Dust Actors and Large Language Models – An Application

The Setting

In a previous article we introduced Dust (https://dzone.com/articles/dust-open-source-actors-for-java) an open source Actor System for Java 21 and above (https://github.com/dust-ai-mr/dust). We explained the basic ideas behind it and gave a tiny complete example consisting of two Actors ping-ponging messages between each other.

Assuming you have read that article the time has come to move on to bigger things, so in this article we will show you how to use Dust Actors to build a small demonstration application which will:

- Take a topic of interest and a list of names of entities that you are interested in being notified about. For instance
  - topic: Electric Vehicle Charging
  - entities: Companies, Technologies, Locations
- Automatically find valid RSS news feeds that would supply news about that topic
- Automatically set up Actors to periodically read those news feeds, confirm they match the topic, extract the main content of the article and provide a list of matching entities found in the content

We make extensive use of ChatGPT so to run the code you'll have to provide your own key (or see dust-nlp for how to use a local instance of Ollama).

For succinctness we'll use the Groovy scripting language in this article – think of it as Java without the ';'s. We only show the important snippets of the code, the rest can be found at (link).

This application uses code from most of the Dust repos – dust-core, dust-http, dust-html, dust-feeds, and, most importantly dust-nlp.

Pipeline Actors

In the previous article we saw how to create an instance of an Actor class by passing its Props (returned by SomeActor.props()) to a method (actorOf()) which actually created the Actor. A PipelineActor is another way to create instances of Actors except they become 'stages' in a sequential pipe. For instance:

```
actorOf(PipelineActor.props(
      [
            RssLocatorActor.props(topic, chatGPTRef),
            FeedHubActor.props(),
            ServiceManagerActor.props(
                  ContentFilterServiceActor.props(topic, chatGPTRef),
                  4
            ),

            ServiceManagerActor.props(
                  EntitiesExtractionServiceActor.props(entities, chatGPTRef),
                  4),
            LogActor.props()
      ],
      [
            'rss-locator',
            'feeds',
            'topic-filter',
            'entities-extraction',
            'logger'
      ]
      ), 'pipeline'
)
```

This creates just about all the Actors we need to implement the application. It is creating (using actorOf()) an instance of PipelineActor from PipelineActor's props. These props take two arguments – a list of other Actor Props, and a list of names for the Actors to be created. In this case we will have 5 Actors – all children of the PipelineActor (whose name will be 'pipeline'). In order, the stages are:

1. An instance of an RssLocatorActor which is passed the topic and an ActorRef to an Actor interacting with ChatGPT (more on that later).
2. An instance of a FeedHubActor
3. A ServiceManagerActor managing ContentFilterServiceActors
4. A ServiceManagerActor managing EntitiesExtractionServiceActors
5. A logging Actor

A pipeline works as follows – messages sent to the PipelineActor from stage N of the pipe are sent on to stage N+1. If a message is sent to the pipe from outside of it it is sent to stage 1, and if a message leaving the pipe has a 'return to sender' reference it will be sent to that Actor, otherwise it is dropped. In our case messages are introduced into the pipe from RssLocatorActor (as it is given the topic, you can guess its job).

So the pipeline starts by looking for feeds matching the topic ('rss-locator'), it generates messages (just urls for discovered feeds in this case) and sends them back to the pipe, whence they go to 'feeds' who will set up Actors to continuously monitor those RSS feeds, 'feeds' sends any content it finds (as messages) back to the pipe where they go to 'topic-filter'. Here they are checked (using ChatGPT) to see that the article really is about the topic and not something just slightly related (e.g. about EVs but not charging in particular). If it

passes this filter it is sent back to the pipe and picked up by ‘entities-extraction’. This Actor recognizes specified named entities which are passed to the final stage, which simply logs them.

ServiceManager and Service Actors

What is going on with stages 3 & 4 above? This is another design pattern in Dust. A ServiceActor is any Actor designed to accept one message from outside, process it in some fashion and then die when finished.

A ServiceManageActor manages a fixed-size pool of identical Service Actors (defined by the Props in the first argument). The maximum size of the pool is defined by the second argument. When it receives a message it waits until it has space in the pool then it creates a new child (from the supplied Props) and sends it the message *as though it came from the original sender.* If the child is to reply to the sender (which is usually the case) it uses its parent (the ServiceManager) as its *from* address. So Actors send messages to the ServiceManager and appear to receive responses back from the ServiceManager – just who processed it is hidden, and to add insult to injury it costs them their life.

Service managers act as a natural throttler, and since Service Actors only ever process one external message their internal state can be very simple.

News Reader Application

That was a lot to digest, but now we have covered the ideas involved we can get down to the details.

Usually with a Dust application we create the ActorSystem and then some top level Actor under ‘/user’. We let that Actor set everything else up and ‘run’ the application. So our main() method is basically:

```
ActorSystem system = new ActorSystem('news-reader')
// Get topic and entities
URL resource = system.class.classLoader.getResource("reader.json")
Map config = new JsonSlurper().parse(new File(resource.toURI())) as Map

// Root does everything else – it is created under /user by system.context
system.context.actorOf(
      RootActor.props(
         (String)config.topic,
         (List<String>)config.entities
), 'root').waitForDeath()
```

We get our topic and entities from the Json config file, create an instance of a RootActor passing it the defined topic and entities and then wait for it do die (since everything from this point on is running on separate virtual threads we’d simply fall off the end of the application if we did not wait).

RootActor

As a RootActor's only real job is to set things up and since it won't be receiving any messages we simply leave it with default Actor Behavior and set things up in its preStart() method:

```
void preStart() {
      ActorRef chatGPTRef = actorOf(ServiceManagerActor.props(
                  ChatGptAPIServiceActor.props(null, key),
                  4
            ),
            'chat-gpt'
      )
      actorOf( PipelineActor.props( [
            RssLocatorActor.props(topic, chatGPTRef),
            FeedHubActor.props(),
            ServiceManagerActor.props(
                  ContentFilterServiceActor.props(topic, chatGPTRef), 4
            ),
            ServiceManagerActor.props(
                  EntitiesExtractionServiceActor.props(entities, chatGPTRef), 4
            ),
            LogActor.props()
            ], [
                  'rss-locator',
                  'feeds',
                  'topic-filter',
                  'entities-extraction',
                  'logger'
            ]
            ), 'pipeline')
      }
```

This creates two Actors – a pool of ChatGptAPIServiceActors (see dust-nlp) and our previously seen main pipeline. A reference to the ChatGPT processing Actors (chatGPTRef) is passed into the Props of the pipe stages that will need LLM support.

RssLocatorActor

The job of this Actor is to find RSS feeds which might have news articles matching the topic, and we take the obvious approach of asking ChatGPT. *But,* as we all know ChatGPT sometimes tries too hard to be helpful. In particular it may give us urls which don't exist, or which exist but are not RSS feeds, so we need to do a second verification pass on whatever ChatGPT tells us.

RssLocatorActor preStart() sends a StartMsg to itself, which sends an instance of ChatGptRequestResponseMsg to ChatGPT. This asks for a list of feeds matching the topic. ChatGPT sends us this message back with its response and we use a helper method (listFromUtterance) to parse out the list of URLS. We bundle these up into a VerifyFeeds message which we repeatedly send to our self to process the urls one by one. For each each one we try to treat it as a valid feed by fetching it. If we get a response we try to parse it as a syndication message. If it is, we pass its url back to our parent (the pipeline). If not we ignore it.

```
   ActorBehavior createBehavior() {
      (message) -> {
         switch(message) {
            case StartMsg:
               chatGPTRef.tell(new ChatGptRequestResponseMsg(
 """Consider the following topic and give me a numerical list consisting *only* of urls for
RSS feeds that might contain information about the topic: '$topic'. Try hard to find many
real RSS urls. Each entry should consist only of the URL - nothing else. Include no
descriptive text."""
                  ),
                  self
               )
               break

            case ChatGptRequestResponseMsg:
               ChatGptRequestResponseMsg msg = (ChatGptRequestResponseMsg)message
               List<String> urls = listFromUtterance(msg.utterance)
               self.tell(new VerifyFeedsMsg(urls), self)
               break

             /* RSSLocatorActor is a subclass of HttpClientActor (dust-http) which
              * provides a request() method. This does an http GET and sends back the
              * result in an HttpRequestResponseMsg
              */
            case VerifyFeedsMsg:
               verifyFeedsMsg = (VerifyFeedsMsg)message
               if (! verifyFeedsMsg.urls.isEmpty()) {
                  try {
                     request(verifyFeedsMsg.urls.removeFirst())
                  } catch (Exception e) {
                     log.warn e.message
                  }
               }
               break

            /*
             * Verify the feed. Does the site exist and is it a feed ?? If we got something
             * try and parse it as a feed. If this fails then simply warn and move on.
             */
            case HttpRequestResponseMsg:
               HttpRequestResponseMsg msg = (HttpRequestResponseMsg)message

               if (null == msg.exception && msg.response.successful) {
                  try {
                     new SyndFeedInput().build(
                          new XmlReader(msg.response.body().byteStream())
                     )
                     parent.tell(msg.request.url().toString(), self)
                  } catch (Exception e) {
                     log.warn "URL: ${msg.request.url().toString()} is not an RSS feed!"
                  }
               } else
                  log.warn "URL: ${msg.request.url().toString()} does not exist!"

               // Check next URL
               self.tell(verifyFeedsMsg, self)
               break
         }
      }
   }

static class VerifyFeedsMsg implements Serializable {
   List<String> urls
   VerifyFeedsMsg(List<String> urls) { this.urls = urls }
}
```

FeedHubActor

```
ActorBehavior createBehavior() {
  (message) -> {
    switch(message) {
      case String:
        String msg = (String) message
        log.info "adding RSS feed $msg"
        actorOf(
          TransientRssFeedPipeActor.props(msg, 3600*1000L)
        ).tell(new StartMsg(), self)
        break

      // From the TransientRssFeedPipeActor
      case HtmlDocumentMsg:
        parent.tell(message, self)
        break
    }
  }
}
```

When FeedHubActor receives a String it knows it is the URL of a valid, topic-related RSS feed. So it creates an instance of TransientRssFeedPipeActor whose Props parameters are the URL and how often (in milliseconds) to visit the feed for new articles (here every hour).

FeedPipeActors (in dust-feeds) need to receive a StartMsg to start up, so we send it one. The TransientRSSFeedPipeActor manages the whole job of parsing the feed content, visiting the linked referenced sites and sending their content (as HtmlDocumentMsg) to its parent – FeedHubActor. When FeedHubActor receives an HtmlDocumentMsg it sends it on to its parent – the Pipeline.

‘Transient’ here refers to the fact that its Actor state is not saved. So if the application is stopped and restarted it has no idea what it already saw. Repo dust-feeds also contains a persistent version of this Actor for more robust applications.

ContentFilterServiceActor

The job of this Actor is simply to check to see if the HtmlDocumentMsg refers to the topic in some way. We use an LLM again and check the title of the content. This can actually be quite powerful as with an article titled ‘Renault and The Mobility House launch V2G project in France’. The LLM knew V2G is ‘Vehicle to Grid’ and so the article is related to charging – I didn’t…. The Actor itself is quite simple: we ask ChatGPT if the document’s title is associated with the topic. If it is we pass the document on down the pipe; if not, we don’t. In either case we die, as we are a ServiceActor:

```
ActorBehavior createBehavior() {
   (message) -> {
      switch(message) {
         case HtmlDocumentMsg:
            originalMsg = (HtmlDocumentMsg)message
            String request =
   """Does '${originalMsg.title}' refer to '$topic'. Answer simply yes or no."""
            chatGPTRef.tell(new ChatGptRequestResponseMsg(request), self)
            break

         case ChatGptRequestResponseMsg:
            ChatGptRequestResponseMsg msg = (ChatGptRequestResponseMsg)message
            String response = msg.getUtterance()?.toLowerCase()

            if (response?.toLowerCase()?.trim()?.startsWith('yes')) {
               /*
                * The actual pipe stage is my parent (the service manager)
                * and my grand parent is the pipe, so send response back to
                * the pipe as though from my parent
                */
               grandParent.tell(originalMsg, parent)
            }
            stopSelf()
            break
      }
   }
}
```

EntitiesExtractionServiceActor

The last non-trivial Actor in the pipeline (we'll let you write a LoggingActor yourself!) is a service Actor to extract named entities from the core content of the document.

HtmlDocumentMsg (dust-html) has a method, getWholeText(), which analyzes the html content of the document, identifies the core content (strips out ads etc) and returns the resulting plain text.

We then pass this task off to ChatGPT asking it to give us a structured list back.

A helper method (fromEntitiesList()) parses the returned response – which will look something like

```
Companies:
1. Tesla
2. General Motors
Location:
1. Maryland
```

into a list of lists:

> [url-of-source, entity-name, entity-value]

We then send this list to our grandparent (the pipe) as though it came from our parent (the service manager) and the pipe passes it down to the logging stage.

```
ActorBehavior createBehavior() {
   (message) -> {
         switch(message)
         {
            case HtmlDocumentMsg:
               originalMsg = (HtmlDocumentMsg)message
               String mainText = originalMsg.getWholeText()

               if (mainText) {
                  String text = "${originalMsg.title} --- $mainText"
                  chatGPTRef.tell(
                     new ChatGptRequestResponseMsg(
"""Following is a list of entity categories: ${entities.join(', ')}.
  For each category give me a numerical list of mentions in the text.
  Precede each list with its category followed by ':'.
  Do not create new categories. Reply in plain text, not markdown.
  If the entity mentioned is a company use its formal name.\n\n ${text}
"""                      ),
                     self
                  )
               }
               else
                  context.stop(self)
               break

            case ChatGptRequestResponseMsg:
               ChatGptRequestResponseMsg msg =
                  (ChatGptRequestResponseMsg)message
               fromEntitiesList(msg.getUtterance())?.each {
                  if (it.value != []) {
                     grandParent.tell(
                        [originalMsg.source, it.key, it.value] as Serializable,
                        parent
                     )
                  }
               }
               stopSelf()
               break
         }
      }
   }
```

How Did it Do?

This little app is purely to show how Dust Actors interact cleanly with LLMs and how NLP pipelines can be easily constructed. A real application would do much more e.g.

- Check for duplicate content. Different RSS feeds often link to the same content (especially in this case where the feeds are all associated with the same topic).
- Do more with the end result – we simply log the entities. A real application might look for certain entities and trigger further actions on them – e.g. summarizing and presenting the article. This quickly leads to Agentic applications for Dust.
- Use persistent Actors (see dust-core)

That said how did it do? Our reader.json file contains

```
{
  "topic" :  "Electric Vehicle Charging",
  "entities":  ["Company", "Technology", "Product", "Location"]
}
```

And the log shows:

```
RssLocatorActor - URL: https://cleantechnica.com/feed/ does not exist!
RssLocatorActor - URL: https://www.greencarreports.com/rss/news does not exist!
FeedHubActor - adding RSS feed https://chargedevs.com/feed/
RssLocatorActor - URL: https://www.greencarcongress.com/index.xml exists but is not an RSS
feed!
FeedHubActor - adding RSS feed https://www.electrive.com/feed/
RssLocatorActor - URL: https://www.autoblog.com/rss.xml does not exist!
RssLocatorActor - URL: https://www.plugincars.com/feed exists but is not an RSS feed!
FeedHubActor - adding RSS feed https://www.teslarati.com/feed/
[https://www.electrive.com/2024/10/25/mercedes-bmw-get-green-light-for-fast-charging-
network-in-china/, company, [Mercedes-Benz, BMW, Ionity, General Motors, Honda, Hyundai,
Kia, Stellantis, Toyota, PowerX, Ashok Leyland]]
[https://www.electrive.com/2024/10/25/mercedes-bmw-get-green-light-for-fast-charging-
network-in-china/, technology, [Ionchi, Plug&Charge]]
[https://www.electrive.com/2024/10/25/mercedes-bmw-get-green-light-for-fast-charging-
network-in-china/, product, [Piaggio EVs, Switch EiV12 electric buses]]
[https://www.electrive.com/2024/10/25/mercedes-bmw-get-green-light-for-fast-charging-
network-in-china/, location, [China, European Economic Area, Beijing, Qingdao, Nanjing,
North America, North Carolina, Mannheim, Sandy Springs, Japan]]
[https://www.electrive.com/2024/10/24/tesla-appoints-new-head-of-charging-infrastructure-
and-reveals-plans-for-charging-park/, company, [Tesla]]
[https://www.electrive.com/2024/10/24/tesla-appoints-new-head-of-charging-infrastructure-
and-reveals-plans-for-charging-park/, technology, [Supercharger, Megapack, Powerwall, Solar
system, Megapack stationary storage units]]
[https://www.electrive.com/2024/10/24/tesla-appoints-new-head-of-charging-infrastructure-
and-reveals-plans-for-charging-park/, product, [Supercharger charging stations, Megapack
division, Powerwall home power storage system]]
[https://www.electrive.com/2024/10/24/tesla-appoints-new-head-of-charging-infrastructure-
and-reveals-plans-for-charging-park/, location, [California, San Francisco, Los Angeles,
Interstate 5, Lost Hills]]
[https://chargedevs.com/features/paired-powers-ev-chargers-let-customers-mix-and-match-
solar-storage-and-grid-power/, company, [Paired Power]]
[https://chargedevs.com/features/paired-powers-ev-chargers-let-customers-mix-and-match-
solar-storage-and-grid-power/, technology, [EV chargers]]
[https://chargedevs.com/features/paired-powers-ev-chargers-let-customers-mix-and-match-
solar-storage-and-grid-power/, product, [PairTree, PairFleet]]
[https://chargedevs.com/features/paired-powers-ev-chargers-let-customers-mix-and-match-
solar-storage-and-grid-power/, location, [California]]
[https://electrek.co/2024/10/23/lg-dc-fast-charger-us/, company, [LG Business Solutions USA,
LG Electronics]]
[https://electrek.co/2024/10/23/lg-dc-fast-charger-us/, technology, [CCS/NACS, SAE J1772, UL
2594, USB, Power Bank, Over-the-air software updates]]
[https://electrek.co/2024/10/23/lg-dc-fast-charger-us/, product, [LG DC fast charger, LG
EVD175SK-PN, Level 2 chargers, Level 3 chargers, Ultra-fast chargers]]
[https://electrek.co/2024/10/23/lg-dc-fast-charger-us/, location, [US, Texas, Fort Worth,
Nevada, White River Junction]]
[https://electrek.co/2024/10/23/tesla-unveils-oasis-supercharger-concept-solar-farm-
megapacks/, company, [Tesla]]
```

And many more ...