elem => _actorRef.Ask(elem, TimeSpan.FromSeconds(5)))
 .Select(elem => (string)elem)
 .Select(elem => elem.ToLower())
 .RunWith(Sink.Ignore<string>(), _actorMaterializer);
```

Note that the messages received in the actor will be in the same order as the stream elements, i.e. the `parallelism` does not change the ordering of the messages. There is a performance advantage of using `parallelism > 1` even though the actor will only process one message at a time because then there is already a message in the mailbox when the actor has completed previous message.

The actor must reply to the `Sender` for each message from the stream. That reply will complete the `completionStage` of the `Ask` and it will be the element that is emitted downstreams from `SelectAsync`.

```
public class Translator : ReceiveActor
{
 public Translator()
 {
 Receive<string>(word => {
 // ... process message
 string reply = word.ToUpper();
 // reply to the ask
 Sender.Tell(reply, Self);
 });
 }
}
```

The stream can be completed with failure by sending `Akka.Actor.Status.Failure` as reply from the actor.

If the `Ask` fails due to timeout the stream will be completed with `TimeoutException` failure. If that is not desired outcome you can use `Recover` on the `Ask` `CompletionStage`.

If you don't care about the replies you can use `Sink.Ignore` after the `SelectAsync` stage and then actor is effectively a sink of the stream.

The same pattern can be used with [Actor routers](#). Then you can use `SelectAsyncUnordered` for better efficiency if you don't care about the order of the emitted downstream elements (the replies).

### Sink.ActorRefWithAck

The sink sends the elements of the stream to the given `IActorRef` that sends back back-pressure signal. First element is always `OnInitMessage`, then stream is waiting for the given acknowledgement message from the given actor which means that it is ready to process elements. It also requires the given acknowledgement message after each stream element to make back-pressure work.

If the target actor terminates the stream will be cancelled. When the stream is completed successfully the given `OnCompleteMessage` will be sent to the destination actor. When the stream is completed with failure a `Akka.Actor.Status.Failure` message will be sent to the destination actor.

[!NOTE] Using `Sink.ActorRef` or ordinary `Tell` from a `Select` or `ForEach` stage means that there is no back-pressure signal from the destination actor, i.e. if the actor is not consuming the messages fast enough the mailbox of the actor will grow, unless you use a bounded mailbox with zero `mailbox-push-timeout-time` or use a rate limiting stage in front. It's often better to use `Sink.ActorRefWithAck` or `Ask` in `SelectAsync`, though.

## Source.Queue

`Source.Queue` can be used for emitting elements to a stream from an actor (or from anything running outside the stream). The elements will be buffered until the stream can process them. You can `offer` elements to the queue and they will be emitted to the stream if there is demand from downstream, otherwise they will be buffered until request for demand is received.

Use overflow strategy `Akka.Streams.OverflowStrategy.Backpressure` to avoid dropping of elements if the buffer is full.

`ISourceQueueWithComplete.OfferAsync` returns `Task<IQueueOfferResult>` which completes with `QueueOfferResult.Enqueueed` if element was added to buffer or sent downstream. It completes with `QueueOfferResult.Dropped` if element was dropped. It can also complete with `QueueOfferResult.Failure` when stream failed or `QueueOfferResult.QueueClosed` when downstream is completed.

When used from an actor you typically `pipe` the result of the `Task` back to the actor to continue processing.

## Source.ActorRef

Messages sent to the actor that is materialized by `Source.ActorRef` will be emitted to the stream if there is demand from downstream, otherwise they will be buffered until request for demand is received.

Depending on the defined `overflowStrategy` it might drop elements if there is no space available in the buffer. The strategy `OverflowStrategy.Backpressure` is not supported for this Source type, i.e. elements will be dropped if the buffer is filled by sending at a rate that is faster than the stream can consume. You should consider using `Source.Queue` if you want a backpressured actor interface.

The stream can be completed successfully by sending `Akka.Actor.PoisonPill` or `Akka.Actor.Status.Success` to the actor reference.

The stream can be completed with failure by sending `Akka.Actor.Status.Failure` to the actor reference.

The actor will be stopped when the stream is completed, failed or cancelled from downstream, i.e. you can watch it to get notified when that happens.

# Integrating with External Services

Stream transformations and side effects involving external non-stream based services can be performed with `SelectAsync` or `SelectAsyncUnordered`.

For example, sending emails to the authors of selected tweets using an external email service:

```
Task<int> Send(Email mail)
```

We start with the tweet stream of authors:

```
var authors = tweets
 .Where(t => t.HashTags.Contains("Akka.Net"))
 .Select(t => t.Author);
```

Assume that we can lookup their email address using:

```
Task<string> LookupEmail(string handle)
```

Transforming the stream of authors to a stream of email addresses by using the `LocusEmail` service can be done with `SelectAsync`:

```
var emailAddresses = authors
 .SelectAsync(4, author => AddressSystem.LookupEmail(author.Handle))
 .Collect(s => string.IsNullOrEmpty(s) ? null : s);
```

Finally, sending the emails:

```
var sendEmails = emailAddresses.SelectAsync(4, address =>
 EmailServer.Send(
 new Email(to: address, title: "Akka.Net", body: "I like your tweet"))
)
.To(Sink.Ignore<int>());

sendEmails.Run(materializer);
```

`SelectAsync` is applying the given function that is calling out to the external service to each of the elements as they pass through this processing step. The function returns a `Task` and the value of that task will be emitted downstreams. The number of Tasks that shall run in parallel is given as the first argument to `SelectAsync`. These Tasks may complete in any order, but the elements that are emitted downstream are in the same order as received from upstream.

That means that back-pressure works as expected. For example if the `EmailServer.Send` is the bottleneck it will limit the rate at which incoming tweets are retrieved and email addresses looked up.

The final piece of this pipeline is to generate the demand that pulls the tweet authors information through the emailing pipeline: we attach a `Sink.Ignore` which makes it all run. If our email process would return some interesting data for further transformation then we would of course not ignore it but send that result stream onwards for further processing or storage.

Note that `SelectAsync` preserves the order of the stream elements. In this example the order is not important and then we can use the more efficient `SelectAsyncUnordered`:

```
var authors = tweets
 .Where(t => t.HashTags.Contains("Akka.Net"))
 .Select(t => t.Author);

var emailAddresses = authors
 .SelectAsyncUnordered(4, author => AddressSystem.LookupEmail(author.Handle))
 .Collect(s => string.IsNullOrEmpty(s) ? null : s);

var sendEmails = emailAddresses.SelectAsyncUnordered(4, address =>
 EmailServer.Send(
 new Email(to: address, title: "Akka.Net", body: "I like your tweet"))
```

```

)
 .To(Sink.Ignore<int>()));

sendEmails.Run(materializer);

```

In the above example the services conveniently returned a `Task` of the result. If that is not the case you need to wrap the call in a `Task`.

For a service that is exposed as an actor, or if an actor is used as a gateway in front of an external service, you can use `Ask`:

```

var akkaTweets = tweets.Where(t => t.HashTags.Contains("Akka.NET"));

var saveTweets = akkaTweets
 .SelectAsync(4, tweet => database.Ask<DbResult>(new Save(tweet), TimeSpan.FromSeconds(3)))
 .To(Sink.Ignore<DbResult>());

```

Note that if the `Ask` is not completed within the given timeout the stream is completed with failure. If that is not desired outcome you can use `Recover` on the `Ask Task`.

## Illustrating ordering and parallelism

Let us look at another example to get a better understanding of the ordering and parallelism characteristics of `SelectAsync` and `SelectAsyncUnordered`.

Several `SelectAsync` and `SelectAsyncUnordered` tasks may run concurrently. The number of concurrent tasks are limited by the downstream demand. For example, if 5 elements have been requested by downstream there will be at most 5 tasks in progress.

`SelectAsync` emits the task results in the same order as the input elements were received. That means that completed results are only emitted downstream when earlier results have been completed and emitted. One slow call will thereby delay the results of all successive calls, even though they are completed before the slow call.

`SelectAsyncUnordered` emits the task results as soon as they are completed, i.e. it is possible that the elements are not emitted downstream in the same order as received from upstream. One slow call will thereby not delay the results of faster successive calls as long as there is downstream demand of several elements.

Here is a fictive service that we can use to illustrate these aspects.

```

public class SometimesSlowService
{
 private readonly AtomicCounter runningCount = new AtomicCounter();

 public Task<string> Convert(string s)
 {
 Console.WriteLine($"running {s} {runningCount.IncrementAndGet()}");

 return Task.Run(() =>
 {
 if(s != "" && char.IsLower(s[0]))
 Thread.Sleep(500);
 else
 Thread.Sleep(20);

 Console.WriteLine($"completed {s} {runningCount.GetAndDecrement()}");
 });

 return s.ToUpper();
 }
}

```

Elements starting with a lower case character are simulated to take longer time to process.

Here is how we can use it with `SelectAsync`:

```
var service = new SometimesSlowService();
var settings = ActorMaterializerSettings.Create(sys).WithInputBuffer(4, 4);
var materializer = sys.Materializer(settings);

Source.From(new[] {"a", "B", "C", "D", "e", "F", "g", "H", "i", "J"})
 .Select(x =>
{
 Console.WriteLine($"before {x}");
 return x;
})
 .SelectAsync(4, service.Convert)
 .RunForEach(x => Console.WriteLine($"after: {x}"), materializer);
```

The output may look like this:

```
before: a
before: B
before: C
before: D
running: a (1)
running: B (2)
before: e
running: C (3)
before: F
running: D (4)
before: g
before: H
completed: C (3)
completed: B (2)
completed: D (1)
completed: a (0)
after: A
after: B
running: e (1)
after: C
after: D
running: F (2)
before: i
before: J
running: g (3)
running: H (4)
completed: H (2)
completed: F (3)
completed: e (1)
completed: g (0)
after: E
after: F
running: i (1)
after: G
after: H
running: J (2)
completed: J (1)
completed: i (0)
after: I
after: J
```

Note that `after` lines are in the same order as the `before` lines even though elements are `completed` in a different order. For example `H` is `completed` before `g`, but still emitted afterwards.

The numbers in parenthesis illustrates how many calls that are in progress at the same time. Here the downstream demand and thereby the number of concurrent calls are limited by the buffer size (4) of the `ActorMaterializerSettings`.

Here is how we can use the same service with `SelectAsyncUnordered`:

```
var service = new SometimesSlowService(_output);
var settings = ActorMaterializerSettings.Create(sys).WithInputBuffer(4, 4);
var materializer = sys.Materializer(settings);

var result = Source.From(new[] {"a", "B", "C", "D", "e", "F", "g", "H", "i", "J"})
 .Select(x =>
{
 Console.WriteLine($"before: {x}");
 return x;
})
 .SelectAsync(4, service.Convert)
 .RunForEach(x => Console.WriteLine($"after: {x}"), materializer);
```

The output may look like this:

```
before: a
before: B
before: C
before: D
running: a (1)
running: B (2)
before: e
running: C (3)
before: F
running: D (4)
before: g
before: H
completed: B (3)
completed: C (1)
completed: D (2)
after: B
after: D
running: e (2)
after: C
running: F (3)
before: i
before: J
completed: F (2)
after: F
running: g (3)
running: H (4)
completed: H (3)
after: H
completed: a (2)
after: A
running: i (3)
running: J (4)
completed: J (3)
after: J
completed: e (2)
after: E
completed: g (1)
after: G
completed: i (0)
after: I
```

Note that `after` lines are not in the same order as the `before` lines. For example `H` overtakes the slow `G`.

The numbers in parenthesis illustrates how many calls that are in progress at the same time. Here the downstream demand and thereby the number of concurrent calls are limited by the buffer size (4) of the `ActorMaterializerSettings`.

## Integrating with Reactive Streams

`Reactive Streams` defines a standard for asynchronous stream processing with non-blocking back pressure. It makes it possible to plug together stream libraries that adhere to the standard. Akka Streams is one such library.

- Reactive Streams: <http://reactive-streams.org/>

The two most important interfaces in Reactive Streams are the `IPublisher` and `ISubscriber`.

```
Reactive.Streams.IPublisher
Reactive.Streams.ISubscriber
```

Let us assume that a library provides a publisher of tweets:

```
IPublisher<Tweet> Tweets
```

and another library knows how to store author handles in a database:

```
ISubscriber<Author> Storage
```

Using an Akka Streams Flow we can transform the stream and connect those:

```
var authors = Flow.Create<Tweet>()
 .Where(t => t.HashTags.Contains("Akka.net"))
 .Select(t => t.Author);

Source.FromPublisher(tweets)
 .Via(authors)
 .To(Sink.FromSubscriber(storage))
 .Run(materializer);
```

The `Publisher` is used as an input `Source` to the flow and the `Subscriber` is used as an output `Sink`.

A `Flow` can also be converted to a `RunnableGraph<IProcessor<In, Out>>` which materializes to a `IProcessor` when `Run()` is called. `Run()` itself can be called multiple times, resulting in a new `Processor` instance each time.

```
var processor = authors.ToProcessor().Run(materializer);
tweets.Subscribe(processor);
processor.Subscribe(storage);
```

A publisher can be connected to a subscriber with the `Subscribe` method.

It is also possible to expose a `Source` as a `Publisher` by using the Publisher-`Sink`:

```
var authorPublisher = Source.FromPublisher(tweets)
 .Via(authors)
 .RunWith(Sink.AsPublisher<Author>(fanout: false), materializer);

authorPublisher.Subscribe(storage);
```

A publisher that is created with `Sink.AsPublisher(fanout = false)` supports only a single subscription. Additional subscription attempts will be rejected with an `IllegalStateException`.

A publisher that supports multiple subscribers using fan-out/broadcasting is created as follows:

```
ISubscriber<Author> Storage
ISubscriber<Author> Alert

var authorPublisher = Source.FromPublisher(tweets)
 .Via(authors)
 .RunWith(Sink.AsPublisher<Author>(fanout: true), materializer);

authorPublisher.Subscribe(storage);
authorPublisher.Subscribe(alert);
```

The input buffer size of the stage controls how far apart the slowest subscriber can be from the fastest subscriber before slowing down the stream.

To make the picture complete, it is also possible to expose a `Sink` as a `Subscriber` by using the `Subscriber- source` :

```
var tweetSubscriber = authors.To(Sink.FromSubscriber(storage))
 .RunWith(Source.AsSubscriber<Tweet>(), materializer);

tweets.Subscribe(tweetSubscriber);
```

It is also possible to use re-wrap `Processor` instances as a `Flow` by passing a factory function that will create the `Processor` instances:

```
Func<IMaterializer, IProcessor<int, int>> createProcessor =
 mat => Flow.Create<int>().ToProcessor().Run(mat);

var flow = Flow.FromProcessor(()=> createProcessor(materializer));
```

Please note that a factory is necessary to achieve reusability of the resulting `Flow`.

## Implementing Reactive Streams Publisher or Subscriber

As described above any Akka Streams `Source` can be exposed as a Reactive Streams `Publisher` and any `Sink` can be exposed as a Reactive Streams `Subscriber`. Therefore we recommend that you implement Reactive Streams integrations with built-in stages or `custom stages`.

For historical reasons the `ActorPublisher` and `ActorSubscriber` are provided to support implementing Reactive Streams `Publisher` class and `Subscriber` class with an `Actor` class.

These can be consumed by other Reactive Stream libraries or used as an Akka Streams `Source` class or `Sink` class.

[!WARNING] `ActorPublisher` class and `ActorSubscriber` class will probably be deprecated in future versions of Akka.

[!WARNING] `ActorPublisher` class and `ActorSubscriber` class cannot be used with remote actors, because if signals of the Reactive Streams protocol (e.g. `Request`) are lost the the stream may deadlock.

## ActorPublisher

Extend `Akka.Streams.Actor.ActorPublisher` to implement a stream publisher that keeps track of the subscription life cycle and requested elements.

Here is an example of such an actor. It dispatches incoming jobs to the attached subscriber:

```

public sealed class Job
{
 public Job(string payload)
 {
 Payload = payload;
 }

 public string Payload { get; }
}

public sealed class JobAccepted
{
 public static JobAccepted Instance { get; } = new JobAccepted();

 private JobAccepted() { }
}

public sealed class JobDenied
{
 public static JobDenied Instance { get; } = new JobDenied();

 private JobDenied() { }
}

public class JobManager : Actors.ActorPublisher<Job>
{
 public static Props Props { get; } = Props.Create<JobManager>();

 private List<Job> _buffer;
 private const int MaxBufferSize = 100;

 public JobManager()
 {
 _buffer = new List<Job>();
 }

 protected override bool Receive(object message)
 {
 return message.Match()
 .With<Job>(job =>
 {
 if (_buffer.Count == MaxBufferSize)
 Sender.Tell(JobDenied.Instance);
 else
 {
 Sender.Tell(JobAccepted.Instance);
 if (_buffer.Count == 0 && TotalDemand > 0)
 OnNext(job);
 else
 {
 _buffer.Add(job);
 DeliverBuffer();
 }
 }
 })
 .With<Request>(DeliverBuffer)
 .With<Cancel>(() => Context.Stop(Self))
 .WasHandled;
 }

 private void DeliverBuffer()
 {
 if (TotalDemand > 0)
 {
 // totalDemand is a Long and could be larger than
 // what _buffer.Take and Skip can accept
 }
 }
}

```

```

 if (TotalDemand < int.MaxValue)
 {
 var use = _buffer.Take((int) TotalDemand).ToList();
 _buffer = _buffer.Skip((int) TotalDemand).ToList();
 use.ForEach(OnNext);
 }
 else
 {
 var use = _buffer.Take(int.MaxValue).ToList();
 _buffer = _buffer.Skip(int.MaxValue).ToList();
 use.ForEach(OnNext);
 DeliverBuffer();
 }
 }
}

```

You send elements to the stream by calling `OnNext`. You are allowed to send as many elements as have been requested by the stream subscriber. This amount can be inquired with `TotalDemand`. It is only allowed to use `OnNext` when `IsActive` and `TotalDemand > 0`, otherwise `OnNext` will throw `IllegalStateException`.

When the stream subscriber requests more elements the `ActorPublisherMessage.Request` message is delivered to this actor, and you can act on that event. The `TotalDemand` is updated automatically.

When the stream subscriber cancels the subscription the `ActorPublisherMessage.Cancel` message is delivered to this actor. After that subsequent calls to `OnNext` will be ignored.

You can complete the stream by calling `OnComplete`. After that you are not allowed to call `OnNext`, `OnError` and `OnComplete`.

You can terminate the stream with failure by calling `OnError`. After that you are not allowed to call `OnNext`, `OnError` and `OnComplete`.

If you suspect that this `ActorPublisher` may never get subscribed to, you can set the `SubscriptionTimeout` property to provide a timeout after which this Publisher should be considered canceled. The actor will be notified when the timeout triggers via an `ActorPublisherMessage.SubscriptionTimeoutExceeded` message and MUST then perform cleanup and stop itself.

If the actor is stopped the stream will be completed, unless it was not already terminated with failure, completed or canceled.

More detailed information can be found in the API documentation.

This is how it can be used as input `Source` to a `Flow`:

```

var jobManagerSource = Source.ActorPublisher<Job>(JobManager.Props);
var actorRef = Flow.Create<Job>()
 .Select(job => job.Payload.ToUpper())
 .Select(elem =>
 {
 Console.WriteLine(elem);
 return elem;
 })
 .To(Sink.Ignore<string>())
 .RunWith(jobManagerSource, materializer);

actorRef.Tell(new Job("a"));
actorRef.Tell(new Job("b"));
actorRef.Tell(new Job("c"));

```

You can only attach one subscriber to this publisher. Use a `Broadcast`-element or attach a `Sink.AsPublisher(true)` to enable multiple subscribers.

## ActorSubscriber

Extend `Akka.Streams.Actor.ActorSubscriber` to make your class a stream subscriber with full control of stream back pressure. It will receive `OnNext`, `OnComplete` and `OnError` messages from the stream. It can also receive other, non-stream messages, in the same way as any actor.

Here is an example of such an actor. It dispatches incoming jobs to child worker actors:

```

public class Message
{
 public int Id { get; }

 public IActorRef ReplyTo { get; }

 public Message(int id, IActorRef replyTo)
 {
 Id = id;
 ReplyTo = replyTo;
 }
}

public class Work
{
 public Work(int id)
 {
 Id = id;
 }

 public int Id { get; }
}

public class Reply
{
 public Reply(int id)
 {
 Id = id;
 }

 public int Id { get; }
}

public class Done
{
 public Done(int id)
 {
 Id = id;
 }

 public int Id { get; }
}

public class WorkerPool : Actors.ActorSubscriber
{
 public static Props Props { get; } = Props.Create<WorkerPool>();

 private class Strategy : MaxInFlightRequestStrategy
 {
 private readonly Dictionary<int, IActorRef> _queue;

 public Strategy(int max, Dictionary<int, IActorRef> queue) : base(max)
 {
 _queue = queue;
 }

 public override int InFlight => _queue.Count;
 }
}

```

```

private const int MaxQueueSize = 10;
private readonly Dictionary<int, IActorRef> _queue;
private readonly Router _router;

public WorkerPool()
{
 _queue = new Dictionary<int, IActorRef>();
 var routees = new Routee[]
 {
 new ActorRefRoutee(Context.ActorOf<Worker>()),
 new ActorRefRoutee(Context.ActorOf<Worker>()),
 new ActorRefRoutee(Context.ActorOf<Worker>())
 };
 _router = new Router(new RoundRobinRoutingLogic(), routees);
 RequestStrategy = new Strategy(MaxQueueSize, _queue);
}

public override IRequestStrategy RequestStrategy { get; }

protected override bool Receive(object message)
{
 return message.Match()
 .With<OnNext>(next =>
 {
 var msg = next.Element as Message;
 if (msg != null)
 {
 _queue.Add(msg.Id, msg.ReplyTo);
 if (_queue.Count > MaxQueueSize)
 throw new IllegalStateException($"Queued too many : {_queue.Count}");
 _router.Route(new Work(msg.Id), Self);
 }
 })
 .With<Reply>(reply =>
 {
 _queue[reply.Id].Tell(new Done(reply.Id));
 _queue.Remove(reply.Id);
 })
 .WasHandled;
}

public class Worker : ReceiveActor
{
 public Worker()
 {
 Receive<Work>(work =>
 {
 //...
 Sender.Tell(new Reply(work.Id));
 });
 }
}

```

Subclass must define the `RequestStrategy` to control stream back pressure. After each incoming message the `ActorSubscriber` will automatically invoke the `IRequestStrategy.RequestDemand` and propagate the returned demand to the stream.

- The provided `WatermarkRequestStrategy` is a good strategy if the actor performs work itself.
- The provided `MaxInFlightRequestStrategy` is useful if messages are queued internally or delegated to other actors.
- You can also implement a custom `IRequestStrategy` or call `Request` manually together with `ZeroRequestStrategy` or some other strategy. In that case you must also call `Request` when the actor is started or when it is ready, otherwise it will not receive any elements.

More detailed information can be found in the API documentation.

This is how it can be used as output `sink` to a `Flow`:

```
var n = 118;
Source.From(Enumerable.Range(1, n))
 .Select(x => new Message(x, replyTo))
 .RunWith(Sink.ActorSubscriber<Message>(WorkerPool.Props), materializer);
```

# Error Handling in Streams

When a stage in a stream fails this will normally lead to the entire stream being torn down. Each of the stages downstream gets informed about the failure and each upstream stage sees a cancellation.

In many cases you may want to avoid complete stream failure, this can be done in a few different ways:

- `Recover` to emit a final element then complete the stream normally on upstream failure
- `RecoverWithRetries` to create a new upstream and start consuming from that on failure
- Restarting sections of the stream after a backoff
- Using a supervision strategy for stages that support it

In addition to these built in tools for error handling, a common pattern is to wrap the stream inside an actor, and have the actor restart the entire stream on failure.

## Recover

`Recover` allows you to emit a final element and then complete the stream on an upstream failure. Deciding which exceptions should be recovered is done through a `delegate`. If an exception does not have a matching case the stream is failed.

Recovering can be useful if you want to gracefully complete a stream on failure while letting downstream know that there was a failure.

```
Source.From(Enumerable.Range(0, 6)).Select(n =>
{
 if (n < 5)
 return n.ToString();

 throw new ArithmeticException("Boom!");
})
.Recover(exception =>
{
 if (exception is ArithmeticException)
 return new Option<string>("stream truncated");
 return Option<string>.None;
})
.RunForEach(Console.WriteLine, materializer);
```

This will output:

```
0
1
2
3
4
stream truncated
```

## Recover with retries

`RecoverWithRetries` allows you to put a new upstream in place of the failed one, recovering stream failures up to a specified maximum number of times.

Deciding which exceptions should be recovered is done through a `delegate`. If an exception does not have a matching case the stream is failed.

```
var planB = Source.From(new List<string> {"five", "six", "seven", "eight"});

Source.From(Enumerable.Range(0, 10)).Select(n =>
{
 if (n < 5)
 return n.ToString();

 throw new ArithmeticException("Boom!");
})
.RecoverWithRetries(attempts: 1, partialFunc: exception =>
{
 if (exception is ArithmeticException)
 return planB;
 return null;
})
.RunForeach(Console.WriteLine, materializer);
```

This will output:

```
0
1
2
3
4
five
six
seven
eight
```

## Delayed restarts with a backoff stage

Just as Akka provides the [backoff supervision pattern for actors](#), Akka streams also provides a `RestartSource`, `RestartSink` and `RestartFlow` for implementing the so-called *exponential backoff supervision strategy*, starting a stage again when it fails, each time with a growing time delay between restarts.

This pattern is useful when the stage fails or completes 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 WebSocket connection fails due to the HTTP server it's running on going down, perhaps because it is overloaded. By using an exponential backoff, we avoid going into a tight reconnect loop, which both gives the HTTP server some time to recover, and it avoids using needless resources on the client side.

The following snippet shows how to create a backoff supervisor using `Akka.Streams.Dsl.RestartSource` which will supervise the given `source`. The `source` in this case is a `HttpResponseMessage`, produced by `HttpClient`. If the stream fails or completes at any point, the request will be made again, in increasing intervals of 3, 6, 12, 24 and finally 30 seconds (at which point it will remain capped due to the `maxBackoff` parameter):

[!code-csharpRestartDocTests.cs]

```
var httpClient = new HttpClient();

var restartSource = RestartSource.WithBackoff(() =>
{
 // Create a source from a task
 return Source.FromTask(
 httpClient.GetAsync("http://example.com/eventstream") // Make a single request
)
})
```

```

 .Select(c => c.Content.ReadAsStringAsync())
 .Select(c => c.Result);
 },
 minBackoff: TimeSpan.FromSeconds(3),
 maxBackoff: TimeSpan.FromSeconds(30),
 randomFactor: 0.2 // adds 20% "noise" to vary the intervals slightly
);

```

Using a `randomFactor` to add a little bit of additional variance to the backoff intervals is highly recommended, in order to avoid multiple streams re-start at the exact same point in time, for example because they were stopped due to a shared resource such as the same server going down and re-starting after the same configured interval. By adding additional randomness to the re-start intervals the streams will start in slightly different points in time, thus avoiding large spikes of traffic hitting the recovering server or other resource that they all need to contact.

The above `RestartSource` will never terminate unless the `Sink` it's fed into cancels. It will often be handy to use it in combination with a `Killswitch`, so that you can terminate it when needed:

[!code-csharpRestartDocTests.cs]

```

var killSwitch = restartSource
 .ViaMaterialized(KillSwitches.Single<string>(), Keep.Right)
 .ToMaterialized(Sink.ForEach<string>(evt => Console.WriteLine($"Got event: {evt}")), Keep.Left)
 .Run(Materializer);

DoSomethingElse();

killSwitch.Shutdown();

```

Sinks and flows can also be supervised, using `Akka.Streams.Dsl.RestartSink` and `Akka.Streams.Dsl.RestartFlow`. The `RestartSink` is restarted when it cancels, while the `RestartFlow` is restarted when either the in port cancels, the out port completes, or the out port sends an error.

## Supervision Strategies

[!NOTE] The stages that support supervision strategies are explicitly documented to do so, if there is nothing in the documentation of a stage saying that it adheres to the supervision strategy it means it fails rather than applies supervision..

The error handling strategies are inspired by actor supervision strategies, but the semantics have been adapted to the domain of stream processing. The most important difference is that supervision is not automatically applied to stream stages but instead something that each stage has to implement explicitly.

For many stages it may not even make sense to implement support for supervision strategies, this is especially true for stages connecting to external technologies where for example a failed connection will likely still fail if a new connection is tried immediately.

For stages that do implement supervision, the strategies for how to handle exceptions from processing stream elements can be selected when materializing the stream through use of an attribute.

There are three ways to handle exceptions from application code:

- `Stop` - The stream is completed with failure.
- `Resume` - The element is dropped and the stream continues.
- `Restart` - The element is dropped and the stream continues after restarting the stage. Restarting a stage means that any accumulated state is cleared. This is typically performed by creating a new instance of the stage.

By default the stopping strategy is used for all exceptions, i.e. the stream will be completed with failure when an exception is thrown.

```
var source = Source.From(Enumerable.Range(0, 6)).Select(x => 100/x);
var result = source.RunWith(Sink.Aggregate<int, int>(0, (sum, i) => sum + i), materializer);
// division by zero will fail the stream and the
// result here will be a Task completed with Failure(DivideByZeroException)
```

The default supervision strategy for a stream can be defined on the settings of the materializer.

```
Decider decider = cause => cause is DivideByZeroException
 ? Directive.Resume
 : Directive.Stop;
var settings = ActorMaterializerSettings.Create(system).WithSupervisionStrategy(decider);
var materializer = system.Materializer(settings);

var source = Source.From(Enumerable.Range(0, 6)).Select(x => 100/x);
var result = source.RunWith(Sink.Aggregate<int, int>(0, (sum, i) => sum + i), materializer);
// the element causing division by zero will be dropped
// result here will be a Task completed with Success(228)
```

Here you can see that all `DivideByZeroException` will resume the processing, i.e. the elements that cause the division by zero are effectively dropped.

[!NOTE] Be aware that dropping elements may result in deadlocks in graphs with cycles, as explained in [Graph cycles, liveness and deadlocks](#).

The supervision strategy can also be defined for all operators of a flow.

```
Decider decider = cause => cause is DivideByZeroException
 ? Directive.Resume
 : Directive.Stop;

var flow = Flow.Create<int>()
 .Where(x => 100 / x < 50)
 .Select(x => 100 / (5 - x))
 .WithAttributes(ActorAttributes.CreateSupervisionStrategy(decider));
var source = Source.From(Enumerable.Range(0, 6)).Via(flow);
var result = source.RunWith(Sink.Aggregate<int, int>(0, (sum, i) => sum + i), materializer);
// the elements causing division by zero will be dropped
// result here will be a Future completed with Success(150)
```

`Restart` works in a similar way as `Resume` with the addition that accumulated state, if any, of the failing processing stage will be reset.

```
Decider decider = cause => cause is ArgumentException
 ? Directive.Restart
 : Directive.Stop;

var flow = Flow.Create<int>()
 .Scan(0, (acc, x) =>
{
 if(x < 0)
 throw new ArgumentException("negative not allowed");
 return acc + x;
})
 .WithAttributes(ActorAttributes.CreateSupervisionStrategy(decider));
var source = Source.From(new [] {1,3,-1,5,7}).Via(flow);
var result = source.Limit(1000).RunWith(Sink.Seq<int>(), materializer);
// the negative element cause the scan stage to be restarted,
// i.e. start from 0 again
// result here will be a Task completed with Success(List(0, 1, 4, 0, 5, 12))
```

## Errors from SelectAsync

Stream supervision can also be applied to the tasks of `SelectAsync` and `SelectAsyncUnordered` even if such failures happen in the task rather than inside the stage itself. .

Let's say that we use an external service to lookup email addresses and we would like to discard those that cannot be found.

We start with the tweet stream of authors:

```
var authors = tweets
 .Where(t => t.HashTags.Contains("Akka.Net"))
 .Select(t => t.Author);
```

Assume that we can lookup their email address using:

```
Task<string> LookupEmail(string handle)
```

The `Task` is completed with `Failure` if the email is not found.

Transforming the stream of authors to a stream of email addresses by using the `LookupEmail` service can be done with `SelectAsync` and we use `Deciders.ResumingDecider` to drop unknown email addresses:

```
var emailAddresses = authors.Via(
 Flow.Create<Author>()
 .SelectAsync(4, author => AddressSystem.LookupEmail(author.Handle))
 .WithAttributes(ActorAttributes.CreateSupervisionStrategy(Deciders.ResumingDecider)));
```

If we would not use `Resume` the default stopping strategy would complete the stream with failure on the first `Task` that was completed with `Failure` .

# Working with streaming IO

Akka Streams provides a way of handling File IO and TCP connections with Streams. While the general approach is very similar to the [Actor based TCP handling using Akka IO](#), by using Akka Streams you are freed of having to manually react to back-pressure signals, as the library does it transparently for you.

## Streaming TCP

### Accepting connections: Echo Server

In order to implement a simple EchoServer we bind to a given address, which returns a

`Source<Tcp.IncomingConnection, Task<Tcp.ServerBinding>>`, which will emit an `IncomingConnection` element for each new connection that the Server should handle:

[!code-csharp StreamTcpDocTests.cs]

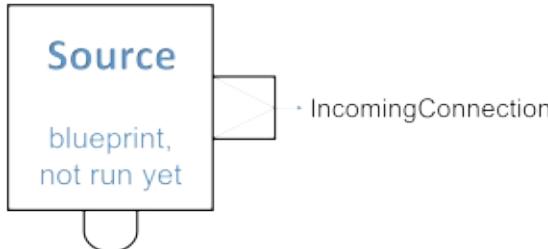
```
// define an incoming request processing logic
Flow<ByteString, ByteString, NotUsed> echo = Flow.Create<ByteString>();

Tcp.ServerBinding binding = await Sys.TcpStream()
 .BindAndHandle(echo, Materializer, "localhost", 9000);

Console.WriteLine($"Server listening at {binding.LocalAddress}");

// close server after everything is done
await binding.Unbind();
```

Sys.TcpStream().Bind("127.0.0.1", 8888)



materializes to `Task<ServerBinding>`

Next, we simply handle each incoming connection using a `Flow` which will be used as the processing stage to handle and emit `ByteString` from and to the TCP Socket. Since one `ByteString` does not have to necessarily correspond to exactly one line of text (the client might be sending the line in chunks) we use the `Framing.Delimiter` helper to chunk the inputs up into actual lines of text. The last boolean argument indicates that we require an explicit line ending even for the last message before the connection is closed. In this example we simply add exclamation marks to each incoming text message and push it through the flow:

[!code-csharp StreamTcpDocTests.cs]

```
Source<Tcp.IncomingConnection, Task<Tcp.ServerBinding>> connections =
 Sys.TcpStream().Bind("127.0.0.1", 8888);

connections.RunForeach(connection =>
{
 Console.WriteLine($"New connection from: {connection.RemoteAddress}");
```

```

var echo = Flow.Create<ByteString>()
 .Via(Framing.Delimiter(
 ByteString.FromString("\n"),
 maximumFrameLength: 256,
 allowTruncation: true))
 .Select(c => c.ToString())
 .Select(c => c + "!!!\n")
 .Select(ByteString.FromString);

connection.HandleWith(echo, Materializer);
}, Materializer);

```

Notice that while most building blocks in Akka Streams are reusable and freely shareable, this is not the case for the incoming connection Flow, since it directly corresponds to an existing, already accepted connection its handling can only ever be materialized once.

Closing connections is possible by cancelling the incoming connection `Flow` from your server logic (e.g. by connecting its downstream to a `Sink.Cancelled` and its upstream to a `Source.Empty`). It is also possible to shut down the server's socket by cancelling the `IncomingConnection source connections``.

[!code-csharpStreamTcpDocTests.cs]

```

var closed = Flow.FromSinkAndSource(Sink.Cancelled<ByteString>(), Source.Empty<ByteString>());
connection.HandleWith(closed, Materializer);

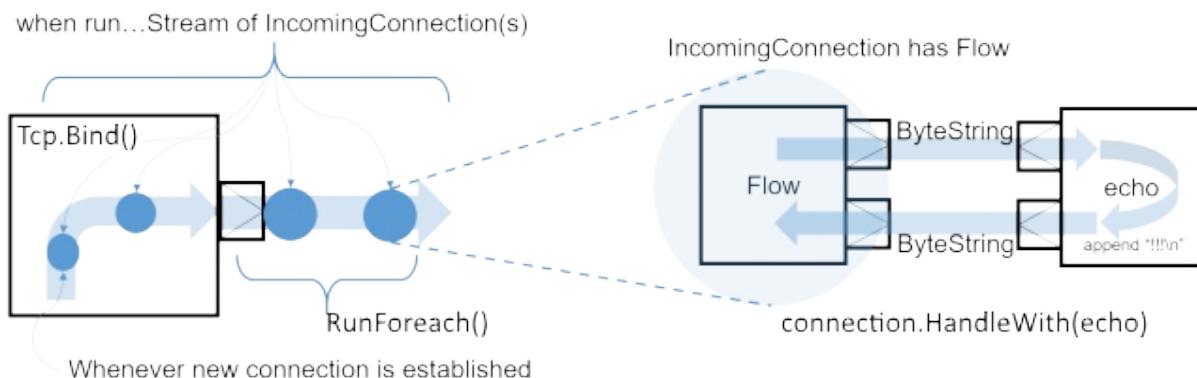
```

We can then test the TCP server by sending data to the TCP Socket using `netcat` (on Windows it is possible to use Linux Subsystem for Windows):

```

echo -n "Hello World" | netcat 127.0.0.1 8888
Hello World!!!

```



## Connecting: REPL Client

In this example we implement a rather naive Read Evaluate Print Loop client over TCP. Let's say we know a server has exposed a simple command line interface over TCP, and would like to interact with it using Akka Streams over TCP. To open an outgoing connection socket we use the `OutgoingConnection` method:

[!code-csharpStreamTcpDocTests.cs]

```

var connection = Sys.TcpStream().OutgoingConnection("127.0.0.1", 8888);

var replParser = Flow.Create<string>().TakeWhile(c => c != "q")
 .Concat(Source.Single("BYE"))
 .Select(elem => ByteString.FromString($"{elem}\n"));

```

```

var repl = Flow.Create<ByteString>()
 .Via(Framing.Delimiter(
 ByteString.FromString("\n"),
 maximumFrameLength: 256,
 allowTruncation: true))
 .Select(c => c.ToString())
 .Select(text =>
{
 Output.WriteLine($"Server: {text}");
 return text;
})
.Select(text => ReadLine("> "))
.Via(replParser);

connection.Join(repl).Run(Materializer);

```

The `repl` flow we use to handle the server interaction first prints the servers response, then awaits on input from the command line (this blocking call is used here just for the sake of simplicity) and converts it to a `ByteString` which is then sent over the wire to the server. Then we simply connect the TCP pipeline to this processing stage—at this point it will be materialized and start processing data once the server responds with an initial message.

A resilient REPL client would be more sophisticated than this, for example it should split out the input reading into a separate `SelectAsync` step and have a way to let the server write more data than one `ByteString` chunk at any given time, these improvements however are left as exercise for the reader.

## Avoiding deadlocks and liveness issues in back-pressured cycles

When writing such end-to-end back-pressed systems you may sometimes end up in a situation of a loop, in which either side is waiting for the other one to start the conversation. One does not need to look far to find examples of such back-pressure loops. In the two examples shown previously, we always assumed that the side we are connecting to would start the conversation, which effectively means both sides are back-pressed and can not get the conversation started. There are multiple ways of dealing with this which are explained in depth in [Graph cycles, liveness and deadlocks](#), however in client-server scenarios it is often the simplest to make either side simply send an initial message.

[!NOTE] In case of back-pressed cycles (which can occur even between different systems) sometimes you have to decide which of the sides has start the conversation in order to kick it off. This can be often done by injecting an initial message from one of the sides—a conversation starter.

To break this back-pressure cycle we need to inject some initial message, a “conversation starter”. First, we need to decide which side of the connection should remain passive and which active. Thankfully in most situations finding the right spot to start the conversation is rather simple, as it often is inherent to the protocol we are trying to implement using Streams. In chat-like applications, which our examples resemble, it makes sense to make the Server initiate the conversation by emitting a “hello” message:

[!code-csharp StreamTcpDocTests.cs]

```

connections.RunForEach(connection =>
{
 // server logic, parses incoming commands
 var commandParser = Flow.Create<string>().TakeWhile(c => c != "BYE").Select(c => c + "!");

 var welcomeMessage = $"Welcome to: {connection.LocalAddress}, you are: {connection.RemoteAddress}";
 var welcome = Source.Single(welcomeMessage);

 var serverLogic = Flow.Create<ByteString>()
 .Via(Framing.Delimiter(
 ByteString.FromString("\n"),

```

```

 maximumFrameLength: 256,
 allowTruncation: true))
 .Select(c => c.ToString())
 .Select(command =>
 {
 serverProbe.Tell(command);
 return command;
 })
 .Via(commandParser)
 .Merge(welcome)
 .Select(c => c + "\n")
 .Select(ByteString.FromString);

 connection.HandleWith(serverLogic, Materializer);
}, Materializer);

```

To emit the initial message we merge a `Source` with a single element, after the command processing but before the framing and transformation to `ByteString` this way we do not have to repeat such logic.

In this example both client and server may need to close the stream based on a parsed command - `BYE` in the case of the server, and `q` in the case of the client. This is implemented by taking from the stream until `q` and concatenating a `Source` with a single `BYE` element which will then be sent after the original source completed.

## Using framing in your protocol

Streaming transport protocols like TCP just pass streams of bytes, and does not know what is a logical chunk of bytes from the application's point of view. Often when implementing network protocols you will want to introduce your own framing. This can be done in two ways: An end-of-frame marker, e.g. end line `\n`, can do framing via `Framing.Delimiter`. Or a length-field can be used to build a framing protocol.

## Streaming File IO

Akka Streams provide simple Sources and Sinks that can work with `ByteString` instances to perform IO operations on files.

Streaming data from a file is as easy as creating a `FileIO.FromFile` given a target file, and an optional `chunkSize` which determines the buffer size determined as one "element" in such stream:

```

var file = new FileInfo("example.csv");
var result = FileIO.FromFile(file)
 .To(Sink.Ignore<ByteString>())
 .Run(materializer);

```

Please note that these processing stages are backed by Actors and by default are configured to run on a pre-configured threadpool-backed dispatcher dedicated for File IO. This is very important as it isolates the blocking file IO operations from the rest of the ActorSystem allowing each dispatcher to be utilised in the most efficient way. If you want to configure a custom dispatcher for file IO operations globally, you can do so by changing the `akka.stream.blocking-io-dispatcher`, or for a specific stage by specifying a custom Dispatcher in code, like this:

```

FileIO.FromFile(file)
 .WithAttributes(ActorAttributes.CreateDispatcher("custom-blocking-io-dispatcher"));

```

# Pipelining and Parallelism

Akka Streams processing stages (be it simple operators on Flows and Sources or graph junctions) are "fused" together and executed sequentially by default. This avoids the overhead of events crossing asynchronous boundaries but limits the flow to execute at most one stage at any given time.

In many cases it is useful to be able to concurrently execute the stages of a flow, this is done by explicitly marking them as asynchronous using the `Async` method. Each processing stage marked as asynchronous will run in a dedicated actor internally, while all stages not marked asynchronous will run in one single actor.

We will illustrate through the example of pancake cooking how streams can be used for various processing patterns, exploiting the available parallelism on modern computers. The setting is the following: both Chris and Bartosz like to make pancakes, but they need to produce sufficient amount in a cooking session to make all of the children happy. To increase their pancake production throughput they use two frying pans. How they organize their pancake processing is markedly different.

## Pipelining

Bartosz uses the two frying pans in an asymmetric fashion. The first pan is only used to fry one side of the pancake then the half-finished pancake is flipped into the second pan for the finishing fry on the other side. Once the first frying pan becomes available it gets a new scoop of batter. As an effect, most of the time there are two pancakes being cooked at the same time, one being cooked on its first side and the second being cooked to completion. This is how this setup would look like implemented as a stream:

```
// Takes a scoop of batter and creates a pancake with one side cooked
var fryingPan1 = Flow.Create<ScoopOfBatter>().Select(batter => HalfCookedPancake());

// Finishes a half-cooked pancake
var fryingPan2 = Flow.Create<HalfCookedPancake>().Select(halfCooked => Pancake());

// With the two frying pans we can fully cook pancakes
var pancakeChef = Flow.Create<ScoopOfBatter>().Via(fryingPan1.Async()).Via(fryingPan2.Async());
```

The two `Select` stages in sequence (encapsulated in the "frying pan" flows) will be executed in a pipelined way, basically doing the same as Bartosz with his frying pans:

1. A `ScoopOfBatter` enters `fryingPan1`
2. `fryingPan1` emits a `HalfCookedPancake` once `fryingPan2` becomes available
3. `fryingPan2` takes the `HalfCookedPancake`
4. at this point `fryingPan1` already takes the next scoop, without waiting for `fryingPan2` to finish

The benefit of pipelining is that it can be applied to any sequence of processing steps that are otherwise not parallelisable (for example because the result of a processing step depends on all the information from the previous step). One drawback is that if the processing times of the stages are very different then some of the stages will not be able to operate at full throughput because they will wait on a previous or subsequent stage most of the time. In the pancake example frying the second half of the pancake is usually faster than frying the first half, `fryingPan2` will not be able to operate at full capacity \*1.

[!NOTE] Asynchronous stream processing stages have internal buffers to make communication between them more efficient. For more details about the behavior of these and how to add additional buffers refer to [Buffers and working with rate](#).

## Parallel processing

Chris uses the two frying pans symmetrically. He uses both pans to fully fry a pancake on both sides, then puts the results on a shared plate. Whenever a pan becomes empty, he takes the next scoop from the shared bowl of batter. In essence he parallelizes the same process over multiple pans. This is how this setup will look like if implemented using streams:

```
var fryingPan = Flow.Create<ScoopOfBatter>().Select(batter => Pancake());

var pancakeChef = Flow.FromGraph(GraphDsl.Create(b =>
{
 var dispatchBatter = b.Add(new Balance<ScoopOfBatter>(2));
 var mergePancakes = b.Add(new Merge<Pancake>(2));

 // Using two frying pans in parallel, both fully cooking a pancake from the batter.
 // We always put the next scoop of batter to the first frying pan that becomes available.
 b.From(dispatchBatter.Out(0)).Via(fryingPan.Async()).To(mergePancakes.In(0));
 // Notice that we used the "fryingPan" flow without importing it via builder.Add().
 // Flows used this way are auto-imported, which in this case means that the two
 // uses of "fryingPan" mean actually different stages in the graph.
 b.From(dispatchBatter.Out(1)).Via(fryingPan.Async()).To(mergePancakes.In(1));

 return new FlowShape<ScoopOfBatter, Pancake>(dispatchBatter.In, mergePancakes.Out);
}));
```

The benefit of parallelizing is that it is easy to scale. In the pancake example it is easy to add a third frying pan with Chris' method, but Bartosz cannot add a third frying pan, since that would require a third processing step, which is not practically possible in the case of frying pancakes.

One drawback of the example code above that it does not preserve the ordering of pancakes. This might be a problem if children like to track their "own" pancakes. In those cases the `Balance` and `Merge` stages should be replaced by strict round-robin balancing and merging stages that put in and take out pancakes in a strict order.

A more detailed example of creating a worker pool can be found in the cookbook: [Balancing jobs to a fixed pool of workers](#)

## Combining pipelining and parallel processing

The two concurrency patterns that we demonstrated as means to increase throughput are not exclusive. In fact, it is rather simple to combine the two approaches and streams provide a nice unifying language to express and compose them.

First, let's look at how we can parallelize pipelined processing stages. In the case of pancakes this means that we will employ two chefs, each working using Bartosz's pipelining method, but we use the two chefs in parallel, just like Chris used the two frying pans. This is how it looks like if expressed as streams:

```
var pancakeChef = Flow.FromGraph(GraphDsl.Create(b =>
{
 var dispatchBatter = b.Add(new Balance<ScoopOfBatter>(2));
 var mergePancakes = b.Add(new Merge<Pancake>(2));

 // Using two pipelines, having two frying pans each, in total using
 // four frying pans
 b.From(dispatchBatter.Out(0)).Via(fryingPan1.Async()).Via(fryingPan2.Async()).To(mergePancakes.In(0));
 b.From(dispatchBatter.Out(1)).Via(fryingPan1.Async()).Via(fryingPan2.Async()).To(mergePancakes.In(1));

 return new FlowShape<ScoopOfBatter, Pancake>(dispatchBatter.In, mergePancakes.Out);
}));
```

The above pattern works well if there are many independent jobs that do not depend on the results of each other, but the jobs themselves need multiple processing steps where each step builds on the result of the previous one. In our case individual pancakes do not depend on each other, they can be cooked in parallel, on the other hand it is not possible to fry both sides of the same pancake at the same time, so the two sides have to be fried in sequence.

It is also possible to organize parallelized stages into pipelines. This would mean employing four chefs:

- the first two chefs prepare half-cooked pancakes from batter, in parallel, then putting those on a large enough flat surface.
- the second two chefs take these and fry their other side in their own pans, then they put the pancakes on a shared plate.

This is again straightforward to implement with the streams API:

```
var pancakeChefs1 = Flow.FromGraph(GraphDs1.Create(b =>
{
 var dispatchBatter = b.Add(new Balance<ScoopOfBatter>(2));
 var mergeHalfPancakes = b.Add(new Merge<HalfCookedPancake>(2));

 // Two chefs work with one frying pan for each, half-frying the pancakes then putting
 // them into a common pool
 b.From(dispatchBatter.Out(0)).Via(fryingPan1.Async()).To(mergeHalfPancakes.In(0));
 b.From(dispatchBatter.Out(1)).Via(fryingPan1.Async()).To(mergeHalfPancakes.In(1));

 return new FlowShape<ScoopOfBatter, HalfCookedPancake>(dispatchBatter.In, mergeHalfPancakes.Out);
}));

var pancakeChefs2 = Flow.FromGraph(GraphDs1.Create(b =>
{
 var dispatchHalfPancakes = b.Add(new Balance<HalfCookedPancake>(2));
 var mergePancakes = b.Add(new Merge<Pancake>(2));

 // Two chefs work with one frying pan for each, finishing the pancakes then putting
 // them into a common pool
 b.From(dispatchHalfPancakes.Out(0)).Via(fryingPan2.Async()).To(mergePancakes.In(0));
 b.From(dispatchHalfPancakes.Out(1)).Via(fryingPan2.Async()).To(mergePancakes.In(1));

 return new FlowShape<HalfCookedPancake, Pancake>(dispatchHalfPancakes.In, mergePancakes.Out);
}));

var kitchen = pancakeChefs1.Via(pancakeChefs2);
```

This usage pattern is less common but might be usable if a certain step in the pipeline might take wildly different times to finish different jobs. The reason is that there are more balance-merge steps in this pattern compared to the parallel pipelines. This pattern rebalances after each step, while the previous pattern only balances at the entry point of the pipeline. This only matters however if the processing time distribution has a large deviation.

\*1 Bartosz's reason for this seemingly suboptimal procedure is that he prefers the temperature of the second pan to be slightly lower than the first in order to achieve a more homogeneous result.

# Testing streams

Verifying behaviour of Akka Stream sources, flows and sinks can be done using various code patterns and libraries. Here we will discuss testing these elements using:

- simple sources, sinks and flows;
- sources and sinks in combination with `TestProbe` from the `Akka.Testkit` module;
- sources and sinks specifically crafted for writing tests from the `Akka.Streams.Testkit` module.

It is important to keep your data processing pipeline as separate sources, flows and sinks. This makes them easily testable by wiring them up to other sources or sinks, or some test harnesses that `Akka.Testkit` or `Akka.Streams.Testkit` provide.

## Built in sources, sinks and combinator

Testing a custom sink can be as simple as attaching a source that emits elements from a predefined collection, running a constructed test flow and asserting on the results that sink produced. Here is an example of a test for a sink:

```
var sinkUnderTest = Flow.Create<int>()
 .Select(x => x*2)
 .ToMaterialized(Sink.Aggregate<int, int>(0, (sum, i) => sum + i), Keep.Right);

var task = Source.From(Enumerable.Range(1, 4)).RunWith(sinkUnderTest, materializer);
task.Wait(TimeSpan.FromMilliseconds(500)).Should().BeTrue();
task.Result.Should().Be(20);
```

The same strategy can be applied for sources as well. In the next example we have a source that produces an infinite stream of elements. Such source can be tested by asserting that first arbitrary number of elements hold some condition. Here the `Grouped` combinator and `Sink.First` are very useful.

```
var sourceUnderTest = Source.Repeat(1).Select(x => x*2);

var task = sourceUnderTest.Grouped(10).RunWith(Sink.First<IEnumerable<int>>(), materializer);
task.Wait(TimeSpan.FromMilliseconds(500)).Should().BeTrue();
task.Result.ShouldAllBeEquivalentTo(Enumerable.Repeat(2, 10));
```

When testing a flow we need to attach a source and a sink. As both stream ends are under our control, we can choose sources that tests various edge cases of the flow and sinks that ease assertions.

```
var flowUnderTest = Flow.Create<int>().TakeWhile(x => x < 5);

var task = Source.From(Enumerable.Range(1, 10))
 .Via(flowUnderTest)
 .RunWith(Sink.Aggregate<int, List<int>>(new List<int>(), (list, i) =>
{
 list.Add(i);
 return list;
}), materializer);

task.Wait(TimeSpan.FromMilliseconds(500)).Should().BeTrue();
task.Result.ShouldAllBeEquivalentTo(Enumerable.Range(1, 4));
```

## TestKit

Akka Stream offers integration with Actors out of the box. This support can be used for writing stream tests that use familiar `TestProbe` from the `Akka.testkit` API.

One of the more straightforward tests would be to materialize stream to a `Task` and then use `pipe` pattern to pipe the result of that future to the probe.

```
var sourceUnderTest = Source.From(Enumerable.Range(1, 4)).Grouped(2);

var expected = new[] {Enumerable.Range(1, 2), Enumerable.Range(3, 2)}.AsEnumerable();
var probe = CreateTestProbe();

sourceUnderTest.Grouped(2)
 .RunWith(Sink.First<IEnumerable<int>>(), materializer)
 .PipeTo(probe.Ref);

probe.ExpectMsg(expected);
```

Instead of materializing to a task, we can use a `Sink.ActorRef` that sends all incoming elements to the given `IActorRef`. Now we can use assertion methods on `TestProbe` and expect elements one by one as they arrive. We can also assert stream completion by expecting for `onCompleteMessage` which was given to `Sink.ActorRef`.

```
var sourceUnderTest = Source.Tick(TimeSpan.FromSeconds(0), TimeSpan.FromMilliseconds(200), "Tick");

var probe = CreateTestProbe();
var cancellable = sourceUnderTest.To(Sink.ActorRef<string>(probe.Ref, "completed")).Run(materializer);

probe.ExpectMsg("Tick");
probe.ExpectNoMsg(TimeSpan.FromMilliseconds(100));
probe.ExpectMsg("Tick", TimeSpan.FromMilliseconds(200));
cancellable.Cancel();
probe.ExpectMsg("completed");
```

Similarly to `Sink.ActorRef` that provides control over received elements, we can use `Source.ActorRef` and have full control over elements to be sent.

```
var sinkUnderTest = Flow.Create<int>()
 .Select(x => x.ToString())
 .ToMaterialized(Sink.Aggregate<string, string>("", (s, s1) => s + s1), Keep.Right);

var t = Source.ActorRef<int>(8, OverflowStrategy.Fail)
 .ToMaterialized(sinkUnderTest, Keep.Both)
 .Run(materializer);

var actorRef = t.Item1;
var task = t.Item2;

actorRef.Tell(1);
actorRef.Tell(2);
actorRef.Tell(3);
actorRef.Tell(new Status.Success("done"));

task.Wait(TimeSpan.FromMilliseconds(500)).Should().BeTrue();
task.Result.Should().Be("123");
```

## Streams TestKit

You may have noticed various code patterns that emerge when testing stream pipelines. Akka Stream has a separate `Akka.Streams.Testkit` module that provides tools specifically for writing stream tests. This module comes with two main components that are `TestSource` and `TestSink` which provide sources and sinks that materialize to probes that

allow fluent API.

[!NOTE] Be sure to add the module `Akka.Streams.Testkit` to your dependencies.

A sink returned by `TestSink.Probe` allows manual control over demand and assertions over elements coming downstream.

```
var sourceUnderTest = Source.From(Enumerable.Range(1, 4)).Where(x => x%2 == 0).Select(x => x*2);

sourceUnderTest.RunWith(this.SinkProbe<int>(), materializer)
 .Request(2)
 .ExpectNext(4, 8)
 .ExpectComplete();
```

A source returned by `TestSource.Probe` can be used for asserting demand or controlling when stream is completed or ended with an error.

```
var sinkUnderTest = Sink.Cancelled<int>();

this.SourceProbe<int>()
 .ToMaterialized(sinkUnderTest, Keep.Left)
 .Run(materializer)
 .ExpectCancellation();
```

You can also inject exceptions and test sink behaviour on error conditions.

```
var sinkUnderTest = Sink.First<int>();

var t = this.SourceProbe<int>()
 .ToMaterialized(sinkUnderTest, Keep.Both)
 .Run(materializer);
var probe = t.Item1;
var task = t.Item2;

probe.SendError(new Exception("boom"));

task.Wait(TimeSpan.FromMilliseconds(500)).Should().BeTrue();
task.Exception.Message.Should().Be("boom");
```

Test source and sink can be used together in combination when testing flows.

```
var flowUnderTest = Flow.Create<int>().SelectAsyncUnordered(2, sleep => Task.Run(() =>
{
 Thread.Sleep(10*sleep);
 return sleep;
}));

var t = this.SourceProbe<int>()
 .Via(flowUnderTest)
 .ToMaterialized(this.SinkProbe<int>(), Keep.Both)
 .Run(materializer);

var pub = t.Item1;
var sub = t.Item2;

sub.Request(3);
pub.SendNext(3);
pub.SendNext(2);
pub.SendNext(1);

sub.ExpectNextUnordered(1, 2, 3);

pub.SendError(new Exception("Power surge in the linear subroutine C-47!"));
```

```
var ex = sub.ExpectError();
ex.Message.Should().Contain("C-47");
```

## Fuzzing Mode

For testing, it is possible to enable a special stream execution mode that exercises concurrent execution paths more aggressively (at the cost of reduced performance) and therefore helps exposing race conditions in tests. To enable this setting add the following line to your configuration:

```
akka.stream.materializer.debug.fuzzing-mode = on
```

[!WARNING] Never use this setting in production or benchmarks. This is a testing tool to provide more coverage of your code during tests, but it reduces the throughput of streams. A warning message will be logged if you have this setting enabled.

# Source stages

These built-in sources are available from `akka.stream.scaladsl.Source` :

## FromEnumerator

Stream the values from an `Enumerator`, requesting the next value when there is demand. The enumerator will be created anew for each materialization, which is the reason the method takes a function rather than an enumerator directly.

If the enumerator perform blocking operations, make sure to run it on a separate dispatcher.

**emits** the next value returned from the enumerator

**completes** when the enumerator reaches its end

## From

Stream the values of an `IEnumerable<T>`.

**emits** the next value of the enumerable

**completes** when the last element of the enumerable has been emitted

## Single

Stream a single object

**emits** the value once

**completes** when the single value has been emitted

## Repeat

Stream a single object repeatedly

**emits** the same value repeatedly when there is demand

**completes** never

## Cycle

Stream iterator in cycled manner. Internally new iterator is being created to cycle the one provided via argument meaning when original iterator runs out of elements process will start all over again from the beginning of the iterator provided by the evaluation of provided parameter. If method argument provides empty iterator stream will be terminated with exception.

**emits** the next value returned from cycled iterator

**completes** never

## Tick

A periodical repetition of an arbitrary object. Delay of first tick is specified separately from interval of the following ticks.

**emits** periodically, if there is downstream backpressure ticks are skipped

**completes** never

## FromTask

Send the single value of the `Task` when it completes and there is demand. If the task fails the stream is failed with that exception.

**emits** the task completes

**completes** after the task has completed

## Unfold

Stream the result of a function as long as it returns not `null`, the value inside the option consists of a tuple where the first value is a state passed back into the next call to the function allowing to pass a state. The first invocation of the provided fold function will receive the `zero` state.

Can be used to implement many stateful sources without having to touch the more low level `GraphStage` API.

**emits** when there is demand and the unfold function over the previous state returns non null value

**completes** when the unfold function returns an null value

## UnfoldAsync

Just like `Unfold` but the fold function returns a `Task` which will cause the source to complete or emit when it completes.

Can be used to implement many stateful sources without having to touch the more low level `GraphStage` API.

**emits** when there is demand and unfold state returned task completes with not null value

**completes** when the task returned by the unfold function completes with an null value

## Empty

Complete right away without ever emitting any elements. Useful when you have to provide a source to an API but there are no elements to emit.

**emits** never

**completes** directly

## Maybe

Materialize a `TaskCompletionSource<T>` that if completed with a `T` will emit that `T` and then complete the stream, or if completed with `null` complete the stream right away.

**emits** when the returned promise is completed with not null value

**completes** after emitting not null value, or directly if the promise is completed with null value

## Failed

Fail directly with a user specified exception.

**emits** never

**completes** fails the stream directly with the given exception

## Lazily

Defers creation and materialization of a `Source` until there is demand.

**emits** depends on the wrapped `Source`

**completes** depends on the wrapped `Source`

## ActorPublisher

Wrap an actor extending `ActorPublisher` as a source.

**emits** depends on the actor implementation

**completes** when the actor stops

## ActorRef

Materialize an `IActorRef`, sending messages to it will emit them on the stream. The actor contain a buffer but since communication is one way, there is no back pressure. Handling overflow is done by either dropping elements or failing the stream, the strategy is chosen by the user.

**emits** when there is demand and there are messages in the buffer or a message is sent to the actorref

**completes** when the actorref is sent `Akka.Actor.Status.Success` or `PoisonPill`

## PreMaterialize

Materializes this Source, immediately returning (1) its materialized value, and (2) a new Source that can consume elements 'into' the pre-materialized one.

Useful for when you need a materialized value of a Source when handing it out to someone to materialize it for you.

## Combine

Combine several sources, using a given strategy such as merge or concat, into one source.

**emits** when there is demand, but depending on the strategy

**completes** when all sources has completed

## UnfoldResource

Wrap any resource that can be opened, queried for next element (in a blocking way) and closed using three distinct functions into a source.

**emits** when there is demand and read function returns value

**completes** when read function returns `None`

## UnfoldResourceAsync

Wrap any resource that can be opened, queried for next element (in a blocking way) and closed using three distinct functions into a source. Functions return `Task` to achieve asynchronous processing

**emits** when there is demand and `Task` from read function returns value

**completes** when `Task` from read function returns `None`

## Queue

Materialize a `SourceQueue` onto which elements can be pushed for emitting from the source. The queue contains a buffer, if elements are pushed onto the queue faster than the source is consumed the overflow will be handled with a strategy specified by the user. Functionality for tracking when an element has been emitted is available through

`SourceQueue.Offer`.

**emits** when there is demand and the queue contains elements

**completes** when downstream completes

## AsSubscriber

Integration with Reactive Streams, materializes into a `Reactive.Streams.ISubscriber`.

## FromPublisher

Integration with Reactive Streams, subscribes to a `Reactive.Streams.IPublisher`.

## ZipN

Combine the elements of multiple streams into a stream of sequences.

**emits** when all of the inputs has an element available

**completes** when any upstream completes

## ZipWithN

Combine the elements of multiple streams into a stream of sequences using a combiner function.

**emits** when all of the inputs has an element available

**completes** when any upstream completes

## Sink stages

These built-in sinks are available from `Akka.Stream.DSL.Sink`:

### First

Materializes into a `Task` which completes with the first value arriving, after this the stream is canceled. If no element is emitted, the task is failed.

**cancels** after receiving one element

**backpressures** never

### FirstOrDefault

Materializes into a `Task<T>` which completes with the first value arriving, or a `default(T)` if the stream completes without any elements emitted.

**cancels** after receiving one element

**backpressures** never

## Last

Materializes into a `Task` which will complete with the last value emitted when the stream completes. If the stream completes with no elements the task is failed.

**cancels** never

**backpressures** never

## LastOrDefault

Materialize a `Task<T>` which completes with the last value emitted when the stream completes. if the stream completes with no elements the task is completed with `default(T)`.

**cancels** never

**backpressures** never

## Ignore

Consume all elements but discards them. Useful when a stream has to be consumed but there is no use to actually do anything with the elements.

**cancels** never

**backpressures** never

## Cancelled

Immediately cancel the stream

**cancels** immediately

## Sink

Collect values emitted from the stream into a collection, the collection is available through a `Task` or which completes when the stream completes. Note that the collection is bounded to `int.MaxValue`, if more element are emitted the sink will cancel the stream

**cancels** If too many values are collected

## Foreach

Invoke a given procedure for each element received. Note that it is not safe to mutate shared state from the procedure.

The sink materializes into a `Task` which completes when the stream completes, or fails if the stream fails.

Note that it is not safe to mutate state from the procedure.

**cancels** never

**backpressures** when the previous procedure invocation has not yet completed

## ForeachParallel

Like `Foreach` but allows up to `parallelism` procedure calls to happen in parallel.

**cancels** never

**backpressures** when the previous parallel procedure invocations has not yet completed

## OnComplete

Invoke a callback when the stream has completed or failed.

**cancels** never

**backpressures** never

## Aggregate

Fold over emitted element with a function, where each invocation will get the new element and the result from the previous fold invocation. The first invocation will be provided the `zero` value.

Materializes into a task that will complete with the last state when the stream has completed.

This stage allows combining values into a result without a global mutable state by instead passing the state along between invocations.

**cancels** never

**backpressures** when the previous fold function invocation has not yet completed

## Sum

Apply a reduction function on the incoming elements and pass the result to the next invocation. The first invocation receives the two first elements of the flow.

Materializes into a task that will be completed by the last result of the reduction function.

**cancels** never

**backpressures** when the previous reduction function invocation has not yet completed

## Combine

Combine several sinks into one using a user specified strategy

**cancels** depends on the strategy

**backpressures** depends on the strategy

## ActorRef

Send the elements from the stream to an `IActorRef`. No backpressure so care must be taken to not overflow the inbox.

**cancels** when the actor terminates

**backpressures** never

## ActorRefWithAck

Send the elements from the stream to an `IActorRef` which must then acknowledge reception after completing a message, to provide back pressure onto the sink.

**cancels** when the actor terminates

**backpressures** when the actor acknowledgment has not arrived.

## PreMaterialize

Materializes this Sink, immediately returning (1) its materialized value, and (2) a new Sink that can consume elements 'into' the pre-materialized one.

Useful for when you need a materialized value of a Sink when handing it out to someone to materialize it for you.

## ActorSubscriber

Create an actor from a `Props` upon materialization, where the actor implements `ActorSubscriber`, which will receive the elements from the stream.

Materializes into an `IActorRef` to the created actor.

**cancels** when the actor terminates

**backpressures** depends on the actor implementation

## AsPublisher

Integration with Reactive Streams, materializes into a `Reactive.Streams.IPublisher`.

## FromSubscriber

Integration with Reactive Streams, wraps a `Reactive.Streams.ISubscriber` as a sink

# Additional Sink and Source converters

Sources and sinks for integrating with `System.IO.Stream` can be found on `StreamConverters`. As they are blocking APIs the implementations of these stages are run on a separate dispatcher configured through the `akka.stream.blocking-io-dispatcher`.

## FromOutputStream

Create a sink that wraps an `Stream`. Takes a function that produces an `Stream`, when the sink is materialized the function will be called and bytes sent to the sink will be written to the returned `Stream`.

Materializes into a `Task` which will complete with a `IOResult` when the stream completes.

Note that a flow can be materialized multiple times, so the function producing the `Stream` must be able to handle multiple invocations.

The `Stream` will be closed when the stream that flows into the `Sink` is completed, and the `Sink` will cancel its inflow when the `Stream` is no longer writable.

## AsInputStream

Create a sink which materializes into an `Stream` that can be read to trigger demand through the sink. Bytes emitted through the stream will be available for reading through the `Stream`.

The `Stream` will be ended when the stream flowing into this `Sink` completes, and the closing the `Stream` will cancel the inflow of this `Sink`.

## FromInputStream

Create a source that wraps an `Stream`. Takes a function that produces an `Stream`, when the source is materialized the function will be called and bytes from the `Stream` will be emitted into the stream.

Materializes into a `Task` which will complete with a `IOResult` when the stream completes.

Note that a flow can be materialized multiple times, so the function producing the `Stream` must be able to handle multiple invocations.

The `Stream` will be closed when the `Source` is canceled from its downstream, and reaching the end of the `Stream` will complete the `Source`.

## AsOutputStream

Create a source that materializes into an `Stream`. When bytes are written to the `Stream` they are emitted from the source

The `Stream` will no longer be writable when the `Source` has been canceled from its downstream, and closing the `Stream` will complete the `Source`.

# File IO Sinks and Sources

Sources and sinks for reading and writing files can be found on [FileIO](#).

## FromFile

Emit the contents of a file, as `ByteString`'s, materializes into a `Task` which will be completed with a `IOResult` upon reaching the end of the file or if there is a failure.

## ToFile

Create a sink which will write incoming `ByteString`'s to a given file.

# Flow stages

All flows by default backpressure if the computation they encapsulate is not fast enough to keep up with the rate of incoming elements from the preceding stage. There are differences though how the different stages handle when some of their downstream stages backpressure them.

Most stages stop and propagate the failure downstream as soon as any of their upstreams emit a failure. This happens to ensure reliable teardown of streams and cleanup when failures happen. Failures are meant to be to model unrecoverable conditions, therefore they are always eagerly propagated. For in-band error handling of normal errors (dropping elements if a map fails for example) you should use the supervision support, or explicitly wrap your element types in a proper container that can express error or success states (for example `try` in C#).

## Simple processing stages

These stages can transform the rate of incoming elements since there are stages that emit multiple elements for a single input (e.g. `concatMany`) or consume multiple elements before emitting one output (e.g. `where`). However, these rate transformations are data-driven, i.e. it is the incoming elements that define how the rate is affected. This is in contrast with [Backpressure aware stages](#) which can change their processing behavior depending on being backpressured by downstream or not.

### AlsoTo

Attaches the given `Sink` to this `Flow`, meaning that elements that pass through will also be sent to the `Sink`.

**emits** when an element is available and demand exists both from the Sink and the downstream

**backpressures** when downstream or Sink backpressures

**completes** when upstream completes

### Select

Transform each element in the stream by calling a mapping function with it and passing the returned value downstream.

**emits** when the mapping function returns an element

**backpressures** when downstream backpressures

**completes** when upstream completes

### SelectMany

Transform each element into zero or more elements that are individually passed downstream.

**emits** when the mapping function returns an element or there are still remaining elements from the previously calculated collection

**backpressures** when downstream backpressures or there are still available elements from the previously calculated collection

**completes** when upstream completes and all remaining elements have been emitted

### StatefulSelectMany

Transform each element into zero or more elements that are individually passed downstream. The difference to `SelectMany` is that the transformation function is created from a factory for every materialization of the flow.

**emits** when the mapping function returns an element or there are still remaining elements from the previously calculated collection

**backpressures** when downstream backpressures or there are still available elements from the previously calculated collection

**completes** when upstream completes and all remaining elements have been emitted

### Where

Filter the incoming elements using a predicate. If the predicate returns true the element is passed downstream, if it returns false the element is discarded.

**emits** when the given predicate returns true for the element

**backpressures** when the given predicate returns true for the element and downstream backpressures

**completes** when upstream completes

## Collect

Apply a partial function to each incoming element, if the partial function is defined for a value the returned value is passed downstream. Can often replace `where` followed by `Select` to achieve the same in one single stage.

**emits** when the provided partial function is defined for the element

**backpressures** the partial function is defined for the element and downstream backpressures

**completes** when upstream completes

## Grouped

Accumulate incoming events until the specified number of elements have been accumulated and then pass the collection of elements downstream.

**emits** when the specified number of elements has been accumulated or upstream completed

**backpressures** when a group has been assembled and downstream backpressures

**completes** when upstream completes

## Sliding

Provide a sliding window over the incoming stream and pass the windows as groups of elements downstream.

Note: the last window might be smaller than the requested size due to end of stream.

**emits** the specified number of elements has been accumulated or upstream completed

**backpressures** when a group has been assembled and downstream backpressures

**completes** when upstream completes

## Scan

Emit its current value which starts at `zero` and then applies the current and next value to the given function emitting the next current value.

Note that this means that scan emits one element downstream before and upstream elements will not be requested until the second element is required from downstream.

**emits** when the function scanning the element returns a new element

**backpressures** when downstream backpressures

**completes** when upstream completes

## ScanAsync

Just like `Scan` but receiving a function that results in a `Task` to the next value.

**emits** when the `Task` resulting from the function scanning the element resolves to the next value

**backpressures** when downstream backpressures

**completes** when upstream completes and the last `Task` is resolved

## Aggregate

Start with current value `zero` and then apply the current and next value to the given function, when upstream complete the current value is emitted downstream.

**emits** when upstream completes

**backpressures** when downstream backpressures

**completes** when upstream completes

## AggregateAsync

Just like `Aggregate` but receiving a function that results in a `Task` to the next value.

**emits** when upstream completes and the last `Task` is resolved

**backpressures** when downstream backpressures

**completes** when upstream completes and the last `Task` is resolved

## Skip

Skip `n` elements and then pass any subsequent element downstream.

**emits** when the specified number of elements has been skipped already

**backpressures** when the specified number of elements has been skipped and downstream backpressures

**completes** when upstream completes

## Take

Pass `n` incoming elements downstream and then complete

**emits** while the specified number of elements to take has not yet been reached

**backpressures** when downstream backpressures

**completes** when the defined number of elements has been taken or upstream completes

## TakeWhile

Pass elements downstream as long as a predicate function return true for the element include the element when the predicate first return false and then complete.

**emits** while the predicate is true and until the first false result

**backpressures** when downstream backpressures

**completes** when predicate returned false or upstream completes

## SkipWhile

Skip elements as long as a predicate function return true for the element

**emits** when the predicate returned false and for all following stream elements

**backpressures** predicate returned false and downstream backpressures

**completes** when upstream completes

## Recover

Allow sending of one last element downstream when a failure has happened upstream.

Throwing an exception inside `Recover` *will* be logged on ERROR level automatically.

**emits** when the element is available from the upstream or upstream is failed and pf returns an element

**backpressures** when downstream backpressures, not when failure happened

**completes** when upstream completes or upstream failed with exception pf can handle

## RecoverWith

Allow switching to alternative Source when a failure has happened upstream.

Throwing an exception inside `RecoverWith` *will* be logged on ERROR level automatically.

**emits** the element is available from the upstream or upstream is failed and pf returns alternative Source

**backpressures** downstream backpressures, after failure happened it backpressures to alternative Source

**completes** upstream completes or upstream failed with exception pf can handle

## RecoverWithRetries

RecoverWithRetries allows to switch to alternative Source on flow failure. It will stay in effect after a failure has been recovered up to `attempts` number of times so that each time there is a failure it is fed into the `function` and a new Source may be materialized. Note that if you pass in 0, this won't attempt to recover at all. Passing -1 will behave exactly the same as `RecoverWith`.

Since the underlying failure signal OnError arrives out-of-band, it might jump over existing elements. This stage can recover the failure signal, but not the skipped elements, which will be dropped.

**emits** when element is available from the upstream or upstream is failed and element is available from alternative Source

**backpressures** when downstream backpressures

**completes** when upstream completes or upstream failed with exception function can handle

## SelectError

While similar to `Recover` this stage can be used to transform an error signal to a different one *without* logging it as an error in the process. So in that sense it is NOT exactly equivalent to `Recover(e -> throw e2)` since recover would log the `e2` error.

Since the underlying failure signal OnError arrives out-of-band, it might jump over existing elements. This stage can recover the failure signal, but not the skipped elements, which will be dropped.

Similarly to `Recover` throwing an exception inside `SelectError` *will* be logged on ERROR level automatically.

**emits** when element is available from the upstream or upstream is failed and function returns an element

**backpressures** when downstream backpressures

**completes** when upstream completes or upstream failed with exception function can handle

## Detach

Detach upstream demand from downstream demand without detaching the stream rates.

**emits** when the upstream stage has emitted and there is demand

**backpressures** when downstream backpressures

**completes** when upstream completes

## Throttle

Limit the throughput to a specific number of elements per time unit, or a specific total cost per time unit, where a function has to be provided to calculate the individual cost of each element.

**emits** when upstream emits an element and configured time per each element elapsed

**backpressures** when downstream backpressures

**completes** when upstream completes

# Asynchronous processing stages

These stages encapsulate an asynchronous computation, properly handling backpressure while taking care of the asynchronous operation at the same time (usually handling the completion of a Task).

## SelectAsync

Pass incoming elements to a function that return a `Task` result. When the task arrives the result is passed downstream. Up to `n` elements can be processed concurrently, but regardless of their completion time the incoming order will be kept when results complete. For use cases where order does not matter `SelectAsyncUnordered` can be used.

If a Task fails, the stream also fails (unless a different supervision strategy is applied)

**emits** when the Task returned by the provided function finishes for the next element in sequence

**backpressures** when the number of tasks reaches the configured parallelism and the downstream backpressures

**completes** when upstream completes and all tasks has been completed and all elements has been emitted

## SelectAsyncUnordered

Like `SelectAsync` but `Task` results are passed downstream as they arrive regardless of the order of the elements that triggered them.

If a Task fails, the stream also fails (unless a different supervision strategy is applied)

**emits** any of the tasks returned by the provided function complete

**backpressures** when the number of tasks reaches the configured parallelism and the downstream backpressures

**completes** upstream completes and all tasks has been completed and all elements has been emitted

## Timer driven stages

These stages process elements using timers, delaying, dropping or grouping elements for certain time durations.

### TakeWithin

Pass elements downstream within a timeout and then complete.

**emits** when an upstream element arrives

**backpressures** downstream backpressures

**completes** upstream completes or timer fires

### SkipWithin

Skip elements until a timeout has fired

**emits** after the timer fired and a new upstream element arrives

**backpressures** when downstream backpressures

**completes** upstream completes

### GroupedWithin

Chunk up the stream into groups of elements received within a time window, or limited by the given number of elements, whichever happens first.

**emits** when the configured time elapses since the last group has been emitted

**backpressures** when the group has been assembled (the duration elapsed) and downstream backpressures

**completes** when upstream completes

### InitialDelay

Delay the initial element by a user specified duration from stream materialization.

**emits** upstream emits an element if the initial delay already elapsed

**backpressures** downstream backpressures or initial delay not yet elapsed

**completes** when upstream completes

### Delay

Delay every element passed through with a specific duration.

**emits** there is a pending element in the buffer and configured time for this element elapsed

**backpressures** differs, depends on `OverflowStrategy` set

**completes** when upstream completes and buffered elements has been drained

# Backpressure aware stages

These stages are aware of the backpressure provided by their downstreams and able to adapt their behavior to that signal.

## Conflate

Allow for a slower downstream by passing incoming elements and a summary into an aggregate function as long as there is backpressure. The summary value must be of the same type as the incoming elements, for example the sum or average of incoming numbers, if aggregation should lead to a different type `ConflateWithSeed` can be used:

**emits** when downstream stops backpressuring and there is a conflated element available

**backpressures** when the aggregate function cannot keep up with incoming elements

**completes** when upstream completes

## ConflateWithSeed

Allow for a slower downstream by passing incoming elements and a summary into an aggregate function as long as there is backpressure. When backpressure starts or there is no backpressure element is passed into a `seed` function to transform it to the summary type.

**emits** when downstream stops backpressuring and there is a conflated element available

**backpressures** when the aggregate or seed functions cannot keep up with incoming elements

**completes** when upstream completes

## Batch

Allow for a slower downstream by passing incoming elements and a summary into an aggregate function as long as there is backpressure and a maximum number of batched elements is not yet reached. When the maximum number is reached and downstream still backpressures batch will also backpressure.

When backpressure starts or there is no backpressure element is passed into a `seed` function to transform it to the summary type.

Will eagerly pull elements, this behavior may result in a single pending (i.e. buffered) element which cannot be aggregated to the batched value.

**emits** when downstream stops backpressuring and there is a batched element available

**backpressures** when batched elements reached the max limit of allowed batched elements & downstream backpressures

**completes** when upstream completes and a "possibly pending" element was drained

## BatchWeighted

Allow for a slower downstream by passing incoming elements and a summary into an aggregate function as long as there is backpressure and a maximum weight batched elements is not yet reached. The weight of each element is determined by applying `costFunction`. When the maximum total weight is reached and downstream still backpressures batch will also backpressure.

Will eagerly pull elements, this behavior may result in a single pending (i.e. buffered) element which cannot be aggregated to the batched value.

**emits** downstream stops backpressuring and there is a batched element available

**backpressures** batched elements reached the max weight limit of allowed batched elements & downstream backpressures

**completes** upstream completes and a "possibly pending" element was drained

## Expand

Allow for a faster downstream by expanding the last incoming element to an `Enumerator`

**emits** when downstream stops backpressuring

**backpressures** when downstream backpressures

**completes** when upstream completes

## Buffer (Backpressure)

Allow for a temporarily faster upstream events by buffering `size` elements. When the buffer is full backpressure is applied.

**emits** when downstream stops backpressuring and there is a pending element in the buffer

**backpressures** when buffer is full

**completes** when upstream completes and buffered elements has been drained

## Buffer (Drop)

Allow for a temporarily faster upstream events by buffering `size` elements. When the buffer is full elements are dropped according to the specified `overflowStrategy` :

- `dropHead` drops the oldest element in the buffer to make space for the new element
- `dropTail` drops the youngest element in the buffer to make space for the new element
- `dropBuffer` drops the entire buffer and buffers the new element
- `dropNew` drops the new element

**emits** when downstream stops backpressuring and there is a pending element in the buffer

**backpressures** never (when dropping cannot keep up with incoming elements)

**completes** upstream completes and buffered elements has been drained

## Buffer (Fail)

Allow for a temporarily faster upstream events by buffering `size` elements. When the buffer is full the stage fails the flow with a `BufferOverflowException`.

**emits** when downstream stops backpressuring and there is a pending element in the buffer

**backpressures** never, fails the stream instead of backpressuring when buffer is full

**completes** when upstream completes and buffered elements has been drained

# Nesting and flattening stages

These stages either take a stream and turn it into a stream of streams (nesting) or they take a stream that contains nested streams and turn them into a stream of elements instead (flattening).

## PrefixAndTail

Take up to `n` elements from the stream (less than `n` only if the upstream completes before emitting `n` elements) and returns a pair containing a strict sequence of the taken element and a stream representing the remaining elements.

**emits** when the configured number of prefix elements are available. Emits this prefix, and the rest as a substream

**backpressures** when downstream backpressures or substream backpressures

**completes** when prefix elements has been consumed and substream has been consumed

## GroupBy

Demultiplex the incoming stream into separate output streams.

**emits** an element for which the grouping function returns a group that has not yet been created. Emits the new group there is an element pending for a group whose substream backpressures

**completes** when upstream completes (Until the end of stream it is not possible to know whether new substreams will be needed or not)

## SplitWhen

Split off elements into a new substream whenever a predicate function return `true`.

**emits** an element for which the provided predicate is true, opening and emitting a new substream for subsequent elements

**backpressures** when there is an element pending for the next substream, but the previous is not fully consumed yet, or the substream backpressures

**completes** when upstream completes (Until the end of stream it is not possible to know whether new substreams will be needed or not)

## SplitAfter

End the current substream whenever a predicate returns `true`, starting a new substream for the next element.

**emits** when an element passes through. When the provided predicate is true it emits the element \* and opens a new substream for subsequent element

**backpressures** when there is an element pending for the next substream, but the previous is not fully consumed yet, or the substream backpressures

**completes** when upstream completes (Until the end of stream it is not possible to know whether new substreams will be needed or not)

## ConcatMany

Transform each input element into a `source` whose elements are then flattened into the output stream through concatenation. This means each source is fully consumed before consumption of the next source starts.

**emits** when the current consumed substream has an element available

**backpressures** when downstream backpressures

**completes** when upstream completes and all consumed substreams complete

## MergeMany

Transform each input element into a `Source` whose elements are then flattened into the output stream through merging. The maximum number of merged sources has to be specified.

**emits** when one of the currently consumed substreams has an element available

**backpressures** when downstream backpressures

**completes** when upstream completes and all consumed substreams complete

# Time aware stages

Those stages operate taking time into consideration.

## InitialTimeout

If the first element has not passed through this stage before the provided timeout, the stream is failed with a `TimeoutException`.

**emits** when upstream emits an element

**backpressures** when downstream backpressures

**completes** when upstream completes or fails if timeout elapses before first element arrives

**cancels** when downstream cancels

## CompletionTimeout

If the completion of the stream does not happen until the provided timeout, the stream is failed with a `TimeoutException`.

**emits** when upstream emits an element

**backpressures** when downstream backpressures

**completes** when upstream completes or fails if timeout elapses before upstream completes

**cancels** when downstream cancels

## IdleTimeout

If the time between two processed elements exceeds the provided timeout, the stream is failed with a `TimeoutException`. The timeout is checked periodically, so the resolution of the check is one period (equals to timeout value).

**emits** when upstream emits an element

**backpressures** when downstream backpressures

**completes** when upstream completes or fails if timeout elapses between two emitted elements

**cancels** when downstream cancels

## BackpressureTimeout

If the time between the emission of an element and the following downstream demand exceeds the provided timeout, the stream is failed with a `TimeoutException`. The timeout is checked periodically, so the resolution of the check is one period (equals to timeout value).

**emits** when upstream emits an element

**backpressures** when downstream backpressures

**completes** when upstream completes or fails if timeout elapses between element emission and downstream demand.

**cancels** when downstream cancels

## KeepAlive

Injects additional (configured) elements if upstream does not emit for a configured amount of time.

**emits** when upstream emits an element or if the upstream was idle for the configured period

**backpressures** when downstream backpressures

**completes** when upstream completes

**cancels** when downstream cancels

## InitialDelay

Delays the initial element by the specified duration.

**emits** when upstream emits an element if the initial delay is already elapsed

**backpressures** when downstream backpressures or initial delay is not yet elapsed

**completes** when upstream completes

**cancels** when downstream cancels

# Fan-in stages

These stages take multiple streams as their input and provide a single output combining the elements from all of the inputs in different ways.

## Merge

Merge multiple sources. Picks elements randomly if all sources has elements ready.

**emits** when one of the inputs has an element available

**backpressures** when downstream backpressures

**completes** when all upstreams complete (This behavior is changeable to completing when any upstream completes by setting `eagerComplete=true`.)

## MergeSorted

Merge multiple sources. Waits for one element to be ready from each input stream and emits the smallest element.

**emits** when all of the inputs have an element available

**backpressures** when downstream backpressures

**completes** when all upstreams complete

## MergePreferred

Merge multiple sources. Prefer one source if all sources has elements ready.

**emits** when one of the inputs has an element available, preferring a defined input if multiple have elements available

**backpressures** when downstream backpressures

**completes** when all upstreams complete (This behavior is changeable to completing when any upstream completes by setting `eagerComplete=true`.)

## MergePrioritized

Merge multiple sources. Prefer sources depending on priorities if all sources has elements ready. If a subset of all sources has elements ready the relative priorities for those sources are used to prioritise.

**emits** when one of the inputs has an element available, preferring inputs based on their priorities if multiple have elements available

**backpressures** when downstream backpressures

**completes** when all upstreams complete (This behavior is changeable to completing when any upstream completes by setting `eagerComplete=true`.)

## Zip

Combines elements from each of multiple sources into tuples and passes the tuples downstream.

**emits** when all of the inputs have an element available

**backpressures** when downstream backpressures

**completes** when any upstream completes

## ZipWith

Combines elements from multiple sources through a `combine` function and passes the returned value downstream.

**emits** when all of the inputs have an element available

**backpressures** when downstream backpressures

**completes** when any upstream completes

## ZipWithIndex

Zips elements of current flow with its indices.

**emits** when upstream emits an element and is paired with their index

**backpressures** when downstream backpressures

**completes** when any upstream completes

## Concat

After completion of the original upstream the elements of the given source will be emitted.

**emits** when the current stream has an element available; if the current input completes, it tries the next one

**backpressures** when downstream backpressures

**completes** when all upstreams complete

## Prepend

Prepends the given source to the flow, consuming it until completion before the original source is consumed.

If materialized values needs to be collected `prependMat` is available.

**emits** when the given stream has an element available; if the given input completes, it tries the current one

**backpressures** when downstream backpressures

**completes** when all upstreams complete

## OrElse

If the primary source completes without emitting any elements, the elements from the secondary source are emitted. If the primary source emits any elements the secondary source is cancelled.

Note that both sources are materialized directly and the secondary source is backpressured until it becomes the source of elements or is cancelled.

Signal errors downstream, regardless which of the two sources emitted the error.

**emits** when an element is available from first stream or first stream closed without emitting any elements and an element is available from the second stream

**backpressures** when downstream backpressures

**completes** the primary stream completes after emitting at least one element, when the primary stream completes without emitting and the secondary stream already has completed or when the secondary stream completes

## Interleave

Emits a specifiable number of elements from the original source, then from the provided source and repeats. If one source completes the rest of the other stream will be emitted.

**emits** when element is available from the currently consumed upstream

**backpressures** when upstream backpressures

**completes** when both upstreams have completed

## Fan-out stages

These have one input and multiple outputs. They might route the elements between different outputs, or emit elements on multiple outputs at the same time.

## Unzip

Takes a stream of two element tuples and unzips the two elements into two different downstreams.

**emits** when all of the outputs stops backpressuring and there is an input element available

**backpressures** when any of the outputs backpressures

**completes** when upstream completes

## UnzipWith

Splits each element of input into multiple downstreams using a function

**emits** when all of the outputs stops backpressuring and there is an input element available

**backpressures** when any of the outputs backpressures

**completes** when upstream completes

## broadcast

Emit each incoming element each of `n` outputs.

**emits** when all of the outputs stops backpressuring and there is an input element available

**backpressures** when any of the outputs backpressures

**completes** when upstream completes

## Balance

Fan-out the stream to several streams. Each upstream element is emitted to the first available downstream consumer.

**emits** when any of the outputs stops backpressuring; emits the element to the first available output

**backpressures** when all of the outputs backpressure

**completes** when upstream completes

# Watching status stages

## WatchTermination

Materializes to a `Task` that will be completed with Done or failed depending whether the upstream of the stage has been completed or failed. The stage otherwise passes through elements unchanged.

**emits** when input has an element available

**backpressures** when output backpressures

**completes** when upstream completes

## Monitor

Materializes to a `FlowMonitor` that monitors messages flowing through or completion of the stage. The stage otherwise passes through elements unchanged. Note that the `FlowMonitor` inserts a memory barrier every time it processes an event, and may therefore affect performance.

**emits** when upstream emits an element

**backpressures** when downstream **backpressures**

**completes** when upstream completes

# Introduction

This is a collection of patterns to demonstrate various usage of the Akka Streams API by solving small targeted problems in the format of "recipes". The purpose of this page is to give inspiration and ideas how to approach various small tasks involving streams. The recipes in this page can be used directly as-is, but they are most powerful as starting points: customization of the code snippets is warmly encouraged.

This part also serves as supplementary material for the main body of documentation. It is a good idea to have this page open while reading the manual and look for examples demonstrating various streaming concepts as they appear in the main body of documentation.

If you need a quick reference of the available processing stages used in the recipes see [Overview of built-in stages and their semantics](#)

# Working with Flows

In this collection we show simple recipes that involve linear flows. The recipes in this section are rather general, more targeted recipes are available as separate sections ([Buffers and working with rate](#), [Working with streaming IO](#)).

## Logging elements of a stream

**Situation:** During development it is sometimes helpful to see what happens in a particular section of a stream.

The simplest solution is to simply use a `Select` operation and use `WriteLine` to print the elements received to the console. While this recipe is rather simplistic, it is often suitable for a quick debug session.

```
var mySource = Source.Empty<string>();

var loggedSource = mySource.Select(element =>
{
 Console.WriteLine(element);
 return element;
});
```

Another approach to logging is to use `Log()` operation which allows configuring logging for elements flowing through the stream as well as completion and erroring.

```
// customise log levels
mySource.Log("before-select")
 .WithAttributes(Attributes.CreateLogLevels(onElement: LogLevel.WarningLevel))
 .Select>Analyse);

// or provide custom logging adapter
mySource.Log("custom", null, Logging.GetLogger(sys, "customLogger"));
```

## Flattening a stream of sequences

**Situation:** A stream is given as a stream of sequence of elements, but a stream of elements needed instead, streaming all the nested elements inside the sequences separately.

The `SelectMany` operation can be used to implement a one-to-many transformation of elements using a mapper function in the form of `In => IEnumerable<Out>`. In this case we want to map a `Enumerable` of elements to the elements in the collection itself, so we can just call `SelectMany(x => x)`.

```
Source<List<Message>, NotUsed > myData = someDataSource;
Source<Message, NotUsed> flattened = myData.SelectMany(x => x);
```

## Draining a stream to a strict collection

**Situation:** A possibly unbounded sequence of elements is given as a stream, which needs to be collected into a collection while ensuring boundedness

A common situation when working with streams is one where we need to collect incoming elements into a collection. This operation is supported via `Sink.Seq` which materializes into a `Task<IEnumerable<T>>`.

The function `Limit` or `Take` should always be used in conjunction in order to guarantee stream boundedness, thus preventing the program from running out of memory.

For example, this is best avoided:

```
// Dangerous: might produce a collection with 2 billion elements!
var f = mySource.RunWith(Sink.Seq<string>(), materializer);
```

Rather, use `Limit` or `Take` to ensure that the resulting `Enumerable` will contain only up to `max` elements:

```
var MAX_ALLOWED_SIZE = 100;

// OK. Task will fail with a `StreamLimitReachedException`
// if the number of incoming elements is larger than max
var limited = mySource.Limit(MAX_ALLOWED_SIZE).RunWith(Sink.Seq<string>(), materializer);

// OK. Collect up until max-th elements only, then cancel upstream
var ignoreOverflow = mySource.Take(MAX_ALLOWED_SIZE).RunWith(Sink.Seq<string>(), materializer);
```

## Calculating the digest of a ByteString stream

**Situation:** A stream of bytes is given as a stream of `ByteStrings` and we want to calculate the cryptographic digest of the stream.

This recipe uses a `GraphStage` to host a `HashAlgorithm` class (part of the .Net Cryptography API). When the stream starts, the `onPull` handler of the stage is called, which just bubbles up the `pull` event to its upstream. As a response to this pull, a `ByteString` chunk will arrive (`onPush`) which we store, then it will pull for the next chunk.

Eventually the stream of `ByteStrings` depletes and we get a notification about this event via `onUpstreamFinish`. At this point we want to emit the digest value, but we cannot do it with `push` in this handler directly since there may be no downstream demand. Instead we call `emit` which will temporarily replace the handlers, emit the provided value when demand comes in and then reset the stage state. It will then complete the stage.

```
public class DigestCalculator : GraphStage<FlowShape<ByteString, ByteString>>
{
 private readonly string _algorithm;

 private sealed class Logic : GraphStageLogic
 {
 private readonly HashAlgorithm _digest;
 private ByteString _bytes;

 public Logic(DigestCalculator calculator) : base(calculator.Shape)
 {
 _digest = HashAlgorithm.Create(calculator._algorithm);
 _bytes = ByteString.Empty;
 }

 @Override
 protected void onPull()
 {
 _bytes += pull();
 }

 @Override
 protected void onUpstreamFinish()
 {
 emit(_digest.ComputeHash(_bytes));
 reset();
 }
 }
}
```

```

 SetHandler(calculator.Out, onPull: () => { Pull(calculator.In); });

 SetHandler(calculator.In, onPush: () =>
 {
 _bytes += Grab(calculator.In);
 Pull(calculator.In);
 }, onUpstreamFinish: () =>
 {
 Emit(calculator.Out, ByteString.Create(_digest.ComputeHash(_bytes.ToArray())));
 CompleteStage();
 });
 }
}

public DigestCalculator(string algorithm)
{
 _algorithm = algorithm;
 Shape = new FlowShape<ByteString, ByteString>(In, Out);
}

public Inlet<ByteString> In { get; } = new Inlet<ByteString>("DigestCalculator.in");

public Outlet<ByteString> Out { get; } = new Outlet<ByteString>("DigestCalculator.out");

public override FlowShape<ByteString, ByteString> Shape { get; }

protected override GraphStageLogic CreateLogic(Attributes inheritedAttributes) => new Logic(this);
}

var data = Source.Empty<ByteString>();
var digest = data.Via(new DigestCalculator("SHA-256"));

```

## Parsing lines from a stream of ByteStrings

**Situation:** A stream of bytes is given as a stream of `ByteStrings` containing lines terminated by line ending characters (or, alternatively, containing binary frames delimited by a special delimiter byte sequence) which needs to be parsed.

The `Framing` helper object contains a convenience method to parse messages from a stream of `ByteStrings` :

```

var rawData = Source.Empty<ByteString>();
var linesStream = rawData
 .Via(Framing.Delimiter(delimiter: ByteString.FromString("\r\n"), maximumFrameLength: 10, allowTruncation: true))
 .Select(b => b.DecodeString());

```

## Implementing reduce-by-key

**Situation:** Given a stream of elements, we want to calculate some aggregated value on different subgroups of the elements.

The "hello world" of reduce-by-key style operations is `wordcount` which we demonstrate below. Given a stream of words we first create a new stream that groups the words according to the `identity` function, i.e. now we have a stream of streams, where every substream will serve identical words.

To count the words, we need to process the stream of streams (the actual groups containing identical words).

`GroupBy` returns a `SubFlow`, which means that we transform the resulting substreams directly. In this case we use the `Reduce` combinator to aggregate the word itself and the number of its occurrences within a tuple `(String, Integer)`. Each substream will then emit one final value—precisely such a pair—when the overall input completes. As a last step we merge back these values from the substreams into one single output stream.

One noteworthy detail pertains to the `MaximumDistinctWords` parameter: this defines the breadth of the `groupBy` and `merge` operations. Akka Streams is focused on bounded resource consumption and the number of concurrently open inputs to the `merge` operator describes the amount of resources needed by the `merge` itself. Therefore only a finite number of substreams can be active at any given time. If the `GroupBy` operator encounters more keys than this number then the stream cannot continue without violating its resource bound, in this case `GroupBy` will terminate with a failure.

```
var words = Source.Empty<string>();
var counts = words
 // split the words into separate streams first
 .GroupBy(MaximumDistinctWords, x => x)
 // transform each element to pair with number of words in it
 .Select(x => Tuple.Create(x, 1))
 // add counting logic to the streams
 .Sum((l, r) => Tuple.Create(l.Item1, l.Item2 + r.Item2))
 // get a stream of word counts
 .MergeSubstreams();
```

By extracting the parts specific to `wordcount` into

- a `GroupKey` function that defines the groups
- a `Select` map each element to value that is used by the reduce on the substream
- a `Reduce` function that does the actual reduction

we get a generalized version below:

```
public Flow<TIn, Tuple< TKey, TOut>, NotUsed> ReduceByKey<TIn, TKey, TOut>(int maximumGroupSize,
 Func<TIn, TKey> groupKey,
 Func<TIn, TOut> map,
 Func<TOut, TOut, TOut> reduce)
{
 return (Flow<TIn, Tuple< TKey, TOut>, NotUsed>)
 Flow.Create<TIn>()
 .GroupBy(maximumGroupSize, groupKey)
 .Select(e => Tuple.Create(groupKey(e), map(e)))
 .Sum((l, r) => Tuple.Create(l.Item1, reduce(l.Item2, r.Item2)))
 .MergeSubstreams();
}

var counts = words.Via(ReduceByKey(MaximumDistinctWords,
 groupKey: (string word) => word,
 map: word => 1,
 reduce: (l, r) => l + r));
```

[!NOTE] Please note that the reduce-by-key version we discussed above is sequential in reading the overall input stream, in other words it is **NOT** a parallelization pattern like MapReduce and similar frameworks.

## Sorting elements to multiple groups with `groupBy`

**Situation:** The `GroupBy` operation strictly partitions incoming elements, each element belongs to exactly one group. Sometimes we want to map elements into multiple groups simultaneously.

To achieve the desired result, we attack the problem in two steps:

- first, using a function `TopicMapper` that gives a list of topics (groups) a message belongs to, we transform our stream of `Message` to a stream of `(Message, Topic)` where for each topic the message belongs to a separate pair will be emitted. This is achieved by using `SelectMany`
- Then we take this new stream of message topic pairs (containing a separate pair for each topic a given message belongs to) and feed it into `GroupBy`, using the topic as the group key.

```

Func<Message, ImmutableHashSet<Topic>> topicMapper = ExtractTopics;
var elements = Source.Empty<Message>();
var messageAndTopic = elements.SelectMany(msg =>
{
 var topicsForMessage = topicMapper(msg);
 // Create a (Msg, Topic) pair for each of the topics
 // the message belongs to
 return topicsForMessage.Select(t => Tuple.Create(msg, t));
});

var multiGroups = messageAndTopic.GroupBy(2, tuple => tuple.Item2).Select(tuple =>
{
 var msg = tuple.Item1;
 var topic = tuple.Item2;

 // do what needs to be done
});

```

## Working with Graphs

In this collection we show recipes that use stream graph elements to achieve various goals.

### Triggering the flow of elements programmatically

**Situation:** Given a stream of elements we want to control the emission of those elements according to a trigger signal. In other words, even if the stream would be able to flow (not being backpressured) we want to hold back elements until a trigger signal arrives.

This recipe solves the problem by simply zipping the stream of `Message` elements with the stream of `Trigger` signals. Since `Zip` produces pairs, we simply map the output stream selecting the first element of the pair.

```

var elements = Source.Empty<Message>();
var triggerSource = Source.Empty<Trigger>();
var sink = Sink.Ignore<Message>().MapMaterializedValue(_ => NotUsed.Instance);

var graph = RunnableGraph.FromGraph(GraphDsl.Create(b =>
{
 var zip = b.Add(new Zip<Message, Trigger>());

 b.From(elements).To(zip.In0);
 b.From(triggerSource).To(zip.In1);
 b.From(zip.Out).Via(Flow.Create<Tuple<Message, Trigger>>().Select(t => t.Item1)).To(sink);

 return ClosedShape.Instance;
}));

```

Alternatively, instead of using a `Zip`, and then using `Select` to get the first element of the pairs, we can avoid creating the pairs in the first place by using `ZipWith` which takes a two argument function to produce the output element. If this function would return a pair of the two argument it would be exactly the behavior of `Zip` so `ZipWith` is a generalization of zipping.

```

var graph = RunnableGraph.FromGraph(GraphDsl.Create(b =>
{
 var zip = b.Add(ZipWith.Apply((Message msg, Trigger trigger) => msg));

 b.From(elements).To(zip.In0);
 b.From(triggerSource).To(zip.In1);
 b.From(zip.Out).To(sink);

 return ClosedShape.Instance;
});

```

```
});
```

## Balancing jobs to a fixed pool of workers

**Situation:** Given a stream of jobs and a worker process expressed as a `Flow` create a pool of workers that automatically balances incoming jobs to available workers, then merges the results.

We will express our solution as a function that takes a worker flow and the number of workers to be allocated and gives a flow that internally contains a pool of these workers. To achieve the desired result we will create a `Flow` from a graph.

The graph consists of a `Balance` node which is a special fan-out operation that tries to route elements to available downstream consumers. In a `for` loop we wire all of our desired workers as outputs of this balancer element, then we wire the outputs of these workers to a `Merge` element that will collect the results from the workers.

To make the worker stages run in parallel we mark them as asynchronous with `Async`.

```
public Flow<TIn, TOut, NotUsed> Balancer<TIn, TOut>(Flow<TIn, TOut, NotUsed> worker, int workerCount)
{
 return Flow.FromGraph(GraphDsl.Create(b =>
 {
 var balancer = b.Add(new Balance<TIn>(workerCount, waitForAllDownstreams: true));
 var merge = b.Add(new Merge<TOut>(workerCount));

 for (var i = 0; i < workerCount; i++)
 b.From(balancer).Via(worker.Async()).To(merge);

 return new FlowShape<TIn, TOut>(balancer.In, merge.Out);
 }));
}

var myJobs = Source.Empty<Job>();
var worker = Flow.Create<Job>().Select(j => new Done(j));
var processedJobs = myJobs.Via(Balancer(worker, 3));
```

## Working with rate

This collection of recipes demonstrate various patterns where rate differences between upstream and downstream needs to be handled by other strategies than simple backpressure.

### Dropping elements

**Situation:** Given a fast producer and a slow consumer, we want to drop elements if necessary to not slow down the producer too much.

This can be solved by using a versatile rate-transforming operation, `Conflate`. Conflate can be thought as a special `Sum` operation that collapses multiple upstream elements into one aggregate element if needed to keep the speed of the upstream unaffected by the downstream.

When the upstream is faster, the sum process of the `Conflate` starts. Our reducer function simply takes the freshest element. This is shown as a simple dropping operation.

```
var droppyStream = Flow.Create<Message>().Conflate((lastMessage, newMessage) => newMessage);
```

There is a more general version of `Conflate` named `ConflateWithSeed` that allows to express more complex aggregations, more similar to a `Aggregate`.

## Dropping broadcast

**Situation:** The default `Broadcast` graph element is properly backpressured, but that means that a slow downstream consumer can hold back the other downstream consumers resulting in lowered throughput. In other words the rate of `Broadcast` is the rate of its slowest downstream consumer. In certain cases it is desirable to allow faster consumers to progress independently of their slower siblings by dropping elements if necessary.

One solution to this problem is to append a `Buffer` element in front of all of the downstream consumers defining a dropping strategy instead of the default `Backpressure`. This allows small temporary rate differences between the different consumers (the buffer smooths out small rate variances), but also allows faster consumers to progress by dropping from the buffer of the slow consumers if necessary.

```
var mysink1 = Sink.Ignore<int>();
var mysink2 = Sink.Ignore<int>();
var mysink3 = Sink.Ignore<int>();

var graph = RunnableGraph.FromGraph(GraphDsl.Create(mysink1, mysink2, mysink3, Tuple.Create,
(builder, sink1, sink2, sink3) =>
{
 var broadcast = builder.Add(new Broadcast<int>(3));

 builder.From(broadcast).Via(Flow.Create<int>().Buffer(10, OverflowStrategy.DropHead)).To(sink1);
 builder.From(broadcast).Via(Flow.Create<int>().Buffer(10, OverflowStrategy.DropHead)).To(sink2);
 builder.From(broadcast).Via(Flow.Create<int>().Buffer(10, OverflowStrategy.DropHead)).To(sink3);

 return ClosedShape.Instance;
}));
```

## Collecting missed ticks

**Situation:** Given a regular (stream) source of ticks, instead of trying to backpressure the producer of the ticks we want to keep a counter of the missed ticks instead and pass it down when possible.

We will use `conflateWithSeed` to solve the problem. The seed version of `conflate` takes two functions:

- A seed function that produces the zero element for the folding process that happens when the upstream is faster than the downstream. In our case the seed function is a constant function that returns 0 since there were no missed ticks at that point.
- A fold function that is invoked when multiple upstream messages needs to be collapsed to an aggregate value due to the insufficient processing rate of the downstream. Our folding function simply increments the currently stored count of the missed ticks so far.

As a result, we have a flow of `Int` where the number represents the missed ticks. A number 0 means that we were able to consume the tick fast enough (i.e. zero means: 1 non-missed tick + 0 missed ticks)

```
var missed = Flow.Create<Tick>()
 .ConflateWithSeed(seed: _ => 0, aggregate: (missedTicks, tick) => missedTicks + 1);
```

## Create a stream processor that repeats the last element seen

**Situation:** Given a producer and consumer, where the rate of neither is known in advance, we want to ensure that none of them is slowing down the other by dropping earlier unconsumed elements from the upstream if necessary, and repeating the last value for the downstream if necessary.

We have two options to implement this feature. In both cases we will use `GraphStage` to build our custom element. In the first version we will use a provided initial value `initial` that will be used to feed the downstream if no upstream element is ready yet. In the `onPush()` handler we just overwrite the `currentValue` variable and immediately relieve

the upstream by calling `pull()`. The downstream `onPull` handler is very similar, we immediately relieve the downstream by emitting `currentValue`.

```
public sealed class HoldWithInitial<T> : GraphStage<FlowShape<T, T>>
{
 private sealed class Logic : GraphStageLogic
 {
 private readonly HoldWithInitial<T> _holder;
 private T _currentValue;

 public Logic(HoldWithInitial<T> holder) : base(holder.Shape)
 {
 _holder = holder;
 _currentValue = holder._initial;

 SetHandler(holder.In, onPush: () =>
 {
 _currentValue = Grab(holder.In);
 Pull(holder.In);
 });

 SetHandler(holder.Out, onPull: () => Push(holder.Out, _currentValue));
 }

 public override void PreStart() => Pull(_holder.In);
 }

 private readonly T _initial;

 public HoldWithInitial(T initial)
 {
 _initial = initial;
 Shape = new FlowShape<T, T>(In, Out);
 }

 public Inlet<T> In { get; } = new Inlet<T>("HoldWithInitial.in");

 public Outlet<T> Out { get; } = new Outlet<T>("HoldWithInitial.out");

 public override FlowShape<T, T> Shape { get; }

 protected override GraphStageLogic CreateLogic(Attributes inheritedAttributes) => new Logic(this);
}
```

While it is relatively simple, the drawback of the first version is that it needs an arbitrary initial element which is not always possible to provide. Hence, we create a second version where the downstream might need to wait in one single case: if the very first element is not yet available.

We introduce a boolean variable `waitingFirstValue` to denote whether the first element has been provided or not (alternatively an `Option` can be used for `currentValue` or if the element type is a value type a null can be used with the same purpose). In the downstream `onPull()` handler the difference from the previous version is that we check if we have received the first value and only emit if we have. This leads to that when the first element comes in we must check if there possibly already was demand from downstream so that we in that case can push the element directly.

```
public sealed class HoldWithWait<T> : GraphStage<FlowShape<T, T>>
{
 private sealed class Logic : GraphStageLogic
 {
 private readonly HoldWithWait<T> _holder;
 private T _currentValue;
 private bool _waitingFirstValue = true;

 public Logic(HoldWithWait<T> holder) : base(holder.Shape)
 {
```

```

 _holder = holder;

 SetHandler(holder.In, onPush: () =>
 {
 _currentValue = Grab(holder.In);
 if (_waitingFirstValue)
 {
 _waitingFirstValue = false;
 if(IsAvailable(holder.Out))
 Push(holder.Out, _currentValue);
 }
 Pull(holder.In);
 });

 SetHandler(holder.Out, onPull: () =>
 {
 if(!_waitingFirstValue)
 Push(holder.Out, _currentValue);
 });
 }

 public override void PreStart() => Pull(_holder.In);
}

public HoldWithWait()
{
 Shape = new FlowShape<T, T>(In, Out);
}

public Inlet<T> In { get; } = new Inlet<T>("HoldWithWait.in");

public Outlet<T> Out { get; } = new Outlet<T>("HoldWithWait.out");

public override FlowShape<T, T> Shape { get; }

protected override GraphStageLogic CreateLogic(Attributes inheritedAttributes) => new Logic(this);
}

```

## Globally limiting the rate of a set of streams

**Situation:** Given a set of independent streams that we cannot merge, we want to globally limit the aggregate throughput of the set of streams.

One possible solution uses a shared actor as the global limiter combined with `SelectAsync` to create a reusable `Flow` that can be plugged into a stream to limit its rate.

As the first step we define an actor that will do the accounting for the global rate limit. The actor maintains a timer, a counter for pending permit tokens and a queue for possibly waiting participants. The actor has an `open` and `closed` state. The actor is in the `open` state while it has still pending permits. Whenever a request for permit arrives as a `WantToPass` message to the actor the number of available permits is decremented and we notify the sender that it can pass by answering with a `MayPass` message. If the amount of permits reaches zero, the actor transitions to the `closed` state. In this state requests are not immediately answered, instead the reference of the sender is added to a queue. Once the timer for replenishing the pending permits fires by sending a `ReplenishTokens` message, we increment the pending permits counter and send a reply to each of the waiting senders. If there are more waiting senders than permits available we will stay in the `closed` state.

```

public sealed class WantToPass
{
 public static readonly WantToPass Instance = new WantToPass();

 private WantToPass() { }
}

```

```

public sealed class MayPass
{
 public static readonly MayPass Instance = new MayPass();

 private MayPass() { }

}

public sealed class ReplenishTokens
{
 public static readonly ReplenishTokens Instance = new ReplenishTokens();

 private ReplenishTokens() { }

}

public class Limiter : ReceiveActor
{
 public static Props Props(int maxAvailableTokens, TimeSpan tokenRefreshPeriod, int tokenRefreshAmount)
 => Akka.Actor.Props.Create(() => new Limiter(maxAvailableTokens, tokenRefreshPeriod, tokenRefreshAmount));

 private readonly int _maxAvailableTokens;
 private readonly int _tokenRefreshAmount;
 private ImmutableList<IActorRef> _waitForQueue;
 private int _permitTokens;
 private readonly ICancelable _replenishTimer;

 public Limiter(int maxAvailableTokens, TimeSpan tokenRefreshPeriod, int tokenRefreshAmount)
 {
 _maxAvailableTokens = maxAvailableTokens;
 _tokenRefreshAmount = tokenRefreshAmount;

 _waitForQueue = ImmutableList.Create<IActorRef>();
 _permitTokens = maxAvailableTokens;
 _replenishTimer = Context.System.Scheduler.ScheduleTellRepeatedlyCancelable(initialDelay: tokenRefreshPeriod,
 interval: tokenRefreshPeriod, receiver: Self, message: ReplenishTokens.Instance, sender: Nobody.Instance);

 Become(Open);
 }

 private void Open(object message)
 {
 message.Match()
 .With<ReplenishTokens>(() =>
 {
 _permitTokens = Math.Min(_permitTokens + _tokenRefreshAmount, _maxAvailableTokens);
 })
 .With<WantToPass>(() =>
 {
 _permitTokens--;
 Sender.Tell(MayPass.Instance);
 if(_permitTokens == 0)
 Become(Closed);
 });
 }

 private void Closed(object message)
 {
 message.Match()
 .With<ReplenishTokens>(() =>
 {
 _permitTokens = Math.Min(_permitTokens + _tokenRefreshAmount, _maxAvailableTokens);
 ReleaseWaiting();
 })
 .With<WantToPass>(() =>
 {
 _waitForQueue
 });
 }
}

```

```

 _waitFor = _waitFor.Add(Sender);
 });
}

private void ReleaseWaiting()
{
 var toBeReleased = _waitFor.GetRange(0, _permitTokens);
 _waitFor = _waitFor.RemoveRange(0, _permitTokens);
 _permitTokens -= toBeReleased.Count;
 toBeReleased.ForEach(s => s.Tell(MayPass.Instance));
 if(_permitTokens > 0)
 Become(Open);
}

protected override void PostStop()
{
 _replenishTimer.Cancel();
 _waitFor.ForEach(s => s.Tell(new Status.Failure(new IllegalStateException("Limiter stopped"))));
}
}

```

To create a Flow that uses this global limiter actor we use the `SelectAsync` function with the combination of the `Ask` pattern. We also define a timeout, so if a reply is not received during the configured maximum wait period the returned task from `Ask` will fail, which will fail the corresponding stream as well.

```

public Flow<T, T, NotUsed> LimitGlobal<T>(IActorRef limiter, TimeSpan maxAllowedWait)
=> Flow.Create<T>().SelectAsync(4, element =>
{
 var limiterTriggerTask = limiter.Ask<T>(WantToPass.Instance, maxAllowedWait);
 return limiterTriggerTask.ContinueWith(t => element);
});

```

[!NOTE] The global actor used for limiting introduces a global bottleneck. You might want to assign a dedicated dispatcher for this actor.

## Working with IO

### Chunking up a stream of ByteStrings into limited size ByteStrings

**Situation:** Given a stream of ByteStrings we want to produce a stream of ByteStrings containing the same bytes in the same sequence, but capping the size of ByteStrings. In other words we want to slice up ByteStrings into smaller chunks if they exceed a size threshold.

This can be achieved with a single `GraphStage`. The main logic of our stage is in `EmitChunk()` which implements the following logic:

- if the buffer is empty, and upstream is not closed we pull for more bytes, if it is closed we complete
- if the buffer is nonEmpty, we split it according to the `chunkSize`. This will give a next chunk that we will emit, and an empty or non-empty remaining buffer.

Both `OnPush()` and `OnPull()` calls `EmitChunk()` the only difference is that the push handler also stores the incoming chunk by appending to the end of the buffer.

```

public class Chunker : GraphStage<FlowShape<ByteString, ByteString>>
{
 private sealed class Logic : GraphStageLogic
 {
 private readonly Chunker _chunker;
 private ByteString _buffer = ByteString.Empty;

```

```

public Logic(Chunker chunker) : base(chunker.Shape)
{
 _chunker = chunker;

 SetHandler(chunker.Out, onPull: () =>
 {
 if (IsClosed(chunker.In))
 EmitChunk();
 else
 Pull(chunker.In);
 });

 SetHandler(chunker.In, onPush: () =>
 {
 var element = Grab(chunker.In);
 _buffer += element;
 EmitChunk();
 }, onUpstreamFinish: () =>
 {
 if (_buffer.IsEmpty)
 CompleteStage();
 // elements left in buffer, keep accepting downstream pulls
 // and push from buffer until buffer is emitted
 });
}

private void EmitChunk()
{
 if (_buffer.IsEmpty)
 {
 if (IsClosed(_chunker.In))
 CompleteStage();
 else
 Pull(_chunker.In);
 }
 else
 {
 var t = _buffer.SplitAt(_chunker._chunkSize);
 var chunk = t.Item1;
 var nextBuffer = t.Item2;

 _buffer = nextBuffer;
 Push(_chunker.Out, chunk);
 }
}

private readonly int _chunkSize;

public Chunker(int chunkSize)
{
 _chunkSize = chunkSize;
 Shape = new FlowShape<ByteString, ByteString>(In, Out);
}

public Inlet<ByteString> In { get; }= new Inlet<ByteString>("Chunker.in");

public Outlet<ByteString> Out { get; } = new Outlet<ByteString>("Chunker.out");

public override FlowShape<ByteString, ByteString> Shape { get; }

protected override GraphStageLogic CreateLogic(Attributes inheritedAttributes) => new Logic(this);
}

var rawBytes = Source.Empty<ByteString>();
var chunkStream = rawBytes.Via(new Chunker(ChunkLimit));

```

## Limit the number of bytes passing through a stream of ByteStrings

**Situation:** Given a stream of ByteStrings we want to fail the stream if more than a given maximum of bytes has been consumed.

This recipe uses a `GraphStage` to implement the desired feature. In the only handler we override, `onPush()` we just update a counter and see if it gets larger than `maximumBytes`. If a violation happens we signal failure, otherwise we forward the chunk we have received.

```
public class ByteLimiter : GraphStage<FlowShape<ByteString, ByteString>>
{
 private sealed class Logic : GraphStageLogic
 {
 private long _count;

 public Logic(ByteLimiter limiter) : base(limiter.Shape)
 {
 SetHandler(limiter.In, onPush: () =>
 {
 var chunk = Grab(limiter.In);
 _count += chunk.Count;
 if (_count > limiter._maximumBytes)
 FailStage(new IllegalStateException("Too much bytes"));
 else
 Push(limiter.Out, chunk);
 });

 SetHandler(limiter.Out, onPull: () => Pull(limiter.In));
 }
 }

 private readonly long _maximumBytes;

 public ByteLimiter(long maximumBytes)
 {
 _maximumBytes = maximumBytes;
 Shape = new FlowShape<ByteString, ByteString>(In, Out);
 }

 public Inlet<ByteString> In { get; } = new Inlet<ByteString>("ByteLimiter.in");

 public Outlet<ByteString> Out { get; } = new Outlet<ByteString>("ByteLimiter.out");

 public override FlowShape<ByteString, ByteString> Shape { get; }

 protected override GraphStageLogic CreateLogic(Attributes inheritedAttributes) => new Logic(this);
}

var limiter = Flow.Create<ByteString>().Via(new ByteLimiter(SizeLimit));
```

## Compact ByteStrings in a stream of ByteStrings

**Situation:** After a long stream of transformations, due to their immutable, structural sharing nature ByteStrings may refer to multiple original ByteString instances unnecessarily retaining memory. As the final step of a transformation chain we want to have clean copies that are no longer referencing the original ByteStrings.

The recipe is a simple use of Select, calling the `Compact()` method of the `ByteString` elements. This does copying of the underlying arrays, so this should be the last element of a long chain if used.

```
var data = Source.Empty<ByteString>();
var compacted = data.Select(b => b.Compact());
```

## Injecting keep-alive messages into a stream of ByteStrings

**Situation:** Given a communication channel expressed as a stream of ByteStrings we want to inject keep-alive messages but only if this does not interfere with normal traffic.

There is a built-in operation that allows to do this directly:

```
var injectKeepAlive = Flow.Create<ByteString>().KeepAlive(TimeSpan.FromSeconds(1), () => keepAliveMessage);
```

# Akka I/O

The I/O extension provides an non-blocking, event driven API that matches the underlying transports mechanism.

## Getting Started

Every I/O Driver has a special actor, called the `manager`, that serves as an entry point for the API. The `manager` for a particular driver is accessible through an extension method on `ActorSystem`. The following example shows how to get a reference to the TCP manager.

```
using Akka.Actor;
using Akka.IO;

...
var system = ActorSystem.Create("example");
var manager = system.Tcp();
```

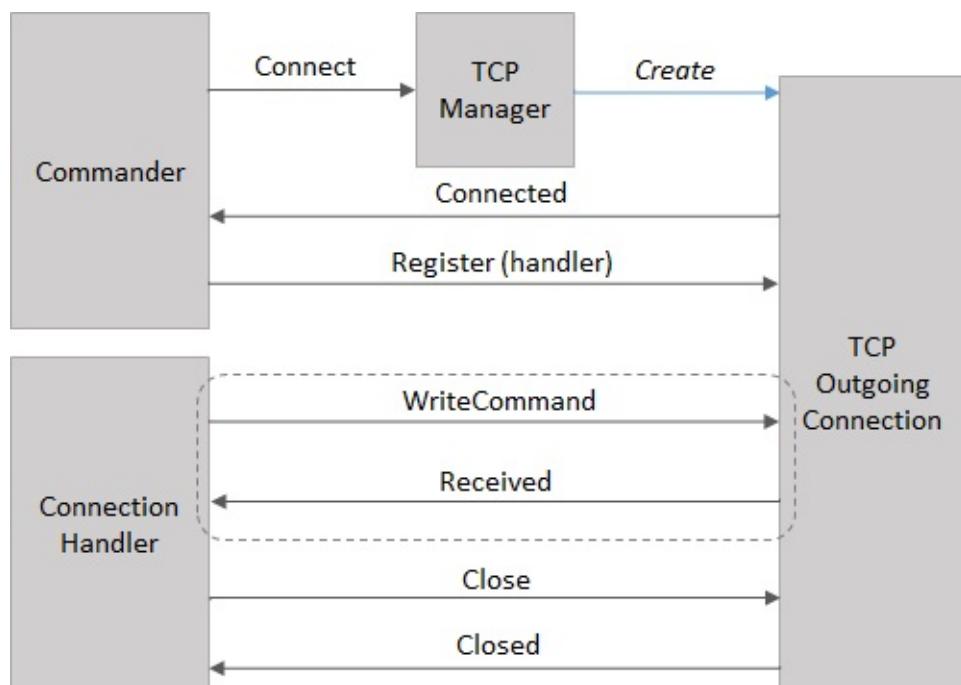
## TCP Driver

### Client Connection

To create a connection an actor sends a `Tcp.Connect` message to the TCP Manager. Once the connection is established the connection actor sends a `Tcp.Connected` message to the `commander`, which registers the `connection handler` by replying with a `Tcp.Register` message.

Once this handshake is completed, the handler and connection communicate with `Tcp.WriteCommand` and `Tcp.Received` messages.

The following diagram illustrate the actors involved in establishing and handling a connection.



The following example shows a simple Telnet client. The client send lines entered in the console to the TCP

The following example shows a simple Telnet client. The client send lines entered in the console to the TCP connection, and write data received from the network to the console.

[!code-csharpMain]

```
public class TelnetClient : UntypedActor
{
 public TelnetClient(string host, int port)
 {
 var endpoint = new DnsEndPoint(host, port);
 Context.System.Tcp().Tell(new Tcp.Connect(endpoint));
 }

 protected override void OnReceive(object message)
 {
 if (message is Tcp.Connected)
 {
 var connected = message as Tcp.Connected;
 Console.WriteLine("Connected to {0}", connected.RemoteAddress);

 // Register self as connection handler
 Sender.Tell(new Tcp.Register(Self));
 ReadConsoleAsync();
 Become(Connected(Sender));
 }
 else if (message is Tcp.CommandFailed)
 {
 Console.WriteLine("Connection failed");
 }
 else Unhandled(message);
 }

 private UntypedReceive Connected(IActorRef connection)
 {
 return message =>
 {
 if (message is Tcp.Received) // data received from network
 {
 var received = message as Tcp.Received;
 Console.WriteLine(Encoding.ASCII.GetString(received.Data.ToArray()));
 }
 else if (message is string) // data received from console
 {
 connection.Tell(Tcp.Write.Create(ByteString.FromString((string)message + "\n")));
 ReadConsoleAsync();
 }
 else if (message is Tcp.PeerClosed)
 {
 Console.WriteLine("Connection closed");
 }
 else Unhandled(message);
 };
 }

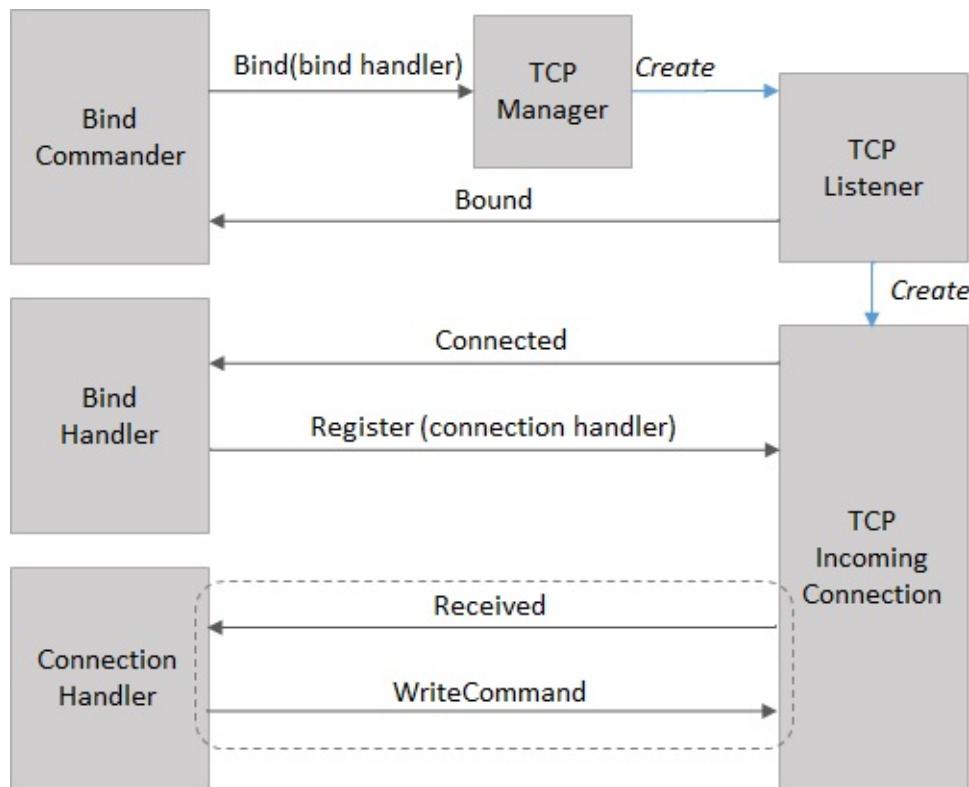
 private void ReadConsoleAsync()
 {
 Task.Factory.StartNew(self => Console.In.ReadLineAsync().PipeTo((ICanTell)self), Self);
 }
}
```

## Server Connection

To accept connections, an actor sends an `Tcp.Bind` message to the TCP manager, passing the `bind handler` in the message. The `bind commander` will receive a `Tcp.Bound` message when the connection is listening.

The `bind handler` will receive a `Tcp.Connected` message for each accepted connection, and needs to register the connection handler by replying with a `Tcp.Register` message. Thereafter it proceeds the same as a client connection.

The following diagram illustrate the actor and messages.



The following code example shows a simple server that echo's data received from the network.

[!code-csharp>Main]

```

public class EchoServer : UntypedActor
{
 public EchoServer(int port)
 {
 Context.System.Tcp().Tell(new Tcp.Bind(Self, new IPEndPoint(IPAddress.Any, port)));
 }

 protected override void OnReceive(object message)
 {
 if (message is Tcp.Bound)
 {
 var bound = message as Tcp.Bound;
 Console.WriteLine("Listening on {0}", bound.LocalAddress);
 }
 else if (message is Tcp.Connected)
 {
 var connection = Context.ActorOf(Props.Create(() => new EchoConnection(Sender)));
 Sender.Tell(new Tcp.Register(connection));
 }
 else Unhandled(message);
 }
}

```

[!code-csharp>Main]

```

public class EchoConnection : UntypedActor
{
}

```

```
private readonly IActorRef _connection;

public EchoConnection(IActorRef connection)
{
 _connection = connection;
}

protected override void OnReceive(object message)
{
 if (message is Tcp.Received)
 {
 var received = message as Tcp.Received;
 if (received.Data[0] == 'x')
 Context.Stop(Self);
 else
 _connection.Tell(Tcp.Write.Create(received.Data));
 }
 else Unhandled(message);
}
```

# Serialization

One of the core concepts of any actor system like Akka.NET is the notion of message passing between actors. Since Akka.NET is network transparent, these actors can be located locally or remotely. As such the system needs a common exchange format to package messages into so that it can send them to receiving actors. In Akka.NET, messages are plain objects and thus easily converted to a byte array. The process of converting objects into byte arrays is known as serialization.

Akka.NET itself uses `Protocol Buffers` to serialize internal messages (i.e. cluster gossip messages). However, the serialization mechanism in Akka.NET allows you to write custom serializers and to define which serializer to use for what. As shown in the examples further down this page, these serializers can be mixed and matched depending on preference or need.

There are many other uses for serialization other than messaging. It's possible to use these serializers for ad-hoc purposes as well.

## Usage

### Configuration

For Akka.NET to know which `Serializer` to use when (de-)serializing objects, two sections need to be defined in the application's configuration. The `akka.actor.serializers` section is where names are associated to implementations of the `Serializer` to use.

```
akka {
 actor {
 serializers {
 json = "Akka.Serialization.NewtonsoftJsonSerializer"
 bytes = "Akka.Serialization.ByteArraySerializer"
 }
 }
}
```

The `akka.actor.serialization-bindings` section is where object types are associated to a `Serializer` by the names defined in the previous section.

```
akka {
 actor {
 serializers {
 json = "Akka.Serialization.NewtonsoftJsonSerializer"
 bytes = "Akka.Serialization.ByteArraySerializer"
 myown = "MySampleProject.MySerializer, MyAssembly"
 }

 serialization-bindings {
 "System.Byte[]" = bytes
 "System.Object" = json
 "MySampleProject.MyOwnSerializable, MyAssembly" = myown
 }
 }
}
```

In case of ambiguity, a message implements several of the configured classes, the most specific configured class will be used, i.e. the one of which all other candidates are superclasses. If this condition cannot be met, because e.g. `ISerializable` and `MyOwnSerializable` both apply and neither is a subtype of the other, a warning will be issued.

Akka.NET provides serializers for POCO's (Plain Old C# Objects) and for `Google.Protobuf.IMessage` by default, so normally you don't need to add configuration for that.

## Verification

Normally, messages sent between local actors (i.e. same CLR) do not undergo serialization. For testing, sometimes, it may be desirable to force serialization on all messages (both remote and local). If you want to do this in order to verify that your messages are serializable you can enable the following config option:

```
akka {
 actor {
 serialize-messages = on
 }
}
```

If you want to verify that your `Props` are serializable you can enable the following config option:

```
akka {
 actor {
 serialize-creators = on
 }
}
```

[!WARNING] We recommend having these config options turned on only when you're running tests. Turning these options on in production is pointless, as it would negatively impact the performance of local message passing without giving any gain.

## Programmatic

As mentioned previously, Akka.NET uses serialization for message passing. However the system is much more robust than that. To programmatically (de-)serialize objects using Akka.NET serialization, a reference to the main serialization class is all that is needed.

```
using Akka.Actor;
using Akka.Serialization;
ActorSystem system = ActorSystem.Create("example");

// Get the Serialization Extension
Serialization serialization = system.Serialization;

// Have something to serialize
string original = "woohoo";

// Find the Serializer for it
Serializer serializer = serialization.FindSerializerFor(original);

// Turn it into bytes
byte[] bytes = serializer.ToBinary(original);

// Turn it back into an object,
// the nulls are for the class manifest and for the classloader
string back = (string)serializer.FromBinary(bytes, original.GetType());

// Voilá!
Assert.AreEqual(original, back);
```

## Customization

Akka.NET makes it extremely easy to create custom serializers to handle a wide variety of scenarios. All serializers in Akka.NET inherit from `Akka.Serialization.Serializer`. So to create a custom serializer, all that is needed is a class that inherits from this base class.

### Creating new Serializers

A custom `Serializer` has to inherit from `Akka.Serialization.Serializer` and can be defined like the following:

[!code-csharpMain]

```
public class MySerializer : Serializer
{
 public MySerializer(ExtendedActorSystem system) : base(system)
 {

 /// <summary>
 /// This is whether <see cref="FromBinary"/> requires a <see cref="Type"/> or not
 /// </summary>
 public override bool IncludeManifest { get; } = false;

 /// <summary>
 /// Completely unique value to identify this implementation of the
 /// <see cref="Serializer"/> used to optimize network traffic
 /// 0 - 40 is reserved by Akka.NET itself
 /// </summary>
 public override int Identifier => 1234567;

 /// <summary>
 /// Serializes the given object to an Array of Bytes
 /// </summary>
 public override byte[] ToBinary(object obj)
 {
 // Put the real code that serializes the object here
 throw new NotImplementedException();
 }

 /// <summary>
 /// Deserializes the given array, using the type hint (if any, see <see cref="IncludeManifest"/> above
 ///)
 /// </summary>
 public override object FromBinary(byte[] bytes, Type type)
 {
 // Put the real code that deserializes here
 throw new NotImplementedException();
 }
}
```

The only thing left to do for this class would be to fill in the serialization logic in the `ToBinary(object)` method and the deserialization logic in the `FromBinary(byte[], Type)`. Afterwards the configuration would need to be updated to reflect which name to bind to and the classes that use this serializer.

### Serializer with String Manifest

The `Serializer` illustrated above supports a class based manifest (type hint). For serialization of data that need to evolve over time the `SerializerWithStringManifest` is recommended instead of `Serializer` because the manifest (type hint) is a `String` instead of a `Type`. That means that the class can be moved/removed and the serializer can still deserialize old data by matching on the String. This is especially useful for `Persistence`.

The manifest string can also encode a version number that can be used in `FromBinary` to deserialize in different ways to migrate old data to new domain objects.

If the data was originally serialized with `Serializer` and in a later version of the system you change to `SerializerWithStringManifest` the manifest string will be the full class name if you used `IncludeManifest=true`, otherwise it will be the empty string.

This is how a `SerializerWithStringManifest` looks like: [!code-csharpMain]

```
public class MyOwnSerializer2 : SerializerWithStringManifest
{
 private const string CustomerManifest = "customer";
 private const string UserManifest = "user";

 public MyOwnSerializer2(ExtendedActorSystem system) : base(system)
 {
 }

 /// <summary>
 /// Completely unique value to identify this implementation of the
 /// <see cref="Serializer"/> used to optimize network traffic
 /// 0 - 40 is reserved by Akka.NET itself
 /// </summary>
 public override int Identifier { get; } = 1234567;

 /// <summary>
 /// The manifest (type hint) that will be provided in the fromBinary method Use <see cref="string.Empty"
 //> if manifest is not needed.
 /// </summary>
 public override string Manifest(object obj)
 {
 switch (obj)
 {
 case Customer _: return CustomerManifest;
 case User _: return UserManifest;
 }

 throw new NotImplementedException();
 }

 /// <summary>
 /// Serializes the given object to an Array of Bytes
 /// </summary>
 public override byte[] ToBinary(object obj)
 {
 // Put the real code that serializes the object here
 switch (obj)
 {
 case Customer c: return Encoding.UTF8.GetBytes(c.Name);
 case User c: return Encoding.UTF8.GetBytes(c.Name);
 }

 throw new NotImplementedException();
 }

 /// <summary>
 /// Deserializes the given array, using the type hint
 /// </summary>
 public override object FromBinary(byte[] bytes, string manifest)
 {
 switch (manifest)
 {
 case CustomerManifest: return new Customer(Encoding.UTF8.GetString(bytes));
 case UserManifest: return new User(Encoding.UTF8.GetString(bytes));
 default: throw new SerializationException();
 }
 }
}
```

```
}
```

You must also bind it to a name in your `Configuration` and then list which classes that should be serialized using it.

It's recommended to throw `SerializationException` in `FromBinary` if the manifest is unknown. This makes it possible to introduce new message types and send them to nodes that don't know about them. This is typically needed when performing rolling upgrades, i.e. running a cluster with mixed versions for while. `SerializationException` is treated as a transient problem in the TCP based remoting layer. The problem will be logged and message is dropped. Other exceptions will tear down the TCP connection because it can be an indication of corrupt bytes from the underlying transport.

## Serializing ActorRefs

All actors are serializable using the default protobuf serializer, but in cases were custom serializers are used, we need to know how to (de-)serialize them properly. In the general case, the local address to be used depends on the type of remote address which shall be the recipient of the serialized information. Use

`Serialization.SerializedActorPath(actorRef)` like this:

```
using Akka.Actor;
using Akka.Serialization;
// Serialize
// (beneath toBinary)
string id = Serialization.SerializedActorPath(theActorRef);

// Then just serialize the identifier however you like

// Deserialize
// (beneath fromBinary)
IActorRef deserializedActorRef = extendedSystem.Provider.ResolveActorRef(id);
// Then just use the IActorRef
```

This assumes that serialization happens in the context of sending a message through the remote transport. There are other uses of serialization, though, e.g. storing actor references outside of an actor application (database, etc.). In this case, it is important to keep in mind that the address part of an actor's path determines how that actor is communicated with. Storing a local actor path might be the right choice if the retrieval happens in the same logical context, but it is not enough when deserializing it on a different network host: for that it would need to include the system's remote transport address. An actor system is not limited to having just one remote transport per se, which makes this question a bit more interesting. To find out the appropriate address to use when sending to `remoteAddr` you can use

`IActorRefProvider.GetExternalAddressFor(remoteAddr)` like this:

.[!code-csharpMain]

```
public class ExternalAddress : ExtensionIdProvider<ExternalAddressExtension>
{
 public override ExternalAddressExtension CreateExtension(ExtendedActorSystem system) =>
 new ExternalAddressExtension(system);
}

public class ExternalAddressExtension : IExtension
{
 private readonly ExtendedActorSystem _system;

 public ExternalAddressExtension(ExtendedActorSystem system)
 {
 _system = system;
 }

 public Address AddressFor(Address remoteAddr)
```

```

 {
 return _system.Provider.GetExternalAddressFor(remoteAddr)
 ?? throw new InvalidOperationException($"cannot send to {remoteAddr}");
 }
}

public class Test
{
 private ExtendedActorSystem ExtendedSystem =>
 ActorSystem.Create("test").AsInstanceOf<ExtendedActorSystem>();

 public string SerializeTo(IActorRef actorRef, Address remote)
 {
 return actorRef.Path.ToSerializationFormatWithAddress(
 new ExternalAddress().Get(ExtendedSystem).AddressFor(remote));
 }
}

```

[!NOTE] `ActorPath.ToSerializationFormatWithAddress` differs from `ToString` if the address does not already have `host` and `port` components, i.e. it only inserts address information for local addresses.

`ToSerializationFormatWithAddress` also adds the unique id of the actor, which will change when the actor is stopped and then created again with the same name. Sending messages to a reference pointing the old actor will not be delivered to the new actor. If you do not want this behavior, e.g. in case of long term storage of the reference, you can use `ToStringWithAddress`, which does not include the unique id.

This requires that you know at least which type of address will be supported by the system which will deserialize the resulting actor reference; if you have no concrete address handy you can create a dummy one for the right protocol using `new Address(protocol, "", "", 0)` (assuming that the actual transport used is as lenient as Akka's `RemoteActorRefProvider`).

## Deep serialization of Actors

The recommended approach to do deep serialization of internal actor state is to use [Akka Persistence](#).

## How to setup Hyperion as default serializer

Starting from Akka.NET v1.5, default Newtonsoft.Json serializer will be replaced in the favor of [Hyperion](#). This change may break compatibility with older actors still using json serializer for remoting or persistence. If it's possible, it's advised to migrate to it already. To do so, first you need to reference hyperion serializer as NuGet package inside your project:

```
Install-Package Akka.Serialization.Hyperion -pre
```

Then bind `hyperion` serializer using following HOCON configuration in your actor system settings:

```

akka {
 actor {
 serializers {
 hyperion = "Akka.Serialization.HyperionSerializer, Akka.Serialization.Hyperion"
 }
 serialization-bindings {
 "System.Object" = hyperion
 }
 }
}

```

# Using the MultiNode TestKit

If you intend to contribute to any of the high availability modules in Akka.NET, such as Akka.Remote and Akka.Cluster, you will need to familiarize yourself with the MultiNode Testkit and the test runner.

The MultiNodeTestkit consists of three binaries within Akka.NET:

- `Akka.MultiNodeTestRunner` - custom Xunit2 test runner for executing the specs.
- `Akka.NodeTestRunner` - test runner for an individual node process launched by `Akka.MultiNodeTestRunner`.
- `Akka.Remote.TestKit` - the MultiNode TestKit itself.

## MultiNode Specs

The multi node specs are different from traditional specs in that they are intended to run across multiple machines in parallel, to simulate multiple logical nodes participating in a network or cluster.

Here's an example of a multi node spec from the Akka.Cluster.Tests project:

```
public class JoinInProgressMultiNodeConfig : MultiNodeConfig
{
 public RoleName First { get; }
 public RoleName Second { get; }

 public JoinInProgressMultiNodeConfig()
 {
 First = Role("first");
 Second = Role("second");

 CommonConfig = MultiNodeLoggingConfig.LoggingConfig.WithFallback(DebugConfig(true))
 .WithFallback(ConfigurationFactory.ParseString(@"
 akka.stdout-loglevel = DEBUG
 akka.cluster {
 # simulate delay in gossip by turning it off
 gossip-interval = 300 s
 failure-detector {
 threshold = 4
 acceptable-heartbeat-pause = 1 second
 }
 }").WithFallback(MultiNodeClusterSpec.ClusterConfig()));

 NodeConfig(new List<RoleName> { First }, new List<Config>
 {
 ConfigurationFactory.ParseString("akka.cluster.roles =[frontend]")
 });
 NodeConfig(new List<RoleName> { Second }, new List<Config>
 {
 ConfigurationFactory.ParseString("akka.cluster.roles =[backend]")
 });
 }
}

public class JoinInProgressSpec : MultiNodeClusterSpec
{
 readonly JoinInProgressMultiNodeConfig _config;

 public JoinInProgressSpec() : this(new JoinInProgressMultiNodeConfig())
 {
 }

 private JoinInProgressSpec(JoinInProgressMultiNodeConfig config) : base(config)
 {
 }
}
```

```

 _config = config;
 }

 [MultiNodeFact]
 public void AClusterNodeMustSendHeartbeatsImmediatelyWhenJoiningToAvoidFalseFailureDetectionDueToDelayedGos
 sip()
 {
 RunOn(StartClusterNode, _config.First);

 EnterBarrier("first-started");

 RunOn(() => Cluster.Join(GetAddress(_config.First)), _config.Second);

 RunOn(() =>
 {
 var until = Deadline.Now + TimeSpan.FromSeconds(5);
 while (!until.IsOverdue)
 {
 Thread.Sleep(200);
 Assert.True(Cluster.FailureDetector.IsAvailable(GetAddress(_config.Second)));
 }
 }, _config.First);

 EnterBarrier("after");
 }
}

```

The `MultiNodeFact` attribute is what's used to distinguish a multi-node spec from a typical spec, so you'll need to decorate your multi-node specs with this attribute.

## Designing a MultiNode Spec

A multi-node spec gives us the ability to do the following:

1. Launch multiple independent processes each running their own `ActorSystem`;
2. Define individual configurations for each node;
3. Run specific commands on individual nodes or groups of nodes;
4. Create barriers that are used to synchronize nodes at specific points within a test; and
5. Test assertions across one or more nodes.

[!NOTE] Everything that's available in the default `Akka.TestKit` is also available inside the `Akka.Remote.TestKit`, but it's worth bearing in mind that `Akka.Remote.TestKit` only works with the `Akka.MultiNodeTestRunner` and uses Xunit 2.0 internally.

### Step 1 - Subclass `MultiNodeConfig`

The first thing to do is define a configuration for each node you want to include in the test, so in order to do that we have to create a test-specific implementation of `MultiNodeConfig`.

```

public class JoinInProgressMultiNodeConfig : MultiNodeConfig
{
 public RoleName First { get; }
 public RoleName Second { get; }

 public JoinInProgressMultiNodeConfig()
 {
 First = Role("first");
 Second = Role("second");

 CommonConfig = MultiNodeLoggingConfig.LoggingConfig.WithFallback(DebugConfig(true))
 .WithFallback(ConfigurationFactory.ParseString(@"
akka.stdout-logger.level = DEBUG

```

```

 akka.cluster {
 # simulate delay in gossip by turning it off
 gossip-interval = 300 s
 failure-detector {
 threshold = 4
 acceptable-heartbeat-pause = 1 second
 }
 }").WithFallback(MultiNodeClusterSpec.ClusterConfig()));

 NodeConfig(new List<RoleName> { First }, new List<Config>
{
 ConfigurationFactory.ParseString("akka.cluster.roles =[frontend]")
});
 NodeConfig(new List<RoleName> { Second }, new List<Config>
{
 ConfigurationFactory.ParseString("akka.cluster.roles =[backend]")
});
 }
}
}

```

In the `JoinInProgressMultiNodeConfig`, we define two `RoleName`s for the two nodes who will be participating in this multi node spec, and then we define a `Config` object and have it set to the `CommonConfig` property, which is shared across all nodes.

Also we configured each node to represent specific role `[frontend,backend]` in the cluster. You can attach arbitrary config instance(s) to individual node or group of nodes by calling `NodeConfig(IEnumerable<RoleName> roles, IEnumerable<Config> configs)`.

## Step 2 - Define a Class for Your Spec, Inherit from `MultiNodeSpec`

The next step is to subclass `MultiNodeSpec` and create a class that each of your individual nodes will run.

```

public class JoinInProgressSpec : MultiNodeClusterSpec
{
 readonly JoinInProgressMultiNodeConfig _config;

 public JoinInProgressSpec() : this(new JoinInProgressMultiNodeConfig())
 {
 }

 private JoinInProgressSpec(JoinInProgressMultiNodeConfig config) : base(config)
 {
 _config = config;
 }
}

```

Decorate each of the independent tests with the `MultiNodeFact` attribute - the `MultiNodeTestRunner` will pick these up once it runs.

You'll need to pass in a copy of your `MultiNodeConfig` object into the constructor of your base class, like this:

```

protected JoinInProgressSpec() : this(new JoinInProgressMultiNodeConfig())
{
}

private JoinInProgressSpec(JoinInProgressMultiNodeConfig config) : base(config)
{
 _config = config;
}

```

The second constructor overload can be used for allowing individual nodes to run with non-shared configurations.

## Step 3 - Write the Actual Test Methods

Decorate each of the independent tests with the `MultiNodeFact` attribute - the `MultiNodeTestRunner` will pick these up once it runs.

```
public class JoinInProgressSpec : MultiNodeClusterSpec
{
 readonly JoinInProgressMultiNodeConfig _config;

 public JoinInProgressSpec() : this(new JoinInProgressMultiNodeConfig())
 {
 }

 private JoinInProgressSpec(JoinInProgressMultiNodeConfig config) : base(config)
 {
 _config = config;
 }

 [MultiNodeFact]
 public void AClusterNodeMustSendHeartbeatsImmediatelyWhenJoiningToAvoidFalseFailureDetectionDueToDelayedGos
 skip()
 {
 RunOn(StartClusterNode, _config.First);

 EnterBarrier("first-started");

 RunOn(() => Cluster.Join(GetAddress(_config.First)), _config.Second);

 RunOn(() =>
 {
 var until = Deadline.Now + TimeSpan.FromSeconds(5);
 while (!until.IsOverdue)
 {
 Thread.Sleep(200);
 Assert.True(Cluster.FailureDetector.IsAvailable(GetAddress(_config.Second)));
 }
 }, _config.First);

 EnterBarrier("after");
 }
}
```

So a couple of special methods to pay attention to....

- `RunOn(Action thunk, params RoleName[] roles)` - this will run a method ONLY on the specified `roles`.
- `EnterBarrier(string barrierName)` - this creates a named barrier and waits for all nodes to synchronize on this barrier before moving onto the next portion of the spec.

There's also the `TestConductor` property, which you can use for doing things like disconnecting a node from the spec:

```
public void AClusterOf3MembersMustNotReachConvergenceWhileAnyNodesAreUnreachable()
{
 var thirdAddress = GetAddress(_config.Third);
 EnterBarrier("before-shutdown");

 RunOn(() =>
 {
 //kill 'third' node
 TestConductor.Exit(_config.Third, 0).Wait();
 MarkNodeAsUnavailable(thirdAddress);
 }, _config.First);

 RunOn(() => Within(TimeSpan.FromSeconds(28), () =>
 {
 //third becomes unreachable
 }));
}
```

```

 AwaitAssert(() => ClusterView.UnreachableMembers.Count.ShouldBe(1));
 AwaitSeenSameState(GetAddress(_config.First), GetAddress(_config.Second));
 // still one unreachable
 ClusterView.UnreachableMembers.Count.ShouldBe(1);
 ClusterView.UnreachableMembers.First().Address.ShouldBe(thirdAddress);
 ClusterView.Members.Count.ShouldBe(3);
 }, _config.First, _config.Second);

 EnterBarrier("after-2");
}

```

If you have multiple phases that need to be executed as part of a test, you can write them like this:

```

[MultiNodeFact]
public void ConvergenceSpecTests()
{
 AClusterOf3MembersMustReachInitialConvergence();
 AClusterOf3MembersMustNotReachConvergenceWhileAnyNodesAreUnreachable();
 AClusterOf3MembersMustNotMoveANewJoiningNodeToUpWhileThereIsNoConvergence();
}

```

This unfortunate design is a byproduct of Xunit and how it recreates the entire test class on each method.

## Running MultiNode Specs

To actually run this specification, we have to execute the `Akka.MultiNodeTestRunner.exe` against the .DLL that contains our specs.

Here's the set of arguments that the MultiNodeTestRunner takes:

```

Akka.MultiNodeTestRunner.exe path-to-dll # path to DLL containing tests
[-Dmultinode.enable-filesystem=(on|off)] # writes test output to disk
[-Dmultinode.spec="fully qualified spec method name"] # execute a specific test method
 # instead of all of them

```

Here's an example of what invoking the test runner might look like if all of our multinodetests were packaged into `Akka.MultiNodeTests.dll`.

```
C:\> Akka.MultiNodeTestRunner.exe "Akka.MultiNodeTests.dll" -Dmultinode.enable-filesystem=on
```

The output of a multi node test run will include the results for each specification for every node participating in the test. Here's a sample of what the final output at the end of a full test run looks like:

```
posh~git ~ akka.net [di-testkit]
erLogging/user/$c/$b to akka://TestRunnerLogging/user/$c/$b was not delivered. 1 dead letters encountered.
[RUNNER][1:25 AM]: Test run completed in [00:03:58.6902300] with 6/6 specs passed.
[RUNNER][1:25 AM]: Results for Akka.MultiNodeTests.LeaderLeavingSpecConfig+ALeaderLeavingMultiNode.ALeaderThatIsLeavingMustBeMovedToLeavingThenExitingThenRemovedThenBeShutDownAndThenANewLeaderShouldBeElected
[INFO][7/11/2015 1:25:54 AM][Thread 0021][akka://TestRunnerLogging/user/$b/$b] Message CompleteTask from akka://TestRunnerLogging/user/$b/$b to akka://TestRunnerLogging/user/$b/$b was not delivered. 2 dead letters encountered.
[RUNNER][1:25 AM]: Start time: 7/11/2015 1:21:56 AM
[RUNNER][1:25 AM]: --> Node 1: PASS [00:00:33.5018670 elapsed]
[RUNNER][1:25 AM]: --> Node 2: PASS [00:00:33.4968654 elapsed]
[RUNNER][1:25 AM]: --> Node 3: PASS [00:00:33.4978657 elapsed]
[RUNNER][1:25 AM]: End time: 7/11/2015 1:22:33 AM
[RUNNER][1:25 AM]: FINAL RESULT: PASS after 00:00:36.6026292.
[RUNNER][1:25 AM]: Results for Akka.MultiNodeTests.JoinSeedNodeMultiNode.JoinSeedNodeSpecs
[RUNNER][1:25 AM]: Start time: 7/11/2015 1:22:33 AM
[RUNNER][1:25 AM]: --> Node 1: PASS [00:00:39.7791706 elapsed]
[RUNNER][1:25 AM]: --> Node 2: PASS [00:00:39.7781703 elapsed]
[RUNNER][1:25 AM]: --> Node 3: PASS [00:00:39.7821704 elapsed]
[RUNNER][1:25 AM]: --> Node 4: PASS [00:00:39.7791706 elapsed]
[RUNNER][1:25 AM]: --> Node 5: PASS [00:00:39.7781703 elapsed]
[RUNNER][1:25 AM]: End time: 7/11/2015 1:23:15 AM
[RUNNER][1:25 AM]: FINAL RESULT: PASS after 00:00:42.7534884.
[RUNNER][1:25 AM]: Results for Akka.MultiNodeTests.JoinInProgressMultiNode.ACusterNodeMustSendHeartbeatsImmediatelyWhenJoiningToAvoidFalseFailureDetectionDueToDelayedGossip
[RUNNER][1:25 AM]: Start time: 7/11/2015 1:23:15 AM
[RUNNER][1:25 AM]: --> Node 1: PASS [00:00:29.0400216 elapsed]
[RUNNER][1:25 AM]: --> Node 2: PASS [00:00:29.0390188 elapsed]
[RUNNER][1:25 AM]: End time: 7/11/2015 1:23:47 AM
[RUNNER][1:25 AM]: FINAL RESULT: PASS after 00:00:32.0594508.
[RUNNER][1:25 AM]: Results for Akka.MultiNodeTests.ConvergenceWithAccrualFailureDetectorMultiNode.ConvergenceSpecTests
[RUNNER][1:25 AM]: Start time: 7/11/2015 1:23:47 AM
[RUNNER][1:25 AM]: --> Node 1: PASS [00:00:38.9396059 elapsed]
[RUNNER][1:25 AM]: --> Node 2: PASS [00:00:38.9396058 elapsed]
[RUNNER][1:25 AM]: --> Node 3: PASS [00:00:38.9456068 elapsed]
[RUNNER][1:25 AM]: --> Node 4: PASS [00:00:38.9386056 elapsed]
[RUNNER][1:25 AM]: End time: 7/11/2015 1:24:29 AM
[RUNNER][1:25 AM]: FINAL RESULT: PASS after 00:00:41.9655347.
[RUNNER][1:25 AM]: Results for Akka.MultiNodeTests.ConvergenceWithFailureDetectorPuppetMultiNode.ConvergenceSpecTests
[RUNNER][1:25 AM]: Start time: 7/11/2015 1:24:29 AM
[RUNNER][1:25 AM]: --> Node 1: PASS [00:00:33.6051543 elapsed]
[RUNNER][1:25 AM]: --> Node 2: PASS [00:00:33.6051543 elapsed]
[RUNNER][1:25 AM]: --> Node 3: PASS [00:00:33.6071565 elapsed]
[RUNNER][1:25 AM]: --> Node 4: PASS [00:00:33.6051548 elapsed]
[RUNNER][1:25 AM]: End time: 7/11/2015 1:25:06 AM
[RUNNER][1:25 AM]: FINAL RESULT: PASS after 00:00:36.5535695.
[RUNNER][1:25 AM]: Results for Akka.MultiNodeTests.ClusterDeathWatchMultiNode.ClusterDeathWatchSpecTests
[RUNNER][1:25 AM]: Start time: 7/11/2015 1:25:06 AM
[RUNNER][1:25 AM]: --> Node 1: PASS [00:00:39.5712322 elapsed]
[RUNNER][1:25 AM]: --> Node 2: PASS [00:00:39.5152290 elapsed]
[RUNNER][1:25 AM]: --> Node 3: PASS [00:00:39.5182297 elapsed]
[RUNNER][1:25 AM]: --> Node 4: PASS [00:00:39.5782317 elapsed]
```

## Akka.Remote Overview

Akka.NET uses the "Home Depot" extensibility model - the base [Akka NuGet package](#) provides all of the capabilities you need to create actors, `IActorRef`s, and pass messages, but the Akka.NET project ships dozens of additional modules which take the capabilities of Akka.NET and extend them to do new things!

**Akka.Remote** is the most powerful of all of these additional packages, as it is what brings the capability to build an `ActorSystem` across multiple processes over a computer network.

## Akka.Remote's Capabilities

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

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

## Distributed by Default

Everything in Akka.NET is designed to work in a distributed setting: all interactions of actors use purely message passing and everything is asynchronous. This effort has been undertaken to ensure that all functions are available equally when running within a single machine or on a cluster of hundreds of machines. The key for enabling this is to go from remote to local by way of optimization instead of trying to go from local to remote by way of generalization. See [this classic paper](#) for a detailed discussion on why the second approach is bound to fail.

## Ways in which Transparency is Broken

What is true of Akka need not be true of the application which uses it, since designing for distributed execution poses some restrictions on what is possible. The most obvious one is that all messages sent over the wire must be serializable. While being a little less obvious this includes closures which are used as actor factories (i.e. within Props) if the actor is to be created on a remote node.

Another consequence is that everything needs to be aware of all interactions being fully asynchronous, which in a computer network might mean that it may take several minutes for a message to reach its recipient (depending on configuration). It also means that the probability for a message to be lost is much higher than within one CLR, where it is close to zero (still: no hard guarantee!).

Message size can also be a concern. While in-process messages are only bound by CLR restrictions, physical memory and operating system, remote transport layer sets the maximum size to 128 kB by default (minimum: 32 kB). If any of the messages sent remotely is larger than that, maximum frame size in the config file has to be changed to

appropriate value:

```
akka {
 dot-netty.tcp {
 # Maximum frame size: 4 MB
 maximum-frame-size = 4000000b
 }
}
```

Messages exceeding the maximum size will be dropped.

You also have to be aware that some protocols (e.g. UDP) might not support arbitrarily large messages.

## How is Remoting Used?

We took the idea of transparency to the limit in that there is nearly no API for the remoting layer of Akka.NET: it is purely driven by configuration. Just write your application according to the principles outlined in the previous sections, then specify remote deployment of actor sub-trees in the configuration file. This way, your application can be scaled out without having to touch the code. The only piece of the API which allows programmatic influence on remote deployment is that Props contain a field which may be set to a specific Deploy instance; this has the same effect as putting an equivalent deployment into the configuration file (if both are given, configuration file wins).

## Peer-to-Peer vs. Client-Server

Akka Remoting is a communication module for connecting actor systems in a peer-to-peer fashion, and it is the foundation for Akka Clustering. The design of remoting is driven by two (related) design decisions:

1. Communication between involved systems is symmetric: if a system A can connect to a system B then system B must also be able to connect to system A independently.
2. The role of the communicating systems are symmetric in regards to connection patterns: there is no system that only accepts connections, and there is no system that only initiates connections. The consequence of these decisions is that it is not possible to safely create pure client-server setups with predefined roles (violates assumption 2) and using setups involving Network Address Translation or Load Balancers (violates assumption 1).

For client-server setups it is better to use HTTP or [Akka I/O](#).

## Use Cases

Akka.Remote is most commonly used in distributed applications that run across the network, some examples include:

1. Client applications (WPF, Windows Forms) with duplex communication requirements with remote servers;
2. Server-to-Server applications;
3. Embedded Akka.NET applications; (like [this RaspberryPi example!](#))
4. and any application that uses Akka.Cluster or any of its modules.

[!NOTE] Akka.Remote largely serves as plumbing for Akka.Cluster and the other "high availability" modules within Akka.NET. The use cases for using Akka.Remote by itself are largely limited to scenarios that don't require the elasticity and fault-tolerance needs that Akka.Cluster fulfills.

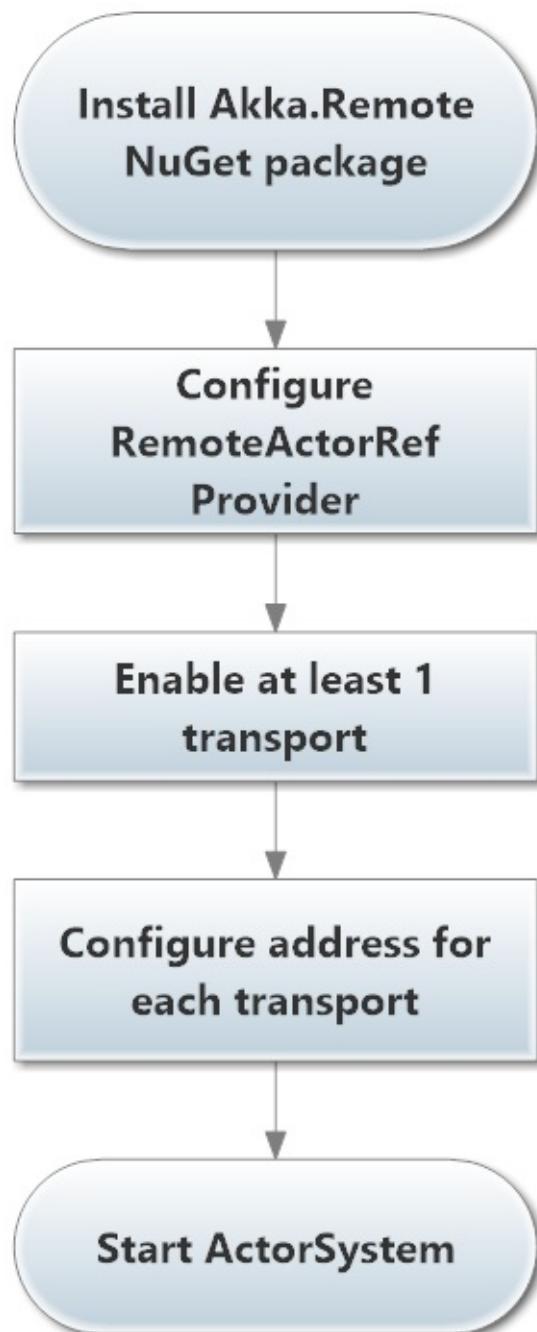
That being said, it's a good idea to understand how Akka.Remote works if you intend to use clustering. So keep reading!

## Marking Points for Scaling Up with Routers

In addition to being able to run different parts of an actor system on different nodes of a cluster, it is also possible to scale up onto more cores by multiplying actor sub-trees which support parallelization (think for example a search engine processing different queries in parallel). The clones can then be routed to in different fashions, e.g. round-robin. The only thing necessary to achieve this is that the developer needs to declare a certain actor as `WithRouter`, then—in its stead—a router actor will be created which will spawn up a configurable number of children of the desired type and route to them in the configured fashion. Once such a router has been declared, its configuration can be freely overridden from the configuration file, including mixing it with the remote deployment of (some of) the children. Read more about this in [Routers](#).

## Enabling Akka.Remote

Enabling Akka.Remote in your own applications is simple:



First you need to install the Akka.Remote NuGet package, which you can do like this:

```
PS> Install-Package Akka.Remote
```

Next, you'll need to enable the `RemoteActorRefProvider` inside [HOCON configuration](#) and bind your transport to an accessible IP address and port combination. Here's an example:

```
akka {
 actor {
 provider = remote
 }

 remote {
 dot-netty.tcp {
 port = 8080
 }
 }
}
```

```

 hostname = localhost
 }
}
}

```

See [Akka Remote Reference Config File](#) for additional information on HOCON settings available in akka remote.

## Addresses, Transports, Endpoints, and Associations

In the above section we mentioned that you have to bind a *transport* to an IP address and port, we did in that in HOCON inside the `dot-netty.tcp` section. Why did we have to do any of that?

Well, let's take a step back to define some key terms you'll need to be familiar with in order to use Akka.Remote:

- **Transport** - a "transport" refers to an actual network transport, such as TCP or UDP. By default Akka.Remote uses a `DotNetty` TCP transport, but you could write your own transport and use that instead of you wish.
- **Address** - this refers to an IP address and port combination, just like any other IP-enabled protocol. You can also use a hostname instead of an IP address, but the hostname must be resolved to an IP address first.
- **Endpoint** - an "endpoint" is a specific address binding for a transport. If I open a TCP transport at `localhost:8080` then I've created an *endpoint* for that transport at that address.
- **Association** - an "association" is a connection between two endpoints, each belonging to a different `ActorSystem`. Must have a valid *outbound* endpoint and a valid *inbound* endpoint in order to create the association.

These terms form the basis for all remote interaction between `ActorSystem` instances, so they're critically important to learn and distinguish.

So in the case of our previous example, `localhost:8080` is the inbound (listening) endpoint for the `DotNetty` TCP transport of the `ActorSystem` we configured.

## How to Form Associations between Remote Systems

So imagine we have the following two actor systems configured to both use the `dot-netty.tcp` Akka.Remote transport:

### Client

```

akka {
 actor {
 provider = remote
 }
 remote {
 dot-netty.tcp {
 port = 0 # bound to a dynamic port assigned by the OS
 hostname = localhost
 }
 }
}

```

### Server

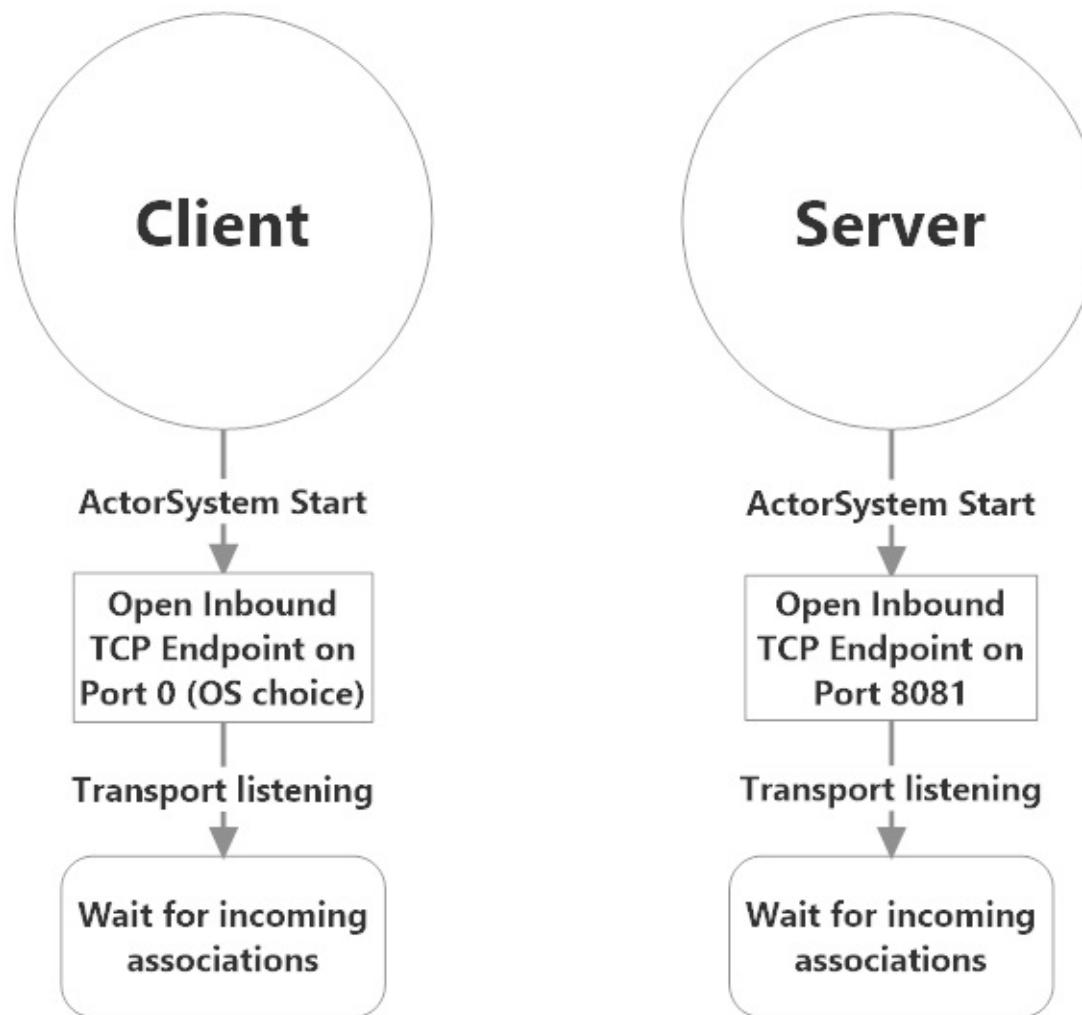
```

akka {
 actor {
 provider = remote
 }
 remote {
 dot-netty.tcp {

```

```
 port = 8081 #bound to a specific port
 hostname = localhost
 }
}
}
```

Here's what the initial state of those two systems would look like upon starting both ActorSystem S.



Both `ActorSystem` instances start, open their transports and bind them to the configured addresses (which creates an *inbound* endpoint for each) and then waits for incoming association attempts from elsewhere.

In order to actually form an association between the client and the server, *one of the nodes has to attempt contact with the other*. Remote associations are formed lazily!

## Addressing a Remote ActorSystem

In order to form an association with a remote `ActorSystem`, we have to have an `Address` for that `ActorSystem`.

All local Akka.NET actors have an `Address` too, as part of their `ActorPath`.

A local `ActorPath` look like this:

## All parts form an "ActorPath"

### Protocol



A remote `ActorPath` looks like this:

## All parts form an "ActorPath"

### Protocol

### Address



Each `ActorPath` consists of four parts:

1. **Protocol** - this defines the protocol used to communicate with this actor. Default local protocol is in-memory message passing.
2. **ActorSystem** - the name of the `ActorSystem` to which this actor belongs.
3. **Address** - refers to the inbound endpoint you can use to communicate with this actor via the protocol. There's a default address for local-only actors and it always get committed from local `ActorPaths`.
4. **Path** - refers to the path of this actor in the hierarchy.

When you want to connect to a remote `ActorSystem`, two important changes occur to the address:

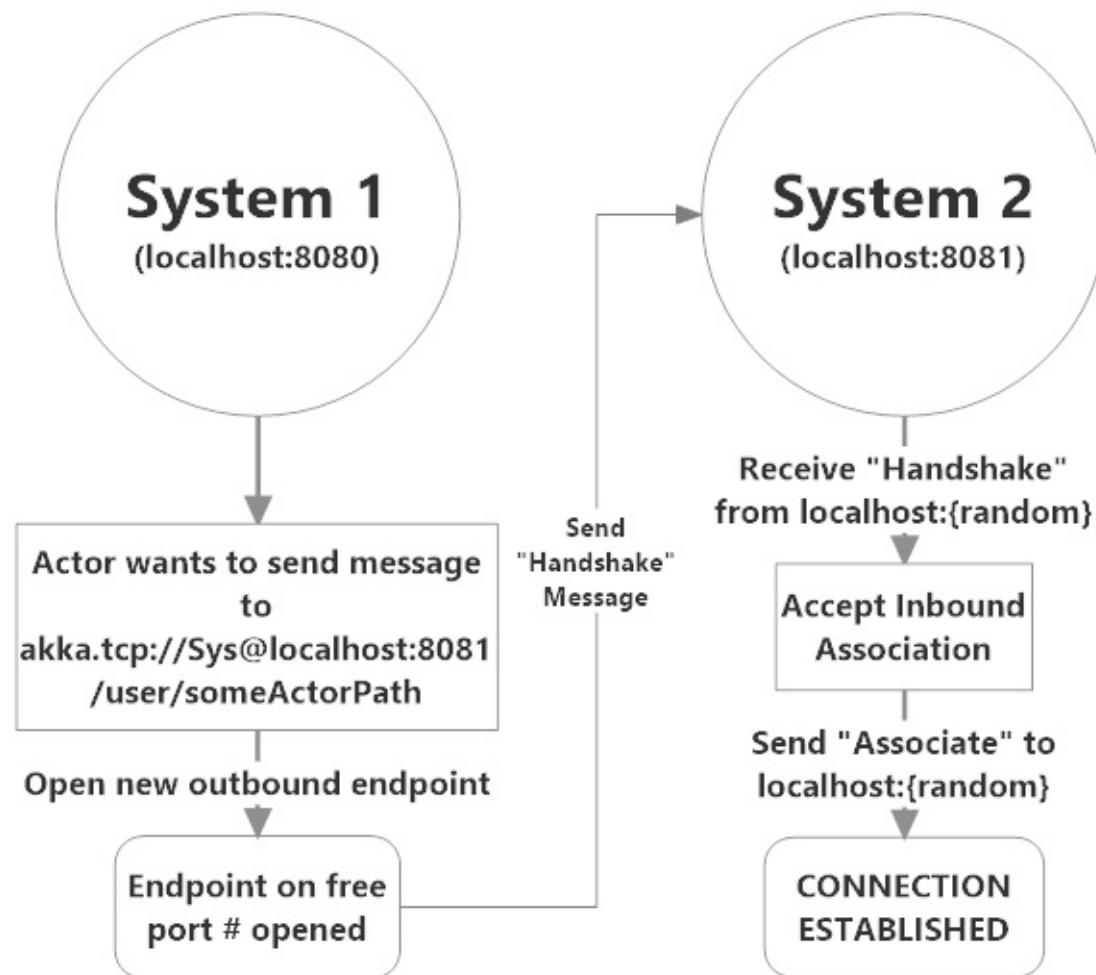
1. **The protocol gets augmented with the protocol of the network transport** - so in this case, since we're using the DotNetty TCP transport the protocol for communicating with all remote actors in our `ActorSystem` changes from `akka://` to `akka.tcp://`. When you deploy an actor remotely or send a message to a remote actor via `ActorSelection`, specifying this protocol is what tells your local `ActorSystem` how to deliver this message to the remote one!
2. **The address gets populated with the inbound endpoint on the transport** - `localhost:9001` in this case. This lets your local system know how to attempt to establish an *outbound endpoint* to the remote `ActorSystem`.

[!NOTE] For more information about addressing in Akka.NET, see [Actor References, Paths and Addresses](#)

Here's how we actually use a remote `Address` to form an association between two remote `ActorSystem` instances.

## The Association Process

This information exposes some of the Akka.Remote internals to you, but it's important to know because without this information it's very difficult to troubleshoot association problems in production - *which you should anticipate as a product of imperfect networks*.



The association process begins when **System 1** has an actor who wants to send a message to an `ActorSelection` belonging to an actor who resides on **System 2**.

The `RemoteActorRefProvider` built into System 1, upon seeing the remote address in the `ActorSelection`, will check to see if a remote connection to System 2 is already open. Since there isn't one, it will open a new outbound endpoint using its TCP transport (which, internally, will create a new TCP socket on a new port - but that's beyond the scope of this course) and send an "handshake" message to System 2.

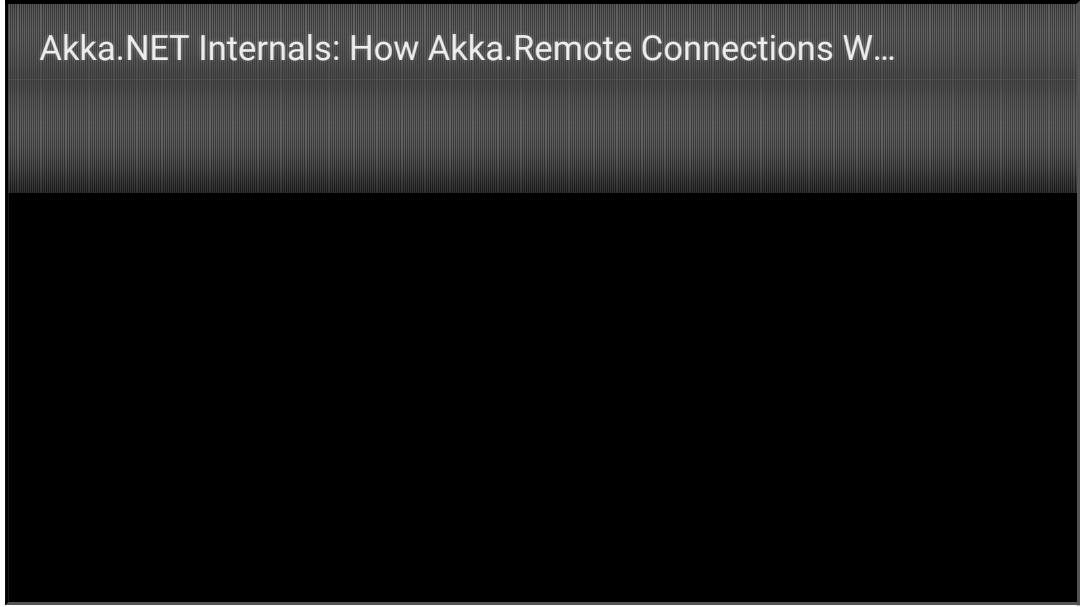
System 2 will receive the handshake, accept the inbound association, and reply with an "Associate" message which will complete the association process.

System 1 will then finally deliver the message contained in the `ActorSelection` to the appropriate actor on System 2.

That's how associations work in a nutshell!

## Internals: How Akka.Remote Associations Work

We have a video that illustrates how this process works - this video was really designed for Akka.NET contributors who work on Akka.Remote, but there's a lot of benefit in understanding it as an end-user of Akka.NET too!



Akka.NET Internals: How Akka.Remote Connections W...

## Additional Resources

- [Akka.NET Internals: How Akka.Remote Connections Work \(Video\)](#)
- [ChatClient Akka.Remote example \(Code Sample\)](#)

# Akka.Remote Transports

In the [Akka.Remote overview](#) we introduced the concept of "transports" for Akka.Remote.

A "transport" refers to an actual network transport, such as TCP or UDP. By default Akka.Remote uses a [DotNetty](#) TCP transport, but you could write your own transport and use that instead of you wish.

In this section we'll expand a bit more on what transports are and how Akka.Remote can support multiple transports simultaneously.

## What are Transports?

Transports in Akka.Remote are abstractions on top of actual network transports, such as TCP and UDP sockets, and in truth transports have pretty simple requirements.

[!NOTE] most of the information below are things you, as an Akka.NET user, do not need to care about 99% of the time. Feel free to skip to the [Akka.Remote's Built-in Transports](#) section.

Transports **do not need to care** about:

- **Serialization** - that's handled by Akka.NET itself;
- **Connection-oriented behavior** - the association process inside Akka.Remote ensures this, even over connectionless transports like UDP;
- **Reliable delivery** - for system messages this is handled by Akka.Remote and for user-defined messages this is taken care of at the application level through something like the [AtLeastOnceDeliveryActor](#) class, part of Akka.Persistence;
- **Handling network failures** - all a transport needs to do is forward that information back up to Akka.Remote.

Transports **do need to care** about:

- **IP addressing and ports** - all Akka.NET endpoints have to be resolved to a reachable IP address and port number;
- **Message delivery** - getting bytes from point A to point B;
- **Message framing** - distinguishing individual messages within a network stream;
- **Disconnect and error reporting** - let Akka.Remote know when a disconnect or transport error occurred;
- **Preserving message order (optional)** - some transports like UDP make no such guarantees, but in general it's recommended that the underlying transport preserve the write-order of messages on the read-side;
- **DNS resolution (optional)** - sockets don't use [DNS](#) out of the box, but in order to make lives of Akka.NET users easy it's recommended that transports support DNS resolution. All built-in Akka.NET transports support DNS resolution by default.

Transports are just plumbing for Akka.Remote - they carry out their tasks and keep things simple and performant.

## Akka.Remote's Built-in Transports

Out of the box Akka.NET uses a socket-based transport built on top of the [DotNetty](#).

[!NOTE] DotNetty supports both TCP and UDP, but currently only TCP support is included within Akka.NET. TCP is what most Akka.Remote and Akka.Cluster users use.

To enable the DotNetty TCP transport, we need to add a section for it inside our `remote` section in [HOCON configuration](#):

```

akka {
 actor {
 provider = remote
 }
 remote {
 dot-netty.tcp {
 port = 8081 #bound to a specific port
 hostname = localhost
 }
 }
}

```

## Using Custom Transports

Akka.Remote supports the ability to load third-party-defined transports at startup time - this is accomplished through defining a transport-specific configuration section within the `akka.remote` section in HOCON.

Let's say, for instance, you found a third party NuGet package that implemented [Google's Quic protocol](#) and wanted to use that transport inside your Akka.Remote application. Here's how you'd configure your application to use it.

```

akka{
 remote {
 enabled-transports = ["akka.remote.google-quic"]
 google-quic {
 transport-class = "Google.Quic.QuicTransport, Akka.Remote.Quic"
 applied-adapters = []
 transport-protocol = quic
 port = 0
 hostname = localhost
 }
 }
}

```

You'd define a custom HOCON section (`akka.remote.google-quic`) and let Akka.Remote know that it should read that section for a transport definition inside `akka.remote.enabled-transports`.

[!NOTE] To implement a custom transport yourself, you need to implement the `Akka.Remote.Transport.Transport` abstract class.

One important thing to note is the `akka.remote.google-quic.transport-protocol` setting - this specifies the address scheme you will use to address remote actors via the Quic protocol.

A remote address for an actor on this transport will look like:

```

akka.quic://MySystem@localhost:9001/user/actor #quic
akka.tcp://MySystem@localhost:9002/user/actor #helios.tcp

```

So the protocol you use in your remote `ActorSelection`s will need to use the string provided inside the `transport-protocol` block in your HOCON configuration.

## Running Multiple Transports Simultaneously

One of the most productive features of Akka.Remote is its ability to allow you to support multiple transports simultaneously within a single `ActorSystem`.

Suppose we created support for an http transport - here's what running both the DotNetty TCP transport and our *imaginary* http transport at the same time would look like in HOCON configuration.

```

akka{
 remote {
 enabled-transports = ["akka.remote.dot-netty.tcp", "akka.remote.magic.http"]
 dot-netty.tcp {
 port = 8081
 hostname = localhost
 }
 magic.http {
 port = 8082 # needs to be on a different port or IP than TCP
 hostname = localhost
 }
 }
}

```

Both TCP and HTTP are enabled in this scenario. But how do I know which transport is being used when I send a message to a `RemoteActorRef`? That's indicated by the protocol scheme used in the `Address` of the remote actor:

```

akka.tcp://MySystem@localhost:8081/user/actor #dot-netty.tcp
akka.udp://MySystem@localhost:8082/user/actor #magic.http

```

So if you want to send a message to a remote actor over HTTP, you'd write something like this:

```

var as = MyActorSystem.ActorSelection("akka.http://RemoteSystem@localhost:8082/user/actor");
as.Tell("remote message!"); //delivers message to remote system, if they're also using this transport

```

## Transport Caveats and Constraints

There are a couple of important caveats to bear in mind with transports in Akka.Remote.

- Each transport must have its own distinct protocol scheme (`transport-protocol` in HOCON) - no two transports can share the same scheme.
- Only one instance of a given transport can be active at a time, for the reason above.

## Separating Physical IP Address from Logical Address

One common DevOps issue that comes up often with Akka.Remote is something along the lines of the following:

"I want to be able to send a message to `machine1.foobar.com` as my `ActorSystem` inbound endpoint, but the socket can't bind to that domain name (it can only bind to an IP.) How do I make it so I can send messages from other remote systems to `machine1.foobar.com`?"

This can be solved through a built-in configuration property that is supported on every Akka.Remote transport, including third-party ones called the `public-hostname` property.

For instance, we can bind a DotNetty TCP transport to listen on all addresses (`0.0.0.0`) so we can accept messages from multiple network interfaces (which is more common in server-side environments than you might think) but still register itself as listening on `machine1.foobar.com`:

```

akka{
 remote {
 enabled-transports = ["akka.remote.dot-netty.tcp", "akka.remote.dot-netty.udp"]
 dot-netty.tcp {
 port = 8081
 hostname = 0.0.0.0 # listen on all interfaces
 public-hostname = "machine1.foobar.com"
 }
 }
}

```

```
}
```

This configuration allows the `ActorSystem`'s DotNetty TCP transport to listen on all interfaces, but when it associates with a remote system it'll tell the remote system that its address is actually `machine1.foobar.com`.

Why is this distinction important? Why do we care about registering an publicly accessible hostname with our `ActorSystem`? Because in the event that other systems need to connect or reconnect to this process, *they need to have a reachable address*.

By default, Akka.Remote assumes that `hostname` is publicly accessible and will use that as the `public-hostname` value. But in the even that it's not AND some of your Akka.NET applications might need to contact this process then you need to set a publicly accessible hostname.

## Additional Resources

- [Message Framing](#) by Stephen Cleary

# Remotely Deploying Actors

Deploying an actor means two things simultaneously:

1. Creating an actor instance with specific, explicitly configured properties and
2. Getting an `IActorRef` to that actor.

With Akka.Remote we get a new exciting detail: the **network location to which an actor is deployed becomes a configuration detail**.

## Remote Deployment Example

That's right - we can *deploy code over the network* with Akka.Remote.

Here's what that concept looks like expressed as Akka.NET code:

### Shared Actor / Message Code

```
/*
 * Create an actor and a message type that gets shared between Deployer and DeployTarget
 * in a common DLL
 */
/// <summary>
/// Actor that just replies the message that it received earlier
/// </summary>
public class EchoActor : ReceiveActor
{
 public EchoActor()
 {
 Receive<Hello>(hello =>
 {
 Console.WriteLine("[{0}]: {1}", Sender, hello.Message);
 Sender.Tell(hello);
 });
 }
}

public class Hello
{
 public Hello(string message)
 {
 Message = message;
 }

 public string Message { get; private set; }
}
```

### DeployTarget (process that gets deployed onto)

```
class Program
{
 static void Main(string[] args)
 {
 using (var system = ActorSystem.Create("DeployTarget", ConfigurationFactory.ParseString(@"
akka {
 actor.provider = remote
 remote {
 dot-netty.tcp {
 port = 8090
 hostname = localhost
 }
 }
}
```

**Deployer** (process that does deploying)

```
class Program
{
 class SayHello { }

 class HelloActor : ReceiveActor
 {
 private IActorRef _remoteActor;
 private int _helloCounter;
 private ICCancelable _helloTask;

 public HelloActor(IActorRef remoteActor)
 {
 _remoteActor = remoteActor;
 Receive<Hello>(hello =>
 {
 Console.WriteLine("Received {1} from {0}", Sender, hello.Message);
 });

 Receive<SayHello>(sayHello =>
 {
 _remoteActor.Tell(new Hello("hello"+_helloCounter++));
 });
 }

 protected override void PreStart()
 {
 _helloTask = Context.System.Scheduler.ScheduleTellRepeatedlyCancelable(TimeSpan.FromSeconds(1),
 TimeSpan.FromSeconds(1), Context.Self, new SayHello(), ActorRefs.NoSender);
 }

 protected override void PostStop()
 {
 _helloTask.Cancel();
 }
 }

 static void Main(string[] args)
 {
 using (var system = ActorSystem.Create("Deployer", ConfigurationFactory.ParseString(@"
 akka {
 actor{
 provider = remote
 deployment {
 /remoteecho {
 remote = ""akka.tcp://DeployTarget@localhost:8090"""
 }
 }
 remote {
 dot-netty.tcp {
 port = 0
 hostname = localhost
 }
 }
 }
 }"))
 }
}
```

```
var remoteAddress = Address.Parse("akka.tcp://DeployTarget@localhost:8090");
//deploy remotely via config
var remoteEcho1 = system.ActorOf(Props.Create(() => new EchoActor()), "remoteecho");

//deploy remotely via code
var remoteEcho2 =
 system.ActorOf(
 Props.Create(() => new EchoActor())
 .WithDeploy(Deploy.None.WithScope(new RemoteScope(remoteAddress))), "coderemoteecho");

system.ActorOf(Props.Create(() => new HelloActor(remoteEcho1)));
system.ActorOf(Props.Create(() => new HelloActor(remoteEcho2)));

Console.ReadKey();
}

}
```

So what happens in this sample? Well first, let's take a look at the output if we run this with just one deployer.

```
D:\Repositories\petabridge\akka-remoting-course\0 - Akka.Remote Intro\Rem... - □ ×

Received hello8 from [akka.tcp://DeployTarget@localhost:8090/remote akka.tcp/Deployer@localhost:19600/user/coderemoteecho]
Received hello8 from [akka.tcp://DeployTarget@localhost:8090/remote akka.tcp/Deployer@localhost:19600/user/remoteecho]
Received hello9 from [akka.tcp://DeployTarget@localhost:8090/remote akka.tcp/Deployer@localhost:19600/user/remoteecho]
Received hello9 from [akka.tcp://DeployTarget@localhost:8090/remote akka.tcp/Deployer@localhost:19600/user/coderemoteecho]
Received hello10 from [akka.tcp://DeployTarget@localhost:8090/remote akka.tcp/Deployer@localhost:19600/user/remoteecho]
Received hello10 from [akka.tcp://DeployTarget@localhost:8090/remote akka.tcp/Deployer@localhost:19600/user/coderemoteecho]
Received hello11 from [akka.tcp://DeployTarget@localhost:8090/remote akka.tcp/Deployer@localhost:19600/user/remoteecho]
Received hello11 from [akka.tcp://DeployTarget@localhost:8090/remote akka.tcp/Deployer@localhost:19600/user/coderemoteecho]
Received hello12 from [akka.tcp://DeployTarget@localhost:8090/remote akka.tcp/Deployer@localhost:19600/user/remoteecho]
Received hello12 from [akka.tcp://DeployTarget@localhost:8090/remote akka.tcp/Deployer@localhost:19600/user/coderemoteecho]
Received hello13 from [akka.tcp://DeployTarget@localhost:8090/remote akka.tcp/Deployer@localhost:19600/user/remoteecho]
Received hello13 from [akka.tcp://DeployTarget@localhost:8090/remote akka.tcp/Deployer@localhost:19600/user/coderemoteecho]
```

```
[[akka.tcp://Deployer@localhost:19600/user/$c]]: hello2
[[akka.tcp://Deployer@localhost:19600/user/$b]]: hello2
[[akka.tcp://Deployer@localhost:19600/user/$b]]: hello3
[[akka.tcp://Deployer@localhost:19600/user/$c]]: hello3
[[akka.tcp://Deployer@localhost:19600/user/$b]]: hello4
[[akka.tcp://Deployer@localhost:19600/user/$c]]: hello4
[[akka.tcp://Deployer@localhost:19600/user/$c]]: hello5
[[akka.tcp://Deployer@localhost:19600/user/$b]]: hello5
[[akka.tcp://Deployer@localhost:19600/user/$c]]: hello6
[[akka.tcp://Deployer@localhost:19600/user/$b]]: hello6
[[akka.tcp://Deployer@localhost:19600/user/$b]]: hello7
[[akka.tcp://Deployer@localhost:19600/user/$c]]: hello7
[[akka.tcp://Deployer@localhost:19600/user/$c]]: hello8
[[akka.tcp://Deployer@localhost:19600/user/$b]]: hello8
[[akka.tcp://Deployer@localhost:19600/user/$b]]: hello9
[[akka.tcp://Deployer@localhost:19600/user/$c]]: hello9
[[akka.tcp://Deployer@localhost:19600/user/$b]]: hello10
[[akka.tcp://Deployer@localhost:19600/user/$c]]: hello10
[[akka.tcp://Deployer@localhost:19600/user/$c]]: hello11
[[akka.tcp://Deployer@localhost:19600/user/$b]]: hello11
[[akka.tcp://Deployer@localhost:19600/user/$b]]: hello12
[[akka.tcp://Deployer@localhost:19600/user/$c]]: hello12
[[akka.tcp://Deployer@localhost:19600/user/$c]]: hello13
[[akka.tcp://Deployer@localhost:19600/user/$b]]: hello13
```

As far as the **DeployTarget** is concerned, it's receiving messages from a `RemoteActorRef` belonging to the **Deployer**, hence why we can see `akka.tcp://Deployer@localhost:19600/user/$c` as the `Sender` address.

But wait! Wait a minute! We didn't actually create any actors in the DeployTarget process!!!!

That's because the Deployer created the actors... but it created them INSIDE DeployTarget's process. Over the network.

## Syntax

In the above example, it's this piece of HOCON configuration:

```
deployment {
 /remoteecho {
 remote = "akka.tcp://DeployTarget@localhost:8090"
 }
}
```

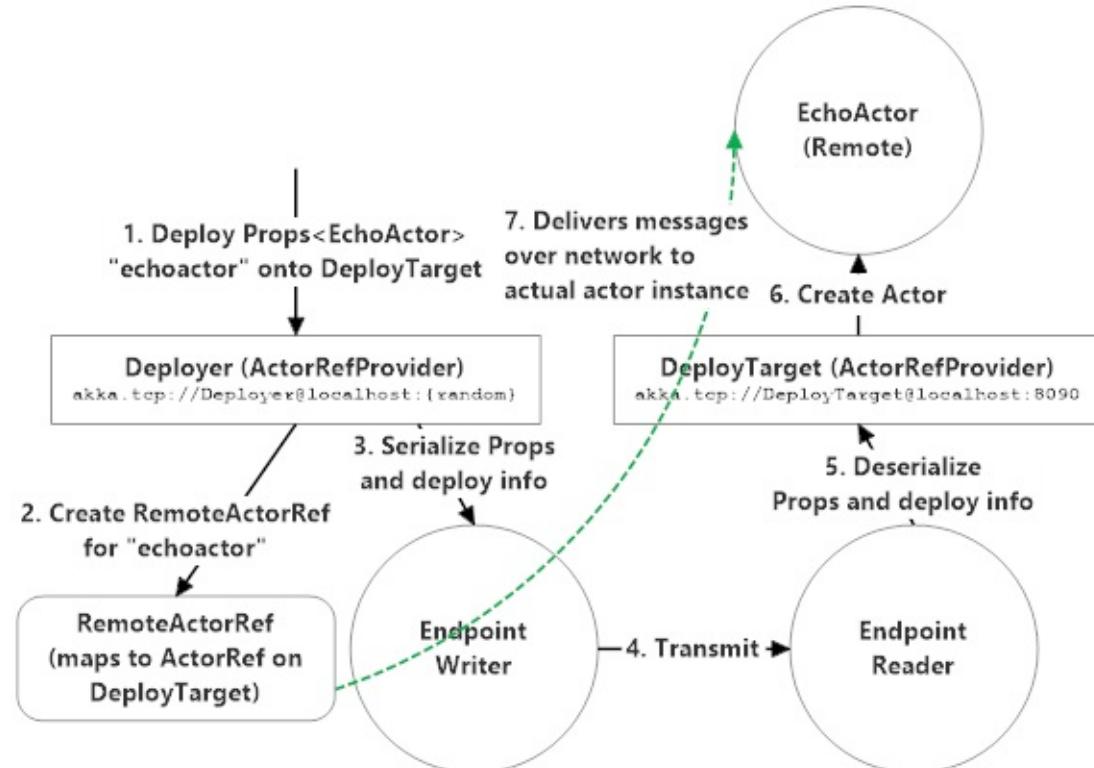
And this piece of C# code:

```
Props.Create(() => new EchoActor()).WithDeploy(Deploy.None.WithScope(new RemoteScope(remoteAddress)))
```

That actually specify to deploy the configured actor onto the specified remote `Address`, which belongs to the DeployTarget in this case.

## How Remote Deployment Actually Works

The process of remote deployment feels magical, but it's actually pretty simple.



Rather than wait for a `RemoteActorRef` to send us a message, we create one in advance - knowing full well that we're going to be able to deploy one onto the remote actor system *if we're able to form an association with the remote system*. If we can't form a remote association with the `DeployTarget` system, then the `RemoteActorRef` we've just created will map to `Deadletters` and nothing will happen.

However, in the happy event that we can form a remote association with `DeployTarget` the following things will happen:

1. A local name on for the actor we're remote deploying will be reserved on Deployer, since all actors must have a unique name;
2. The `Props` for the `EchoActor` will be serialized, including all of the constructor arguments for the `EchoActor` class, along with anything else in the actor's deployment such as router details, dispatcher configuration, and so forth;
3. The serialized `Props` and all of the relevant actor path and name information is transmitted over the network by Deployer's `EndpointWriter` and received by `DeployTarget`'s `EndpointWriter`;
4. The `EndpointWriter` determines that this is a special "deploy remote actor" message, and tells a special system actor (the `RemoteDaemon`) to create a new `EchoActor` instance; and
5. Going forward, all messages sent to the `RemoteActorRef` are automatically sent to the `EchoActor`.

And once all of that is done, we've successfully deployed an `EchoActor` over the network from Deployer to `DeployTarget`.

## Important Things to Know About Remote Actor Deployments

Here are some important things to remember about remote actor deployments:

1. All names for remote actors are determined *by the deploying ActorSystem*. 1000 Deployer instances could all deploy an actor named "echoactor" onto the same `DeployTarget` instance and all 1000 of those operations would be successful. That's because the local actor created on `DeployTarget` has an `ActorPath` that looks like `akka.tcp://DeployTarget@localhost:8090/remote/akka.tcp/Deployer@localhost:19600/user/echoactor/` - the full `Address` of each Deployer `ActorSystem` is appended to the front of the `ActorPath`, thereby guaranteeing that each remote deployed actor name is unique to the Deployer.
2. The C# code that defines the `EchoActor` type and the message types it expects must be present on **both the Deployer and the DeployTarget**, or the deployment fails because the `Props` can't be deserialized.
3. All of the constructor arguments for `EchoActor` and any other remote-deployed actor must be serializable, again, because otherwise it can't be deployed.

## When to Use Remote Deployment

When would you want to remotely deploy a new actor versus just sending a message to a remote actor that already exists somewhere else on the network?

There are two common scenarios for when you would want to deploy an actor remotely:

1. **Work distribution** - you need to be able to push work onto remote machines using a remote or clustered router. This is exactly what the [Cluster.WebCrawler sample](#) does.
2. **Accessing machine-specific resources** - if you wanted to gather `PerformanceCounter` data from an array of remote machines in order to get specific metrics from each machine, remote deployment is an effective tool for accomplishing that.

## Additional Resources



- [Akka.Remote: How to Remotely Deploy Actors \(Video\)](#)
- [Akka.NET remote deployment with F#](#)
- [Akka.NET Remote Deploy \(Code Sample\)](#)

# Detecting and Handling Network Failures with Remote DeathWatch

Programmers often fall victim to the [fallacies of distributed computing](#) when building networked applications, so in this section we want to spend some time address both the realities of network programming AND how to deal with those realities powerfully with Akka.Remote.

Akka.Remote gives us the ability to handle and respond to network failures by extending [Actor life cycle monitoring \(DeathWatch\)](#) to work across the network.

## Syntax

Setting up DeathWatch is done the same way locally as it's done over the network (because, [Location Transparency](#).)

```
public class MyActor : ReceiveActor{
 public MyActor(){
 Receive<Foo>(x => {
 // notify us if the actor who sent
 // this message dies in the future.
 Context.Watch(Sender);
 });

 // message we'll receive anyone we DeathWatch
 // dies, OR if the network terminates
 Receive<Terminated>(t => {
 // ...
 });
 }
}
```

`Context.Watch` is how we establish a DeathWatch for an actor, even one that exists on a remote process!

`Context.Watch` is how we create the DeathWatch subscription, and the `Terminated` message is what we receive in the event the actor we're watching does die.

## When Will You Receive a `Terminated` Message?

Under Akka.Remote, you will receive a `Terminated` message from remote DeathWatch under the following two circumstances:

1. **The remote actor was gracefully terminated**, meaning that the actor or an actor above it on the hierarchy was explicitly terminated. The network is still healthy.
2. **The association between the two remote actor systems has disassociated**; this can mean that there was a network failure or that the remote process crashed. In either case, we assume that the remote actor is dead and receive a `Terminated` message.

## Different Types of `DeathWatch` Notifications

One important piece of data that [the `Terminated` message](#) gives is you some explanation as to WHY the actor you were watching has died.

```
public sealed class Terminated : IAutoReceivedMessage, IPossiblyHarmful
```

```
{
 public Terminated(IActorRef actorRef, bool existenceConfirmed, bool addressTerminated)
 {
 ActorRef = actorRef;
 ExistenceConfirmed = existenceConfirmed;
 AddressTerminated = addressTerminated;
 }

 public IActorRef ActorRef { get; private set; }

 public bool AddressTerminated { get; private set; }

 public bool ExistenceConfirmed { get; private set; }

 public override string ToString()
 {
 return "<Terminated>: " + ActorRef + " - ExistenceConfirmed=" + ExistenceConfirmed;
 }
}
```

The `Terminated.AddressTerminated` property returns **true** if the reason why we're receiving this death watch notification is because the association between our current `ActorSystem` and the remote `ActorSystem` that `Terminated.ActorRef` lived on failed.

# Akka.Remote Security

There are 2 ways you may like to achieve network security when using Akka.Remote:

- Transport Layer Security (introduced with Akka.Remote Version 1.2)
- Virtual Private Networks

## Akka.Remote with TLS (Transport Layer Security)

The release of Akka.NET version 1.2.0 introduces the default [DotNetty](#) transport and the ability to configure [TLS](#) security across Akka.Remote Actor Systems. In order to use TLS, you must first install a valid SSL certificate on all Akka.Remote hosts that you intend to use TLS.

Once you've installed valid SSL certificates, TLS is enabled via your HOCON configuration by setting `enable-ssl = true` and configuring the `ssl` HOCON configuration section like below:

```
akka {
 loglevel = DEBUG
 actor {
 provider = remote
 }
 remote {
 dot-netty.tcp {
 port = 0
 hostname = 127.0.0.1
 enable-ssl = true
 log-transport = true
 ssl {
 suppress-validation = true
 certificate {
 # valid ssl certificate must be installed on both hosts
 path = "<valid certificate path>"
 password = "<certificate password>"

 # flags is optional: defaults to "default-flag-set" key storage flag
 # other available storage flags:
 # exportable | machine-key-set | persist-key-set | user-key-set | user-protected
 flags = ["default-flag-set"]
 }
 }
 }
 }
}
```

## Akka.Remote with Virtual Private Networks

The absolute best practice for securing remote Akka.NET applications today is to make the network around the applications secure - don't use public, open networks! Instead, use a private network to restrict machines that can contact Akka.Remote processes to ones who have your VPN credentials.

Some options for doing this:

- [OpenVPN](#) - for "do it yourself" environments;
- [Azure Virtual Networks](#) - for Windows Azure customers; and
- [Amazon Virtual Private Cloud \(VPC\)](#) - for Amazon Web Services customers.

# Akka.Cluster Overview

## What is a "Cluster"?

A cluster represents a fault-tolerant, elastic, decentralized peer-to-peer network of Akka.NET applications with no single point of failure or bottleneck. Akka.Cluster is the module that gives you the ability to create these applications.

## What Does Akka.Cluster Do?

The best way to begin introducing Akka.Cluster is with brief overview of what it does. Akka.Cluster is the [package](#) that brings clustering support to Akka.NET, and it accomplishes this by adding the following capabilities to Akka.NET:

- Makes it easy to create peer-to-peer networks of Akka.NET applications
- Allows peers to automatically discover new nodes and removed dead ones automatically with no configuration changes
- Allows user-defined classes to subscribe to notifications about changes in the availability of nodes in the cluster
- Introduces the concept of "roles" to distinguish different Akka.NET applications within a cluster; and
- Allows you to create clustered routers, which are an extension of the built-in Akka.NET routers, except that clustered routers automatically adjust their routees list based on node availability.

## Benefits of Akka.Cluster

In short, these are the benefits of a properly designed cluster:

- **Fault-tolerant:** clusters recover from failures (especially network partitions) elegantly.
- **Elastic:** clusters are inherently elastic, and can scale up/down as needed.
- **Decentralized:** it's possible to have multiple **equal** replicas of a given microservice or piece of application state running simultaneously throughout a cluster
- **Peer-to-peer:** New nodes can contact existing peers, be notified about other peers, and fully integrate themselves into the network without any configuration changes.
- **No single point of failure/bottleneck:** multiple nodes are able to service requests, increasing throughput and fault-tolerance.

## How is Clustering Different From Remoting?

Akka.Cluster is a layer of abstraction on top of Akka.Remote, that puts Remoting to use for a specific structure: clusters of applications. Under the hood, Akka.Remote powers Akka.Cluster, so anything you could do with Akka.Remote is also supported by Akka.Cluster.

Generally, Akka.Remote serves as plumbing for Akka.Cluster and other "high availability" modules within Akka.NET. You would generally only use Akka.Remote by itself in scenarios that don't require the elasticity and fault-tolerance needs that Akka.Cluster provides.

Essentially, Akka.Cluster extends Akka.Remote to provide the basis of scalable applications.

## Use Cases

Akka.Cluster lends itself naturally to [high-availability](#) scenarios.

To put it bluntly, you should use clustering in any scenario where you have some or all of the following conditions:

- A sizable traffic load
- Non-trivial to perform
- An expectation of fast response times
- The need for elastic scaling (e.g. bursty workloads)
- A microservices architecture

Some of the use cases where Akka.Cluster emerges as a natural fit are in:

1. Analytics
2. Marketing Automation
3. Multiplayer Games
4. Devices Tracking / Internet of Things
5. Alerting & Monitoring Systems
6. Recommendation Engines
7. Dynamic Pricing
8. ...and many more!

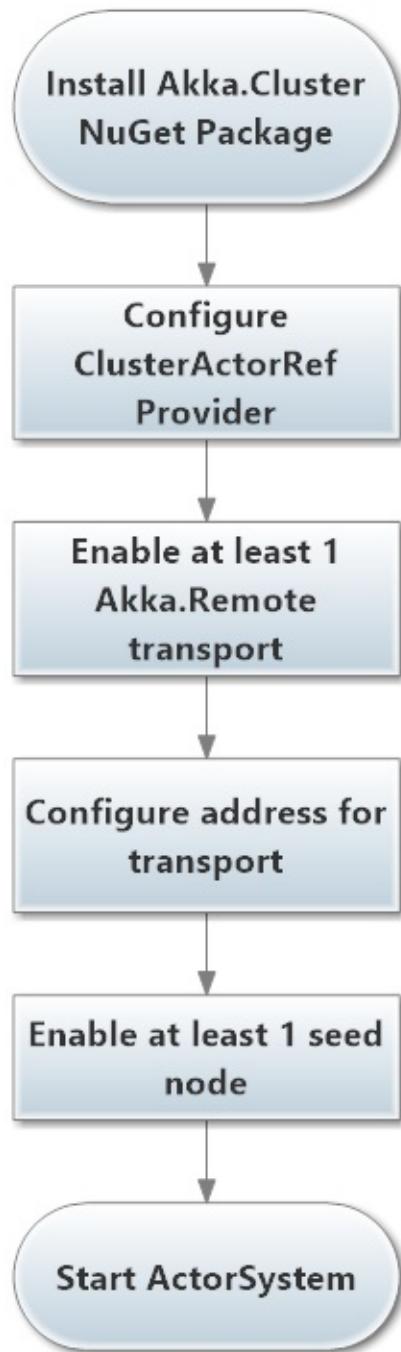
## Key Terms

Akka.Cluster is concept-heavy, so let's clarify a few terms:

- **Node**: a logical member of a cluster.
- **Cluster**: a set of nodes joined through the membership service. Multiple Akka.NET applications can be a part of a single cluster.
- **Gossip**: underlying messages powering the cluster itself.
- **Leader**: single node within the cluster who adds/removes nodes from the cluster.
- **Role**: a named responsibility or application within the cluster. A cluster can have multiple Akka.NET applications in it, each with its own role. A node may exist in 0+ roles simultaneously.
- **Convergence**: when a quorum (simple majority) of gossip messages agree on a change in state of a cluster member.

## Enabling Akka.Cluster

Now that we've gone over some of the concepts and distributed programming concerns behind Akka.Cluster's design, let's focus on how to actually use it inside our own Akka.NET applications.



The first step towards using Akka.Cluster is to install the [Akka.Cluster NuGet package](#), which you can do inside the Package Manager Console in Visual Studio:

```
PM> Install-Package Akka.Cluster
```

Once you've installed Akka.Cluster, we need to update our HOCON configuration to turn on the `ClusterActorRefProvider`, configure an Akka.Remote transport, and enable at least 1 seed node.

[!NOTE] Akka.Cluster depends on Akka.Remote.

## Seed Node Configuration

```
akka {
```

```

actor.provider = cluster
remote {
 dot-netty.tcp {
 port = 8081
 hostname = localhost
 }
}
cluster {
 seed-nodes = ["akka.tcp://ClusterSystem@localhost:8081"]
}
}

```

In this instance, we're configuring this node to act as a seed node to the cluster, so it uses *its own Akka.NET Address* inside the `cluster.seed-nodes` property.

You can, and should, specify multiple seed nodes inside this field - and seed nodes should refer to themselves.

[!NOTE] if you're using dedicated seed nodes (such as [Lighthouse](#)), you should run at least 2 or 3. If you only have one seed node and that machine crashes, the cluster will continue operating but no new members can join the cluster!

## Non-Seed Node Configuration

```

akka {
 actor.provider = cluster
 remote {
 dot-netty.tcp {
 port = 0 #let os pick random port
 hostname = localhost
 }
 }
 cluster {
 seed-nodes = ["akka.tcp://ClusterSystem@localhost:8081"]
 }
}

```

In this case, we've created a non-seed node - it binds its Akka.Remote transport to a random port assigned by the operating system, but it connects to the seed node we assigned in the previous section.

[!NOTE] All nodes in an Akka.NET cluster must have the same `ActorSystem` name.

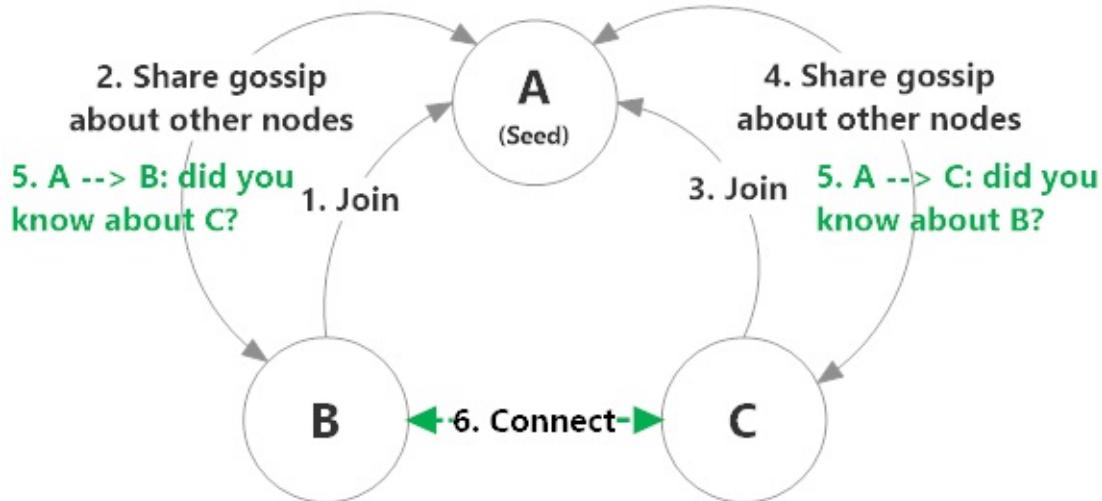
This is important! Even if you're running multiple separate Akka.NET applications inside a single Akka.NET cluster, they must all share the same `ActorSystem` name - otherwise they will not be permitted to join the cluster.

## Cluster Gossip

*This is the most important concept within Akka.Cluster.* This is how nodes are able to join and leave clusters without any configuration changes.

"[Gossip](#)" is the ongoing flow of messages that are passed between nodes in a cluster, updating cluster members of the state of each member of the cluster.

When a node wants to join a cluster, it must first contact one of its configured seed nodes. Once the node has been able to connect to one seed node, it will begin receiving gossip messages containing information about other members of the cluster.



So in the example above:

1. **B** contacts its configured seed node **A** and requests to join the cluster.
2. **A** marks **B** as up and begins to share gossip information with **B** about other nodes in the cluster, but there aren't any other nodes connected at the moment.
3. **C** contacts **A** and requests to join the cluster.
4. **A** welcomes **C** to the cluster and begins sharing gossip information with node **C**.
5. **B** and **C** are both notified about each other by node **A**.
6. **B** and **C** connect to each other and establish communication.

Gossip messages will occur regularly over time whenever there is any change in the status of a member of the cluster, such as when a node joins the cluster, leaves the cluster, becomes unreachable by other nodes, etc.

Generally, you will not interact with gossip messages at the application level. But you do need to be aware of them and know that they power the cluster. To learn more about gossip and event types, see "[Working With Cluster Gossip](#)."

## Nodes

A node is a logical member of a cluster. A node is defined by the address at which it is reachable (hostname:port tuple). Because of this, more than one node can exist simultaneously on a given machine.

### Seed Nodes

A seed node is a well-known contact point that a new node must contact in order to join the cluster. Seed nodes function as the service-discovery mechanism of Akka.Cluster.

[!NOTE] [Lighthouse](#) is a pre-built, dedicated seed node tool that you can use. It's extremely lightweight and only needs to be upgraded when Akka.Cluster itself is upgraded. If you're hosted on a platform like Azure or AWS, you can also tap into the platform-specific APIs to accomplish the same effect.

## How a Cluster Forms

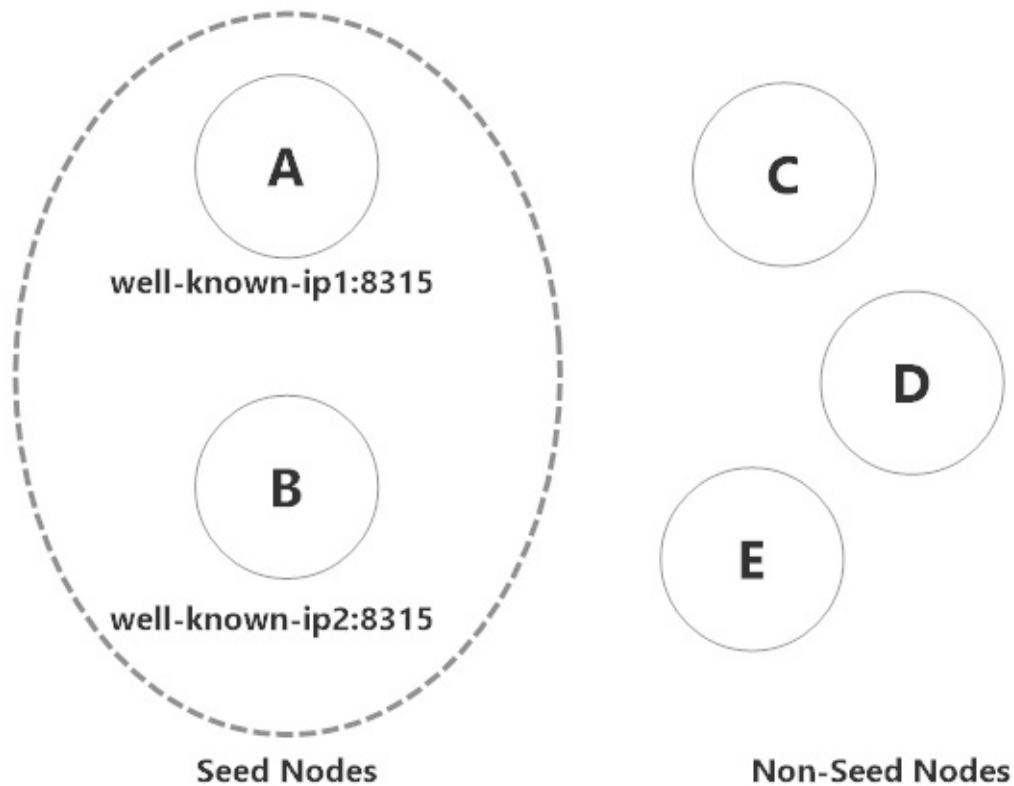
This is what the process of a node joining the cluster looks like:

Clusters initially consist of two distinct parts:

1. **Seed nodes** - nodes that reside at well-known locations on the network and
2. **Non-seed nodes** - nodes whose initial locations are unknown, and these nodes contact seed nodes in order to form the cluster.

We can picture the initial state of a 5-node Akka.NET cluster to look like this:

## Initial Cluster State (Deploying 5 Nodes)

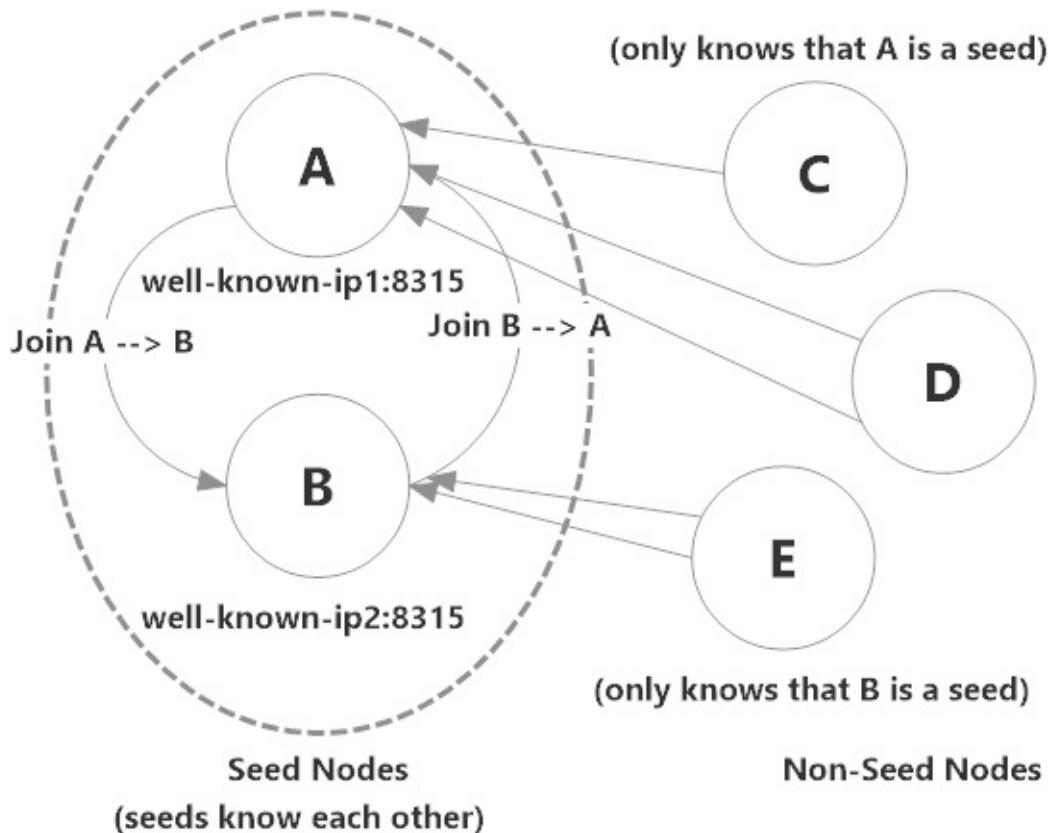


**A** and **B** are both seed nodes, and listen on IP address / port combinations that are baked into the configurations of **C**, **D**, and **E**.

**A** and **B** also know *each other's locations*, so they can communicate with each other initially also.

[!NOTE] A special rule applies the first time a cluster forms: the first seed node declared in a `akka.cluster.seed-nodes` list *must be up*. Otherwise the cluster will not form. This is designed to prevent a split brain from forming the first time a cluster launches. And in general, as a best practice: **always use an identical seed node list on every node, including the seed nodes**. This will give you the most consistent behavior and results.

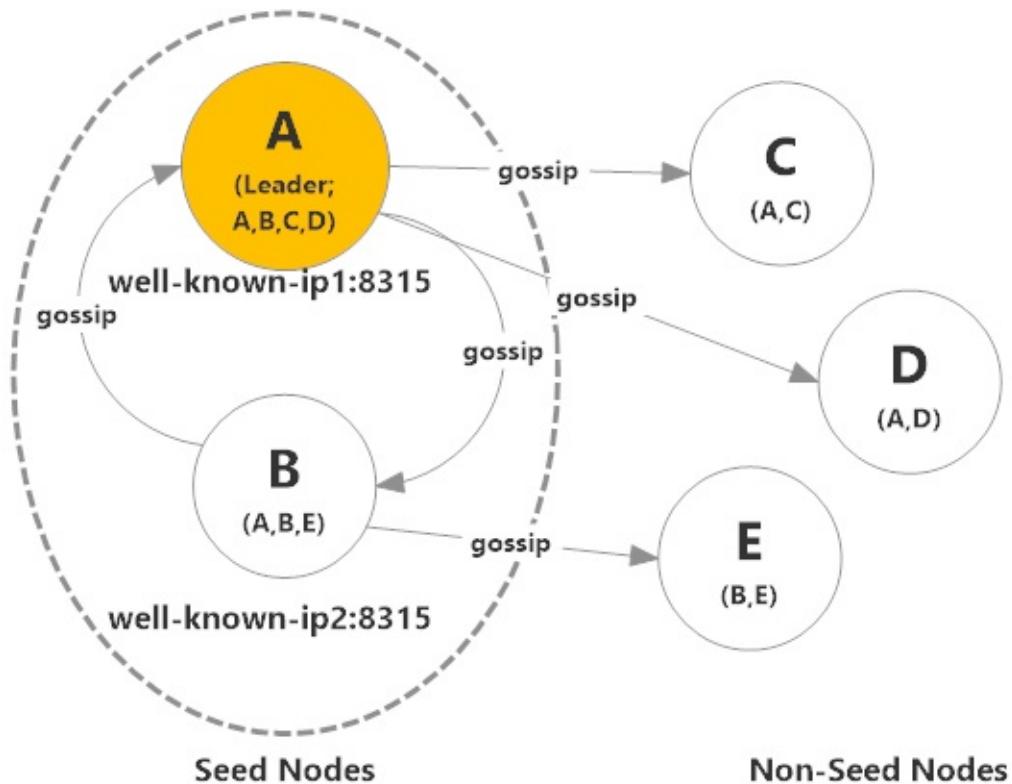
## State 1 - Joining the Cluster (Everyone attempts to Join Seed Nodes)



All nodes initially attempt to connect to a seed node - in this instance the nodes are configured in the following way:

- **E** knows how to contact **B**;
- **C** and **D** know how to contact **A**; and
- **A** and **B** know how to contact each other.

## State 2 - Leader Elected, Marking Nodes Up (Gossip Begins)

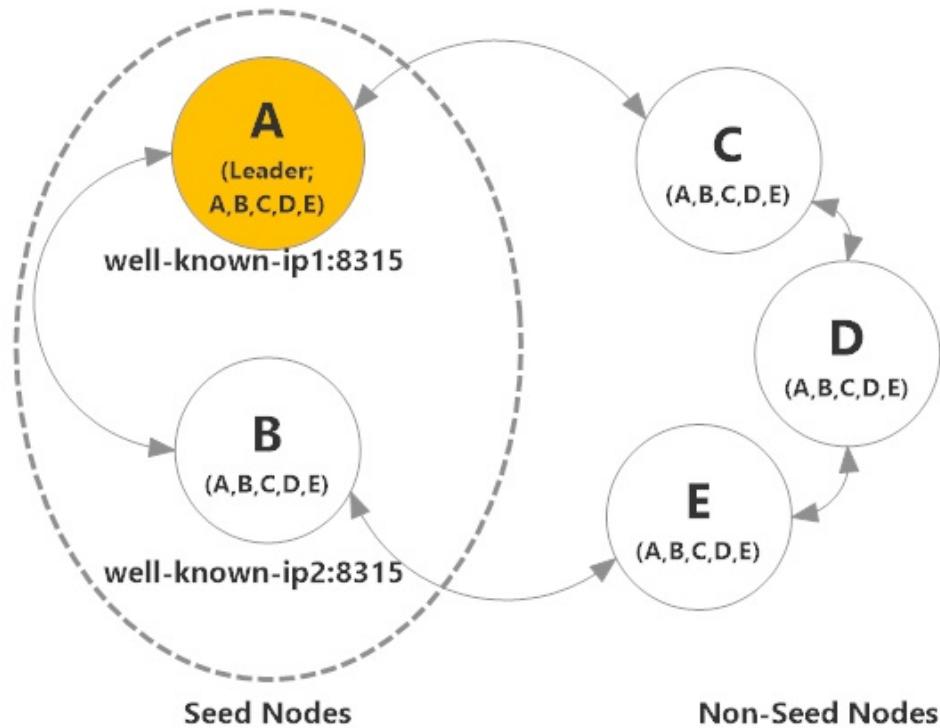


During the process of initial contact within the cluster, a leader will be elected. In this case node **A** is elected to be the leader of the cluster.

**A** will begin marking nodes as up, beginning with the nodes it knows about: **A, B, C, and D**. **A** does not know about node **E** yet, so it's not marked as up.

Gossip information about the membership of the cluster will begin to spread to all nodes, and the nodes will all begin to connect to each other to form a mesh network.

## State 3 - Gossip Spreads, Ring is Formed (Communication Established between Non-Seeds)



After the gossip has had a chance to propagate across all nodes and the leader has marked everyone as up, every node will be connected to every other node and the cluster will have formed. Every node can now participate in any user-defined cluster operations.

### Leader Election

The cluster leader is chosen by a leader election algorithm that randomly picks a leader from the available set of nodes when the cluster forms. Usually, the leader is one of the seed nodes.

### Cluster vs. Role Leader

Each role within the cluster also has a leader, just for that role. Its primary responsibility is enforcing a minimum number of "up" members within the role (if specified in the [cluster config](#)).

### Reachability

Nodes send each other [heartbeats](#) on an ongoing basis. If a node misses enough heartbeats, this will trigger [unreachable](#) gossip messages from its peers. The leader will wait for the node to either become reachable again, restart or get downed. Until that happens, the cluster is not in a consistent state and the leader indicates that it is unable to perform its duties. If the gossip from a quorum of cluster nodes agree that the node is unreachable ("convergence"), the leader will mark it as down and begin removing the node from the cluster. You can control how long the cluster waits for unreachable nodes through the `auto-down-unreachable-after` setting.

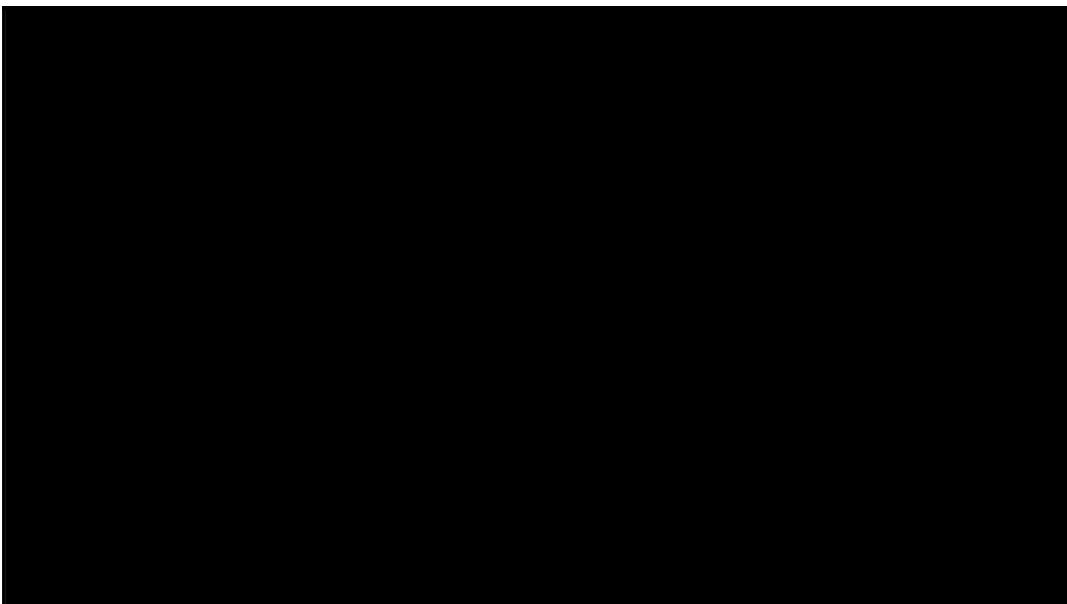
When marked as unreachable, the node can restart and join the cluster again, however the association will only be formed if that node is identified as the same node that became unreachable. If you use dynamic addressing (port 0), starting a node again might result in a different port being assigned upon restart. The result of that is that the cluster remains in an inconsistent state, waiting to the unreachable node to either become reachable or get downed.

A node might also exit the cluster gracefully, preventing it from being marked as unreachable in the first place. Akka.net uses IDowningProvider to take the nodes through all the stages of existing the cluster. Starting in Akka.NET 1.2 [CoordinatedShutdown](#) was introduced allowing the user to easily invoke that mechanism.

## Location Transparency

[Location transparency](#) is the underlying principle powering all of Akka.Remote and Akka.Cluster. The key point is that in a cluster, it's entirely possible that the actors you interface with to do work can be living on any node in the cluster... and you don't have to worry about which one.

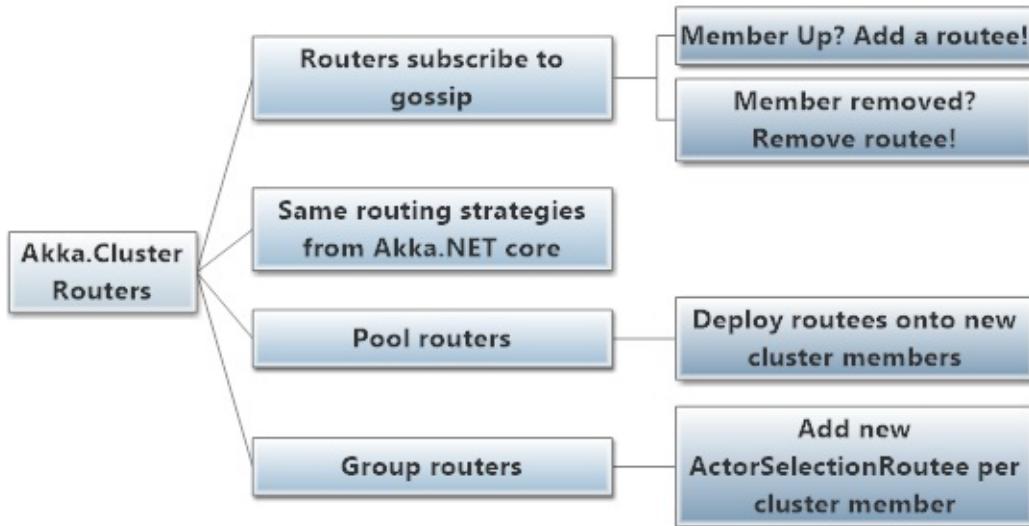
## Additional Resources



- [How to Create Scalable Clustered Akka.NET Apps Using Akka.Cluster](#)
- [Video: Introduction to Akka.Cluster](#)
- [Gossip Protocol](#)
- [High-availability scenarios](#)
- [Microservices](#)

# Akka.Cluster Routing

`Akka.Cluster` extends the capabilities of both `Pool` and `Group` routers to work across entire clusters of Akka.NET applications, and can automatically add or remove routees as new nodes join and exit the cluster.



## How Routers Use Cluster Gossip

How the gossip is used depends on the type of the router. Clustered `Pool` routers will automatically [remote-deploy](#) routees onto nodes they discover as a result of changes in cluster membership. `Group` routers will add new `ActorSelectionRoutee`s to their routees list instead.

[INOTE] this section refers to gossip events, such as `ClusterEvent.MemberUp`, which are [covered here](#).

Clustered routers subscribe to gossip messages from the `cluster` object (which [you can also do in a user-defined actor](#)), and they use the information they dynamically receive from the cluster to add or remove routees on the fly.

A `ClusterEvent.MemberUp` message will cause the cluster to add a new routee (although this depends on some of the router configuration options) and a `ClusterEvent.MemberRemoved` will cause the clustered router to remove any routees who were on the affected node.

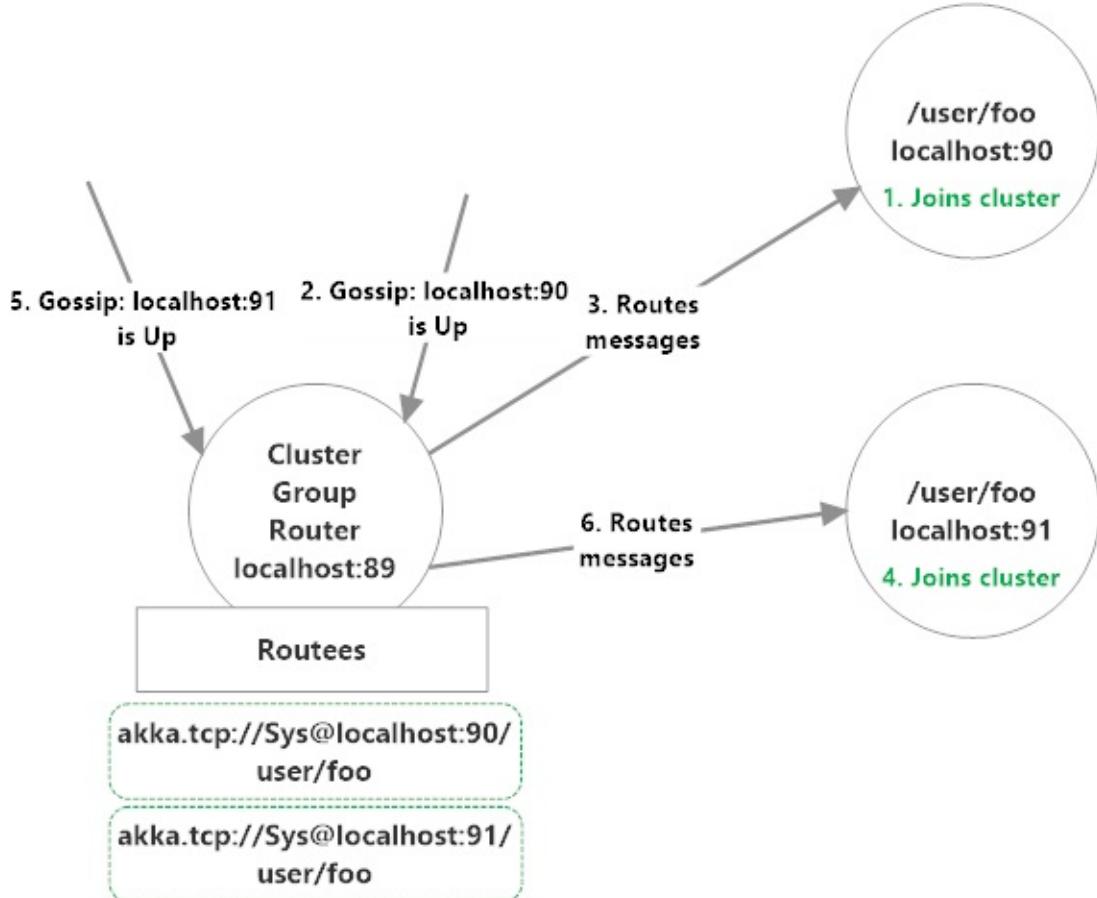
## Cluster Routing Strategies

All of the routing strategies that are available in Akka.NET core are also available in Akka.Cluster - `RoundRobin` and `ConsistentHash` work just as well across the network as they do locally.

## Types of Clustered Routers

### Clustered Group Routers

The first type of router we're going to look at is clustered `Group` routers.



So what's happening in the diagram above? In this setup, we have three distinct nodes:

1. `localhost:89`, who's already **Up**, and is the node which has a clustered `Group` router running on it;
2. `localhost:90`, who's **not Up** yet but will be routed to; and
3. `localhost:91`, who's **not Up** yet and will also be routed to.

We're going to route messages from `localhost:89` to all of the actors who live at path `/user/foo` on each of the new nodes who join. So here's the sequence of events that occurs:

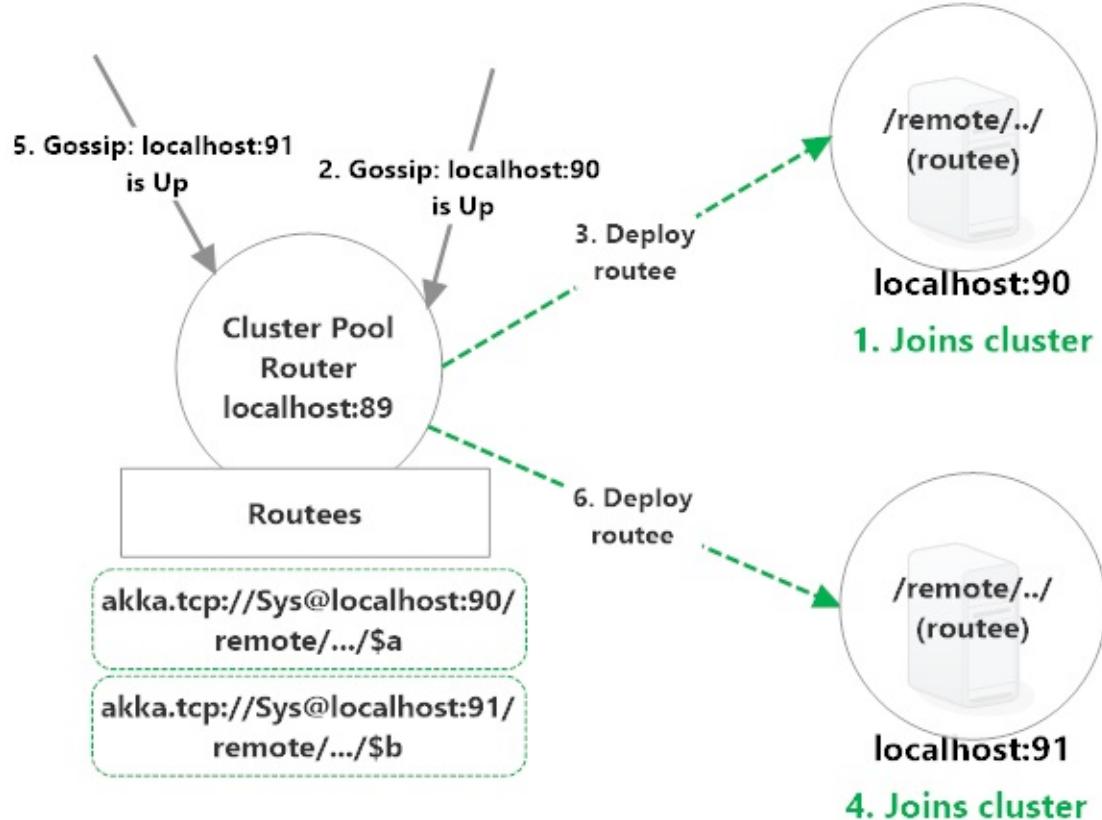
1. `localhost:90` joins the cluster and a `ClusterEvent.MemberUp` message gets generated on each member.
2. `Group` router on `localhost:89` receives the `ClusterEvent.MemberUp` message;
3. `Group` router adds `akka.tcp://Sys@localhost:90/user/foo` as a routee and begins routing messages to it over the network;
4. `localhost:91` joins the cluster and a `ClusterEvent.MemberUp` message gets generated on each member.
5. `Group` router on `localhost:89` receives the `ClusterEvent.MemberUp` message;
6. `Group` router adds `akka.tcp://Sys@localhost:91/user/foo` as a routee and begins routing messages to it over the network.

If either `localhost:90` or `localhost:91` was removed from the cluster at some point in the future, their routee would be removed from the `Group` router running on `localhost:89`.

## Clustered Pool Routers

Clustered `Pool` routers differ from `Group` routers in that they deploy their routees remotely onto their target nodes, versus routing messages to pre-defined actor paths that may or may not exist on the remote machines.

Here's a `Pool` scenario equivalent to the `Group` one we showed earlier:



This is a virtually identical setup as the `Group` router:

1. **localhost:89**, who's already **Up**, and is the node which has a clustered `Pool` router running on it;
2. **localhost:90**, who's **not Up** yet but will be deployed onto and routed to; and
3. **localhost:91**, who's **not Up** yet and will also be deployed onto and routed to.

**localhost:89** is going to deploy routees onto **localhost:90** and **localhost:91** and subsequently route messages to them.

Here's how that occurs:

1. **localhost:90** joins the cluster and a `ClusterEvent.MemberUp` message gets generated on each member;
2. `Pool` router on **localhost:89** receives the `ClusterEvent.MemberUp` message;
3. `Pool` router REMOTE DEPLOYS `akka.tcp://Sys@localhost:90/remote akka/tcp/localhost:89/$a` as a routee and begins routing messages to it over the network;
4. **localhost:91** joins the cluster and a `ClusterEvent.MemberUp` message gets generated on each member;
5. `Pool` router on **localhost:89** receives the `ClusterEvent.MemberUp` message;
6. `Pool` router REMOTE DEPLOYS `akka.tcp://Sys@localhost:91/remote akka/tcp/localhost:89/$b` as a routee and begins routing messages to it over the network.

If either **localhost:90** or **localhost:91** dies following this setup the remote deployed routee will be killed and the `Pool` router will be notified via `Deathwatch`. If there's capacity available (we'll explain in a moment,) the `Pool` router will replace the dead routee by deploying a new routee onto another qualified node.

## Cluster Router Config

Here are the essential options you will use to configure cluster-aware routers.

### All Clustered Routers

- `enabled` : this must be set to `on` in order for this to be a clustered router.
- `allow-local-routees` : determines if routee actors are allowed to be located on the same node as router actor, or only on remote nodes. Setting this to `off` means that all the routees for this router will exist on different nodes in the cluster.
- `nr-of-instances` : this is the maximum number of total routees that this router will route to.
- `use-role` : depends on whether router is a `Pool` or `Group` router

## Clustered Group Router Config

- `nr-of-instances` : this is the maximum number of total routees that this router will route to.
- `max-nr-of-instances-per-node` : this does not apply to `Group` routers.
- `routees.paths` : the comma-separated path(s) of the routees on each node in the cluster.
  - This setting can use what's called a *wildcard* path, meaning we don't care about the name of the actor in the `*` position. e.g. [in the Webcrawler sample](#), as long as the actor being deployed is named `coordinators` and the grandparent is named `api`, then this deployment configuration can be safely reused. You can write `ActorSelection $` using wildcard paths also!
  - You do *not* need to specify `/user` at the start of each path. It is implied.
- `use-role` : the `Group` router will only route to routees at the specified `paths` on nodes marked with the given role. Can only specify one role here.

## HOCON for Clustered Group Routers

Here's an example of what the HOCON for a clustered `Group` router looks like, [taken from the WebCrawler sample](#):

```
akka {
 actor{
 provider = cluster
 deployment {
 /api/myClusterGroupRouter {
 router = broadcast-group # routing strategy
 routees.paths = ["/user/api"] # path of routee on each node
 nr-of-instances = 3 # max number of total routees
 cluster {
 enabled = on
 allow-local-routees = on
 use-role = crawler
 }
 }
 }
 }
}
```

## Clustered Pool Router Config

- `nr-of-instances` : this is the maximum number of total routees that this router will first deploy, and then route to.
- `max-nr-of-instances-per-node` : the maximum number of routees that the `Pool` router will deploy onto a given cluster node.
  - Note that `nr-of-instances` defines total number of routees, but number of routees per node will not be exceeded, i.e. if you define `nr-of-instances = 50` and `max-nr-of-instances-per-node = 2`, the router will deploy 2 routees per new node in the cluster, up to 25 nodes.
- `use-role` : router will only deploy routees onto nodes in the cluster marked with the given role. Can only specify one role here.

## HOCON for Clustered Pool Routers

Here's an example of what the HOCON for a clustered `Pool` router looks like:

```
akka {
 actor{
 provider = cluster
 deployment {
 /api/myClusterPoolRouter {
 router = round-robin-pool # routing strategy
 nr-of-instances = 10 # max number of total routees
 cluster {
 enabled = on
 allow-local-routees = on
 use-role = crawler
 max-nr-of-instances-per-node = 1
 }
 }
 }
 }
}
```

## Additional Resources

- [Scalable WebCrawler Sample](#)
- [Cluster Routing Config](#)

# Akka.Cluster Configuration

In this section, we'll review the most important and commonly used Cluster configuration options. To see the full list of options, check out the [full Cluster conf file here](#).

## Critical Configuration Options

Here are the most common options for configuring a `node` in the cluster:

- `min-nr-of-members` : the minimum number of member nodes required to be present in the cluster before the leader marks nodes as `up`.
  - `min size per role` : the minimum number of member nodes for the given role required to be present in the cluster before the role leader marks nodes with the specified role as `up` .
- `seed-nodes` : The addresses of the `seed nodes` to join automatically at startup. Comma-separated list of URI strings. A node may list itself as a seed node and "self-join."
- `roles` : the `roles` that this node is to fulfill in the cluster. List of strings, e.g. `roles = ["A", "B"]` .

You can see all of these options being used in the following HOCON:

```
akka {
 actor.provider = cluster
 remote {
 dot-netty.tcp {
 port = 8081
 hostname = localhost
 }
 }
 cluster {
 seed-nodes = ["akka.tcp://ClusterSystem@localhost:8081"] # address of seed node
 roles = ["crawler", "logger"] # roles this member is in
 role."crawler".min-nr-of-members = 3 # crawler role minimum node count
 }
}
```

## Specifying Minimum Cluster Sizes

One feature of Akka.Cluster that can be useful in a number of scenarios is the ability to specify a minimum cluster size, i.e. "this cluster must have at least 3 nodes present before it can be considered 'up'."

Akka.Cluster supports this behavior at both a cluster-wide and a per-role level.

### Cluster-Wide Minimum Size

If you want to specify a cluster-wide minimum size, then we need to set the `cluster.min-nr-of-members` property inside our HOCON configuration, like this:

```
akka {
 actor.provider = cluster
 remote {
 dot-netty.tcp {
 port = 8081
 hostname = localhost
 }
 }
}
```

```

cluster {
 seed-nodes = ["akka.tcp://ClusterSystem@localhost:8081"]
 min-nr-of-members = 3
}
}

```

In this case, the leader of the cluster will delay firing any `ClusterEvent.MemberUp` messages until 3 nodes have connected to the leader and have not become unreachable. So you can use this setting in combination with your own user-defined actors who subscribe to the `cluster` for gossip messages (`MemberUp`, specifically) to know if your cluster has reached its minimum size or not.

## Per-Role Minimum Size

If you want to delay nodes of a specific role from being marked as up until a certain minimum has been reached, you can accomplish that via HOCON too!

```

akka {
 actor.provider = cluster
 remote {
 dot-netty.tcp {
 port = 8081
 hostname = localhost
 }
 }
 cluster {
 seed-nodes = ["akka.tcp://ClusterSystem@localhost:8081"]
 roles = ["crawlerV1", "crawlerV2"]
 role."crawlerV1".min-nr-of-members = 3
 }
}

```

This tells the role leader for `crawlerV1` to not mark any of those nodes as up until at least three nodes with role `crawlerV1` have joined the cluster.

## Additional Resources

- [Cluster.conf](#): the full set of configuration options

# Using the Cluster ActorSystem Extension Object

`Akka.Cluster` is actually an `ActorSystem` extension that you can use to access membership information and `cluster gossip` directly.

## Getting a Reference to the `cluster`

You can get a direct reference to the `cluster` extension like so (drawn from the [SimpleClusterListener](#) example in the [Akka.NET project](#)):

```
using Akka.Actor;
using Akka.Cluster;
using Akka.Event;

namespace Samples.Cluster.Simple
{
 public class SimpleClusterListener : UntypedActor
 {
 // get direct reference to the Cluster extension
 protected Akka.Cluster.Cluster Cluster = Akka.Cluster.Cluster.Get(Context.System);
 }
}
```

## Working With Cluster Gossip

We've shown a number of examples and referred to many different types of `ClusterEvent` messages throughout *Akka.NET Clustering* thus far. We're going to take a moment to show you all of the different types of messages you can subscribe to from the `cluster` actor system extension and how the current state of the cluster can be replayed to new subscribers.

### Subscribing and Unsubscribing from Cluster Gossip

We've seen a few code samples that showed actors who subscribed to gossip messages from the `cluster` actor system extension, but let's review the `Subscribe` and `Unsubscribe` methods and see what they do.

#### Subscribing to Gossip

This is an example of subscribing to `IMemberEvents` from the `Cluster`:

```
// subscribe to all future IMemberEvents and get current state as snapshot
Cluster.Subscribe(Self, new[] { typeof(ClusterEvent.IMemberEvent) });
```

#### Unsubscribing from Gossip

Let's suppose you need to unsubscribe from cluster gossip - here's how you can accomplish that.

```
Cluster.Unsubscribe(Self); // unsub from ALL events
Cluster.Unsubscribe(Self, typeof(IMemberEvent)); // unsub from just IMemberEvent
```

These two `Unsubscribe` calls are virtually identical - the difference is that the first call unsubscribes `self` from ALL cluster events going forward. The second call only unsubscribes self from cluster messages of type `IMemberEvent`.

Subscribing and unsubscribing from cluster events are pretty straightforward - now let's take a closer look at the different classes of messages and events we can subscribe to.

## Cluster Gossip Event Types

Gossip events fall into three categories:

1. Member events
2. Reachability events
3. Metrics events (not yet implemented)

## Cluster Event Categories

### Member events

Member events refer to nodes joining / leaving / being removed from the cluster. These events are used by [Akka.Cluster routers](#) to automatically adjust their routee lists.

### Reachability events

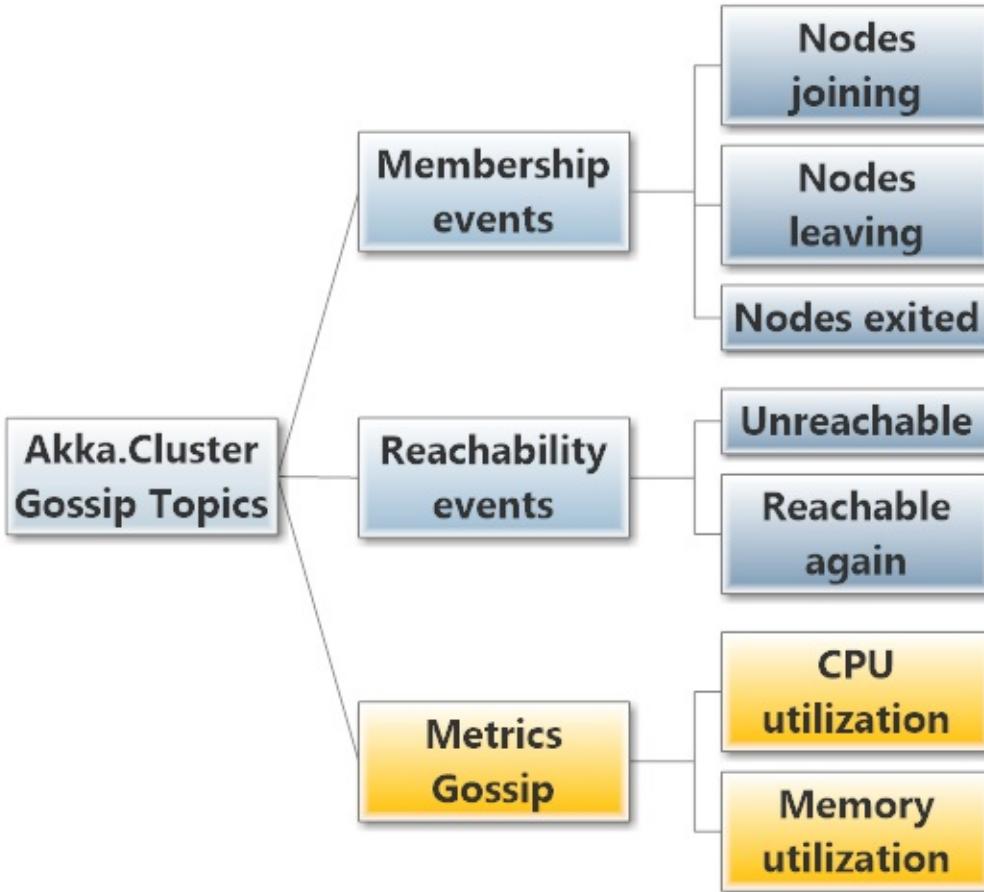
Reachability events refer to connectivity between nodes.

If node A can no longer reach node B, then B is considered to be "unreachable" to node A. If B becomes unreachable to a large number of nodes, the leader is going to mark the node as "down" and remove it from the cluster.

However, if B is able to communicate with A again then A will begin gossiping that B is once again "reachable."

### Gossip Message Classes

There's a variety of different types of information we can collect from Akka.Cluster via the `cluster`. These gossip classes fall into the categories described above.



- `ClusterEvent.IMemberEvent` - you can subscribe to messages that implement this interface in order to get data about changes in cluster membership.
- `ClusterEvent.IReachabilityEvent` - subscribe to messages that implement this interface in order to receive events about nodes being unreachable or reachable.
- `ClusterEvent.IClusterDomainEvent` - **subscribes you to ALL cluster messages.**
- `ClusterEvent.LeaderChanged` - subscribe to notifications about changes in the cluster leader.
- `ClusterEvent.RoleLeaderChanged` - subscribe to notifications about changes in the role leader.
- `ClusterShuttingDown` - receive shutdown notifications for the entire cluster (these events rarely happen.)

As shown above, you can subscribe to these events to get pieces of information you need in order to begin working with the cluster, such as knowing when 2 members of a particular role are up or knowing when the leader becomes unreachable.

Let's expand the `SimpleClusterListener` example from earlier to shows it subscribing to and handling `ClusterEvent` messages.

```

public class SimpleClusterListener : UntypedActor
{
 protected ILoggingAdapter Log = Context.GetLogger();
 protected Akka.Cluster.Cluster Cluster = Akka.Cluster.Cluster.Get(Context.System);

 /// <summary>
 /// Need to subscribe to cluster changes
 /// </summary>
 protected override void PreStart()
 {
 // subscribe to IMemberEvent and UnreachableMember events
 Cluster.Subscribe(Self, ClusterEvent.InitialStateAsEvents,
 new []{ typeof(ClusterEvent.IMemberEvent), typeof(ClusterEvent.UnreachableMember) });
 }
}

```

```

/// <summary>
/// Re-subscribe on restart
/// </summary>
protected override void PostStop()
{
 Cluster.Unsubscribe(Self);
}

protected override void OnReceive(object message)
{
 var up = message as ClusterEvent.MemberUp;
 if (up != null)
 {
 var mem = up;
 Log.Info("Member is Up: {0}", mem.Member);
 } else if(message is ClusterEvent.UnreachableMember)
 {
 var unreachable = (ClusterEvent.UnreachableMember) message;
 Log.Info("Member detected as unreachable: {0}", unreachable.Member);
 }
 else if (message is ClusterEvent.MemberRemoved)
 {
 var removed = (ClusterEvent.MemberRemoved) message;
 Log.Info("Member is Removed: {0}", removed.Member);
 }
 else if (message is ClusterEvent.IMemberEvent)
 {
 //IGNORE
 }
 else
 {
 Unhandled(message);
 }
}
}

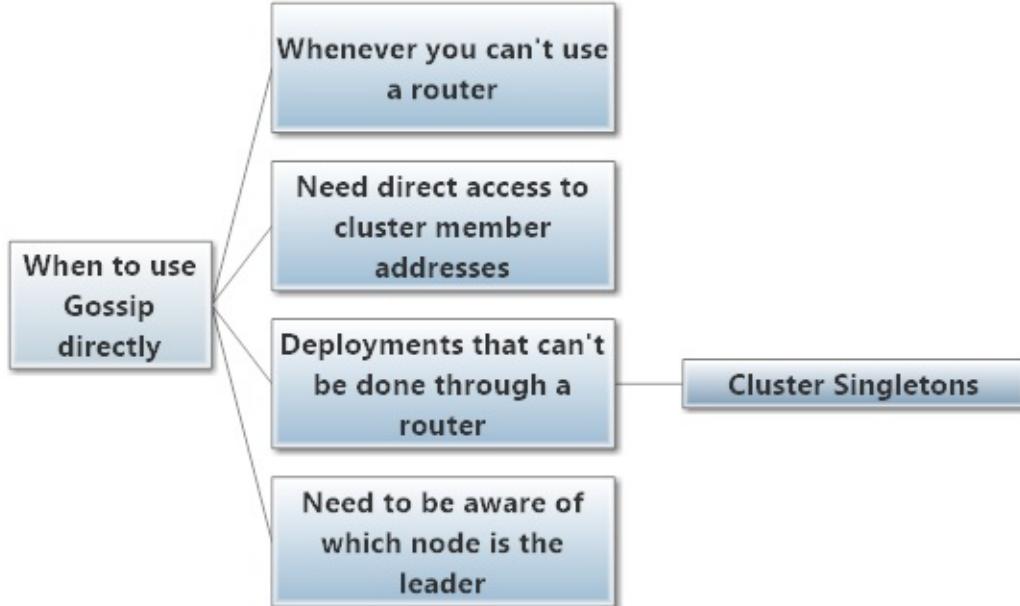
```

Inside the `SimpleClusterListener` we subscribe to messages of type of `ClusterEvent.IMemberEvent` and `ClusterEvent.UnreachableMember`, a class that implements `ClusterEvent.IReachabilityEvent`.

The `cluster` will periodically broadcast these messages as a result in changes in the membership of the cluster and their reachability, and they'll be delivered to the `SimpleClusterListener` as messages that can be handled inside the `OnReceive` method.

## When to Work with Gossip Directly

So when would you need to work with cluster gossip messages directly rather than use a clustered router?



In reality, you will use clustered routers 99% of the time. Using cluster gossip directly is only necessary in scenarios where a clustered router isn't a good fit, such as some of the examples given in the diagram above.

## Getting Cluster State

You can get the current state of the `cluster` two times:

1. On initial subscription
2. On demand

### Getting Cluster State on Initial Subscription

If you use [this overload of `Cluster.Subscribe`](#), you will get the state of the cluster right after subscribing to the event of your choice. Once you receive the initial state, you will not automatically be sent full state events going forward (see on-demand section below).

Here is how to subscribe to `cluster` events and also get the initial state of the cluster:

```

// subscribe to all future IMemberEvents and get current state as snapshot
Cluster.Subscribe(Self, ClusterEvent.SubscriptionInitialStateMode.InitialStateAsSnapshot,
 new[] { typeof(ClusterEvent.IMemberEvent) });

// subscribe to all future IMemberEvents and get current state as event stream
Cluster.Subscribe(Self, ClusterEvent.SubscriptionInitialStateMode.InitialStateAsEvents,
 new[] { typeof(ClusterEvent.IMemberEvent) });

```

Both of these methods accomplish the same goal - subscribing `Self` to all `IMemberEvent` cluster events going forward, but the difference is in how `Self` will receive the current state of the cluster.

In the first `Subscribe` call, `Self` will receive a `ClusterEvent.CurrentClusterState` message that describes all of the current members of the cluster, their roles, and their membership status.

In the second `Subscribe` call, `Self` will receive a stream of `MemberUp`, `MemberDown`, and other events - essentially an event-sourced version of what you receive in the `ClusterEvent.CurrentClusterState` message payload.

By default, the initial state will be delivered as a snapshot.

Any future membership change events will also produce the events that are broadcast in the initial state ( `MemberUp` , etc) - so those types of events need to be handled either way.

## Getting Cluster State On-Demand

In the above example, the current state of the cluster is delivered initially, but is not delivered afterwards.

To receive a snapshot of the state of the cluster on demand, use the `SendCurrentClusterState` method, like so:

```
// get the current state of cluster
Cluster.SendCurrentClusterState (Self);
```

## Additional Resources

- [SimpleClusterListener example](#)

# Cluster Singleton

For some use cases it is convenient and sometimes also mandatory to ensure that you have exactly one actor of a certain type running somewhere in the cluster.

Some examples:

- single point of responsibility for certain cluster-wide consistent decisions, or coordination of actions across the cluster system
- single entry point to an external system
- single master, many workers
- centralized naming service, or routing logic

Using a singleton should not be the first design choice. It has several drawbacks, such as single-point of bottleneck. Single-point of failure is also a relevant concern, but for some cases this feature takes care of that by making sure that another singleton instance will eventually be started.

The cluster singleton pattern is implemented by `Akka.Cluster.Tools.Singleton.ClusterSingletonManager`. It manages one singleton actor instance among all cluster nodes or a group of nodes tagged with a specific role.

`ClusterSingletonManager` is an actor that is supposed to be started on all nodes, or all nodes with specified role, in the cluster. The actual singleton actor is started by the `ClusterSingletonManager` on the oldest node by creating a child actor from supplied Props. `ClusterSingletonManager` makes sure that at most one singleton instance is running at any point in time.

The singleton actor is always running on the oldest member with specified role. The oldest member is determined by `Akka.Cluster.Member#IsOlderThan`. This can change when removing that member from the cluster. Be aware that there is a short time period when there is no active singleton during the hand-over process.

The cluster failure detector will notice when oldest node becomes unreachable due to things like CLR crash, hard shut down, or network failure. Then a new oldest node will take over and a new singleton actor is created. For these failure scenarios there will not be a graceful hand-over, but more than one active singletons is prevented by all reasonable means. Some corner cases are eventually resolved by configurable timeouts.

You can access the singleton actor by using the provided `Akka.Cluster.Tools.Singleton.ClusterSingletonProxy`, which will route all messages to the current instance of the singleton. The proxy will keep track of the oldest node in the cluster and resolve the singleton's `IActorRef` by explicitly sending the singleton's `ActorSelection` the `Akka.Actor.Identify` message and waiting for it to reply. This is performed periodically if the singleton doesn't reply within a certain (configurable) time. Given the implementation, there might be periods of time during which the `IActorRef` is unavailable, e.g., when a node leaves the cluster. In these cases, the proxy will buffer the messages sent to the singleton and then deliver them when the singleton is finally available. If the buffer is full the `ClusterSingletonProxy` will drop old messages when new messages are sent via the proxy. The size of the buffer is configurable and it can be disabled by using a buffer size of 0.

It's worth noting that messages can always be lost because of the distributed nature of these actors. As always, additional logic should be implemented in the singleton (acknowledgement) and in the client (retry) actors to ensure at-least-once message delivery.

## Potential problems to be aware of

This pattern may seem to be very tempting to use at first, but it has several drawbacks, some of them are listed below:

- the cluster singleton may quickly become a performance bottleneck,

- you can not rely on the cluster singleton to be non-stop available — e.g. when the node on which the singleton has been running dies, it will take a few seconds for this to be noticed and the singleton be migrated to another node,
- in the case of a network partition appearing in a Cluster that is using Automatic Downing, it may happen that the isolated clusters each decide to spin up their own singleton, meaning that there might be multiple singletons running in the system, yet the Clusters have no way of finding out about them (because of the partition).

Especially the last point is something you should be aware of — in general when using the Cluster Singleton pattern you should take care of downing nodes yourself and not rely on the timing based auto-down feature.

[!WARNING] Be very careful when using Cluster Singleton together with Automatic Downing, since it allows the cluster to split up into two separate clusters, which in turn will result in `multiple Singletons` being started, one in each separate cluster!

## An Example

Assume that we need one single entry point to an external system. An actor that receives messages from a JMS queue with the strict requirement that only one JMS consumer must exist to make sure that the messages are processed in order. That is perhaps not how one would like to design things, but a typical real-world scenario when integrating with external systems.

On each node in the cluster you need to start the `clusterSingletonManager` and supply the Props of the singleton actor, in this case the JMS queue consumer.

```
system.ActorOf(ClusterSingletonManager.Props(
 singletonProps: Props.Create<MySingletonActor>(),
 terminationMessage: PoisonPill.Instance,
 settings: ClusterSingletonManagerSettings.Create(system).WithRole("worker")),
 name: "consumer");
```

Here we limit the singleton to nodes tagged with the "worker" role, but all nodes, independent of role, can be used by not specifying `WithRole`.

Here we use an application specific `TerminationMessage` to be able to close the resources before actually stopping the singleton actor. Note that `PoisonPill` is a perfectly fine `TerminationMessage` if you only need to stop the actor.

With the names given above, access to the singleton can be obtained from any cluster node using a properly configured proxy.

```
system.ActorOf(ClusterSingletonProxy.Props(
 singletonManagerPath: "/user/consumer",
 settings: ClusterSingletonProxySettings.Create(system).WithRole("worker")),
 name: "consumerProxy");
```

## Configuration

The following configuration properties are read by the `ClusterSingletonManagerSettings` when created with a `ActorSystem` parameter. It is also possible to amend the `ClusterSingletonManagerSettings` or create it from another config section with the same layout as below. `ClusterSingletonManagerSettings` is a parameter to the `clusterSingletonManager.props` factory method, i.e. each singleton can be configured with different settings if needed.

```
akka.cluster.singleton {
 # The actor name of the child singleton actor.
 singleton-name = "singleton"
```

```

Singleton among the nodes tagged with specified role.
If the role is not specified it's a singleton among all nodes in the cluster.
role = ""

When a node is becoming oldest it sends hand-over request to previous oldest,
that might be leaving the cluster. This is retried with this interval until
the previous oldest confirms that the hand over has started or the previous
oldest member is removed from the cluster (+ akka.cluster.down-removal-margin).
hand-over-retry-interval = 1s

The number of retries are derived from hand-over-retry-interval and
akka.cluster.down-removal-margin (or ClusterSingletonManagerSettings.RemovalMargin),
but it will never be less than this property.
min-number-of-hand-over-retries = 10
}

```

The following configuration properties are read by the `ClusterSingletonProxySettings` when created with a `ActorSystem` parameter. It is also possible to amend the `ClusterSingletonProxySettings` or create it from another config section with the same layout as below. `ClusterSingletonProxySettings` is a parameter to the `ClusterSingletonProxy.props` factory method, i.e. each singleton proxy can be configured with different settings if needed.

```

akka.cluster.singleton-proxy {
 # The actor name of the singleton actor that is started by the ClusterSingletonManager
 singleton-name = ${akka.cluster.singleton.name}

 # The role of the cluster nodes where the singleton can be deployed.
 # If the role is not specified then any node will do.
 role = ""

 # Interval at which the proxy will try to resolve the singleton instance.
 singleton-identification-interval = 1s

 # If the location of the singleton is unknown the proxy will buffer this
 # number of messages and deliver them when the singleton is identified.
 # When the buffer is full old messages will be dropped when new messages are
 # sent via the proxy.
 # Use 0 to disable buffering, i.e. messages will be dropped immediately if
 # the location of the singleton is unknown.
 # Maximum allowed buffer size is 10000.
 buffer-size = 1000
}

```

# Distributed Publish Subscribe in Cluster

How do I send a message to an actor without knowing which node it is running on?

How do I send messages to all actors in the cluster that have registered interest in a named topic?

This pattern provides a mediator actor, `akka.cluster.pubsub.DistributedPubSubMediator`, that manages a registry of actor references and replicates the entries to peer actors among all cluster nodes or a group of nodes tagged with a specific role.

The `DistributedPubSubMediator` actor is supposed to be started on all nodes, or all nodes with specified role, in the cluster. The mediator can be started with the `DistributedPubSub` extension or as an ordinary actor.

The registry is eventually consistent, i.e. changes are not immediately visible at other nodes, but typically they will be fully replicated to all other nodes after a few seconds. Changes are only performed in the own part of the registry and those changes are versioned. Deltas are disseminated in a scalable way to other nodes with a gossip protocol.

You can send messages via the mediator on any node to registered actors on any other node.

There are two different modes of message delivery, explained in the sections `Publish` and `Send` below.

## Publish

This is the true pub/sub mode. A typical usage of this mode is a chat room in an instant messaging application.

Actors are registered to a named topic. This enables many subscribers on each node. The message will be delivered to all subscribers of the topic.

For efficiency the message is sent over the wire only once per node (that has a matching topic), and then delivered to all subscribers of the local topic representation.

You register actors to the local mediator with `DistributedPubSubMediator.Subscribe`. Successful `Subscribe` and `Unsubscribe` is acknowledged with `DistributedPubSubMediator.SubscribeAck` and `DistributedPubSubMediator.UnsubscribeAck` replies. The acknowledgment means that the subscription is registered, but it can still take some time until it is replicated to other nodes.

You publish messages by sending `DistributedPubSubMediator.Publish` message to the local mediator.

Actors are automatically removed from the registry when they are terminated, or you can explicitly remove entries with `DistributedPubSubMediator.Unsubscribe`.

An example of a subscriber actor:

```
public class Subscriber : ReceiveActor
{
 private readonly ILoggingAdapter log = Context.GetLogger();

 public Subscriber()
 {
 var mediator = DistributedPubSub.Get(Context.System).Mediator;

 // subscribe to the topic named "content"
 mediator.Tell(new Subscribe("content", Self));

 Receive<string>(s =>
 {
 log.Info($"Got {s}");
 });
 }
}
```

```

Receive<SubscribeAck>(subscribeAck =>
{
 if (subscribeAck.Subscribe.Topic.Equals("content")
 && subscribeAck.Subscribe.Ref.Equals(Self)
 && subscribeAck.Subscribe.Group == null)
 {
 log.Info("subscribing");
 }
});
}
}

```

Subscriber actors can be started on several nodes in the cluster, and all will receive messages published to the "content" topic.

```

RunOn(() =>
{
 Sys.ActorOf(Props.Create<Subscriber>(), "subscriber1");
}, _first);

RunOn(() =>
{
 Sys.ActorOf(Props.Create<Subscriber>(), "subscriber2");
 Sys.ActorOf(Props.Create<Subscriber>(), "subscriber3");
}, _second);

```

A simple actor that publishes to this "content" topic:

```

public class Publisher : ReceiveActor
{
 public Publisher()
 {
 // activate the extension
 var mediator = DistributedPubSub.Get(Context.System).Mediator;

 Receive<string>(str =>
 {
 var upperCase = str.ToUpper();
 mediator.Tell(new Publish("content", upperCase));
 });
 }
}

```

It can publish messages to the topic from anywhere in the cluster:

```

RunOn(() =>
{
 var publisher = Sys.ActorOf(Props.Create<Publisher>(), "publisher");

 // after a while the subscriptions are replicated
 publisher.Tell("hello");
}, _third);

```

## Topic Groups

Actors may also be subscribed to a named topic with a group id. If subscribing with a group id, each message published to a topic with the `SendOneMessageToEachGroup` flag set to true is delivered via the supplied `RoutingLogic` (default random) to one actor within each subscribing group.

If all the subscribed actors have the same group id, then this works just like `Send` and each message is only delivered to one subscriber.

If all the subscribed actors have different group names, then this works like normal `Publish` and each message is broadcasted to all subscribers.

[!NOTE] Note that if the group id is used it is part of the topic identifier. Messages published with `SendOneMessageToEachGroup=false` will not be delivered to subscribers that subscribed with a group id. Messages published with `SendOneMessageToEachGroup=true` will not be delivered to subscribers that subscribed without a group id.

## Send

This is a point-to-point mode where each message is delivered to one destination, but you still do not have to know where the destination is located. A typical usage of this mode is private chat to one other user in an instant messaging application. It can also be used for distributing tasks to registered workers, like a cluster aware router where the routees dynamically can register themselves.

The message will be delivered to one recipient with a matching path, if any such exists in the registry. If several entries match the path because it has been registered on several nodes the message will be sent via the supplied `RoutingLogic` (default random) to one destination. The `sender()` of the message can specify that local affinity is preferred, i.e. the message is sent to an actor in the same local actor system as the used mediator actor, if any such exists, otherwise route to any other matching entry.

You register actors to the local mediator with `DistributedPubSubMediator.Put`. The `IActorRef` in `Put` must belong to the same local actor system as the mediator. The path without address information is the key to which you send messages. On each node there can only be one actor for a given path, since the path is unique within one local actor system.

You send messages by sending `DistributedPubSubMediator.Send` message to the local mediator with the path (without address information) of the destination actors.

Actors are automatically removed from the registry when they are terminated, or you can explicitly remove entries with `DistributedPubSubMediator.Remove`.

An example of a destination actor:

```
public class Destination : ReceiveActor
{
 private readonly ILoggingAdapter log = Context.GetLogger();

 public Destination()
 {
 // activate the extension
 var mediator = DistributedPubSub.Get(Context.System).Mediator;

 // register to the path
 mediator.Tell(new Put(Self));

 Receive<string>(s =>
 {
 log.Info($"Got {s}");
 });
 }
}
```

Destination actors can be started on several nodes in the cluster, and all will receive messages sent to the path (without address information).

```
RunOn(() =>
{
 Sys.ActorOf(Props.Create<Destination>(), "destination");
}, _first);

RunOn(() =>
{
 Sys.ActorOf(Props.Create<Destination>(), "destination");
}, _second);
```

A simple actor that sends to the path:

```
public class Sender : ReceiveActor
{
 public Sender()
 {
 // activate the extension
 var mediator = DistributedPubSub.Get(Context.System).Mediator;

 Receive<string>(str =>
 {
 var upperCase = str.ToUpper();
 mediator.Tell(new Send(path: "/user/destination", message: upperCase, localAffinity: true));
 });
 }
}
```

It can send messages to the path from anywhere in the cluster:

```
RunOn(() =>
{
 var sender = Sys.ActorOf(Props.Create<Sender>(), "sender");

 // after a while the destinations are replicated
 sender.Tell("hello");
}, _third);
```

It is also possible to broadcast messages to the actors that have been registered with `Put . Send` `DistributedPubSubMediator.SendToAll` message to the local mediator and the wrapped message will then be delivered to all recipients with a matching path. Actors with the same path, without address information, can be registered on different nodes. On each node there can only be one such actor, since the path is unique within one local actor system.

Typical usage of this mode is to broadcast messages to all replicas with the same path, e.g. 3 actors on different nodes that all perform the same actions, for redundancy. You can also optionally specify a property (`AllButSelf`) deciding if the message should be sent to a matching path on the self node or not.

## DistributedPubSub Extension

In the example above the mediator is started and accessed with the

`Akka.Cluster.Tools.PublishSubscribe.DistributedPubSub` extension. That is convenient and perfectly fine in most cases, but it can be good to know that it is possible to start the mediator actor as an ordinary actor and you can have several different mediators at the same time to be able to divide a large number of actors/topics to different mediators. For example you might want to use different cluster roles for different mediators.

The `DistributedPubSub` extension can be configured with the following properties:

```
Settings for the DistributedPubSub extension
```

```
akka.cluster.pub-sub {
 # Actor name of the mediator actor, /system/distributedPubSubMediator
 name = distributedPubSubMediator

 # Start the mediator on members tagged with this role.
 # All members are used if undefined or empty.
 role = ""

 # The routing logic to use for 'Send'
 # Possible values: random, round-robin, broadcast
 routing-logic = random

 # How often the DistributedPubSubMediator should send out gossip information
 gossip-interval = 1s

 # Removed entries are pruned after this duration
 removed-time-to-live = 120s

 # Maximum number of elements to transfer in one message when synchronizing the registries.
 # Next chunk will be transferred in next round of gossip.
 max-delta-elements = 3000

 # The id of the dispatcher to use for DistributedPubSubMediator actors.
 # If not specified default dispatcher is used.
 # If specified you need to define the settings of the actual dispatcher.
 use-dispatcher = ""
}
```

It is recommended to load the extension when the actor system is started by defining it in akka.extensions configuration property. Otherwise it will be activated when first used and then it takes a while for it to be populated.

```
akka.extensions = ["Akka.Cluster.Tools.PublishSubscribe.DistributedPubSubExtensionProvider,Akka.Cluster.Tools"]
```

# Cluster Client

An actor system that is not part of the cluster can communicate with actors somewhere in the cluster via this `clusterClient`. The client can of course be part of another cluster. It only needs to know the location of one (or more) nodes to use as initial contact points. It will establish a connection to a `ClusterReceptionist` somewhere in the cluster. It will monitor the connection to the receptionist and establish a new connection if the link goes down. When looking for a new receptionist it uses fresh contact points retrieved from previous establishment, or periodically refreshed contacts, i.e. not necessarily the initial contact points.

[!NOTE] `clusterClient` should not be used when sending messages to actors that run within the same cluster. Similar functionality as the `ClusterClient` is provided in a more efficient way by `Distributed Publish Subscribe` in Cluster for actors that belong to the same cluster.

Also, note it's necessary to change `akka.actor.provider` from `Akka.Actor.LocalActorRefProvider` to `Akka.Remote.RemoteActorRefProvider` or `Akka.Cluster.ClusterActorRefProvider` when using the cluster client.

```
akka.actor.provider = "Akka.Cluster.ClusterActorRefProvider, Akka.Cluster"
```

or this shorthand notation

```
akka.actor.provider = cluster
```

The receptionist is supposed to be started on all nodes, or all nodes with specified role, in the cluster. The receptionist can be started with the `clusterClientReceptionist` extension or as an ordinary actor.

You can send messages via the `clusterClient` to any actor in the cluster that is registered in the `DistributedPubSubMediator` used by the `ClusterReceptionist`. The `ClusterClientReceptionist` provides methods for registration of actors that should be reachable from the client. Messages are wrapped in `clusterClient.Send`, `clusterClient.SendToAll` or `clusterClient.Publish`.

Both the `ClusterClient` and the `clusterClientReceptionist` emit events that can be subscribed to. The `clusterClient` sends out notifications in relation to having received a list of contact points from the `clusterClientReceptionist`. One use of this list might be for the client to record its contact points. A client that is restarted could then use this information to supersede any previously configured contact points.

The `ClusterClientReceptionist` sends out notifications in relation to having received contact from a `clusterClient`. This notification enables the server containing the receptionist to become aware of what clients are connected.

- 1. ClusterClient.Send** The message will be delivered to one recipient with a matching path, if any such exists. If several entries match the path the message will be delivered to one random destination. The `Sender` of the message can specify that local affinity is preferred, i.e. the message is sent to an actor in the same local actor system as the used receptionist actor, if any such exists, otherwise random to any other matching entry.

- 2. ClusterClient.SendToAll** The message will be delivered to all recipients with a matching path.

- 3. ClusterClient.Publish** The message will be delivered to all recipients Actors that have been registered as subscribers to the named topic.

Response messages from the destination actor are tunneled via the receptionist to avoid inbound connections from other cluster nodes to the client, i.e. the `Sender`, as seen by the destination actor, is not the client itself. The `Sender` of the response messages, as seen by the client, is `deadLetters` since the client should normally send subsequent messages via the `clusterClient`. It is possible to pass the original sender inside the reply messages if the client is supposed to communicate directly to the actor in the cluster.

While establishing a connection to a receptionist the `ClusterClient` will buffer messages and send them when the connection is established. If the buffer is full the `clusterClient` will drop old messages when new messages are sent via the client. The size of the buffer is configurable and it can be disabled by using a buffer size of 0.

It's worth noting that messages can always be lost because of the distributed nature of these actors. As always, additional logic should be implemented in the destination (acknowledgement) and in the client (retry) actors to ensure at-least-once message delivery.

## An Example

On the cluster nodes first start the receptionist. Note, it is recommended to load the extension when the actor system is started by defining it in the `akka.extensions` configuration property:

```
akka.extensions = ["Akka.Cluster.Tools.Client.ClusterClientReceptionistExtensionProvider, Akka.Cluster.Tools"]
```

Next, register the actors that should be available for the client.

```
RunOn(() =>
{
 var serviceA = Sys.ActorOf(Props.Create<Service>(), "serviceA");
 ClusterClientReceptionist.Get(Sys).RegisterService(serviceA);
}, host1);

RunOn(() =>
{
 var serviceB = Sys.ActorOf(Props.Create<Service>(), "serviceB");
 ClusterClientReceptionist.Get(Sys).RegisterService(serviceB);
}, host2, host3);
```

On the client you create the `ClusterClient` actor and use it as a gateway for sending messages to the actors identified by their path (without address information) somewhere in the cluster.

```
RunOn(() =>
{
 var c = Sys.ActorOf(Client.ClusterClient.Props(
 ClusterClientSettings.Create(Sys).WithInitialContacts(initialContacts), "client"));
 c.Tell(new Client.ClusterClient.Send("/user/serviceA", "hello", localAffinity: true));
 c.Tell(new Client.ClusterClient.SendToAll("/user/serviceB", "hi"));
}, client);
```

The `initialContacts` parameter is a `IEnumerable<ActorPath>`, which can be created like this:

```
var initialContacts = new List<ActorPath>()
{
 ActorPath.Parse("akka.tcp://OtherSys@host1:2552/system/receptionist"),
 ActorPath.Parse("akka.tcp://OtherSys@host2:2552/system/receptionist")
};

var settings = ClusterClientSettings.Create(Sys).WithInitialContacts(initialContacts);
```

You will probably define the address information of the initial contact points in configuration or system property. See also Configuration.

## ClusterClientReceptionist Extension

In the example above the receptionist is started and accessed with the

`Akka.Cluster.Tools.Client.ClusterClientReceptionist` extension. That is convenient and perfectly fine in most cases, but it can be good to know that it is possible to start the `akka.cluster.client.ClusterReceptionist` actor as an ordinary actor and you can have several different receptionists at the same time, serving different types of clients.

Note that the `ClusterClientReceptionist` uses the `DistributedPubSub` extension, which is described in [Distributed Publish Subscribe in Cluster](#).

It is recommended to load the extension when the actor system is started by defining it in the `akka.extensions` configuration property:

```
akka.extensions = ["Akka.Cluster.Tools.Client.ClusterClientReceptionistExtensionProvider, Akka.Cluster.Tools"]
```

## Events

As mentioned earlier, both the `ClusterClient` and `ClusterClientReceptionist` emit events that can be subscribed to. The following code snippet declares an actor that will receive notifications on contact points (addresses to the available receptionists), as they become available. The code illustrates subscribing to the events and receiving the `clusterClient` initial state.

```
public class ClientListener : UntypedActor
{
 private readonly IActorRef _targetClient;

 public ClientListener(IActorRef targetClient)
 {
 _targetClient = targetClient;
 }

 protected override void OnReceive(object message)
 {
 Context.Become(ReceiveWithContactPoints(ImmutableHashSet<ActorPath>.Empty));
 }

 protected override void PreStart()
 {
 _targetClient.Tell(SubscribeContactPoints.Instance);
 }

 public UntypedReceive ReceiveWithContactPoints(IImmutableSet<ActorPath> contactPoints)
 {
 return (message) =>
 {
 switch (message)
 {
 // Now do something with the up-to-date "cps"
 case ContactPoints cp:
 Context.Become(ReceiveWithContactPoints(cp.ContactPointsList));
 break;
 // Now do something with an up-to-date "contactPoints + cp"
 case ContactPointAdded cpa:
 Context.Become(ReceiveWithContactPoints(contactPoints.Add(cpa.ContactPoint)));
 break;
 // Now do something with an up-to-date "contactPoints - cp"
 case ContactPointRemoved cpr:
 Context.Become(ReceiveWithContactPoints(contactPoints.Remove(cpr.ContactPoint)));
 break;
 }
 };
 }
}
```

Similarly we can have an actor that behaves in a similar fashion for learning what cluster clients contact a

`ClusterClientReceptionist :`

```

public class ReceptionistListener : UntypedActor
{
 private readonly IActorRef _targetReceptionist;

 public ReceptionistListener(IActorRef targetReceptionist)
 {
 _targetReceptionist = targetReceptionist;
 }

 protected override void OnReceive(object message)
 {
 Context.Become(ReceiveWithContactPoints(ImmutableHashSet<IActorRef>.Empty));
 }

 protected override void PreStart()
 {
 _targetReceptionist.Tell(SubscribeClusterClients.Instance);
 }

 public UntypedReceive ReceiveWithContactPoints(IImmutableSet<IActorRef> contactPoints)
 {
 return (message) =>
 {
 switch (message)
 {
 // Now do something with the up-to-date "c"
 case ClusterClients cc:
 Context.Become(ReceiveWithContactPoints(cc.ClusterClientsList));
 break;
 // Now do something with an up-to-date "clusterClients + c"
 case ClusterClientUp ccu:
 Context.Become(ReceiveWithContactPoints(contactPoints.Add(ccu.ClusterClient)));
 break;
 // Now do something with an up-to-date "clusterClients - c"
 case ClusterClientUnreachable ccun:
 Context.Become(ReceiveWithContactPoints(contactPoints.Remove(ccun.ClusterClient)));
 break;
 }
 };
 }
}

```

## Configuration

The `ClusterClientReceptionist` extension (or `ClusterReceptionistSettings`) can be configured with the following properties:

```

#####
Akka Cluster Tools Reference Config File
#####

This is the reference config file that contains all the default settings.
Make your edits/overrides in your application.conf.

//receptionist-ext-config
Settings for the ClusterClientReceptionist extension
akka.cluster.client.receptionist {
 # Actor name of the ClusterReceptionist actor, /system/receptionist
 name = receptionist
}
```

```

Start the receptionist on members tagged with this role.
All members are used if undefined or empty.
role = ""

The receptionist will send this number of contact points to the client
number-of-contacts = 3

The actor that tunnel response messages to the client will be stopped
after this time of inactivity.
response-tunnel-receive-timeout = 30s

The id of the dispatcher to use for ClusterReceptionist actors.
If not specified default dispatcher is used.
If specified you need to define the settings of the actual dispatcher.
use-dispatcher = ""

How often failure detection heartbeat messages should be received for
each ClusterClient
heartbeat-interval = 2s

Number of potentially lost/delayed heartbeats that will be
accepted before considering it to be an anomaly.
The ClusterReceptionist is using the akka.remote.DeadlineFailureDetector, which
will trigger if there are no heartbeats within the duration
heartbeat-interval + acceptable-heartbeat-pause, i.e. 15 seconds with
the default settings.
acceptable-heartbeat-pause = 13s

Failure detection checking interval for checking all ClusterClients
failure-detection-interval = 2s
}

//receptionist-ext-config

//cluster-client-config
Settings for the ClusterClient
akka.cluster.client {
 # Actor paths of the ClusterReceptionist actors on the servers (cluster nodes)
 # that the client will try to contact initially. It is mandatory to specify
 # at least one initial contact.
 # Comma separated full actor paths defined by a string on the form of
 # "akka.tcp://system@hostname:port/system/receptionist"
 initial-contacts = []

 # Interval at which the client retries to establish contact with one of
 # ClusterReceptionist on the servers (cluster nodes)
 establishing-get-contacts-interval = 3s

 # Interval at which the client will ask the ClusterReceptionist for
 # new contact points to be used for next reconnect.
 refresh-contacts-interval = 60s

 # How often failure detection heartbeat messages should be sent
 heartbeat-interval = 2s

 # Number of potentially lost/delayed heartbeats that will be
 # accepted before considering it to be an anomaly.
 # The ClusterClient is using the akka.remote.DeadlineFailureDetector, which
 # will trigger if there are no heartbeats within the duration
 # heartbeat-interval + acceptable-heartbeat-pause, i.e. 15 seconds with
 # the default settings.
 acceptable-heartbeat-pause = 13s

 # If connection to the receptionist is not established the client will buffer
 # this number of messages and deliver them when the connection is established.
 # When the buffer is full old messages will be dropped when new messages are sent
 # via the client. Use 0 to disable buffering, i.e. messages will be dropped
 # immediately if the location of the singleton is unknown.
 # Maximum allowed buffer size is 10000.
 buffer-size = 1000
}

```

```

If connection to the receptionist is lost and the client has not been
able to acquire a new connection for this long the client will stop itself.
This duration makes it possible to watch the cluster client and react on a more permanent
loss of connection with the cluster, for example by accessing some kind of
service registry for an updated set of initial contacts to start a new cluster client with.
If this is not wanted it can be set to "off" to disable the timeout and retry
forever.
reconnect-timeout = off
}
//##cluster-client-config

Protobuf serializer for ClusterClient messages
akka.actor {
 serializers {
 akka-cluster-client = "Akka.Cluster.Tools.Client.Serialization.ClusterClientMessageSerializer, Akka.Cluster
.Tools"
 }
 serialization-bindings {
 "Akka.Cluster.Tools.Client.IClusterClientMessage, Akka.Cluster.Tools" = akka-cluster-client
 }
 serialization-identifiers {
 "Akka.Cluster.Tools.Client.Serialization.ClusterClientMessageSerializer, Akka.Cluster.Tools" = 15
 }
}

```

The 'akka.cluster.client' configuration properties are read by the `ClusterClientSettings` when created with a `ActorSystem` parameter. It is also possible to amend the `ClusterClientSettings` or create it from another config section with the same layout in the reference config. `ClusterClientSettings` is a parameter to the `ClusterClient.Props()` factory method, i.e. each client can be configured with different settings if needed.

## Failure handling

When the cluster client is started it must be provided with a list of initial contacts which are cluster nodes where receptionists are running. It will then repeatedly (with an interval configurable by `establishing-get-contacts-interval`) try to contact those until it gets in contact with one of them. While running, the list of contacts are continuously updated with data from the receptionists (again, with an interval configurable with `refresh-contacts-interval`), so that if there are more receptionists in the cluster than the initial contacts provided to the client the client will learn about them.

While the client is running it will detect failures in its connection to the receptionist by heartbeats if more than a configurable amount of heartbeats are missed the client will try to reconnect to its known set of contacts to find a receptionist it can access.

## When the cluster cannot be reached at all

It is possible to make the cluster client stop entirely if it cannot find a receptionist it can talk to within a configurable interval. This is configured with the `reconnect-timeout`, which defaults to off. This can be useful when initial contacts are provided from some kind of service registry, cluster node addresses are entirely dynamic and the entire cluster might shut down or crash, be restarted on new addresses. Since the client will be stopped in that case a monitoring actor can watch it and upon `Terminate` a new set of initial contacts can be fetched and a new cluster client started.

# Akka.Cluster.Sharding module

Cluster sharding is useful in cases when you want to contact with cluster actors using their logical id's, but don't want to care about their physical location inside the cluster or manage their creation. Moreover it's able to rebalance them, as nodes join/leave the cluster. It's often used to represent i.e. Aggregate Roots in Domain Driven Design terminology.

Cluster sharding can operate in 2 modes, configured via `akka.cluster.sharding.state-store-mode` HOCON configuration:

1. `persistence` (**default**) depends on Akka.Persistence module. In order to use it, you'll need to specify an event journal accessible by all of the participating nodes. An information about the particular shard placement is stored in a persistent cluster singleton actor known as *coordinator*. In order to guarantee consistent state between different incarnations, coordinator stores its own state using Akka.Persistence event journals.
2. `ddata` (**experimental**, available in versions above 1.3.2) depends on Akka.DistributedData module. It uses Conflict-free Replicated Data Types (CRDT) to ensure eventually consistent shard placement and global availability via node-to-node replication and automatic conflict resolution. In this mode event journals don't have to be configured. At this moment this mode doesn't support `akka.cluster.sharding.remember-entities` option.

Cluster sharding may be active only on nodes in `up` status - so the ones fully recognized and acknowledged by every other node in a cluster.

## QuickStart

Actors managed by cluster sharding are called **entities** and can be automatically distributed across multiple nodes inside the cluster. One entity instance may live only at one node at the time, and can be communicated with via `ShardRegion` without need to know, what it's exact node location is.

Example:

```
// define envelope used to message routing
public sealed class ShardEnvelope
{
 public readonly int ShardId;
 public readonly int EntityId;
 public readonly object Message;

 ...
}

// define, how shard id, entity id and message itself should be resolved
public sealed class MessageExtractor : IMessageExtractor
{
 public string EntityId(object message) => (message as ShardEnvelope)?.EntityId.ToString();

 public string ShardId(object message) => (message as ShardEnvelope)?.ShardId.ToString();

 public object EntityMessage(object message) => (message as ShardEnvelope)?.Message;
}

// register actor type as a sharded entity
var region = await ClusterSharding.Get(system).StartAsync(
 typeName: "my-actor",
 entityProps: Props.Create<MyActor>(),
 settings: ClusterShardingSettings.Create(system),
 messageExtractor: new MessageExtractor());

// send message to entity through shard region
```

```
region.Tell(new ShardEnvelope(shardId: 1, entityId: 1, message: "hello"))
```

In this example, we first specify way to resolve our message recipients in context of sharded entities. For this, specialized message type called `ShardEnvelope` and resolution strategy called `MessageExtractor` have been specified. That part can be customized, and shared among many different shard regions, but it needs to be uniform among all nodes.

Second part of an example is registering custom actor type as sharded entity using `ClusterSharding.Start` or `ClusterSharding.StartAsync` methods. Result is the `IActorRef` to shard region used to communicate between current actor system and target entities. Shard region must be specified once per each type on each node, that is expected to participate in sharding entities of that type. Keep in mind, that it's recommended to wait for the current node to first fully join the cluster before initializing a shard regions in order to avoid potential timeouts.

In some cases, the actor may need to know the `entityId` associated with it. This can be achieved using the `entityPropsFactory` parameter to `ClusterSharding.Start` or `ClusterSharding.StartAsync`. The entity ID will be passed to the factory as a parameter, which can then be used in the creation of the actor.

In case when you want to send message to entities from specific node, but you don't want that node to participate in sharding itself, you can use `ShardRegionProxy` for that.

Example:

```
var proxy = ClusterSharding.Get(system).StartProxy(
 typeName: "my-actor",
 role: null,
 messageExtractor: new MessageExtractor());
```

## Shards

Entities are located and managed automatically. They can also be recreated on the other nodes, as new nodes join the cluster or old ones are leaving it. This process is called rebalancing and for performance reasons it never works over a single entity. Instead all entities are organized and managed in so called shards.

As you may have seen in the examples above shard resolution algorithm is one of the choices you have to make. Good uniform distribution is not an easy task - too small number shards may result in not even distribution of entities across all nodes, while too many of them may increase message routing latency and rebalancing overhead. As a rule of thumb, you may decide to have a number of shards ten times greater than expected maximum number of cluster nodes.

By default rebalancing process always happens from nodes with the highest number of shards, to the ones with the smallest one. This can be configured into by specifying custom implementation of the `IshardAllocationStrategy` interface in `clusterSharding.Start` parameters.

## Passivation

To reduce memory consumption, you may decide to stop entities after some period of inactivity using `Context.SetReceiveTimeout(timeout)`. In order to make cluster sharding aware of stopping entities, **DON'T use `Context.Stop(self)` on the entities**, as this may result in losing messages. Instead send a `Passivate` message message to current entity `Context.Parent` (which is shard itself in this case). This will inform shard to stop forwarding messages to target entity, and buffer them instead until it's terminated. Once that happens, if there are still some messages buffered, entity will be reincarnated and messages flushed to it automatically.

## Remembering entities

By default, when a shard is rebalanced to another node, the entities it stored before migration, are NOT started immediately after. Instead they are recreated ad-hoc, when new messages are incoming. This behavior can be modified by `akka.cluster.sharding.remember-entities = true` configuration. It will instruct shards to keep their state between rebalances - it also comes with extra cost due to necessity of persisting information about started/stopped entities. Additionally a message extractor logic must be aware of `ShardRegion.StartEntity` message:

```
public sealed class ShardEnvelope
{
 public readonly int EntityId;
 public readonly object Message;

 ...
}

public sealed class MessageExtractor : HashCodeMessageExtractor
{
 public MessageExtractor() : base(maxNumberOfShards: 100) { }

 public string EntityId(object message)
 {
 switch(message)
 {
 case ShardEnvelope e: return e.EntityId;
 case ShardRegion.StartEntity start: return start.EntityId;
 }
 }

 public object EntityMessage(object message) => (message as ShardEnvelope)?.Message ?? message;
}
```

Using `ShardRegion.StartEntity` implies, that you're able to infer a shard id given an entity id alone. For this reason, in example above we modified a cluster sharding routing logic to make use of `HashCodeMessageExtractor` - in this variant, shard id doesn't have to be provided explicitly, as it will be computed from the hash of entity id itself. Notice a `maxNumberOfShards`, which is the maximum available number of shards allowed for this type of an actor - this value must never change during a single lifetime of a cluster.

## Retrieving sharding state

You can inspect current sharding stats by using following messages:

- On `GetShardRegionState` shard region will reply with `ShardRegionState` containing data about shards living in the current actor system and what entities are alive on each one of them.
- On `GetClusterShardingStats` shard region will reply with `ClusterShardingStats` having information about shards living in the whole cluster and how many entities alive in each one of them.

## Integrating cluster sharding with persistent actors

One of the most common scenarios, where cluster sharding is used, is to combine them with eventsourced persistent actors from [Akka.Persistence](#) module. However as the entities are incarnated automatically based on provided props, specifying a dedicated, static unique `PersistenceId` for each entity may seem troublesome.

This can be resolved by getting information about shard/entity ids directly from actor's path and constructing unique id from it. For each entity actor path will follow `/user/{typeName}/{shardId}/{entityId}` pattern, where `{typeName}` was the parameter provided to `clusterSharding.Start` method, while `{shardId}` and `{entityId}` were strings returned by

message extractor logic. Given these values we can build consistent, unique `PersistenceId`s on the fly like on the following example:

```
public class Aggregate : PersistentActor
{
 public override string PersistenceId { get; }

 public Aggregate()
 {
 PersistenceId = Context.Parent.Path.Name + "-" + Self.Path.Name;
 }

 ...
}
```

# Distributed data

Akka.DistributedData plugin can be used as in-memory, highly-available, distributed key-value store, where values conform to so called [Conflict-Free Replicated Data Types](#) (CRDT). Those data types can have replicas across multiple nodes in the cluster, where DistributedData plugin has been initialized. We are free to perform concurrent updates on replicas with the same corresponding key without need of coordination (distributed locks or transactions) - all state changes will eventually converge with conflicts being automatically resolved, thanks to the nature of CRDTs. To use distributed data plugin, simply install it via NuGet:

```
install-package Akka.DistributedData -pre
```

Keep in mind, that CRDTs are intended for high-availability, non-blocking read/write scenarios. However they are not a good fit, when you need strong consistency or are operating on big data. If you want to have millions of data entries, this is NOT a way to go. Keep in mind, that all data is kept in memory and, as state-based CRDTs, whole object state is replicated remotely across the nodes, when an update happens. A more efficient implementations (delta-based CRDTs) are considered for the future implementations.

**[!WARNING]** At the present moment, Akka.DistributedData plugin is in state of flux. This means, that its API is unstable and the performance is yet to improve.

## Basic operations

Each CRDT defines few core operations, which are: reads, upserts and deletes. There's no explicit distinction between inserting a value and updating it.

### Reads

To retrieve current data value stored under expected key, you need to send a `Replicator.Get` request directly to a replicator actor reference. You can use `Dsl.Get` helper too:

```
using Akka.DistributedData;
using static Akka.DistributedData.Dsl;

var replicator = DistributedData.Get(system).Replicator;
var key = new ORSetKey<string>("keyA");
var readConsistency = ReadLocal.Instance;

var response = await replicator.Ask<Replicator.IGetResponse>(Get(key, readConsistency));
if (response.IsSuccessfull)
{
 var data = response.Get(key);
}
```

In response, you should receive `Replicator.IGetResponse` message. There are several types of possible responses:

- `GetSuccess` when a value for provided key has been received successfully. To get the value, you need to call `response.Get(key)` with the key, you've sent with the request.
- `NotFound` when no value was found under provided key.
- `GetFailure` when a replicator failed to retrieve value within specified consistency and timeout constraints.
- `DataDeleted` when a value for the provided key has been deleted.

All `Get` requests follows the read-your-own-write rule - if you updated the data, and want to read the state under the same key immediately after, you'll always retrieve modified value, even if the `IGetResponse` message will arrive before `IUpdateResponse`.

## Read consistency

What is a mentioned read consistency? As we said at the beginning, all updates performed within distributed data module will eventually converge. This means, we're not speaking about immediate consistency of a given value across all nodes. Therefore we can precise, what degree of consistency are we expecting:

- `ReadLocal` - we take value based on replica living on a current node.
- `ReadFrom` - value will be merged from states retrieved from some number of nodes, including local one.
- `ReadMajority` - value will be merged from more than a half of cluster nodes replicas (or nodes given a configured role).
- `ReadAll` - before returning, value will be read and merged from all cluster nodes (or the ones with configured role).

## Upserts

To update and replicate the data, you need to send `Replicator.Update` request directly to a replicator actor reference. You can use `Dsl.Update` helper too:

```
using System;
using Akka.Cluster;
using Akka.DistributedData;
using static Akka.DistributedData.Dsl;

var cluster = Cluster.Get(system);
var replicator = DistributedData.Get(system).Replicator;
var key = new ORSetKey<string>("keyA");
var set = ORSet.Create(cluster, "value");
var writeConsistency = new WriteTo(3, TimeSpan.FromSeconds(1));

var response = await replicator.Ask<Replicator.IUpdateResponse>(Update(key, set, writeConsistency, old => old.Merge(set)));
```

Just like in case of reads, there are several possible responses:

- `UpdateSuccess` when a value for provided key has been replicated successfully within provided write consistency constraints.
- `ModifyFailure` when update failed because of an exception within modify function used inside `update` command.
- `UpdateTimeout` when a write consistency constraints has not been fulfilled on time. **Warning:** this doesn't mean, that update has been rolled back! Provided value will eventually propagate its replicas across nodes using gossip protocol, causing the altered state to eventually converge across all of them.
- `DataDeleted` when a value under provided key has been deleted.

You'll always see updates done on local node. When you perform two updates on the same key, second modify function will always see changes done by the first one.

## Write consistency

Just like in case of reads, write consistency allows us to specify level of certainty of our updates before proceeding:

- `WriteLocal` - while value will be disseminated later using gossip, the response will return immediately after local replica update has been acknowledged.

- `WriteTo` - update will immediately be propagated to a given number of replicas, including local one.
- `WriteMajority` - update will propagate to more than a half nodes in a cluster (or nodes given a configured role) before response will be emitted.
- `WriteAll` - update will propagate to all nodes in a cluster (or nodes given a configured role) before response will be emitted.

## Deletes

Any data can be deleted by sending a `Replicator.Delete` request to a local replicator actor reference. You can use `Dsl.Delete` helper too:

```
using Akka.DistributedData;
using static Akka.DistributedData.Dsl;

var replicator = DistributedData.Get(system).Replicator;
var key = new ORSetKey<string>("keyA");
var writeConsistency = WriteLocal.Instance;

var response = await replicator.Ask<Replicator.IDeleteResponse>(Delete(key, writeConsistency))
```

Delete may return one of the 3 responses:

- `DeleteSuccess` when key deletion succeeded within provided consistency constraints.
- `DataDeleted` when data has been deleted already. Once deleted, key can no longer be reused and `DataDeleted` response will be send to all subsequent requests (either reads, updates or deletes). This message will also be used as notification for subscribers.
- `ReplicationDeleteFailure` when operation failed to satisfy specified consistency constraints. **Warning:** this doesn't mean, that delete has been rolled back! Provided operation will eventually propagate its replicas across nodes using gossip protocol, causing the altered state to eventually converge across all of them.

Deletes doesn't specify it's own consistency - it uses the same `IWriteConsistency` interface as updates.

Delete operation doesn't mean, that the entry for specified key has been completely removed. It will still occupy portion of a memory. In case of frequent updates and removals consider to use remove-aware data types such as `ORSet` or `ORDictionary`.

## Subscriptions

You may decide to subscribe to eventual notifications about possible updates by sending `Replicator.Subscribe` message to a local replicator actor reference. All subscribers will be notified periodically (accordingly to `akka.cluster.distributed-data.notify-subscribers-interval` setting, which is 0.5 sec by default). You can also provoke immediate notification of all subscribers by sending `Replicator.FlushChanges` request to the replicator. Example of actor subscription:

```
using System;
using Akka.DistributedData;
using static Akka.DistributedData.Dsl;

class Subscriber : ReceiveActor
{
 public Subscriber()
 {
 var replicator = DistributedData.Get(Context.System).Replicator;
 var key = new ORSetKey<string>("keyA");
 replicator.Tell(Subscribe(key, Self));

 Receive<Replicator.Changed>(changed =>
```

```

 {
 var newValue = changed.Get(key);
 Console.WriteLine($"Received updated value for key '{key}': {newValue}");
 });
}
}

```

All subscribers are removed automatically when terminated. This can be also done explicitly by sending `Replicator.Unsubscribe` message.

## Available replicated data types

Akka.DistributedData specifies several data types, sharing the same `IReplicatedData` interface. All of them share some common members, such as (default) empty value or `Merge` method used to merge two replicas of the same data with automatic conflict resolution. All of those values are also immutable - this means, that any operations, which are supposed to change their state, produce new instance in result:

- `Flag` is a boolean CRDT flag, which default value is always `false`. When a merging replicas have conflicting state, `true` will always win over `false`.
- `GCounter` (also known as growing-only counter) allows only for addition/increment of its state. Trying to add a negative value is forbidden here. Total value of the counter is a sum of values across all of the replicas. In case of conflicts during merge operation, a copy of replica with greater value will always win.
- `PNCounter` allows for both increments and decrements. A total value of the counter is a sum of increments across all replicas decreased by the sum of all decrements.
- `GSet` is an add-only set, which disallows to remove elements once added to it. Merges of GSets are simple unions of their elements. This data type doesn't produce any garbage.
- `ORSet` is implementation of an observed remove add-wins set. It allows to both add and remove its elements any number of times. In case of conflicts when merging replicas, added elements always wins over removed ones.
- `ORDictionary` (also knowns as OR-Map or Observed Remove Map) has similar semantics to OR-Set, however it allows to merge values (which must be CRDTs themselves) in case of concurrent updates.
- `ORMultiDictionary` is a multi-map implementation based on `ORDictionary`, where values are represented as OR-Sets. Use `AddItem` or `RemoveItem` to add or remove elements to the bucket under specified keys.
- `PNCounterDictionary` is a dictionary implementation based on `ORDictionary`, where values are represented as PN-Counters.
- `LWWRegister` (Last Write Wins Register) is a cell for any data type, that implements CRDT semantics. Each modification updates register's timestamp (timestamp generation can be customized, by default it's using UTC date time ticks). In case of merge conflicts, the value with highest update timestamp always wins.
- `LWWDictionary` is a dictionary implementation, which internally uses LLW-Registers to allow to store data of any type to be stored in it. Dictionary entry additions/removals are solved with add-wins semantics, while in-place entry value updates are resolved using last write wins semantics.

Keep in mind, that most of the replicated collections add/remove methods require to provide local instance of the cluster in order to correctly track, to which replica update is originally assigned to.

## Tombstones

One of the issue of CRDTs, is that they accumulate history of changes (including removed elements), producing a garbage, that effectively pile up in memory. While this is still a problem, it can be limited by replicator, which is able to remove data associated with nodes, that no longer exist in the cluster. This process is known as a pruning.

## Settings

There are several different HOCON settings, that can be used to configure distributed data plugin. By default, they all live under `akka.cluster.distributed-data` node:

- `name` of replicator actor. Default: *dDataReplicator*.
- `role` used to limit expected `DistributedData` capability to nodes having that role. None by default.
- `gossip-interval` tells replicator, how often replicas should be gossiped over the cluster. Default: *2 seconds*
- `notify-subscribers-interval` tells, how often replicator subscribers should be notified with replica state changes. Default: *0.5 second*
- `max-delta-elements` limits a maximum number of entries (key-value pairs) to be send in a single gossip information. If there are more modified entries waiting to be gossiped, they will be send in the next round. Default: *1000*
- `use-dispatcher` can be used to specify custom replicator actor message dispatcher. By default it uses an actor system default dispatcher.
- `pruning-interval` tells, how often replicator will check if pruning should be performed. Default: *30 seconds*
- `max-pruning-dissemination` informs, what is the worst expected time for the pruning process to inform whole cluster about pruned node data. Default: *60 seconds*
- `serializer-cache-time-to-live` is used by custom distributed data serializer to determine, for how long serialized replicas should be cached. When sending replica over multiple nodes, it will reuse data already serialized, if it was found in a cache. Default: *10 seconds*.

# Split Brain Resolver

[!NOTE] While this feature is based on [Lightbend Reactive Platform Split Brain Resolver](#) feature description, however its implementation is a result of free contribution and interpretation of Akka.NET team. Lightbend doesn't take any responsibility for the state and correctness of it.

When working with an Akka.NET cluster, you must consider how to handle [network partitions](#) (a.k.a. split brain scenarios) and machine crashes (including .NET CLR/Core and hardware failures). This is crucial for correct behavior of your cluster, especially if you use Cluster Singleton or Cluster Sharding.

## The Problem

One of the common problems present in distributed systems are potential hardware failures. Things like garbage collection pauses, machine crashes or network partitions happen all the time. Moreover it is impossible to distinguish between them. Different cause can have different result on our cluster. A careful balance here is highly desired:

- From one side we may want to detect crashed nodes as fast as possible and remove them from the cluster.
- However, things like network partitions may be only temporary. For this reason it may be more feasible to wait a while for disconnected nodes in hope, that they will be able to reconnect soon.

Networks partitions also bring different problems - the natural result of such event is a risk of splitting a single cluster into two or more independent ones, unaware of each others existence. This comes with certain risks. Even more, some of the Akka.NET cluster features may be unavailable or malfunctioning in such scenario.

To solve this kind of problems we need to determine a common strategy, in which every node will come to the same deterministic conclusion about which node should live and which one should die, even if it won't be able to communicate with others.

Since Akka.NET cluster is working in peer-to-peer mode, it means that there is no single *global* entity which is able to arbitrary define one true state of the cluster. Instead each node has so called failure detector, which tracks the responsiveness and checks health of other connected nodes. This allows us to create a *local* node perspective on the overall cluster state.

In the past the only available opt-in strategy was an auto-down, in which each node was automatically downing others after reaching a certain period of unreachability. While this approach was enough to react on machine crashes, it was failing in face of network partitions: if cluster was split into two or more parts due to network connectivity issues, each one of them would simply consider others as down. This would lead to having several independent clusters not knowing about each other. It is especially disastrous in case of Cluster Singleton and Cluster Sharding features, both relying on having only one actor instance living in the cluster at the same time.

Split brain resolver feature brings ability to apply different strategies for managing node lifecycle in face of network issues and machine crashes. It works as a custom downing provider. Therefore in order to use it, **all of your Akka.NET cluster nodes must define it with the same configuration**. Here's how minimal configuration looks like:

```
akka.cluster {
 downing-provider-class = "Akka.Cluster.SplitBrainResolver, Akka.Cluster"
 split-brain-resolver {
 active-strategy = <your-strategy>
 }
}
```

Keep in mind that split brain resolver will NOT work when `akka.cluster.auto-down-unreachable-after` is used.

# Strategies

This section describes the different split brain resolver strategies. Please keep in mind, that there's no universal solution and each one of them may fail under specific circumstances.

To decide which strategy to use, you can set `akka.cluster.split-brain-resolver.active-strategy` to one of 4 different options:

- `static-quorum`
- `keep-majority`
- `keep-oldest`
- `keep-referee`

All strategies will be applied only after cluster state has reached stability for specified time threshold (no nodes transitioning between different states for some time), specified by `stable-after` setting. Nodes which are joining will not affect this threshold, as they won't be promoted to UP status in face unreachable nodes. For the same reason they won't be taken into account, when a strategy will be applied.

```
akka.cluster.split-brain-resolver {
 # Enable one of the available strategies (see descriptions below):
 # static-quorum, keep-majority, keep-oldest, keep-referee
 active-strategy = off

 # Decision is taken by the strategy when there has been no membership or
 # reachability changes for this duration, i.e. the cluster state is stable.
 stable-after = 20s
}
```

There is no simple way to decide the value of `stable-after`, as shorter value will give you the faster reaction time for unreachable nodes at cost of higher risk of false failures detected - due to temporary network issues. The rule of thumb for this setting is to set `stable-after` to `log10(maxExpectedNumberOfNodes) * 10`.

Remember that if a strategy will decide to down a particular node, it won't shutdown the related `ActorSystem`. In order to do so, use cluster removal hook like this:

```
Cluster.Get(system).RegisterOnMemberRemoved(() => {
 system.Terminate().Wait();
});
```

## Static Quorum

The `static-quorum` strategy works well, when you are able to define minimum required cluster size. It will down unreachable nodes if the number of reachable ones is greater than or equal to a configured `quorum-size`. Otherwise reachable ones will be downed.

When to use it? When you have a cluster with fixed size of nodes or fixed size of nodes with specific role.

Things to keep in mind:

1. If cluster will split into more than 2 parts, each one smaller than the `quorum-size`, this strategy may bring down the whole cluster.
2. If the cluster will grow 2 times beyond `quorum-size`, there is still a potential risk of having cluster splitting into two if a network partition will occur.
3. If during cluster initialization some nodes will become unreachable, there is a risk of putting the cluster down - since strategy will apply before cluster will reach quorum size. For this reason it's a good thing to define `akka.cluster.min-nr-of-members` to a higher value than actual `quorum-size`.

4. Don't forget to add new nodes back once some of them were removed.

This strategy can work over a subset of cluster nodes by defining a specific `role`. This is useful when some of your nodes are more important than others and you can prioritize them during quorum check. You can also use it to to configure a "core" set of nodes, while still being free grow your cluster over initial limit. Of course this will leave your cluster more vulnerable in situation where those "core" nodes will fail.

Configuration:

```
akka.cluster.split-brain-resolver {
 active-strategy = static-quorum

 static-quorum {
 # minimum number of nodes that the cluster must have
 quorum-size = undefined

 # if the 'role' is defined the decision is based only on members with that 'role'
 role = ""
 }
}
```

## Keep Majority

The `keep-majority` strategy will down this part of the cluster, which sees a lesser part of the whole cluster. This choice is made based on the latest known state of the cluster. When cluster will split into two equal parts, the one which contains the lowest address, will survive.

When to use it? When your cluster can grow or shrink very dynamically.

Keep in mind, that:

1. Two parts of the cluster may make their decision based on the different state of the cluster, as it's relative for each node. In practice, the risk of it is quite small.
2. If there are more than 2 partitions, and none of them has reached the majority, the whole cluster may go down.
3. If more than half of the cluster nodes will go down at once, the remaining ones will also down themselves, as they didn't reached the majority (based on the last known cluster state).

Just like in the case of static quorum, you may decide to make decisions based only on a nodes having configured `role`. The advantages here are similar to those of the static quorum.

Configuration:

```
akka.cluster.split-brain-resolver {
 active-strategy = keep-majority

 keep-majority {
 # if the 'role' is defined the decision is based only on members with that 'role'
 role = ""
 }
}
```

## Keep Oldest

The `keep-oldest` strategy, when a network split has happened, will down a part of the cluster which doesn't contain the oldest node.

When to use it? This approach is particularly good in combination with Cluster Singleton, which usually is running on the oldest cluster member. It's also useful, when you have a one starter node configured as `akka.cluster.seed-nodes` for others, which will still allow you to add and remove members using its address.

Keep in mind, that:

1. When the oldest node will get partitioned from others, it will be downed itself and the next oldest one will pick up its role. This is possible thanks to `down-if-alone` setting.
2. If `down-if-alone` option will be set to `off`, a whole cluster will be dependent on the availability of this single node.
3. There is a risk, that if partition will split cluster into two unequal parts i.e. 2 nodes with the oldest one present and 20 remaining ones, the majority of the cluster will go down.
4. Since the oldest node is determined on the latest known state of the cluster, there is a small risk that during partition, two parts of the cluster will both consider themselves having the oldest member on their side. While this is very rare situation, you still may end up having two independent clusters after split occurrence.

Just like in previous cases, a `role` setting can be used to determine the oldest member across all having specified role.

Configuration:

```
akka.cluster.split-brain-resolver {
 active-strategy = keep-oldest

 keep-oldest {
 # Enable downing of the oldest node when it is partitioned from all other nodes
 down-if-alone = on

 # if the 'role' is defined the decision is based only on members with that 'role',
 # i.e. using the oldest member (singleton) within the nodes with that role
 role = ""
 }
}
```

## Keep Referee

The `keep-referee` strategy will simply down the part that does not contain the given referee node.

When to use it? If you have a single node which is running processes crucial to existence of the entire cluster.

Things to keep in mind:

1. With this strategy, cluster will never split into two independent ones, under any circumstances.
2. A referee node is a single point of failure for the cluster.

You can configure a minimum required amount of reachable nodes to maintain operability by using `down-all-if-less-than-nodes`. If a strategy will detect that the number of reachable nodes will go below that minimum it will down the entire partition even when referee node was reachable.

Configuration:

```
akka.cluster.split-brain-resolver {
 active-strategy = keep-referee

 keep-referee {
 # referee address on the form of "akka.tcp://system@hostname:port"
 address = ""
 down-all-if-less-than-nodes = 1
 }
}
```

## Relation to Cluster Singleton and Cluster Sharding

Cluster singleton actors and sharded entities of cluster sharding have their lifecycle managed automatically. This means that there can be only one instance of a target actor at the same time in the cluster, and when detected dead, it will be resurrected on another node. However it's important the the old instance of the actor must be stopped before new one will be spawned, especially when used together with Akka.Persistence module. Otherwise this may result in corruption of actor's persistent state and violate actor state consistency.

Since different nodes may apply their split brain decisions at different points in time, it may be good to configure a time margin necessary to make sure, that other nodes will get enough time to apply their strategies. This can be done using `akka.cluster.down-removal-margin` setting. The shorter it is, the faster reaction time of your cluster will be. However it will also increase the risk of having multiple singleton/sharded entity instances at the same time. It's recommended to set this value to be equal `stable-after` option described above.

## Expected Failover Time

If you're going to use a split brain resolver, you can see that the total failover latency is determined by several values. Defaults are:

- failure detection 5 seconds
- `akka.cluster.split-brain-resolver.stable-after` 20 seconds
- `akka.cluster.down-removal-margin` 20 seconds

This would result in total failover time of 45 seconds. While this value is good for the cluster of 100 nodes, you may decide to lower those values in case of a smaller one i.e. cluster of 20 nodes could work well with timeouts of 13s, which would reduce total failover time to 31 seconds.

# EventBus

## Subscribing to Dead letter messages

The following example demonstrates the capturing of dead letter messages generated from a stopped actor. The dedicated actor will output the message, sender and recipient of the captured dead letter to the console.

```
void Main()
{
 // Setup the actor system
 ActorSystem system = ActorSystem.Create("MySystem");

 // Setup an actor that will handle deadletter type messages
 var deadletterWatchMonitorProps = Props.Create(() => new DeadletterMonitor());
 var deadletterWatchActorRef = system.ActorOf(deadletterWatchMonitorProps, "DeadLetterMonitoringActor");

 // subscribe to the event stream for messages of type "DeadLetter"
 system.EventStream.Subscribe(deadletterWatchActorRef, typeof(DeadLetter));

 // Setup an actor which will simulate a failure/shutdown
 var expendableActorProps = Props.Create(() => new ExpendableActor());
 var expendableActorRef = system.ActorOf(expendableActorProps, "ExpendableActor");

 // simulate the expendable actor failing/stopping
 expendableActorRef.Tell(Akka.Actor.PoisonPill.Instance);

 // try sending a message to the stopped actor
 expendableActorRef.Tell("another message");
}

// A dead letter handling actor specifically for messages of type "DeadLetter"
public class DeadletterMonitor : ReceiveActor
{
 public DeadletterMonitor()
 {
 Receive<DeadLetter>(dl => HandleDeadletter(dl));
 }

 private void HandleDeadletter(DeadLetter dl)
 {
 Console.WriteLine($"DeadLetter captured: {dl.Message}, sender: {dl.Sender}, recipient: {dl.Recipient}")
 }
}

// An expendable actor which will simulate a failure
public class ExpendableActor : ReceiveActor { }
```

sample capture

```
DeadLetter captured: another message, sender: [akka://MySystem/deadLetters], recipient: [akka://MySystem/user/ExpendableActor#1469246785]
```

## Subscribing to messages of type "string"

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

```
var subscriber = system.ActorOf<SomeActor>();
//Subscribe to messages of type string
system.EventStream.Subscribe(subscriber,typeof(string));
//send a message
system.EventStream.Publish("hello"); //this will be forwarded to subscriber
```

## Further reading

- Read more on <http://doc.akka.io/docs/akka/snapshot/java/event-bus.html>

# Logging

For more info see real Akka's documentation: <http://doc.akka.io/docs/akka/2.0/scala/logging.html>

## How to Log

To log in an actor, create a logger and assign it to a private field:

```
private readonly ILoggingAdapter _log = Logging.GetLogger(Context);
```

Use the `Debug`, `Info`, `Warning` and `Error` methods to log.

```
_log.Debug("Some message");
```

## Standard Loggers

Akka.NET comes with two built in loggers.

- **StandardOutLogger**
- **BusLogging**

## Contrib Loggers

These loggers are also available as separate nuget packages

- **Akka.Logger.slf4net** which logs using [slf4net](#)
- **Akka.Logger.Serilog** which logs using [serilog](#). See [Detailed instructions on using Serilog](#).
- **Akka.Logger.NLog** which logs using [NLog](#)

Note that you need to modify the config as explained below.

## NLog Configuration

Example NLog configuration inside your app.config or web.config:

```
akka {
 loggers = ["Akka.Logger.NLog.NLogLogger, Akka.Logger.NLog"]
}
```

The above NLog components can be found on Nuget (<https://www.nuget.org/packages/Akka.Logger.NLog/>)

## Configuring Custom Loggers

To configure a custom logger inside your Akka.Config, you need to use a fully qualified .NET class name like this:

```
akka {
 loggers = ["NameSpace.ClassName, AssemblyName"]
}
```

## Logging Unhandled messages

It is possible to configure akka so that Unhandled messages are logged as Debug log events for debug purposes. This can be achieved using the following configuration setting:

```
akka {
 actor.debug.unhandled = on
}
```

## Example configuration

```
akka {
 stdout-loglevel = DEBUG
 loglevel = DEBUG
 log-config-on-start = on
 actor {
 debug {
 receive = on
 autoreceive = on
 lifecycle = on
 event-stream = on
 unhandled = on
 }
 }
}
```

# Using Serilog

## Setup

Install the package **Akka.Logger.Serilog** via nuget to utilize [Serilog](#), this will also install the required Serilog package dependencies.

```
PM> Install-Package Akka.Logger.Serilog
```

## Example

The following example uses Serilog's **Colored Console** sink available via nuget, there are wide range of other sinks available depending on your needs, for example a rolling log file sink. See serilog's documentation for details on these.

```
PM> Install-Package Serilog.Sinks.ColoredConsole
```

Next, you'll need to configure the global `Log.Logger` and also specify to use the logger in the config when creating the system, for example like this:

```
var logger = new LoggerConfiguration()
 .WriteTo.ColoredConsole()
 .MinimumLevel.Information()
 .CreateLogger();

Serilog.Log.Logger = logger;

var system = ActorSystem.Create("my-test-system", "akka { loglevel=INFO, loggers=[\"Akka.Logger.Serilog.Slf4jLogger, Akka.Logger.Serilog\"]}");
```

## Logging

To log inside an actor, using the normal `string.Format()` syntax, get the logger and log:

```
var log = Context.GetLogger();
...
log.Info("The value is {0}", counter);
```

Or alternatively

```
var log = Context.GetLogger();
...
log.Info("The value is {Counter}", counter);
```

## Extensions

The package **Akka.Logger.Serilog** also includes the extension method `ForContext()` for `ILoggingAdapter` (the object returned by `Context.GetLogger()`). This is analogous to Serilog's `ForContext()` but instead of returning a Serilog `ILogger` it returns an Akka.NET `ILoggingAdapter`. This instance acts as contextual logger that will attach a property to all events logged through it.

However, in order to use it, the parent `ILoggingAdapter` must be constructed through another included generic extension method for `GetLogger()`. For example,

```
using Akka.Logger.Serilog;
...
private readonly ILoggingAdapter _logger = Context.GetLogger<SerilogLoggingAdapter>();
...
private void ProcessMessage(string correlationId)
{
 var contextLogger = _logger.ForContext("CorrelationId", correlationId);
 contextLogger.Info("Processing message");
}
```

If the configured output template is, for example, `"[{CorrelationId}] {Message}{NewLine}"`, and the parameter `correlationId` is `"1234"` then the resulting log would contain the line `[1234] Processing message`.

```
// configure sink with an output template
var logger = new LoggerConfiguration()
 .WriteTo.ColoredConsole(outputTemplate: "[{CorrelationId}] {Message}{NewLine}")
 .MinimumLevel.Information()
 .CreateLogger();
```

## HOCON configuration

In order to be able to change log level without the need to recompile, we need to employ some sort of application configuration. To use Serilog via HOCON configuration, add the following to the **App.config** of the project.

```
<configSections>
 <section name="akka" type="Akka.Configuration.Hocon.AkkaConfigurationSection, Akka" />
</configSections>

...

<akka>
 <hocon>
 <![CDATA[
 akka {
 loglevel=INFO,
 loggers=["Akka.Logger.Serilog.SerilogLogger, Akka.Logger.Serilog"]
 }
]]>
 </hocon>
</akka>
```

The code can then be updated as follows removing the inline HOCON from the actor system creation code. Note in the following example, if a minimum level is not specified, Information level events and higher will be processed. Please read the documentation for [Serilog](#) configuration for more details on this. It is also possible to move serilog configuration to the application configuration, for example if using a rolling log file sink, again, browsing the serilog documentation is the best place for details on that feature.

```
var logger = new LoggerConfiguration()
 .WriteTo.ColoredConsole()
 .MinimumLevel.Information()
```

```
.CreateLogger();
Serilog.Log.Logger = logger;
var system = ActorSystem.Create("my-test-system");
```

# Scheduler

Sometimes the need for making things happen in the future arises, and where do you go look then? Look no further than `ActorSystem`! There you find the `Scheduler` property that returns an instance of `Akka.Actor.Scheduler`, this instance is unique per `ActorSystem` and is used internally for scheduling things to happen at specific points in time.

You can schedule sending of messages to actors and execution of tasks (`Action` delegate). You will get a `Task` back. By providing a `CancellationTokenSource` you can cancel the scheduled task.

The scheduler in Akka is designed for high-throughput of thousands up to millions of triggers. The prime use-case being triggering Actor receive timeouts, Future timeouts, circuit breakers and other time dependent events which happen all-the-time and in many instances at the same time. The implementation is based on a Hashed Wheel Timer, which is a known data structure and algorithm for handling such use cases, refer to the [Hashed and Hierarchical Timing Wheels](#) whitepaper by Varghese and Lauck if you'd like to understand its inner workings.

The Akka scheduler is **not** designed for long-term scheduling (see [Akka.Quartz.Actor](#) instead for this use-case) nor is it to be used for highly precise firing of the events. The maximum amount of time into the future you can schedule an event to trigger is around 8 months, which in practice is too much to be useful since this would assume the system never went down during that period. If you need long-term scheduling we highly recommend looking into alternative schedulers, as this is not the use-case the Akka scheduler is implemented for.

[!WARNING] The default implementation of Scheduler used by Akka is based on job buckets which are emptied according to a fixed schedule. It does not execute tasks at the exact time, but on every tick, it will run everything that is (over)due. The accuracy of the default Scheduler can be modified by the `akka.scheduler.tick-duration` configuration property.

## Some examples

```
var system = ActorSystem.Create("MySystem");
var someActor = system.ActorOf<SomeActor>("someActor");
var someMessage = new SomeMessage() {...};
system
 .Scheduler
 .ScheduleTellRepeatedly(TimeSpan.FromSeconds(0),
 TimeSpan.FromSeconds(5),
 someActor, someMessage, ActorRefs.NoSender); //or ActorRefs.Nobody or something else
```

The above example will schedule "someMessage" to be sent to "someActor" every 5 seconds.

[!WARNING] If you schedule a closure you should be extra careful to not pass or close over unstable references. In practice this means that you should not call methods on the enclosing actor from within the closure. If you need to schedule an invocation it is better to use the `Schedule()` variant accepting a message and an `IActorRef` to schedule a message to self (containing the necessary parameters) and then call the method when the message is received.

## From inside the actor

```
Context.System.Scheduler.ScheduleTellRepeatedly(...);
```

[!WARNING] All scheduled task will be executed when the `ActorSystem` is terminated. i.e. the task may execute before its timeout.

## The scheduler interface

The actual scheduler implementation is defined by config and loaded upon ActorSystem start-up, which means that it is possible to provide a different one using the `akka.scheduler.implementation` configuration property. The referenced class must implement the `Akka.Actor.IScheduler` and `Akka.Actor.IAdvancedScheduler` interfaces

## The cancellable interface

Scheduling a task will result in a `ICancellable` or (or throw an `Exception` if attempted after the scheduler's shutdown). This allows you to cancel something that has been scheduled for execution.

[!WARNING] this does not abort the execution of the task, if it had already been started.

# Circuit Breaker

## Why are they used?

A circuit breaker is used to provide stability and prevent cascading failures in distributed systems. These should be used in conjunction with judicious timeouts at the interfaces between remote systems to prevent the failure of a single component from bringing down all components.

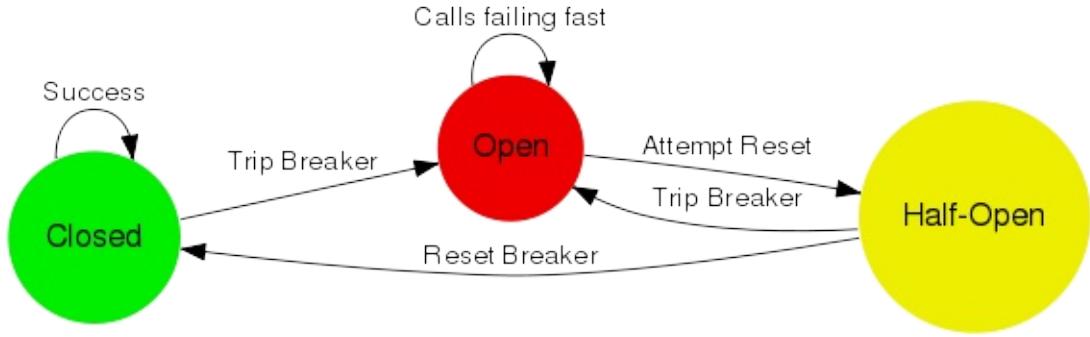
As an example, we have a web application interacting with a remote third party web service. Let's say the third party has oversold their capacity and their database melts down under load. Assume that the database fails in such a way that it takes a very long time to hand back an error to the third party web service. This in turn makes calls fail after a long period of time. Back to our web application, the users have noticed that their form submissions take much longer seeming to hang. Well the users do what they know to do which is use the refresh button, adding more requests to their already running requests. This eventually causes the failure of the web application due to resource exhaustion. This will affect all users, even those who are not using functionality dependent on this third party web service.

Introducing circuit breakers on the web service call would cause the requests to begin to fail-fast, letting the user know that something is wrong and that they need not refresh their request. This also confines the failure behavior to only those users that are using functionality dependent on the third party, other users are no longer affected as there is no resource exhaustion. Circuit breakers can also allow savvy developers to mark portions of the site that use the functionality unavailable, or perhaps show some cached content as appropriate while the breaker is open.

The Akka.NET library provides an implementation of a circuit breaker called `Akka.Pattern.CircuitBreaker` which has the behavior described below.

## What do they do?

- During normal operation, a circuit breaker is in the `Closed` state:
  - Exceptions or calls exceeding the configured `callTimeout` increment a failure counter
  - Successes reset the failure count to zero
  - When the failure counter reaches a `MaxFailures` count, the breaker is tripped into `Open` state
- While in `Open` state:
  - All calls fail-fast with a `CircuitBreakerOpenException`
  - After the configured `ResetTimeout`, the circuit breaker enters a `Half-Open` state
- In `Half-Open` state:
  - The first call attempted is allowed through without failing fast
  - All other calls fail-fast with an exception just as in `Open` state
  - If the first call succeeds, the breaker is reset back to `Closed` state
  - If the first call fails, the breaker is tripped again into the `Open` state for another full `ResetTimeout`
- State transition listeners:
  - Callbacks can be provided for every state entry via `onOpen`, `onClose`, and `onHalfOpen`
  - These are executed in the `ExecutionContext` provided.



## Examples

### Initialization

Here's how a `CircuitBreaker` would be configured for:

- 5 maximum failures
- a call timeout of 10 seconds
- a reset timeout of 1 minute

[!code-csharpMain]

```

public class DangerousActor : ReceiveActor
{
 private readonly ILoggingAdapter _log = Context.GetLogger();

 public DangerousActor()
 {
 var breaker = new CircuitBreaker(
 maxFailures: 5,
 callTimeout: TimeSpan.FromSeconds(10),
 resetTimeout: TimeSpan.FromMinutes(1))
 .OnOpen(NotifyMeOnOpen);
 }

 private void NotifyMeOnOpen()
 {
 _log.Warning("My CircuitBreaker is now open, and will not close for one minute");
 }
}

```

### Call Protection

Here's how the `CircuitBreaker` would be used to protect an asynchronous call as well as a synchronous one:

[!code-csharpMain]

```

public class DangerousActorCallProtection : ReceiveActor
{
 private readonly ILoggingAdapter _log = Context.GetLogger();

 public DangerousActorCallProtection()
 {
 var breaker = new CircuitBreaker(
 maxFailures: 5,
 callTimeout: TimeSpan.FromSeconds(10),
 resetTimeout: TimeSpan.FromMinutes(1))
 .OnOpen(NotifyMeOnOpen);
 }

 protected override void OnReceive(ActorMessage message)
 {
 if (message is IAsyncRequest request)
 {
 breaker.Protect(() => request.Respond("Hello"));
 }
 else
 {
 breaker.Protect(() => Respond("Hello"));
 }
 }
}

```

```
var dangerousCall = "This really isn't that dangerous of a call after all";

Receive<string>(str => str.Equals("is my middle name"), msg =>
{
 breaker.WithCircuitBreaker(() => Task.FromResult(dangerousCall)).PipeTo(Sender);
});

Receive<string>(str => str.Equals("block for me"), msg =>
{
 Sender.Tell(breaker.WithSyncCircuitBreaker(() => dangerousCall));
});
}

private void NotifyMeOnOpen()
{
 _log.Warning("My CircuitBreaker is now open, and will not close for one minute");
}
}
```

## Modules marked "May Change"

To be able to introduce new modules and APIs without freezing them the moment they are released we have introduced the term **may change**.

Concretely **may change** means that an API or module is in early access mode and that it:

- is not guaranteed to be binary compatible in minor releases
- may have its API change in breaking ways in minor releases
- may be entirely dropped from Akka in a minor release

Complete modules can be marked as **may change**, this will can be found in their module description and in the docs.

Individual public APIs can be annotated with `Akka.Annotations.ApiMayChange` to signal that it has less guarantees than the rest of the module it lives in. Please use such methods and classes with care, however if you see such APIs that is the best point in time to try them out and provide feedback (e.g. using the akka-user mailing list, GitHub issues or Gitter) before they are frozen as fully stable API.

Best effort migration guides may be provided, but this is decided on a case-by-case basis for **may change** modules.

The purpose of this is to be able to release features early and make them easily available and improve based on feedback, or even discover that the module or API wasn't useful.

These are the current complete modules marked as **may change**:

## Akka Configuration

Below is the default HOCON configuration for the base `Akka` package.

[!codeAkka.dll HOCON Configuration]

## Akka.Remote Configuration

Below is the default HOCON configuration for the base `Akka.Remote` package.

[!codeAkka.Remote.dll HOCON Configuration]

## Akka.Cluster Configuration

Below is the default HOCON configuration for the base `Akka.Cluster` package.

[!codeAkka.Cluster.dll HOCON Configuration]

## Akka.Persistence Configuration

Below is the default HOCON configuration for the base `Akka.Persistence` package.

[!codeAkka.Persistence.dll HOCON Configuration]

## Akka.Streams Configuration

Below is the default HOCON configuration for the base `Akka.Streams` package.

[!codeAkka.Streams.dll HOCON Configuration]

## Akka.TestKit Configuration

Below is the default HOCON configuration for the base `Akka.TestKit` package.

[!codeAkka.TestKit.dll HOCON Configuration]

Additionally, it's also possible to change the default `IScheduler implementation` in the `Akka.TestKit` to use a `virtualized TestScheduler implementation` that Akka.NET developers can use to artificially advance time forward. To swap in the `TestScheduler`, developers will want to include the HOCON below:

[!codeAkka.TestKit.dll TestScheduler HOCON Configuration]

## Books

### Akka/Akka.NET

- [Reactive Web Applications](#) (Manuel Bernhardt, June 2016)
- [Akka in action](#) (Raymond Roestenburg, Rob Bakker, and Rob Williams, September 2016)
- [Reactive Applications with Akka.NET](#) (Anthony Brown, **Not released yet**, MEAP began January 2016 Publication in Summer 2017)

### Reactive programming

- [Reactive Messaging Patterns with the Actor Model](#) (Vaughn Vernon, July 2015)
- [Functional Reactive Programming](#) (Stephen Blackheath and Anthony Jones, July 2016)
- [Functional and Reactive Domain Modeling](#) (Debasish Ghosh, October 2016)
- [Reactive Design Patterns](#) (Roland Kuhn with Brian Hanafee and Jamie Allen, February 2017)
- [Rx.NET in Action](#) (Tamir Dresher, **Not released yet**, MEAP began July 2015 Publication in March 2017)
- [Reactive Application Development](#) (Duncan K. DeVore, Sean Walsh, and Brian Hanafee, **Not released yet**, MEAP began December 2014 Publication in Summer 2017)

Akka.Net has an official [NuGet package](#).

To install Akka.NET, run the following command in the Package Manager Console:

```
PM> Install-Package Akka -Pre
```

You can also build it locally from the source code.

## Building Akka.NET with Fake

The build has been ported to [Fake](#) to make it even easier to compile.

Clone the source code from GitHub (currently only on the dev branch):

```
git clone https://github.com/akkadotnet/akka.net.git -b dev
```

## Running build task

There is no need to install anything specific before running the build.

Once in the directory, run the build.cmd with the target All:

```
build all
```

The `all` targets runs the following targets in order:

- Build
- Test
- Nuget

## Version management

The build uses the last version number specified in the [RELEASE\\_NOTES.md](#) file.

The release notes are also used in nuget packages.

## Running tests

To run unit tests from the command line, run the following command:

```
build test
```

## Running MultiNodeTests

To run the multiple node specifications from the command line, run the following command:

```
build multinodetests
```

To run the multinode specifications for a subset of akka, you can supply a filter:

```
build multinodetests spec-assembly=<filter>
```

For example to run only the specifications for Akka.Remote the command would be:

```
build multinodetests spec-assembly=remote
```

## Creating Nuget distributions

To create nuget packages locally, run the following command:

```
build nuget
```

To create and publish packages to nuget.org, specify the nuget key:

```
build nuget nugetkey=<key>
```

or to run also unit tests before publishing:

```
build all nugetkey=<key>
```

## Detailed Help from command line

The command line supplies some detailed help on the usage of build.

```
build help
build helpnuget
build helpdocs
build helpmultinodetests
```

# Resources

## Akka.NET Bootcamp

[Akka.NET Bootcamp](#) is a free, self-directed learning course brought to you by the folks at [Petabridge](#). Over the three units of this bootcamp you will learn how to create fully-functional, real-world programs using Akka.NET actors and many other parts of the core Akka.NET framework.

[Start Bootcamp here.](#)

## Blog posts

### Petabridge

- [Akka.NET: What is an Actor?](#) (Aaron Stannard on January 25, 2015)
- [How to Do Asynchronous I/O with Akka.NET Actors Using PipeTo](#) (Aaron Stannard on January 27, 2015)
- [How actors recover from failure](#) (Andrew Skotzko on February 6, 2015)
- [Akka.NET Internals: How Akka.Remote Connections Work](#) (Andrew Skotzko on May 6, 2015)
- [When Should I Use Actor Selection?](#) (Andrew Skotzko on May 20, 2015)
- [Akka.NET: How to Remotely Deploy Actors Using Akka.Remote](#) (Aaron Stannard on June 1, 2015)
- [How to Create Scalable Clustered Applications Using Akka.Cluster](#) (June 13, 2015)
- [Meet the Top Akka.NET Design Patterns](#) (Andrew Skotzko on June 30, 2015)
- [Large Messages and Sockets in Akka.NET](#) (Andrew Skotzko on July 15, 2015)
- [Akka.NET Goes to Wall Street](#) (Andrew Skotzko on August 11, 2015)
- [How to Integrate Akka.NET and ASP.NET \(and Nancy!\)](#) (Aaron Stannard on August 20, 2015)
- [The Top 7 Mistakes Newbies Make with Akka.NET](#) (Aaron Stannard on September 7, 2015)
- [How to Stop an Actor... the Right Way](#) (Andrew Skotzko on September 9, 2015)
- [The New .NET Stack](#) (Aaron Stannard on September 23, 2015)
- [How to Unit Test Akka.NET Actors with Akka.TestKit](#) (November 13, 2015)
- [Creating Persistent Actors in Akka.NET with Akka.Persistence](#) (Aaron Stannard on January 7, 2016)
- [How to Guarantee Delivery of Messages in Akka.NET](#) (Aaron Stannard on March 11, 2016)
- [The Business Case for Actors and Akka.NET](#) (Aaron Stannard on May 10, 2016)
- [Distributing State in Akka.Cluster Applications](#) (Aaron Stannard on July 26, 2016)
- [Designing Akka.NET Applications from Scratch Part 1: Go with the Flow](#) (Aaron Stannard on September 1, 2016)
- [Designing Akka.NET Applications from Scratch Part 2: Hierarchies and SOLID Principles](#) (Aaron Stannard on November 30, 2016)
- [Introduction to Akka.Cluster.Sharding in Akka.NET](#) (Bartosz Syptykowski on January 17, 2017)
- [Technical Overview of Akka.Cluster.Sharding in Akka.NET](#) (Bartosz Syptykowski on January 31, 2017)
- [Introduction to Distributed Publish-Subscribe in Akka.NET](#) (Bartosz Syptykowski on February 14, 2017)
- [Introducing Petabridge.Cmd - a Command-line Management Tool for Akka.NET Applications](#) (Aaron Stannard on June 7, 2017)

### Others

- [Deploying actors with Akka.NET](#) (Roger Johansson on March 9, 2014)
- [FSharp and Akka.net - the functional way](#) (Bartosz Syptykowski on July 5th, 2014)
- [Map reduce with FSharp and Akka.net](#) (Bartosz Syptykowski on July 8th, 2014)
- [Actor supervisors in Akka.NET FSharp API](#) (Bartosz Syptykowski on August 6th, 2014)
- [Hipsterize your backend for The Greater Good with Akka.NET, F# and some DDD flavor](#) (Bartosz Syptykowski on

October 26th, 2014)

- Akka.NET – Concurrency control (Roger Johansson on November 10th, 2014)
- Akka.NET remote deployment with F# (Bartosz Sypytkowski on December 14th, 2014)
- An Actor Model Example with Akka.NET (Claus Sørensen on February 24th, 2015)
- Create your own Akka.NET persistence plugin (Bartosz Sypytkowski on March 28th, 2015)
- Starting Akka.NET (James Conway on April 7, 2015)
- Akka.NET + Azure: Azure ServiceBus integration (Roger Johansson on April 13, 2015)
- Akka.NET application logging in your browser (Bartosz Sypytkowski on July 22nd, 2015)
- Building a framework – The early Akka.NET history (Roger Johansson on July 26, 2015)
- How to create an Akka.NET cluster in F# (Bartosz Sypytkowski on August 8th, 2015)
- How Akka.NET persistence works? (Bartosz Sypytkowski on September 12th, 2015)
- Akka.NET underestimated features - Akka.IO (Bartosz Sypytkowski on November 14th, 2015)
- Random things learned building Akka.NET – Part 1 (Roger Johansson on March 13, 2016)
- Don't Ask, Tell (Bartosz Sypytkowski on May 3rd, 2016)
- Monitoring Akka.NET Systems with Datadog (Creg Shackles on May 25th, 2017)
- Actor model and using of Akka.NET (Nikola Živković on May 28th, 2017)

## Videos

- The Actor Model in F# and Akka.Net (March 17, 2015)
- Streaming ETL w/ Akka.NET (Andrew Skotzko on March 18, 2015)
- Akka.NET Internals: How Akka.Remote Connections Work (Aaron Stannard on May 5, 2015)
- Full-Stack, Message-Oriented Programming with Akka.NET Actors (Andrew Skotzko on 8th December, 2015)
- Akka.NET: The Future of Distributed Programming in .NET (Aaron Stannard on August 31, 2016)
- Life with actors: experience report (Vagif Abilov and Erlend Wiig on February 16th, 2017)
- Akka.NET: The Future of Distributed Programming in .NET (Aaron Stannard on June 8th, 2017)
- Easy Eventual Consistency with Actor Models + Amazon Web Services (Philip Laureano on July 5th, 2017)
- Composing high performance process workflows with Akka Streams (Vagif Abilov on July 5th, 2017)
- Getting real(time) with Akka.NET, React and Redux (Francis Paulin on July 5th, 2017)
- Channel 9: Building distributed applications with Akka.NET (Aaron Stannard on April 3, 2018)

## Courses

- Building Concurrent Applications with the Actor Model in Akka.NET (Pluralsight) (Jason Roberts on 5 August 2015)
- Implementing Logging and Dependency Injection in Akka.NET (Jason Roberts on 22 August 2015)
- Building Reactive Concurrent WPF Applications with Akka.NET (Jason Roberts on 10 September 2015)
- Improving Message Throughput in Akka.NET (Jason Roberts on 1 October 2015)
- Stateful Reactive Concurrent SPAs with SignalR and Akka.NET (Jason Roberts on 4 November 2015)
- Akka.NET Testing Fundamentals (Jason Roberts on 8 December 2015)
- Building Distributed Systems with Akka.NET Clustering (Simon Anderson on 23 March 2016)
- Akka.NET Persistence Fundamentals (Jason Roberts on 23 Jul 2016)

## Podcasts

- .NET Rocks! (August 2016) — Aaron Stannard introduces Akka.NET v1.1
- .NET Rocks! (May 2015) — Aaron Stannard introduces Akka.NET v1.0
- Hanselminutes (April 2015) — Good overview of concepts in Akka.NET and high-level discussion with Aaron Stannard.

- [.NET Rocks! \(November 2014\) — Overview of the project discussed w/ Roger Alsing.](#)

## Code samples / Demos

- [Using Akka.Cluster to build a webcrawler](#)

## Non-English resources

- [Distributed Programming Using Akka.NET Framework \(in Polish\)](#) (Bartosz Syptykowski on March 6, 2015)
- [Intro to Akka.NET \(in Swedish\)](#) (Håkan Canberger on March 23, 2015)
- [Writing scalable and distributed systems with Akka.NET](#) (Nikita Tsukanov on June 5, 2015)
- [Actor Model on .NET \(in Russian\)](#) (Anton Moldovan on October 30, 2015)
- [Actor-based Concurrency with F# and Akka.NET \(in Russian\)](#) (Akim Boyko on 19 December, 2015)

# Making Public API Changes

Akka.NET follows the [semantic versioning methodology](#), and as such the most important convention we have to be mindful of is accurately communicating to our users whether or not Akka.NET is compatible with previous versions of the API.

As such, we have automated procedures designed to ensure that accidental breaking / incompatible changes to the Akka.NET public API can't sail through the pull request process without some human acknowledgement first.

This document outlines how to comply with said procedures.

## API Approvals

The goal of this process is to make conscious decisions about API changes and force the discovery of those changes during the pull request review. Here is how the process works:

- Uses [ApiApprovals](#) and [ApprovalTests](#) to generate a public API of a given assembly.
- The public API gets approved by a human into a `*.approved.txt` file.
- Everytime the API approval test runs the API is generated again into a `*.received.txt` file. If the two files don't match the test fails on the CI server or locally. Locally on the devs machine the predefined Diff viewer pops up (never happens on CI) and the dev has to approve the API changes (therefore making a conscious decision)
- Each PR making public API changes will contain the `*.approved.txt` file in the DIFF and all reviewers can easily see the breaking changes on the public API.

In Akka.NET, the API approval tests can be found in the following test assembly:

```
src/core/Akka.API.Tests
```

The approval file is located at:

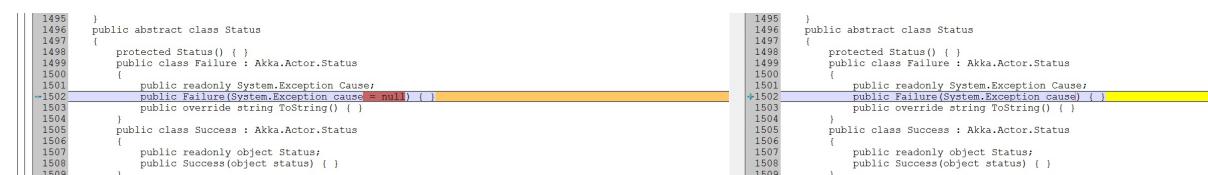
```
src/core/Akka.API.Tests/CoreAPISpec.ApproveCore.approved.txt
```

## Approving a New Change

After modifying some code in Akka.NET that results in a public API change - this can be any change, such as adding an overload to a public method or adding a new public class, you will immediately see an API change when you attempt to run the `Akka.API.Tests` unit tests:



The tests will fail, because the `.approved.txt` file doesn't match the new `.received.txt`, but you will be prompted by [ApprovalTests](#) to view the diff between the two files in your favorite diff viewer:



After you've merged the changes generated from your code into the `approved.txt` file, the tests will pass:

✓  Core (1 test)	Success
▲ ✓  Akka.Tests (1 test)	Success
▲ ✓  Akka.Tests.API (1 test)	Success
▲ ✓  CoreAPISpec (1 test)	Success
✓ ApproveCore	Success

And then once you've merged in those changes, added them to a Git commit, and sent them in a pull request then other Akka.NET contributors will review your pull request and view the differences between the current `approved.txt` file and the one included in your PR:

```
$ git diff
diff --git a/src/core/Akka.Tests/API/CoreAPISpec.ApproveCore.approved.txt b/src.
index fe84b48..1f30d35 100644
--- a/src/core/Akka.Tests/API/CoreAPISpec.ApproveCore.approved.txt
+++ b/src/core/Akka.Tests/API/CoreAPISpec.ApproveCore.approved.txt
@@ -1499,7 +1499,7 @@ namespace Akka.Actor
 public class Failure : Akka.Actor.Status
 {
 public readonly System.Exception Cause;
- public Failure(System.Exception cause) { }
+ public Failure(System.Exception cause = null) { }
 public override string ToString() { }
 }
 public class Success : Akka.Actor.Status
diff --git a/src/core/Akka/Actor/ActorBase.cs b/src/core/Akka/Actor/ActorBase.cs
index 2c9b910..ebcd40a 100644
--- a/src/core/Akka/Actor/ActorBase.cs
+++ b/src/core/Akka/Actor/ActorBase.cs
@@ -38,7 +38,7 @@ public class Failure : Status
 {
 public readonly Exception Cause;
- public Failure(Exception cause) { }
+ public Failure(Exception cause = null) { }
```

## Unacceptable API Changes

The following types of API changes will generally not be approved:

1. Any modification to a commonly used public interface;
2. Changing any public method signature or removing any public members;
3. Renaming public classes or members; and
4. Changing an access modifier from public to private / internal / protected on any member that is or is meant to be used.

# Documentation guidelines

When developers or users have problems with software the usual forum quip is to read the manual. Sometimes in nice tones and others not so nice. It's great when the documentation is succinct and easy to read and comprehend. All too often though there are huge swathes of missing, incomplete or downright wrong bits that leave people more confused than before they read the documentation.

So the call goes out for people to help build up the documentation. Which is great until you have a lot of people with their own ideas how everything should be laid out trying to contribute. To alleviate the confusion guidelines are setup. This document illustrates the documentation guidelines for this project.

There is a ton of work that still needs to be done especially in the API documentation department. Please don't hesitate to join and contribute to the project. We welcome everyone and could use your help.

## Website

When writing documentation for the website, the project uses [Markdown](#) when crafting the documents. The rendering of the website is done with Marked.JS. Thus, any editor based on this will give you the best preview/edit experience, such as [Atom](#) or [StackEdit](#).

To contribute to the website's documentation, go to the github project page [getakka.net](#). Please be sure to read the [Readme.md](#) before getting started to get acquainted with the project's workflow.

## Code

When documenting code, please use the standard .NET convention of [XML documentation comments](#). This allows the project to use tools like Sandcastle to generate the API documentation for the project. The latest stable API documentation can be found [here](#).

Please be mindful to including *useful* comments when documenting a class or method. *Useful* comments means to include full English sentences when summarizing the code and not relying on pre-generated comments from a tool like GhostDoc. Tools like these are great in what they do *if* supplemented with well reasoned grammar.

### **BAD** obviously auto-generated comment

```
/// <summary>
/// Class Serializer.
/// </summary>
public abstract class Serializer
{
 /// <summary>
 /// Froms the binary.
 /// </summary>
 /// <param name="bytes">The bytes.</param>
 /// <param name="type">The type.</param>
 /// <returns>System.Object.</returns>
 public abstract object FromBinary(byte[] bytes, Type type);
}
```

### **GOOD** clear succinct comment

```
/// <summary>
/// A Serializer represents a bimap between an object and an array of bytes representing that object.
/// </summary>
```

```
public abstract class Serializer
{
 /// <summary>
 /// Deserializes a byte array into an object of type <paramref name="type"/>
 /// </summary>
 /// <param name="bytes">The array containing the serialized object</param>
 /// <param name="type">The type of object contained in the array</param>
 /// <returns>The object contained in the array</returns>
 public abstract object FromBinary(byte[] bytes, Type type);
}
```

We've all seen the bad examples at one time or another, but rarely do we see the good examples. A nice rule of thumb to remember is to write the comments you would want to see and read while perusing the API documentation.

# Contributor guidelines

## To be considered while porting Akka to Akka.NET

Here are some guidelines to keep in mind when you're considering making some changes to Akka.NET:

- Be .NET idiomatic, e.g. do not port `Duration` or `Future`, use `Timespan` and `Task<T>`
- Stay as close as possible to the original JVM implementation, <https://github.com/akka/akka>
- Do not add features that does not exist in JVM Akka into the core Akka.NET framework
- Please include relevant unit tests / specs along with your changes if appropriate.
- Try to include descriptive commit messages, even if you squash them before sending a pull request.
- If you aren't sure how something works or want to solicit input from other Akka.NET developers before making a change, you can [create an issue](#) with the `discussion` tag or reach out to [AkkaDotNet on Twitter](#).

## Coding conventions

- Use the default Resharper guidelines for code
  - Private member fields start with `_`, i.e. `_camelCased`
  - PascalCased public and protected Properties and Methods.
  - avoid using `this` when accessing class variables, e.g. BAD `this.fieldName`
  - TODO.. Anyone got a complete list for this?
- 4 spaces for indentation
- use [Allman style](#) brackets for C# code and [1TBS style](#) brackets for HOCON and F# code.
- No protected fields. Create a private field and a protected property instead.

## Tests

- Name your tests using `DisplayName=`

e.g.

```
[Fact(DisplayName=
@"
If a parent receives a Terminated event for a child actor,
the parent should no longer supervise it")]
public void ClearChildUponTerminated()
{
 ...
}
```