Dust Feeds

alan@mentalresonance.com

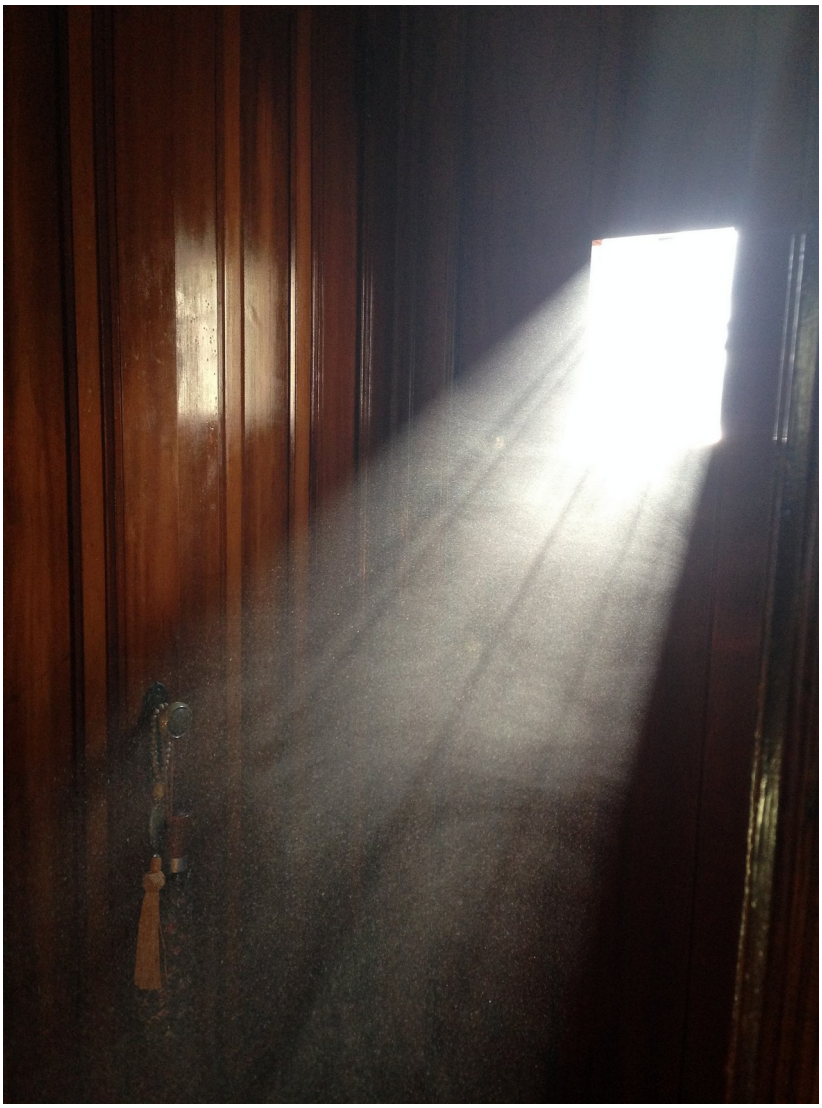
Version 1.0.1

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-feeds, which uses Dust core, html and http libraries, so familiarity with the code and idioms in these libraries is assumed.



1 Introduction

In Dust a ‘feed’ is a generic term meaning, roughly, ‘An Actor that can be pointed at some external resource and return you a stream of messages about it’. In dust-feeds we currently support:

- RSS Feeds – Dust wraps the Rome[1] library with a persistent Actor. The Actor is given the URL of the feed and how long to wait between visits. For each newly listed url in the feed the Actor fetches the content and wraps it in an HtmlDocumentMsg (if the content type is html) or in a RawDocumentMsg otherwise.

  We support two kinds of RSS Actors – one which is to be used within a Pipe (usually at the head of the Pipe), and one which has Pub/Sub behavior, so many clients can subscribe to receive messages from one Feed.

- Web Pages – Dust has simple Actors which can ‘crawl’ web sites. They obey the robots.txt files and permit the creation of filters on the urls of links and the anchor text of links. Note this is used for crawling individual sites – not the entire internet![2]

- Search engines – an Actor interface to an instance of the SearxNG meta search engine (https://docs.searxng.org/). Thus Actors can make requests for internet searches and process responses using all the Dust idioms. Since the protocol for a search is simply request / response the Actors involved are ServiceActors and will run under a ServiceManager.

  Note that most public instances of SearXNG do **not** support usage of the search API (i.e. they only allow browser access) so to be on the safe side you should have an instance of SearXNG installed locally. There is a unit test for SearXNG in the test/ folder but it assumes a local SearXNG so adapt it accordingly.

1 And so supports most versions of RSS and Atom feeds.
2 Happy to support pull requests in this direction :)

2 RSS

Since by there very nature Actors returning data from RSS feeds are long lived we have RSS support in two different forms:

- An Actor that assumes it is being used in a Dust pipeline and which sends result messages back to the pipeline
- A pub/sub Actor which sends result messages to all its subscribed clients

2.1 RssFeedPipeActor

The constructor for RssFeedPipeActor is

```
RssFeedPipeActor(
      String url, // The URL for the RSS feed
      Long intervalMS, // Length of times between visits to the feed
      ActorRef throttler, // A throttler to use or null
      Boolean returnContent // Type of data to send in message
)
```

There are two options for the types of messages which will be generated by this Actor which are determined by returnContent:

- If returnContent is true then the Actor follows the links in the RSS feed and sends as series of HtmlDocumentMsg (if the target url returns html) else RawDocumentMsg.
- If returnContent is false then the Actor sends a series of RssContentMsg each one of which contains information about a corresponding Syndication Entry in the RSS feed:

```
public class RssContentMsg implements Serializable {
    /**
     * Date published
     */
    public Date published;
    /**
     * Content author
     */
    public String author = null;
    /**
     * Content title
     */
    public String title = null;
    /**
     * Content (description)
     */
    public String content;
    /**
     * Link to source content
     */
    public String link;
}
```

In this latter case the client might want to examine the title or the content description to decide whether to follow the link to the source document.

The following Groovy snippet shows how to set up such a pipe:

```
ActorSystem system = new ActorSystem("RssPipelineTest")
system.setPersistenceService(FSTPersistenceService.create())

Props rssProps = RssFeedPipeActor.props(
      "https://cleantechnica.com/feed",
      3600 * 1000
)
Props logProps = LogActor.props()

List propsList = [rssProps, logProps], nameList = ['rss', 'log']

ActorRef pipe = system.context.actorOf(
                  PipelineActor.props(propsList, nameList),
                  'rsspipe'
)
pipe.tell(new StartMsg(), null)
```

rssProps defines our RSS Actor. We will visit the feed every hour. By default the throttler is null and returnContent is true. We then create a pipeline with this as the first stage and a logger as the second stage.

We then send the pipe (and hence the RssFeedPipeActor) a StartMsg. This is required to start the RssFeed and will cause it to visit the Feed for the first time. Subsequently the Actor will visit every hour. The RssFeedPipeActor will generate a series of (probably) HtmlDocumentMsg which get sent up the pipe to the Log Actor which simply outputs some information about the returned web pages to the console. A real application would, of course, have a more sophisticated pipe.

It is possible to pause the Actor and to change the url of the feed by sending the Actor appropriate messages. See the javadoc for details.

2.2 Persistence

Note in the example above we have set up a (global) persistence service. This is because RssFeedPipeActors are persistent – they store information about the last time they visited the feed, so the Actor always knows when to revisit the feed, even across shutdowns.

2.3 RssPubSubActor

RssPubSubActor is a simple instance of a Dust PubSubActor. It is configured with the same parameters as RssFeedPipeActor:

```
RssPubSubActor(String url, Long intervalMS, ActorRef throttler, boolean returnContent)
```

When the Actor is created it will create an suitable configured underlying RssFeedPipeActor and send it a StartMsg.

A client can then (un)subscribe to messages (HtmlDocumentMsg, RawDocumentMsg or a RssContentMsg) by the usual means of sending the RssPubSubActor a PubSubMsg:

```
PubSubMsg(Class<? extends Serializable> clz, Boolean subscribe)
```

where clz is the appropriate message class and subscribe is true to subscribe, false to unsubscribe. See the tests for a complete example.

3 SiteCrawlerPipeActor

As is says on the can, SiteCrawlerPipeActor is a pipeline stage (usually the first) that can crawl and send on web pages from the site in question.

Note this is a **site** crawler and not an **internet** crawler although certainly it could be a primary component of one – pipelines being just a specific kind of Actor are easy and fast to create.

SiteCrawlerPipeActor works as you would expect. Given a url it fires up a PageCrawlerActor to read the page, locate any links *that refer back to the site* and return that page and those links to the site. The page is sent on in the pipe and the links examined. If they have not been processed yet new PageCrawlerActors are created for them and the process continues until there is nothing left to do.

For politeness all http access to the site is routed through a throttler Actor set to a 1 second period. For further politeness if the SiteCrawlerActor finds a robots.txt file then its wishes are respected.

See the tests for a completely simple example.

3.1 Link Filtering

If SiteCrawlerPipeActor is created with an empty props() then the only restrictions on links is they must refer back to the site, but there are two other supported props() configurations.

3.1.1 hRef Filtering

Create the site crawler with:

```
Props props(List<List<String>> hrefFilters)
```

where hrefFilters is a list of lists of String [‘reg-exp-pattern’, ‘root’ or ‘page’]. ‘root’ means the filter applies when on the root (‘/’) page of the site, and page means it applies everywhere else. When links are being examined the process is:

1. Does this link refer back to the site. If it does not reject it. If it does
2. If we have hRef filters check the target of the link (the link’s href parameter) to see if it matches on the regexps, if it does accept the link, otherwise reject it.

So hrefFilters can be used to narrow the crawler to subsets of the entire site.

3.1.2 anchor Filtering

Site crawlers created with

```
Props props(List<List<String>> hrefFilters, List<List<String>> anchorFilters)
```

add an extra level of filtering to the link. If it passes hrefFilters then the *anchor* (text of the link) is checked against anchorFilters (same format as hrefFilters). If the link matches one of these filters it is accepted – otherwise rejected.

4 SearxNG

SearxNGServiceActor is a service Actor (so it should be run under a ServiceManager where it will process one request and die). SearxNG is an open source search engine aggregator – it takes a search request and dispatches it to a number of public search engines and aggregates the responses.

SearxNGServiceActor consumes a SearxNGRequestMsg and replies with a SearxNGResponsetMsg.

```
public class SearxNGRequestMsg implements Serializable {
    /**
     * The query
     */
    public String q;
    /**
     * Optional categories (see SearXNG engine documentation)
     */
    public String categories = null;
    /**
     * Optional engines (see SearXNG engine documentation)
     */
    public String engines = null;
    /**
     * Optional time_range (see SearXNG engine documentation)
     */
    public String time_range = null;
    /**
     * Format of response -- default is json
     */
    public String format = "json";

      ……
}
```

The field q should contain the search term which follows the usual conventions to limit to sites etc etc. The optional fields provide SearxNG with extrac guidance – see SearxNG documentation for details.

```
public class SearxNGResponseMsg implements Serializable {
    /**
     * The original query
     */
    public String query;
    /**
     * The results
     */
    public List<SearxNGResultMsg> results;
    /**
     * Categories (see SearXNG engine documentation)
     */
    public Map<String, String> categories;
    /**
     * Pagination (see SearXNG engine documentation)
     */
    public Map<String, Integer> pagination;
    /**
     * Suggestions (see SearXNG engine documentation)
     */
    public List<String> suggestions;
    /**
     * Metadata (see SearXNG engine documentation)
     */
    public Map<String, Object> metadata;
}
```

Again the main field here is results which is a list of individual search hits:

```
public class SearxNGResultMsg implements Serializable {
    /**
     * Title of document
     */
    public String title;
    /**
     * Url of document
     */
    public String url;
    /**
     * Content of document (summary)
     */
    public String content;
    /**
     * Name of the search engine responsible for this result
     */
    public String engine;
    /**
     * Matching category (if categories were defined in request)
     */
    public String category;
    /**
     * Published date
     */
    public String publishedDate;
    /**
     * All engines that returned this result
     */
    public List<String> engines;
    /**
     * For each engine which returned this result the position in the search
results
     */
    public List<Integer> positions;
    /**
     * Score for this engine
     */
    public Float score;
}
```

There is a simple test for SearxNG in tests. It assumes a local instance is running at port 8081.