Dust

alan@mentalresonance.com

Version 1.1.5

Dust is a JVM based framework for building distributed, scalable applications based on the Actor paradigm.

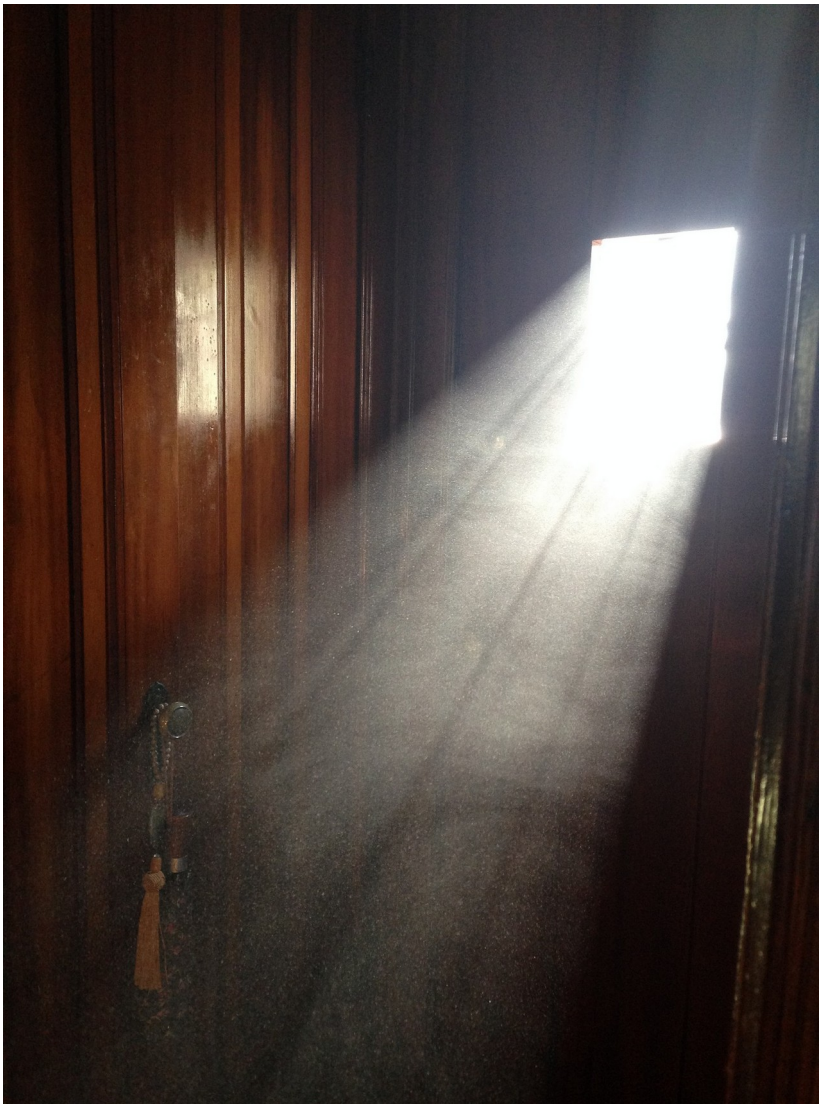
Dust comes in several modules:

- **dust-core –** the heart of Dust which implements Actors, Actor persistence, several core Actor building blocks and the basics of the Entity framework, built on Dust, for digital twinning and similar applications.
- **dust-http** – library for creating Actors which participate in http/REST style interactions
- **dust-html** – library for adding html processing to Actors
- **dust-nlp** – library for NLP processing using LLMs or SpaCy
- **dust-feeds** – library of Actors for constructing pipelines of feed processors, e.g. RSS feeds.

This document discusses dust-core.

The core of Dust is written in pure Java with some higher level functionality in Groovy – a natural extension of Java which may be used as a scripting language. If you know Java (which would be helpful in reading this document) then a good start with Groovy is to think of it as Java ‘without the semi-colons’[1].

1 http://www.groovy-lang.org/



Dust-Core: An Introduction

The advent of (very) virtual threads in Java (Java 19 and on) has opened up the possibility of new (or in the Dust case, old) paradigms of computation. Dust is an exploration of one such paradigm whose origins actually date back to the 1970's – Actors[2].

An Actor is a sealed object whose only form of communication with the world outside of it is by receiving and sending *messages*. Internally what an Actor does in response to specific messages is called its behavior, and an Actor may change its behavior in response to messages it has received.

The world outside of an Actor is usually populated with (possibly millions) of other Actors and application creation is the process of defining the behaviors of classes of Actors and the messages past between them. It is also possible for Actors to interact with non-Actors in the application by the use of closures or futures or similar devices.

A non-JVM implementation of the Actor model is Erlang (and Elixir, which shares Erlang's virtual machine) which is based on a custom VM designed to support the Actor paradigm directly. Thus each Actor gets its own heap and sealing is enforced by the VM. Actors build on the JVM can't be so easily sealed (you can, for example, pass a reference to on object managed internally by an Actor in a message to a second Actor, so sealing here is done by implicit contracts (if it could cause hurt, don't do it).

Actor models lend themselves to being able to create very resilient systems – often using the 'let it crash' approach[3] (where 'it' here is an Actor).

Finally Dust uses a uniform method for addressing Actors which is readily distributed over a network[4]. Since a Dust application consists entirely of Actors sending messages to each others' addresses Dust gives you distributivity and scalability for, in some sense, free.

---

2 https://arxiv.org/vc/arxiv/papers/1008/1008.1459v8.pdf
3 https://erlang.org/download/armstrong_thesis_2003.pdf
4 In this and other regards Dust gained some of its inspiration from 'Classic' Akka. https://doc.akka.io/docs/akka/current/index-classic.html

1 Dust Basics

Dust use some terminology in very specific ways, which we discuss here.

1.1 Actors

A Dust Actor consists of three major parts.

1. A mailbox – this has an associated address and is where messages get sent for a particular Actor.
2. A collection of one or more *Behaviors*. A Behavior defines what the Actor does on receipt of a message in its mailbox. There is always exactly one current Behavior, but an Actor's response to a message when the Actor is in one Behavior might cause it to change Behaviors, so the next message retrieved from the mailbox may be handled by a different Behavior.
3. The main loop. This loop removes the next message from the mailbox (or waits for a message to appear which is then removed) and passes it to the current Behavior who processes it as it sees fit. When the processing of the message is complete the loop repeats. *Messages are always processed in the order received*.

Notice that only one message is being processed at one time in a given Actor. It is a good Dust design practice to divide an Actor's work into many small messages (some of which it may well send to itself) since this prohibits a single Actor from consuming a large part of the hosts systems compute resources.[5]

1.2 Messages and Message Processing

In Dust a message is simply an instance of a *Serializable* class.

It is important to understand the guarantees Dust does (and does not) make regarding message delivery.

1. A message will never be received by its recipient more than once, but we do not guarantee messages will always be delivered (but they usually are – especially when sent within a given host).
2. Actors have a well defined notion of the order in which messages were sent out and the order in which they are received. Thus if Actor A sends messages A1, A2 (in that order) to Actor B then Actor B will receive (assuming no message was lost) and extract from its mailbox A1, A2 (in that order).
3. Only message delivery order is guaranteed, *not* consecutiveness. So in the above Actor B could well process:

---

5 Java's virtual threads have helped greatly here, but it is still good design process to divide work into a series of messages. Message creation and delivery is *very* fast,

A1, P3, Q4, A2, S25

Where messages P and Q were sent by different Actors after the receipt of A1 but before A2.

Note that in the above we made no notion of acknowledging receipt of messages. This, by design, is not built into the message delivery protocols.

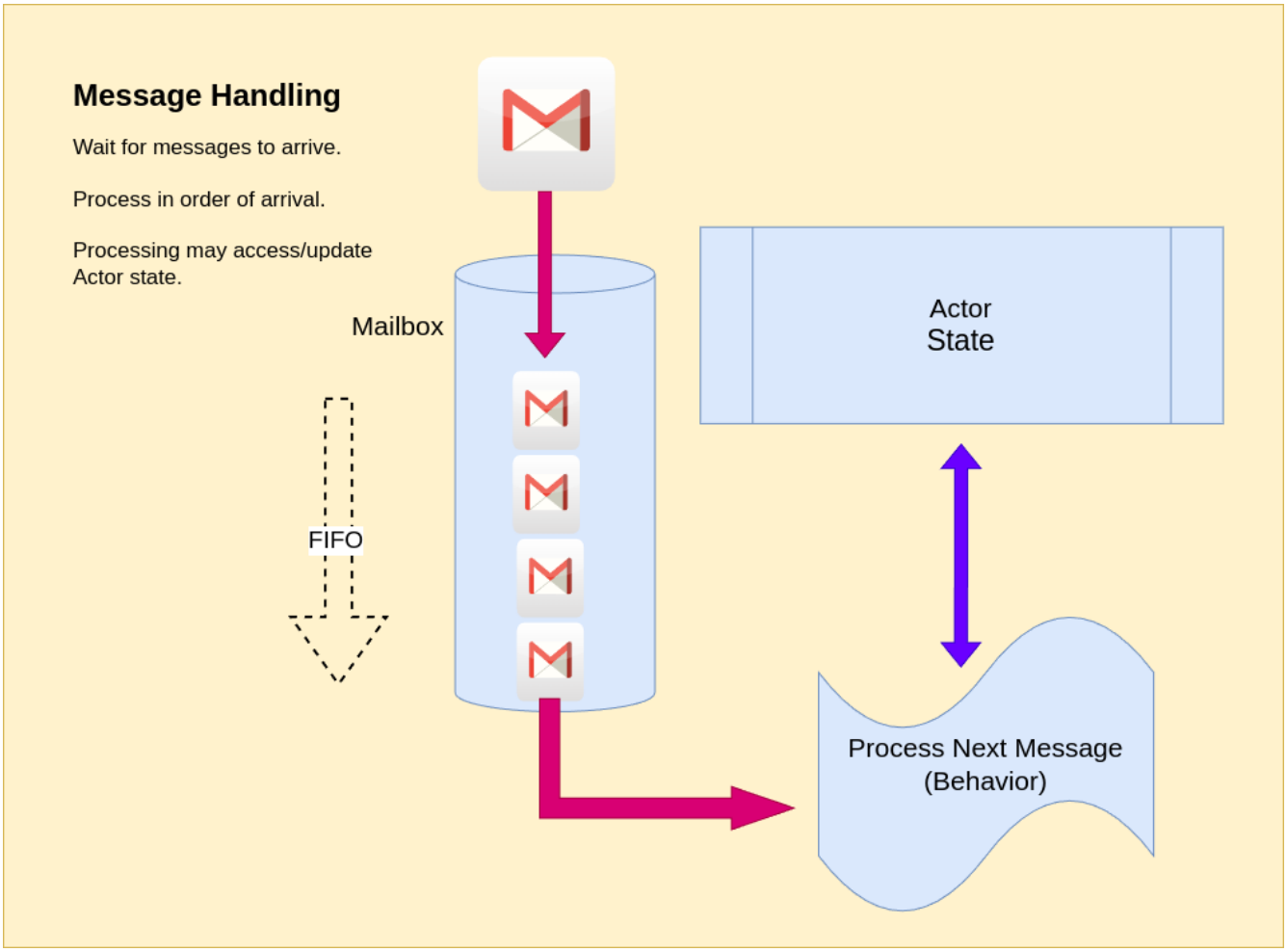
While this may seem to be a strange way of going about things (we are used to acknowledgments and retries being built into the fabric of things) we shall see that in fact it forces us into design patterns that are actually quite powerful.

1.3 Mailbox

Every Actor has a mailbox and it is the mailbox that joins the JVM virtual threads to the Actor model. The mailbox is simply a list of messages received by the Actor in the order they were received. Newly received messages are added to the end of the list and the Actor's (mailbox) thread simply removes messages one at a time from the start of the list and responds to their contents.

If the mailbox is empty the thread simply waits for a new message to come in. Dust gives the Actor no way to access the mailbox directly so the model 'remove and process the next message' is the only way it can access the queue of messages.

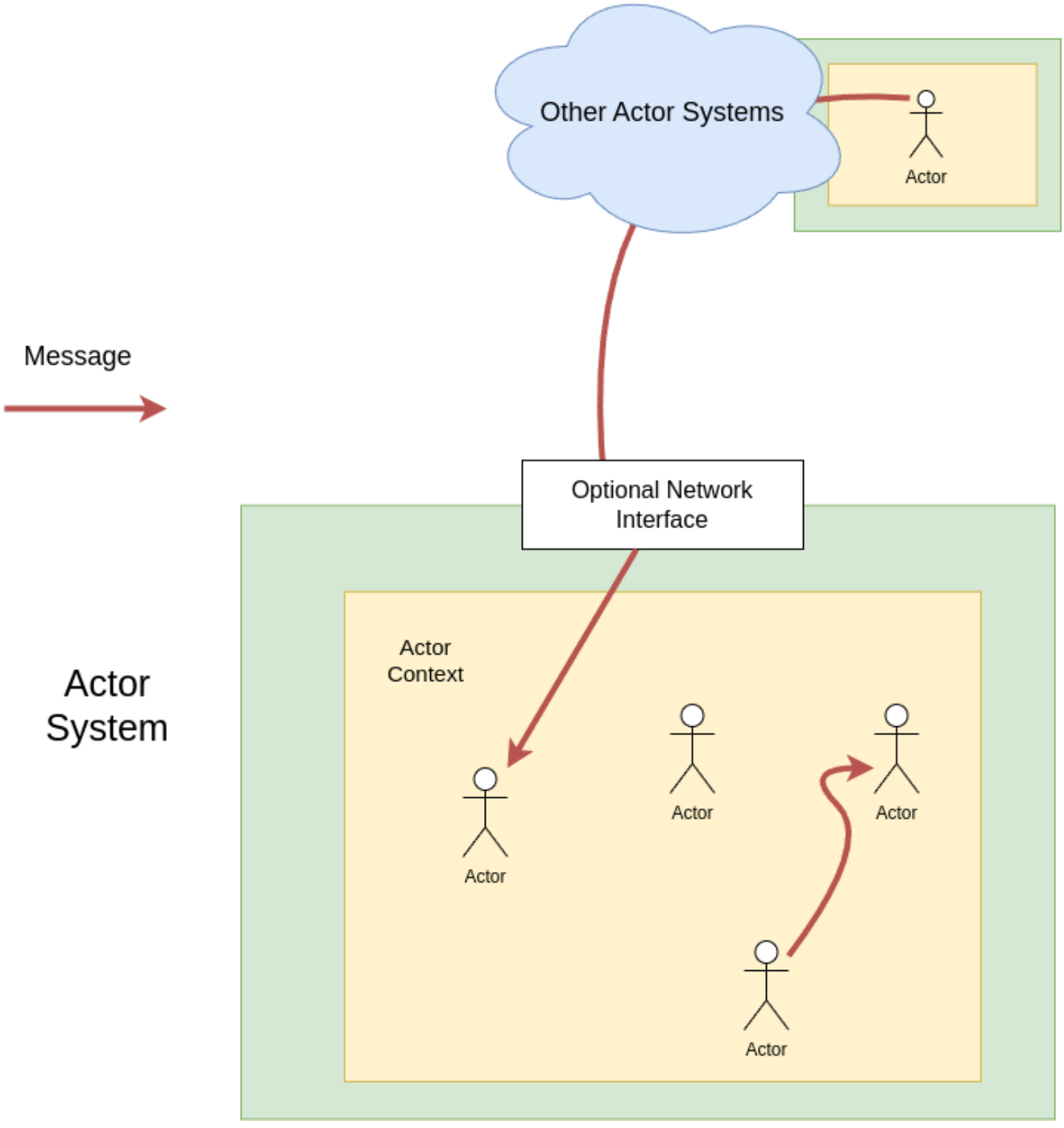
Importantly: *Within an Actor there is only one thread. Internally, Actors are totally sequential.*



Every mailbox (and hence every Actor) has a unique Address which we will discuss below.

1.4 Actor System

The Actor System is a management container for Actors. It creates a set of services for the Actors it contains. An Actor System is created with a name and, optionally a TCP port number. Actors within the same system can address each other (and hence send each other messages) very rapidly. Actors in different Actor Systems (which may be on different machines) address each other using a simple extension of the addressing scheme which factors in the host, name and port of the other System.



The Actor System contains a single **Actor Context**. This allows an Actor to:

- Tell any Actor to stop
- Convert an Actor Address into an actionable Actor Reference (either local or remote). Messages are sent to a particular Actor via an Actor Reference.
- Create ‘top level’ Actors to bootstrap the initialization process

The Actor System creates a special Actor called the Guardian. All other Actors have address that can trace their name back to this guardian. The Guardian Actor is denoted by the initial "/" in paths as we shall see below.

The final action of the ActorSystem upon creation is to create two top level Actors:

/user
/system

All user created Actors will find themselves in paths starting /user.. and some automatically generated system Actors will be in paths under /system.

1.5 Actor Organization – Parent and Children

In the diagram above it looks as though the ActorSystem is just a bag of Actors, but this is not the case. Actors live in a very organized extended family structure.

Excluding the guardian, which is a special case, *only Actors can create Actors*[6]*.* If Actor A creates Actor B then we say A is B's *parent* (or, naturally, B is A's *child*). Other children of the B's parent are called B's *siblings*.

A parent must give a child a unique name when creating it – unique among its siblings. This leads to the natural notion of an Actor *path:* take the Actor's name, prepend that with its parent's name, and repeat the process until you land at the mother of all Actors – the (invisible) guardian.

To make the path even more familiar we separate parent/child boundaries with the '/' symbol (which naturally cannot be part of a name[7]) and so now each Actor has a totally unique path to it e.g.:

/user/building1/firstfloor/rooms/21

Here there a five paths identifying 5 different Actors whose names are user, building1, firstfloor, rooms and 21. If rooms created another child (say 22) then we'd have 6 Actors:

/user/building1/firstfloor/rooms/21

/user/building1/firstfloor/rooms/22

This represents 6 Actors – user, building1, firstfloor, rooms, 21 and 22. Likewise if building1 creates a new child, say 2ndfloor, then we might find ourselves with the following path.

/user/building1/2ndfloor/rooms/21

Caution: user and building1 are the same Actors as above (they share the same paths). 2ndfloor is a new Actor as is rooms (since it lies on a different path than the rooms above).

---

6 This is not quite true – it is possible for non-Actor code to create Actors managed directly by the 'user' Actor. This is used to bootstrap an application.

7 Apart from the aforementioned Guardian Actor whose name is the first "/" in the path. So /user and /*system* have parent "/".

And finally 21 is a new Actor, again because it lives on a different path. Thus we have two different Actors called rooms, and two different Actors whose name is 21.

Names do not uniquely identify Actors; paths do.

There is more to the parent/child relationship than just forming paths – a parent is intimately involved in managing the life-cycle of the child.

- If the child stops normally it sends a message to its parent that it is halting and the child Actor is removed from the path. The parent may respond however it likes to this ‘clean stop’ message (including ignoring it).
- The stopping child Actor tells all of its children to stop, and this process ripples down the tree beneath the child.
- If a child crashes (i.e. throw an Exception) then a message (along with the Exception) is sent to the parent. The parent then has a choice of strategies it can follow to manage the situation:
    - Simply restart the Actor (i.e. start its mailbox thread again) giving it the Exception that caused the error
    - Tell the child to stop
    - Tell the child and all its siblings to stop

1.6 Actor Addresses

As can be seen above every Actor (beneath a common Guardian) has a unique address determined by the path between it and the Guardian. This is the Actor’s address and each address has the corresponding Actor’s mailbox associated with it. This is how messages are targeted for delivery.

1.7 Remote Addresses

All the addresses (paths) we have seen so far begin in the same guardian Actor, and hence are in the same Actor System. How do we address Actors in different Actor Systems (which might indeed be on different machines)?

An Actor System is given a name when created, it may also be given a port number. In which case it sets up a server on this port and this is how we enable (almost complete) transparency with Actor distribution.

We can take an Actor path, say

/*user*/a/b/c

and now we can prefix it with some routing information:

dust://remote-host:port/remote-actor-system-name/user/a/b/c

and we still have a path, but it allows us to send messages to Actors on remote machines.

Sending of messages is totally independent of whether the target Actor is local or remote.

1.8 Dead Letters

1.8.1 Dead Letter

An Actor may hold an Actor Reference to another Actor which has stopped – the reference holder is not notified if such as situation happens (unless it was watch()ing it – more on this later), and so it may well try to send the defunct Actor a message.

The Actor System detects the message has been sent to a defunct Actor and wraps the message in a 'Dead Letter' message. This message is in turn sent to the DeadLetter Actor created by the system.

The DeadLetterActor simply logs the message, sender and target. Other Actors can register with the DeadLetterActor to be notified if a message they sent (or someone else sent) failed to reach its target because its target was no more.

The dead letter Actor is found at /system/deadletters.

1.8.2 ZombieMsg Interface

We will see later on that the Dead Letter mechanism can be used in a 'reverse way': send a message to an Actor you assume to exist, and if it does not yet exist it gets created and receives the message. This is done by hooking the DeadLetter Actor and is a very powerful paradigm but the DeadLetter handler does not know this message isn't really a dead letter so it logs the message anyway. This can cause confusion (why was this a dead letter? Oh it really wasn't ..) so, as a convenience, messages which implement the ZombieMsg interface do not get so logged and confusion is minimized.[8]

8 We hope ...

2 Actors: Creating, Communicating, Locating and Destroying

We have seen the basic notions of Actors, Messages, Actor Addresses and Actor References as wrapped up inside an Actor System, but we have yet to see how this all actually 'works'.

Dust is written in Java and runs on the JVM, so at the end of the day everything is objects[9] and method calls. We will now see how Dust ties together the Java object/inheritance world with that of Actor and messages. We will use the Groovy language, which is a more streamlined version of Java. Groovy, Java and Dust can all readily be combined.

2.1 Structure of an Actor

A user-supplied Actor extends the Dust class 'Actor':

```
class MyActor extends Actor { //1

      static Props props(int i) { Props.create(MyActor.class, i) }   // 2

      MyActor(int i) { // 3
            // Do something clever with I upon construction
      }

      @Override
      protected ActorBehavior createBehavior() { // 4
            (Serializable message) → {
                  log.info "Got message ${message}"
                  sender.tell(new PoisonPill(), self)
            }

      }
}
```

1. Most declarations in Groovy default to public
2. An Actor defines a method (by *convention* called props()) which takes the arguments to be passed to the Actor's constructor and returns an instance of Props via the static Props create() method. Props define how to construct the Actor (which is done via method calls, ***not*** the *new* operator).

   Props.create returns these Props, the first argument is the Actor's class, the rest: the parameters to be passed to the constructor.
3. The Actor is expecting the single parameter, an integer as defined by props().

9 Alan Kay, who claims to have coined the phrase 'Object Oriented Programming' frequently points out that what he had in mind was message passing and not inheritance !

4. ActorBehavior defines how the Actor will response to messages. By default the Actor will begin with the behavior defined by a createBehavior() method. ActorBehavior is really just a Java Lambda which takes a serializable object (message) and reacts to it. In this case by printing out the message and then doing some magic we discuss below.

This is enough code to describe an Actor. It doesn't do a lot – it just prints out any message sent to it and does something with a PoisonPill object. Doesn't sound too good.

2.2 ActorRef

ActorRef is a core class. An instance of it is in essence a handle to an Actor's mailbox, so to send messages to an Actor you have to have an ActorRef for it.

2.3 Actor Fields and Methods

All Actors inherit several fields and methods from the Actor class. The core ones are:

2.3.1 self

The *self* field is the ActorRef to the Actor itself. An Actor sends messages to itself via *self*.

2.3.2 context

The *context* field references an instance of the ActorContext class. This class manages all the Actors in the ActorSystem and allows Actors to interact with each other in various ways.

2.3.3 sender

The *sender* field's value is an ActorRef to the Actor which sent the message currently being processed. So sender only makes sense inside of a Behavior.

2.3.4 parent

The ActorRef of your parent (or null if none).

2.3.5 grandParent

The ActorRef of your parent's parent (or null if none).

2.3.6 actorOf()

The actorOf() method comes with two signatures:

```
actorOf(Props childActorProps, String name) // Create child with given name
actorOf(Props childActorProps)  // Create child with unique random name
```

It is the canonical way in which an Actor creates its children. It returns an ActorRef for the created Actor.

So to create a child from within an Actor we simply supply the appropriate props to the built-in actorOf method:

```
ActorRef ref = actorOf(MyActor.props(42), “answer”)
```

This creates an instance of MyActor (above) as a child of mine whose name is ‘answer’, gives it a mailbox and starts its listening loop. It is then ready to receive messages

2.3.7 actorSelection()

The ActorSelection takes a String describing a path to a (supposedly) running Actor and returns an instance of ActorSelection, which can be thought of as containing all the location information for that Actor. e.g.

ActorSelection localSelection = actorSelection(“/user/foo/bar”)

ActorSelection remoteSelection =
actorSelection(“dust://192.168.1.121:4044/system-name/user/foo/bar”)

ActorSelection relativeSelection = actorSelection(‘c/d’)

ActorSection parentRelativeSelection = actorSelection(‘../foo/bar’)

The first selection is of an Actor named /*user/*foo/bar running in the same ActorSystem as the caller.

The second is an ActorSelection for a similarly named Actor running on a remote system on (probably) a remote machine.

The third is a *relative path* and specifies the actor d, who is a child of c where c is a child of mine. The path ‘./c/d’ is an alias for this child-relative path.

The fourth is a relative path but here it is relative to my parent ‘..’. So .’.,/foo’ is a sibling of mine and so .’.,/foo/bar’ is my nephew (or niece).

ActorSelection supports the getRef() method:

ActorRef refToLocalSelection = localSelection.getRef()

Note: An ActorSelection is a structure built around a path. It does not know if the Actor specified by the path was ever created, or if it is still alive. *If the Actor is not reachable then getRef() returns an ActorRef for the Dead Letter Actor.*

2.3.8 watch() and unWatch()

Actors have a life cycle – they are created, process messages and the they may stop and die (eventually being garbage collected in the JVM).

watch(targetRef) – watches the target. If the target stops you will receive an instance of a Terminated message from it.

unWatch(targetRef) – if the target was being watched then stop the watching. If the target later stops you will not be informed.

2.4 MessageSending

We now have enough for an Actor to create a child Actor and send it a message. *A message can be any Serializable object*.

An ActorRef supports the tell() method:

targetRef.tell(Serializable msg, ActorRef sender)

method. This sends msg to the Actor at targetRef and marks the sender of the message to be the second argument of the tell. This second argument is usually ‘self’ but does not have to be. Thus messages can be ‘sent by proxy’ – a very powerful idiom we shall use later. The second argument can also be null – in effect telling the receiver the sender wishes to remain anonymous.

This code usually runs in some Behavior of the Actor …

```
protected ActorBehavior createBehavior() {

      … Props and Constuctor

      ActorBehavior createBehavior() {
            (Serializable message) → {
                  switch(message) {
                   case StartMsg:
                        actorOf(MyActor.props(42)).tell(“Hello!”, self)
                        break
                   default:
                        log.info “Waiting for start”
                  }
            }
}
```

So what is happening? This Actor is waiting to receive an instance of a StartMsg object (a predefined class which is handy for cases like this). When it gets the message it creates a new child which is a randomly named instance of MyActor giving its constructor the parameter 42.

Since actorOf returns an ActorRef to the child we can call the tell() method on it. We send the actor a String (which is Serializable) and tells it who it came from. Let’s revisit the behavior of MyActor:

```
            @Override
            protected ActorBehavior createBehavior() { // 4
                  (Serializable message) → {
                        log.info “Got message ${message}”
                        sender.tell(new PoisonPill(), self)
                  }

            }
```

Whatever message it receives MyActor prints it out and then sends the *sender of this message a new* message*:* an instance of PoisonPill().

Note: as a convenience an ActorSelection supports an identical tell() method, so rather than writing

```
actorSelection.getRef().tell(x, y)
```

you can simply write

```
actorSelection.tell(x, y)
```

As a further convenience[10] the Actor class defines a method

```
tellSelf(Serializable msg)
```

this is simply implemented as

```
self.tell(msg, self)
```

2.5 Stopping an Actor

There are two ways of gracefully stopping an Actor.

1. Send it a PoisonPill message which is a Dust built-in message class. When a PoisonPill gets removed from the recipient's mailbox for processing the Actor will automatically shut down. This happens before the message would be passed to the current behavior so the user's message processing will never see it.
2. **context.stop(targetActorRef)** has the same effect (to stop the Actor) but it works in a different way. A PoisonPill is handled just like any other message – it is added to the recipient's mailbox and all the messages in the mailbox before it are processed first. A context.stop(), by comparison, waits until any current message processing finishes and then stops the target Actor immediately.

In both cases the postStop() method is called after the Actor stops (see Life Cycle below).

2.6 More on ActorContext

Every Actor has a *context* field which is a reference to its ActorSystem's ActorContext. As well as providing the whole pathname → actor reference machinery it provides some services to Actors:

actorSelection(String path) – as discussed above this returns an ActorRef to the Actor defined by the path. If no such Actor exists the ActorRef returned is to the Dead Letter Actor. The path may be a local or remote path.

stop(ActorRef ref) – stops the referenced Actor immediately after allowing the Actor to finish processing its current message, if it has one.

10 A common Actor idiom is to respond to an external message by generating a series of internal messages and sending them to itself.

stop() - stops the entire Actor system. If it has a server this is stopped, then a complete orderly shutdown of all Actors takes place.

getUserActor() - a convenience method to gain a reference to '/user'

getDeadLetterActor() - a convenience method to gain a reference to the Dead Letter Actor.

actorOf(Props props, String name) – creates an Actor from the given Props as a child of /user with the given name. Returns its ActorRef.

actorOf(Props props) – creates an Actor from the given Props as a child of /user with a random name. Returns its ActorRef.

2.7 More on Behavior

2.7.1 Super Behavior

Quite often an Actor which is a subclass of another will have the interpretation of some messages which are unique to it, but others whose interpretation is shared with its superclass. In this case the common idiom is simply to delegate those messages to its superclass's behavior:

```
ActorBehavior createBehavior() {
      return message → {
            switch(message) {
                  case A → { … }
                  case B → { … }
                  default → super.createBehavior.onMessage(message)
            }
      }
}
```

2.7.2 Becoming Behavior

Depending on the use case Actors may live in different 'states' at different times and may wish to respond to the same message differently at different times. Rather than attempt to model this in one behavior, which could well become a tangled mess, we can specify different behaviors and move between them by using the Actor's **become**() method. For example:

```
ActorBehavior createBehavior() {
      return message → {
            switch(message) {
                  case A → { "do something with A" }
                  case B → { become(newBehavior()) }
                  default → super.createBehavior.onMessage(message)
            }
      }
}
```

```
ActorBehavior newBehavior() {
      return message → {
            switch(message) {
                  case A → { “do something else with A” }
                  case B → { “do something else with B” }
                  default → super.createBehavior.onMessage(message)
            }
      }
}
```

The Actor begins with behavior defined by createBehavior() (by default). Once the Actor receives an instance of B as a message it uses become(). The next message to be processed will be handled by the newBehavior() lambda. Note that in this case its behavior will continue to be defined by this new behavior.

Actors also support a more structured way of managing behavior transitions:

stashBecome(ActorBehavior newBehavior) – pushes the current behavior onto a stack and becomes newBehavior.

unbecome() - pops the last pushed behavior from the stack and becomes it.

2.7.3 Stashing Messages

As we move between behaviors it may be that one behavior does not handle a message but that message will be handled by some other behavior so it should not be dropped. To deal with these situations Dust Actors implement **stash(Object message)** and **unstashAll().**

stash(msg) adds the message to a temporary queue. unstashAll() empties that queue by adding the temporary queue (in order) to the end of the mailbox, so typical patterns for using this are:

ActorBehavior createBehavior() {

```
      return message → {
            switch(message) {
                  case A → { “do something with A” }
                  case B → {
                        become(newBehavior());
                        unstashAll();
                  }
                  default → stash(message)
            }
      }
}


ActorBehavior newBehavior() {
      return message → {
            switch(message) {
                  case C → {  }
                  case D → {
                        become(createBehavior());
                        unstashAll();
                  }
```

```
                    default → stash(message)
                }
            }
        }
```

2.7.4 Internal Message processing

A small number of messages are handled by the base Actor before they would be passed to the subclass Actor's current behavior. These are:

- **WatchMsg** – causes the sender to be added to the recipent's list of watching Actors. So the sender will be notified via a Terminated message when the recipient stops (i.e. implements the watch() method.
- **UnWatchMsg** – implements the unWatch() method.
- **GetChildrenMsg** – on receipt the Actor fills in a list of ActorRefs for its children and sends the message back to the sender. When the sender receives this message it knows that is is a response and not a request so the response does get passed to the current behavior.
- **PoisonPill** – stops the Actor

2.8 A Complete Example

So far we have seen extracts of code covering specific use cases, but not an end to end example. So here we present ping-pong: two Actors which simply relay messages back and forth.

First we need a ping-pong Actor. Its job is to reply to a 'ping' message with a 'pong' message (and vice versa) until it has been received a message a maximum number of times, in which case it tells its sender to die and then kills itself.

```
public class PingActor extends Actor {
    protected Integer maxMessages;

    public static Props props(Integer maxMessages) {        // 1
        return Props.create(PingActor.class, maxMessages);
    }
    /**
     * @param maxMessages - if not null max number of messages before stopping
     */
    public PingActor(Integer maxMessages) {
      this.maxMessages =  maxMessages;
    }

    @Override
    protected ActorBehavior createBehavior() {             // 2
        return message -> {

            switch(message) {
                case PingMsg msg -> {
                        if (-- maxMessages > 0) {
                              log.info “Ping”
                              sender.tell(new PongMsg(), self)  // 3

                        } else {
                              sender.tell(new PoisonPill(), self)  // 4
                              context.stop(self) // 5
                        }
                        break
                case PongMsg msg -> {
                        if (-- maxMessages > 0) {
                              log.info “Pong”
                              sender.tell(new PingMsg(), self)

                        } else {
                              sender.tell(new PosionPill(), self)
                              context.stop(self)
                        }
                        break
                default -> log.warn(self.path + " unexpected message: " + message);
            }
        };
    }

}
      public class PingMsg implements Serializable {}
      public class PongMsg implements Serializable {}
```

1. We will create the Actors with a parameter that specifies the max number of message before stopping, otherwise there would be a lot of output :)
2. Our behavior follows a standard pattern. The lambda passes the message to a switch based on the message class. Here we are only interested in Ping and Pong messages – anything else we simply log.

3. If we haven't used up our message allowance we reply to the sender with a complementary message (Ping/Pong).
4. If we have used up our messages we send the sender a PoisonPill message and
5. stop ourselves.

Having defined the Actors we now need to set then up and then start the ping pong going.

```
ActorSystem system = new ActorSystem("Test")   // 1
ActorContext context = system.context          // 2

context.actorOf(PingActor.props(50000), 'ping') // 3
context.actorOf(PingActor.props(50000), 'pong')

ActorRef ping = context.actorSelection("ping").getRef() // 4
ActorRef pong = context.actorSelection("pong").getRef()

ping.tell(new PingMsg(), pong) // 5
```

1. We need to set up an ActorSystem to use. We call it Test but give it no port since all our Actors are local.
2. Get the context of the Actor system so we can create our Actors
3. context.actorOf() creates the specified Actors as children of /user (predefined by the ActorSystem) so we are defining two PingActors: /*user/*ping and /*user/*pong. This is how we 'bootstrap' our Actors.
4. We get references to our ping and pong Actors
5. We send a PingMsg to the Ping Actor

Note that the thread that creates the ActorSystem should not terminate after sending the PingMsg since that would cause the ActorSystem to shut down taking everything with it.

So, on receipt of the PingMsg the Ping Actor will send a PongMsg back to the Ping Actor and the two will pass messages back and forth until 50,000 messages have passed back and forth at which point both Actors will shut down.

2.9 Scheduling Messages

It is sometimes useful to send a message to some target which is to delivered in the future. For instance an Actor might be expecting a reply to some message it sent and it is willing to wait for a certain length of time to receive the message otherwise it will retry.

Actors have methods:

```
Cancellable scheduleIn(Serializable msg, Long ms)
Cancellable scheduleIn(Serializable msg, Long ms, ActorRef target)
```

for just this purpose. The first method sends the msg to itself after ms milliseconds whereas the second sends the message to the target Actor after ms milliseconds. Both return a

Cancellable which has a simple cancel() method which, if called before the message is sent will cancel the delivery.

scheduleIn is very useful for having Actors repeat a task at regular intervals e.g.Cancellable task;

```
void preStart() {
      task = scheduleIn(new TaskMsg(), 5*60*1000);
}
void postStop() {
      task.cancel()
}
ActorBehavior createBehavior() {
      return message → {
            switch(message) {
                  ….
                  case TaskMsg → {
                        // Do the task
                        ….
                        scheduleIn(message, 5*60*1000);
                  }
            }
      };
}
```

will repeat some task every 5 minutes. Bear in mind:

- scheduleIn determines when the message is sent, not when it is processed. At the scheduled time the message is added to the recipient's mailbox just like any other message so when it is actually processed will depend on what is ahead of it.
- scheduleIn() uses Java's virtual threads and timers. These are usually accurate to a few milliseconds so Dust is at best a 'soft' real time system. Do not use it to control nuclear power plants.
- Notice we cancel() the task in the postStop(). A scheduled message exists somewhat independently of the Actor before it is sent and is not automatically canceled if the Actor is stopped. Canceling the task if the Actor stops is good hygiene – it prevents a DeadLetter being sent.

3 Actor Lifecycle

We have seen how to instantiate Actors and stop Actors and so Actors have a life-cycle. The Dust framework gives access to various points in this life-cycle by means of overridable callbacks:

```
@Override
protected void preStart() { .. }
```

preStart() is called after the Actor is constructed as a Java object and has a mailbox. The main loop is not yet running but it can be queuing received messages. Thus from within preStart() the Actor can send messages to other Actors, including itself. It also has access to **context** and **self** variables so the Actor could stop itself before it even gets started.

```
@Override
protected void postStop() { .. }
```

postStop() is called after the mailbox processing is stopped. Any remaining messages will have been dropped (if the Actor was stopped via context.stop()) but the Actor can still send messages. It also still has access to self, parent and context.

3.1 Actor Stopping Process

When an Actor is told to stop it tells its children to stop and waits until they have stopped before it calls its postStop() method. Naturally this process is recursive and so the entire subtree of Actors beneath a parent Actor is stopped when it is stopped. postStop() gets called by every Actor as it stops.

3.2 Watching Actors

An Actor can **watch** another. This means that when the watched Actor is stopped the watching Actor receives a Terminated message. If an Actor no longer wishes to watch another it can **unwatch** it.

The protected methods **watch**(ActorRef other) and **unwatch**(ActorRef other) are available from within the Actor.

3.3 Supervising Actors

Actors can contain bugs and bugs can raise Java exceptions. Dust gives Actors the ability to control what happens next.

If an Actor throws an Exception it sends a message to its **parent** including the exception. The parent can do one of three things:

1. It can allow the Actor to stop. The Actor's postStop() will be called as usual. This is default response.
2. It can allow the Actor to continue. The message which caused the exception is dropped and the next message is processed as usual.
3. It can restart the Actor. In this case the pending messages are dropped and the loop restarts. But in this case **preStart**() is not called, rather **preRestart**(Throwable t) is called. Here t is the exception that caused the error and this gives the Actor the chance to interpret the root cause and possibly modify its behavior.

In addition to deciding what happens to the erroring Actor the supervising (parent) Actor can decide if the action applies to *only* that Actor (mode MODE_ONE_FOR_ONE) or all of its children (MODE_ALL_FOR_ONE).

The supervision strategy is determined by setting the **supervisor** field in the parent Actor:

```
supervisor = new SuperVisionStrategy(int strategy, int mode)
```

Where strategy and mode are constants in SuperVisionStrategy:

```
public final static int SS_STOP = ActorRef.LC_STOP;
public final static int SS_RESUME = ActorRef.LC_RESUME;
public final static int SS_RESTART = ActorRef.LC_RESTART;

public final static int MODE_ONE_FOR_ONE = 0;
public final static int MODE_ALL_FOR_ONE = 1;
```

3.3.1 Exceptions

As we will see in the next section an Actor may chose to persist its state so that it can be restored when that Actor is restarted. So a clean shutdown of an application means it can be restarted and the persistent Actors pick up where they left off.

If an Actor is stopping because its job is finished then it probably wants to delete its state.

But that happens if an Actor is stopping because:

a) It took an exception

b) One of it ancestors took an exception.

It needs to know if either if these events occurred so it can determine (in its postStop()) what to do with its state.

3.3.1.1 isException

ActorRefs (in particular self) have a public field 'Throwable isException = null'. If self takes an exception then isException is the throwable that cause the Exception and postStop() can act accordingly.

3.3.1.2 ParentException

If an Actor takes an exception then it creates a ParentException message (again containing the Throwable cause) and stops its children but it sets their isException to this throwable. Recursively they stop, but before they do they stop their children by repeating this process. Thus the whole tree of descendants from the Actor that had the original Exception is informed of this Exception and so can decide how their postStop() handlers should response.

4 Persistent Actors

As we will discuss later (in the ‘Entities’ section) Actors can often be used to model real world processes and situations. For instance we might want to model security systems in a building where Actors are responsible for various levels of granularity: An Actor that represents the entire building which receives messages from Actors representing the floors of the building which in turn receive messages from Actors representing individual rooms and areas on the floor. At each level information is rolled up and interpreted before being passed up the chain of command.

Such Actors should never stop and the previously discussed supervision mechanisms means that generally this can be achieved, but things happen: servers go down, some exceptions may be un-handleable etc. So we need some way of recreating whatever important state an Actor had should it need to be restarted.

Actors which subclass **PersistentActor** (which subclasses Actor) get the mechanisms to easily store and restore state.

4.1 Persistence Id

PersistentActor implements a method which return a unique identifier for the Actor which is used as the key for the state in the database.

```
String persistenceId()
```

The default implementation returns self.path but may be overridden as long as it is unique and stable (i.e. does not change across restarts).

4.2 Snapshots

A snapshot is a persisted copy of serialized[11] Actor state. The implementer of the Actor decides what state is important and when it should be saved. The Actor saves its state to a external storage, such as a database, under the key returned by persistenceId().

To save the state the persistent Actor calls its method:

```
void saveSnapshot(Serializable state)
```

After the state is saved the PersistentActor will receive one of two messages: SnapshotSuccessMsg or SnapshotFailureMsg. If the default PersistentActor behavior is delegated to its response is to drop both messages, but place a message in the log if the snapshot failed, so it is a good idea to be sure to handle SnapshotFailureMsg at the very least.

11 Dust uses FST serialization - https://github.com/RuedigerMoeller/fast-serialization which allows the creation of custom serializers.

An Actor has at most one snapshot so snapshots are updated as they are written. A snapshot may be deleted with

```
void deleteSnapshot()
```

which sends the Actor the message DeleteSnapshotFailureMsg or DeleteSnapshotSuccessMsg. Again failure should be handled.

4.3 Restoring Snapshots

A Persistent Actor's life-cycle is slightly different than a plain Actor – it provides a mechanism to restore its state. This is accomplished by defining the recoveryBehavior:

```
ActorBehavior recoveryBehavior() {
      return message → {
            switch(message) {
                  case SnapshotMsg → {
                        Object snapshot = ((SnapshotMsg) message).getSnapshot();
                        // Do something to setup the state then ..
                        become(createBehavior()); // Continue as usual by ..
                        unstashAll(); // .. processing any messages received while
                                      // waiting to recover the state
                  }
                  default → stash(message) // Only handle SnapshotMsg
            }
      }
}
```

The recoveryBehavior expects a SnapshotMsg which Dust will deliver. getSnapshot() retrieves the de-serialized state which recoveryBehavior should use to setup up the Actor and then, typically, move to a different behavior.

If no snapshot has yet been saved getSnapshot() will return null – the SnapshotMsg is always delivered.

4.4 Post Recovery

Persistent Actors can override the method

```
protected void postRecovery() {}
```

Dust calls this method after the SnapshotMsg has been sent but otherwise makes no guarantees just when it will be called. After postRecovery() is called preStart() is called as usual and from there on we are into the usual Actor life-cycle.

4.5 Persistence Storage

Storage mechanisms can be user defined by defining a class that implements:

public interface PersistenceService {

```
    void write(String id, Serializable object) throws Exception;

    /**
     * Delete the object with id if it exists. This allows us to exit cleanly even
      * when a snapshot has not
     * been saved yet.
     * @param id
     * @throws Exception
     */
    void delete(String id) throws Exception;

    Serializable read(String id) throws Exception;

    Serializable read(String id, Class<?> clz) throws Exception;
}
```

By default persistent Actors use the persistence service defined in the ActorSystem. This is set by the method call:

```
  setPersistenceService(PersistenceService persistenceService)
```

On the ActorSystem instance.

There is no default persistence service defined for ActorSystem so a program *must* set this before persistence is used.

A persistent Actor can set its own strategy by overriding the PersistentActor method:

```
protected persistenceServices setPersistence(PersistenceService persistenceService)
  throws IOException
```

Two current out of the box solutions are file system serialization using FST or Gson.

4.5.1 FST

FST persistence can be found in FSTPersistenceService. This defines a factory method to set persistence up:

```
  public static PersistenceService create(String path)
```

and use it in setPersistence():

```
    protected void setPersistence() throws IOException {
        persistenceService = GsonPersistenceService.create(“/home/snapshots”);
    }
```

State will be saved in .snap files in the defined directory.

4.5.2 Gson

An alternative Persistence service uses Gson for serialization and can be found in GsonPersistenceService. This defines a factory method to set persistence up:

```
public static PersistenceService create(String directory, String extension)
```

and use it in setPersistence():

```
    protected void setPersistence() throws IOException {
        persistenceService = GsonPersistenceService.create(
            “/home/snapshots”,
            “json”
        );
    }
```

There is one wrinkle using Gson persistence – it needs to know the Class that represents your Actor’s state in order to de-serialize it. PersistentActors feature an overrideable method:

```
    protected Class<?> getSnapshotClass() {
        return null;
    }
```

which can be used to define the state class.

4.5.3 Jackson

This is a json/file based Persistence service like Gson but using the FasterJackson persistence engine. Gson have different strengths and weaknesses.

- Gson is generally faster than Jackson
- Gson needs to be given the class it is de-serializing, Jackson does notified
- Jackson is more forgiving (via @nnotations) with missing fields (on either side) of the serialize/de-serialize divide. This is a great advantage during development when the serialized objects may be in flux.

4.6 Persistence and Stopping

Any Actor can stop for one of a number of reasons, but this becomes particularly important where Persistence is concerned.

- The Actor’s job is finished and that is why it is stopping. In this case there is no need to save its state (in fact it probably wants to delete it).
- The ActorSystem was shutting down i.e. the App is being turned off. If done gracefully (stopping the Actor System) this will cause the Guardian Actor ‘/’ to stop and therefore

(eventually) all Actors in the ActorSystem will stop. But in this case we probably want persistent Actors to save their state.

Persistent Actors support the method

```
boolean isInShutdown()
```

to detect when the ActorSystem is being shut down. e.g.

```
@Override
void postStop() {
      if (isInShutdown()) {
            saveSnapshot(myState)
      } else // we are stopping ourselves so presuambly do not save state
            deleteSnapshot()
}
```

5 Idiomatic Dust

So far we have seen how to create and instantiate Actors and how to pass messages between them and have them respond to those messages. In a certain sense that is the core of Dust and you now know how to use Actors. Congratulations!

That said possibly the hardest thing about Dust (and the Actor paradigm in general) is how to use it idiomatically. Some idioms are so common and powerful that we have incorporated them into dust-core and we will now discuss idiomatic Dust in more detail. Many of these idioms come about because ….

5.1 Actors are cheap and light weight[12]

On modest desktop hardware Dust can instantiate about 1 million Actors per second and the memory overhead for an Actor is typically < 1 Kbyte. Messages can be passed between local Actors at the rate of several million per second.

These facts drive many of the idioms, may of which are driven from: ***Do not be afraid to spin up an Actor to do a simple, well-contained job, pass on its results and then die.***

The Dust core library contains several lightweight Actors which can be combined in different to build scalable applications.

5.1.1 ServiceManagerActor

A ServiceManagerActor is given a configurable number of ‘slots’ and the Props to create children[13]. Each time it receives a message it looks to see if it has a spare slot – if it does it creates a child and hands off the message to it, while watch() ing the child. If not the message is queued.

12 Hence the name Dust.
13 Only one instance of Props is given, so all the children will be identical.

The child follows a Service pattern[14]: it processes the message, replies if necessary (setting the sender to its parent so it looks as though the Service Manager processed the message) and then it stops itself.

Since the child was being watch()ed by its parent, the ServiceManagerActor knows a slot has opened up, and it can dequeue the next message and repeat the process.

This pool of service workers is very common (and simple). Since the Actors actually doing the processing are created to specifically handle just one task they do not have to manage complicated state and there are no concerns of thread safety. They just do their job and quit.

The ServiceManagerActor is a part of Dust. Here is its implementation in its entirety.

14 By convention we name them XXX..ServiceActor so we know how the are to be used.

```
public class ServiceManagerActor extends Actor {

    Props serviceProps;
    int maxWorkers, currentWorkers;

    // [msg, sender]
    LinkedList<LinkedList<Object>> msgQ = new LinkedList<>();

    public static Props props(Props serviceProps, int maxWorkers) {
        return Props.create(ServiceManagerActor.class, serviceProps, maxWorkers);
    }

    public ServiceManagerActor(Props serviceProps, int maxWorkers) {
        this.serviceProps = serviceProps;
        this.maxWorkers = maxWorkers;
        currentWorkers = 0;
    }

    @Override
    protected void preStart() {
        self.tell(new StartMsg(), self);  // #1 Kick the process off ..
    }

    @Override
    protected ActorBehavior createBehavior() {
        return message -> {
            switch(message) {
                case StartMsg ignored -> {
                     // #2  If we have something to do – do it
                    if (msgQ.size() > 0 && currentWorkers < maxWorkers) {
                        LinkedList<Object> record = msgQ.removeFirst();
                        watch(actorOf(serviceProps))
                              .tell(record.getFirst(),(ActorRef)record.getLast());
                        ++currentWorkers;
                    }
                }
                // A slot has opened up so look to see if we have anything to do
                case Terminated ignored -> {
                    --currentWorkers;
                    self.tell(new StartMsg(), self);
                }

                // This is a task message. If we have a slot available create
                // a new actor, watch it and send the message to it -
                // otherwise queue it
                default -> {
                    if (currentWorkers < maxWorkers) {
                        watch(actorOf(serviceProps)).tell(message, sender);
                        ++currentWorkers;
                    }
                    else {
                        msgQ.add(new LinkedList<>(List.of(message, sender)));
                    }
                }
            }
        };
    }
}
```

1. StartMsg is a simple built-in message in Dust since it can be used in so many scenarios.
2. If we have a slot then create a new watched service actor and send it the next message in the queue.

Dust also contains a persistent version of this Actor – PersistentServiceManagerActor – which behaves in exactly the same way except its message queue is persisted so it can recover.

5.1.2 Delegated Actor

Delegated Actors are a stronger version of Behavior changing. When an Actor uses become() its behavior changes, but any state the Actor manages remains the same. This is fine for many cases where the behavior change is comparatively modest, but in the case where there is a wholesale change in what the Actor is doing having it maintain different states for different behaviors can rapidly become a mess. This is where Delegation comes in.

All Actors have a method:

```
void delegateTo(ActorRef targetRef, ActorBehavior thenBecome, Serializable thenMsg)
```

When called the following occurs:

- While targetRef is running (almost) all messages sent to the caller are immediately passed to the target. The caller's current behavior is bypassed.
- When the target halts:
  - The delegating Actor becomes() thenBecome
  - The delegating Actor sends itself the thenMsg (if not null)

This approach is very useful when an Actor is acting as a 'switch' do vastly different functionality e.g. consulting different 'experts' depending on the topic at hand. The plumbing in front of the switch remains the same – an expert gets chosen, all work is delegated to it until it finishes.

If a message implements the interface *NonDelegatedMsg* that message will not be delegated to targetRef but will be processed by whatever the current behavior of the delegating Actor is.

5.2 Pipelines

5.2.1 PipelineActor

Many computations can be split into a series of smaller, composable computations. The Unix pipe '|' command is a reflection of this where A | B | C means the output of program A is fed into program B whose output is sent to C. Dust supports a similar concept – the PipelineActor.

The PipelineActor's props method looks like:

```
    public static Props props(List<Props> props, List<String> names) {
        return Props.create(PipelineActor.class, props, names);
    }
```

The first list's props define the stages of the pipe (in list order) and the list of names (which should be unique) gives these Actors names.

When the PipelineActor is instantiated it creates its children from the list of Props and gives them the corresponding name, but, importantly, it recognizes the order of these Actors.

The Actors defined by props have a particular property. They respond to messages by creating other messages, but these messages are sent to their parent – the PipelineActor.

Since the PipelineActor keeps a track of the order of its children, when it receives a message from a child it knows who the 'next' child is, and it just sends it to that child.

Dust has a very simple, extensible ReturnableMsg class:

```
public class ReturnableMsg implements Serializable {
    @Getter
    ActorRef sender;

    public ReturnableMsg(ActorRef sender) { this.sender = sender; }
}
```

This is to allow the original sender of a message to be 'baked in' even if that message is passed through several different Actors and is useful in pipelines: If the final stage of the pipeline sends a message to its parent and that message is a Returnablemsg then it is sent to the sender in that message, otherwise the message is dropped.

5.2.2 PipelineBuilderServiceActor

A PipelineActor is determined by its stages, which are determined by a series of Actor props used to build the pipeline. A PipelineBuilderServiceActor is an Actor which consumes messages which define a Pipeline and returns the constructed PipelineActor to the sender. Thus Pipelines may be created *dynamically.*[15] Once the PipelineActor is constructed the PipelineBuilderServiceActor dies.

A PipelineBuilderServiceActor responds to three message types. In order they are

1. StartMsg – this sets things up.
2. A series of StageMsgs – these have pairs of Props/Names which define a stage in the pipe. The order in which these messages are sent determines the order of the stages in the pipe.
3. A StopMsg – this indicates the PipelineActor is complete. The sender receives a PipelineStagesMsg which contains the PipelineActor Props. The recipient can then use actorOf() to build the Pipeline where it wants.

15 This is particularly useful for creating Agentic systems using Large Language Models.

5.3 PubSubActor

A common pattern (in applications in general, not just Actor-based ones) is a source of events, and a number of clients who are interested in hearing about those events. This is the so-called publish/subscribe pattern.

In Dust we split this functionality out into a separate Actor which the event source can spin up. Clients use a common mechanism to subscribe with this Actor, which, when the event source sends it a message, just passes the messages on to all the subscribers.

PubSubActor allows clients to specify the kind of messages they are interested in:

```
public class PubSubActor extends Actor {

    public static Props props() { return Props.create(PubSubActor.class); }
    /*
     * Maps fully qualified class name of message of interest to maps of
     * registrants for that class. Registrants map their Actor Path -> ActorRef
     */
    final protected HashMap<String, HashMap<String, ActorRef>> subs =
                  new HashMap<String, HashMap<String, ActorRef>>();

    protected ActorBehavior createBehavior() {
        return message -> {
            if (message instanceof PubSubMsg msg) {
                HashMap<String, ActorRef> registrants =
                        subs.get(((PubSubMsg)message).fqn);
                if (msg.subscribe) {
                    if (null == registrants) {
                        registrants = new HashMap<String, ActorRef>();
                        registrants.put(sender.path, sender);
                        subs.put(msg.fqn, registrants);
                    } else
                        registrants.computeIfAbsent(sender.path, k -> sender);
                } else {
                    if (null != registrants) {
                        registrants.remove(sender.path);
                    }
                }
            }
            else if (message instanceof _Publish) {
                _Publish msg = (_Publish)message;
                if (msg.subs.size() > 0) {
                    msg.subs.removeFirst().tell(msg.msg, msg.sender);
                    self.tell(message, self);
                }
            }
            else {
                HashMap<String, ActorRef> registrants =
                             subs.get(message.getClass().getName());
                if (null != registrants) {
                    _Publish publish = new _Publish();
                    publish.subs =  new LinkedList<ActorRef>(
                        registrants.values().stream().toList()
                    );
                    publish.sender = sender;
                    publish.msg = message;
                    self.tell(publish, self);
                }
            }
        };
    }

    static protected class _Publish implements Serializable {
        public ActorRef sender;
        public Object msg;
        public LinkedList<ActorRef> subs = new LinkedList<ActorRef>();

        public _Publish() {};
    }
}
```

To subscribe to or unsubscribe from a PubSubActor the client sends it a PubSubMsg specifying the class of message it wishes to subscribe to.

```
public class PubSubMsg implements Serializable {
    public Boolean subscribe = true; // or unsubscribe
    public String fqn;

    /**
     * Subscribe to messages
     * @param clz - class of message to subscribe to
     */
    public PubSubMsg(Class clz) {
        this.fqn = clz.getName();
    }

    /**
     * (Un)Subscribe to messages
     * @param clz - class of message to (un) subscribe
     * @param subscribe - if true then subscribe else unsubscribe
     */
    public PubSubMsg(Class clz, Boolean subscribe) {
        this.fqn = clz.getName();
        this.subscribe = subscribe;
    }
}
```

For as long as the client remains subscribed it will receive matching messages. The sender of these messages will be the ActorRef for the event source, not the PubSubActor.

5.4 ProxyMsg

A common scenario is when one Actor wishes to send a message to another but it requires some intermediate processing (e.g. throttling, below). For this we have the extensible ProxyMsg class. This is a ReturnableMsg, so it has a permanent record of its sender and it has a ActorRef target that may or may not be null.

When a Proxying Actor (an Actor that knows what to do with ProxyMsgs) receives the ProxyMsg subclass it does whatever processing it is required to do and possibly updates the message. It sets the proxied flag to true and then, if target it not null it tells() the target the message, otherwise it tells() the sender (as determined by the sender field) the message.

```
public class ProxyMsg extends ReturnableMsg {

    boolean proxied = false;

    /**
     * Eventual target of the message.
     */
    public ActorRef target = null;

    public ProxyMsg(ActorRef sender) {
        super(sender);
    }

    public ProxyMsg(ActorRef sender, ActorRef target) {
        super(sender);
        this.target = target;
    }
    /**
     * If the target is not null then by convention we say it is not accepted
     * @return true if not accepted else false
     */
    public boolean isProxied() { return proxied; }

    /**
     * If we wish to reprocess this message at the target e.g. send it to myself
     * again then we use notAccepted() to disambiguate this case and then accept
     * so that when I reprocess it I will know
     */
    public void setProxied(boolean value) { proxied = value; }
}
```

5.5 ThrottlingRelayActor

Another common situation is when there is some time-restricted resource access to which needs to be managed. For instance a polite web crawler might want to restrict itself to visiting web pages on a given site to 1 / second. We could construct a web crawling Actor with this limit built in but a better way is to factor this notion of throttling into a generic Actor which can be used in multiple situations.

Dust provides the ThrottlingRelayActor for just this purpose. The idea is simple: the Actor receives ProxyMsgs and runs a scheduleIn loop. A queue of received ProxyMsgs is maintained and is drained by the periodic message in the loop. The ProxyMsg is then marked as proxied and sent to its destination (the target or the sender of the message).

```
public class ThrottlingRelayActor extends Actor {

    Long intervalMS;
    LinkedBlockingQueue<ProxyMsg> q = new LinkedBlockingQueue<>();
    Cancellable pump;

    /**
     * @param intervalMS - interval between send-on attempts in MS
     * @return
     */
    public static Props props(Long intervalMS) {
        return Props.create(ThrottlingRelayActor.class, intervalMS);
    }

    @Override
    public void preStart() {
        pump = scheduleIn(new StartMsg(), intervalMS);
    }

    @Override
    public void postStop() {
        pump.cancel();
    }

    public ThrottlingRelayActor(Long intervalMS) {
        this.intervalMS = intervalMS;
    }

    protected ActorBehavior createBehavior() {
        return message -> {
            switch(message) {
                case StartMsg msg -> {
                    ProxyMsg p;

                    if (null != (p = q.poll()))
                    {
                        p.setProxied(true);
                        if (null != p.target)
                            p.target.tell(p, p.getSender());
                        else
                            p.getSender().tell(p, p.getSender());
                    }
                    pump = scheduleIn(msg, intervalMS);
                }
                case ProxyMsg msg -> {
                    q.add(msg);
                }
                default -> log.error(
                  “%s got not Proxy message %s".formatted(self.path, message));
            }
        };
    }
}
```

5.6 Reaping

Consider an Actor managing a number of identical child Actors – e.g. an Actor ‘companies’ each of whose children is watching, in some sense, news about a particular company. Some

other Actor (perhaps connected to a UI) wants to answer the question ‘show me the most recent articles about all the companies you know about’.

Clearly what we want to do is send a message representing this request to the companies Actor. Logically what we want to do is have the companies Actor send some request to all of it children, and then when all the children have replied with the answer to the ‘question’ the companies Actor should send the cumulative results back to the requester.

We call this idiomatic request ‘reaping’ inspired by the idea of harvesting a field of data.

Now of course we could add code to each parent Actor we wish to exhibit this behavior, but in true Dust tradition we factor this out into an idiom – a ReaperActor.

5.6.1 ReaperActor

An Actor which wishes to reap its children creates a child ReaperActor for this job. The ReaperActor will reply to the requester with the appropriate information and then, in true Dust tradition, die.

The core of this mechanism is the (subclass of) ReapMsg.

A ReapMsg needs to specify two things:

1. The list of ActorRefs to send some message to. This is kept in the field targets.
2. What the message to be sent is. A ReapMsg contains two ways of doing this:
    a) Setting a Class (via the clz field) that has a 0 parameter constructor or
    b) Setting a CopyableMsg (via the copyableMsg field). A CopyableMsg is an interface that simply defines a copy() method

A ReapMsg also contains a ReapResponseMsg (in field response) which in turn contains a map of ActorRef → Object where the ActorRef is one of the targets and Object is the value returned by that target.

The behavior of a ReapActor is as follows:

```
protected ActorBehavior createBehavior() {
    return message -> {
        switch(message) {
            case ReapMsg msg -> {
                client = sender;
                reapMsg = msg;
                if (!reapMsg.targets.isEmpty()) {
                    if (null != reapMsg.clz)
                        for (ActorRef t : reapMsg.targets) {
                            t.tell(
                             reapMsg.clz.getDeclaredConstructor().newInstance(),
                             self
                            );
                        }
                    else if (null != reapMsg.copyableMsg)
                        for (ActorRef t : reapMsg.targets) {
                            t.tell(reapMsg.copyableMsg.copy(), self);
                        }
                    else
                        self.tell(new StopMsg(), self);
                }
                else {
                    self.tell(new StopMsg(), self);
                }
            }
            case StopMsg ignored -> {
                client.tell(reapMsg.response, null);
                cancelDeadMansHandle();
                stopSelf();
            }
            default -> {
                reapMsg.response.results.put(sender, message);
                if (reapMsg.isComplete()) {
                    reapMsg.response.complete = true;
                    self.tell(new StopMsg(), self);
                }
            }
        }
    };
}
```

So on receipt of a ReapMsg it creates new instances / copies of the message to be sent and tells() the targets. It then waits for responses, and adds them to the map. If the map is complete (by comparing its size to the targets size) it sends the completed ReapResponseMsg back to the requester.

The ReapActor sets up a dead man’s handle with a configurable timeout. The ReapActor overrides the dying() method to return partial results. ReapResponseMsg contains a field (complete) that indicates if the ReapActor believes the data is complete or not.

5.6.2 ReapTransformServiceActor

ReapTransformServiceActor is a refined ReaperActor. It is a service Actor so it will stop itself when it job is done.

Its props takes three arguments:

1. ActorRef host – the parent of the children which will be reaped
2. ActorRef client – the Actor to which the response will be sent
3. Function<Object, Serializable> transform – a Function which takes a single argument (the ReapResponseMsg from the Reap), optionally applies a transformation to it which should result in a Serializable. That result will then be tell()'d to the client.

The createBehavior for this Actor is as follows:

```
protected ActorBehavior createBehavior() {
        return (Serializable message) -> {
            switch(message) {
                case StartMsg ignored:
                    host.tell(new GetChildrenMsg(), self);
                    break;

                /*
                 Warning. Common usage pattern is to invoke this Actor on the host
                 itself. But this means self is now a child of host which would
                 result in an attempt to reap self .... so filter me out.
                */
                case GetChildrenMsg msg:
                    List<ActorRef> childs = msg.getChildren()
                        .stream()
                        .filter(child -> child != self).toList();

                    actorOf(ReaperActor.props(10000L)).tell(
                        new ReaperActor.ReapMsg(
                            reapingClz,
                            childs,
                            ReapResponseMsg.class
                        ),
                        self
                    );
                    break;

                case ReapResponseMsg msg:
                    client.tell(transform.apply(msg), parent);
                    stopSelf();
                    break;

                default: super.createBehavior().onMessage(message);
            }
        };
    }
```

Processing is kicked off on receipt of a StartMsg to which the ReapTransformServiceActor reacts by getting the host's children. It is very common for the host to be reaping its own children, in which case the ReapTransformServiceActor will be in the list, so it filters itself out.

It then creates a child ReaperActor which perfoms the Reap (and which will die when it dies). The ReapResponseMsg from this Reap is then passed through the transform and the result sent to the client. Job done, the ReapTransformServiceActor dies.

For instance here is a sample from an actual application that has the notion of ‘Lenses’ which are all children of lensesRef. On receipt of a GetLensesMsg the handler Reaps the children of lensesRef asking for their state and transforms the state into a list of Lens objects which it sends to the requestor:

```
case GetLensesMsg:
       final GetLensesMsg msg = (GetLensesMsg)message
       actorOf(ReapTransformServiceActor.props(
              lensesRef,
              sender,
              GetStateMsg,
              {
                     ReaperActor.ReapMsg.ReapResponseMsg rrm =
                                   (ReaperActor.ReapMsg.ReapResponseMsg)it
                     rrm.results.values().each {
                            LensActor.LensState state =
                                   (LensActor.LensState)((GetStateMsg)it).state
                            msg.lenses <<
                            new Lens(
                                   state.name,
                                   state.description,
                                   state.lensId,
                                   state.topic
                            )
                     }
                     msg
              }
       )).tell(new StartMsg(), self)
       break
```

5.7 Future Idiomatic Dust ?

It is interesting to point out that the Props class is serializable. So it is quite feasible to send messages to an Actor which gives it instructions for how to build its children. This would seem to be a tantalizing dark corner of potentially idiomatic Dust that deserves further exploration.

6 Entities and Digital Twins

Dust Actors are units of computation. We have seen how they can be combined in idiomatic ways to make scalable, reliable systems. However there is another way to interpret what Actors may represent – as Entities.

An Entity is simply something which exists in the 'real world'[16]. It might be a concrete object – a school bus, an office, a company, a person, a network interface or it might be a more nebulous entity – a degree of belief, a competitive relationship, a sentiment, or an inference.

This process of modeling real-world entities with a digital representation is often called Digital Twinning[17].

6.1 EntityActor

We have a core class, EntityActor, which allow easy implementation of Actors which twin a corresponding Entity by simply following conventions and exploiting idiomatic dust.

Entities have state and so are represented by PersistentActors. We also assume that an Entity has a unique identity which we identity with its *name*[18]. How this name is determined is entirely up to the implementer, but there should also be a way, given a name and an Entity type, to determine whatever state the EntityActor might need to be initialized.

For example, suppose we have EntityActors representing companies, and we have a database of company information with a uuid as a key. We would use the uuid as the name of the twinning Actor, and if it needed to initialize itself it could use its name to extract information from the database.

Below is the entire code defining the EntityActor superclass.

First note that we assume the existence of a field T state where T is provided type. If the usual PersistentActor recovery path restores state then we just enter our createBehavior().

If we do not have persisted state then we enter a new behavior which waits to receive state in a EntityStateMsg. This message is to be provided by a user-supplied getState(String name) method where name is, by default, the name of the Entity.

On receipt of an EntityStateMsg the state is updated, behavior switches to createBehavior() and an optionally override postGetState() is called. Finally any stashed message are unstashed.

Finally there is the notion of life-cycle. If an EntityActor is shutting down then the assumption is that whatever it was twinning has come to the end of its lifetime. In this case we want to

16 A better formulation would be 'outside of the computation', but then things can get all Matrixy ..

17 There are many definitions of just what a Digital Twin is, See for example https://www.digitaltwinconsortium.org/ or https://www.mckinsey.com/featured-insights/mckinsey-explainers/what-is-digital-twin-technology. We think EntityActors certainly pass the test – you be the judge.

18 This is one of the few natural cases where the identity of an Actor is determined completely by its name.

delete the snapshot. However if the whole ActorSystem is shutting down the default assumption is we want to save the snapshot, so it is generally a good idea for a subclass to make sure the EntityActor’s postStop() is called.

```
public abstract class EntityActor<T extends Serializable> extends PersistentActor {

    protected T state;

    /**
     * Generally if an EntityActor is stopping it means it is at the end of its
     * lifecycle and so should delete its snapshot. If however we are shutting down
     * then we want to do the opposite and ensure our state is saved.
     */
    @Override
    protected void postStop() {
        if (isInShutdown()) {
            saveSnapshot(state);
        }
        else {
            deleteSnapshot();
        }
    }
    /**
     * Override this to supply state to the Actor if it has not restored any state
     */
    protected void getState(String name) {
       self.tell(new EntityStateMsg<T>(null), self);
    }
    /**
     * To be overriden. Called after we receive an
     * EntityStateMsg and have switched to our createBehavior
     * @param name
     */
    protected void postGetState(String name) {}

    protected ActorBehavior recoveryBehavior() {
        return message -> {
            switch(message)
            {
                case SnapshotMsg msg -> {
                    state = (T) msg.getSnapshot();
                    if (null == state)
                    {
                        become(waitForStateBehavior());
                        getState(self.name);
                    }
                    else {
                        become(createBehavior());
                        unstashAll();
                    }
                }
                default -> stash(message);
            }
        };
    }

    protected ActorBehavior waitForStateBehavior() {
        return message -> {
            switch(message) {
                case  EntityStateMsg msg -> {
                    state = (T) msg.getState();
                    become(createBehavior());
                    postGetState(self.name);
                    unstashAll();
                }
                default -> stash(message);
            }
        };
    }
}
```

6.2 PodManagerActor

Like a ServiceManagerActor a PodManagerActor manages a collection of children all created by the same Props. However in this case the children are identical Entity Actors.

The PodmanagerActor is a PersistentActor and its state is a list of the names of its children. When the PodManagerActor is recovering it recreates its children under these names.

As we have seen above when an EntityActor recovers it has a way to restore its state, either from pre-existing snapshots or by the provided getState() mechanism. Thus a ‘pod’ of Entities is robustly managed.

The class CreateChildMsg

```
public class CreateChildMsg implements Serializable {
    @Getter
    String name;

    @Getter
    Object msg;

    public CreateChildMsg(String name) {
        this.name = name;
        msg = null;
    }

    public CreateChildMsg(String name, Object msg) {
        this.name = name;
        this.msg = msg;
    }
}
```

is how other Actors request a PodManagerActor to create a named child. Optionally the message may itself contain a message, in which case the PodManager sends this message (under the identity of the original sender) to the child.

The default supervision strategy for PodManagerActors is to let a child stop if it encounters an error. Of course PodManagerActor can be subclassed with a different supervision strategy.

```
public class PodManagerActor extends PersistentActor {

    Props childProps;
    HashMap<String, Boolean> kids;

    public static Props props(Props childProps) {
        return Props.create(PodManagerActor.class, childProps);
    }

    public PodManagerActor(Props childProps) { this.childProps = childProps; }

    @Override
    protected ActorBehavior recoveryBehavior() {
        return message -> {
            switch(message)
            {
                case SnapshotMsg msg -> {
                    kids = (HashMap<String, Boolean>) msg.getSnapshot();
                    if (null == kids) kids = new HashMap<>();
                    for (String name :kids.keySet())
                    {
                        watch(actorOf(childProps, name));
                        kids.put(name, true);
                    }
                    become(createBehavior());
                    unstashAll();
                }
                default -> stash(message);
            }
        };
    }

    @Override
    protected ActorBehavior createBehavior() {
        return message -> {
            switch(message) {
                case Terminated ignored -> {
                    kids.remove(sender.name);
                    saveSnapshot(kids);
                }
                case CreateChildMsg msg -> {
                    ActorRef child = actorOf(childProps, msg.getName());
                    kids.put(msg.getName(), true);
                    if (null != msg.getMsg()) {
                        child.tell(msg.getMsg(), sender);
                    }
                    saveSnapshot(kids);
                }
                case DeadLetterProxyMsg msg -> {
                    self.tell(new CreateChildMsg(msg.name, msg.msg), msg.sender);
                }

                case RegisterPodDeadLettersMsg ignored -> {}

                default -> super.createBehavior().onMessage(message);
            }
        };
    }
}
```

6.3 PodDeadLetterActor

While it is easy to create a new Entity under a PodManagerActor by sending the manager a CreateChildMsg it is somewhat clumsy.

Consider a fictitious Dust application: Pipelines read business news articles and perform NLP on them to detect company references. Also assume we have a database of companies that given a company name will give us a unique identifier (say a uuid). Finally assume we are twinning companies with Entity Actors which will track the mentions for a particular company, possibly gauge sentiment or track major business developments etc.

Pipelines can't just send messages to these company tracking Actors because the company might not have been mentioned yet and so no tracking Actor exists. An alternate approach would be to send the message to the pod manager which then checks to see if the child exists and creates it if it doesn't. While this would work pipelines are not really talking to who they want to talk to and are relying on somewhat hidden processing with the manager.

The best approach would be to have the pipeline send the message to the Entity Actor which (somehow) gets created if needed and then receives the messages.

The 'somehow' is the PodDeadLetterActor.

We previously discussed the DeadLetterActor which is the Actor messages are delivered to if the target Actor does not exist. However (surprise) DeadLetterActor is a subclass of PubSubActor so other Actors can subscribe to be informed of dead letters.

When created a PodDeadLetterActor subscribes for dead letters with the DeadLetterActor. It also waits to receive:

```
public  class RegisterPodDeadLettersMsg implements Serializable {
    public List<Class> acceptedMessages = List.of();
}
```

This registers the sender (PodManager) and optionally associates with it a list acceptable message classes (acceptedMessages).

When a dead letter is sent to the PodDeadLetterActor the sender and recipient are included. The recipient is examined and if it is the **child** of a registered subscriber (and optionally the message class matches) then the message is wrapped up in **DeadLetterProxyMsg**() which is sent to the pod manager. On receipt of this the PodManager creates the child and sends it the message (as though from the original sender).

7 Dust and the Outside World

7.1 CompletionServiceActor

The Dust Actor ecosystem is simple. There are Actors and they pass messages between themselves and respond accordingly. However at some point this infrastructure has to meet the ‘outside world’ which is usually modeled as some form of an API.

If the API has access to the ActorSystem then it has access to the context and thence to ActorSelection and the ability to send a message to an addressed Actor. The API though is not usually an Actor (though one could arduously construct such an Actor) and so it cannot easily receive the consequences of any such messages it sent out.

Again we could fill this void by using CompleteableFutures on a case by case basis, but in the Dust tradition (because an Actor weighs as much as a dust particle) we rely on Actor creation to handle this impedance mismatch.

So let us model the following. We have an API which needs to get the response of a computation from an Actor so it can return the response to its caller. The API gets the ActorSystem’s context and uses it to send an appropriate message which is a subclass of:

```
public class CompletionRequestMsg<Response> extends ProxyMsg {

    @Getter
    CompletableFuture<Response> future;

    /**The message is sent to the target.
     * @param target - target of proxy message
     * @param future - user supplied Future to be set
     */
      public CompletionRequestMsg(
            ActorRef target,
            CompletableFuture<Response> future
      ) {
        super(null);

        this.target = target;
        this.future = future;
    }
}
```

The API creates a future and initializes the message with it and the target Actor the message is to be sent to **but** it sends it to a ServiceManagerActor managing a set of Proxy Actors:

```
public class CompletionServiceActor extends Actor {

    CompletableFuture<Object> future;

    public static Props props() {
        return Props.create(CompletionServiceActor.class);
    }

    protected ActorBehavior createBehavior() {
        return message -> {
            switch(message) {
                case CompletionRequestMsg msg -> {
                    future = msg.getFuture();
                    msg.setSender(self);
                    msg.target.tell(msg, self);

                }
                default -> {
                    future.complete(message);
                    context.stop(self);
                }
            }
        };
    }
}
```

When it receives the CompletionRequestMsg the CompletionServiceActor stores the future, sets itself as the sender and sends the message off to the target. Whatever processing is required to process the result takes place eventually resulting in a message being sent back to the service Actor. This message is used to complete the future and the service Actor, having done his job, dies.

Note that the Response message should not be a CompletionRequestMsg.