Analysis of Communication Flow in Mailing Lists

A report submitted in partial fulfilment of the course CS F366

By

P.V.Srinidhi 2012A7PS095G

Under the supervision of

Talasila S.R.K.P

Lecturer

Birla Institute of Technology and Science

Pilani

Acknowledgements

I would like to take this opportunity to thank Mr. Talasila S.R.K.P, Lecturer at the Computer Science department, for providing me with this opportunity to work on the project. I am grateful for the guidance provided by him, during the entire period of work on this project. I would also like to thank Ms. Ipsita Samal, for the guidance she provided, when I first began working on this project.

Table of Contents

Mailing List

A mailing list is a collection of names and addresses used by an individual or an organization to send material to multiple recipients. An electronic mailing list is one that makes use of email for the widespread distribution of information to users on the internet. It generally consists of four things -

- a list of email addresses
- the subscribers at those addresses
- the email messages sent to those addresses
- a single mail address, called reflector, which when designated as the recipient of a message, sends a copy of the message to all the subscribers

Electronic mailing lists usually are fully or partially automated through the use of special mailing list software and a reflector address that are set up on a server capable of receiving email. Incoming messages sent to the reflector address are processed by the software, and, depending on their content, are acted upon internally if, in the case of messages containing commands directed at the software itself or, are distributed to all email addresses subscribed to the mailing list. Depending on the software, additional addresses may be set up for the purpose of sending commands

There are two major categories of electronic mailing list. These are -

- Announcement list – primarily used for one way communication like newsletters or marketing campaigns etc., messages can be posted to only be a set of selected users.
- Discussion list – Here, any subscriber may post messages. On a discussion list, a subscriber uses the mailing list to send messages to all the other subscribers, who may answer in similar fashion. Thus, actual discussion and information exchanges can happen. Mailing lists of this type are usually topic-oriented.

Online Archives

A mailing list archive is a collection of past messages from one or more electronic mailing list. Generally, the provide search and indexing functionalities. Many archives are directly associated with the mailing list of an organization, like gcc mailing list archive , debian mailing list archive etc. While there are archives exclusive to a particular organization available online, there are other which contain archives of many organizations in one place. An example for the latter is www.spinics.net, a site that contains the mailing list archive a many organizations and communities like the Linux kernel, apache, fedora etc. For this project, our initial point of data collection was from the Linux kernel mailing list archive available at spinics.net. From there we moved onto to the exclusive archive of the kernel mailing list available at lkml.org.

Outline of the mailing list parser

The purpose of the project is to do an analysis of the communication flow in mailing list and also to find some features of the mailing list (like the average numbers of messages for a topic) from all the mails received by the subscribers. For this, certain information about each mail is required, like - from address, to addresses, message-id, etc. Since the archives are available online, we need to scrape the required data from the archive. This is achieved through the usage the Scrapy framework.

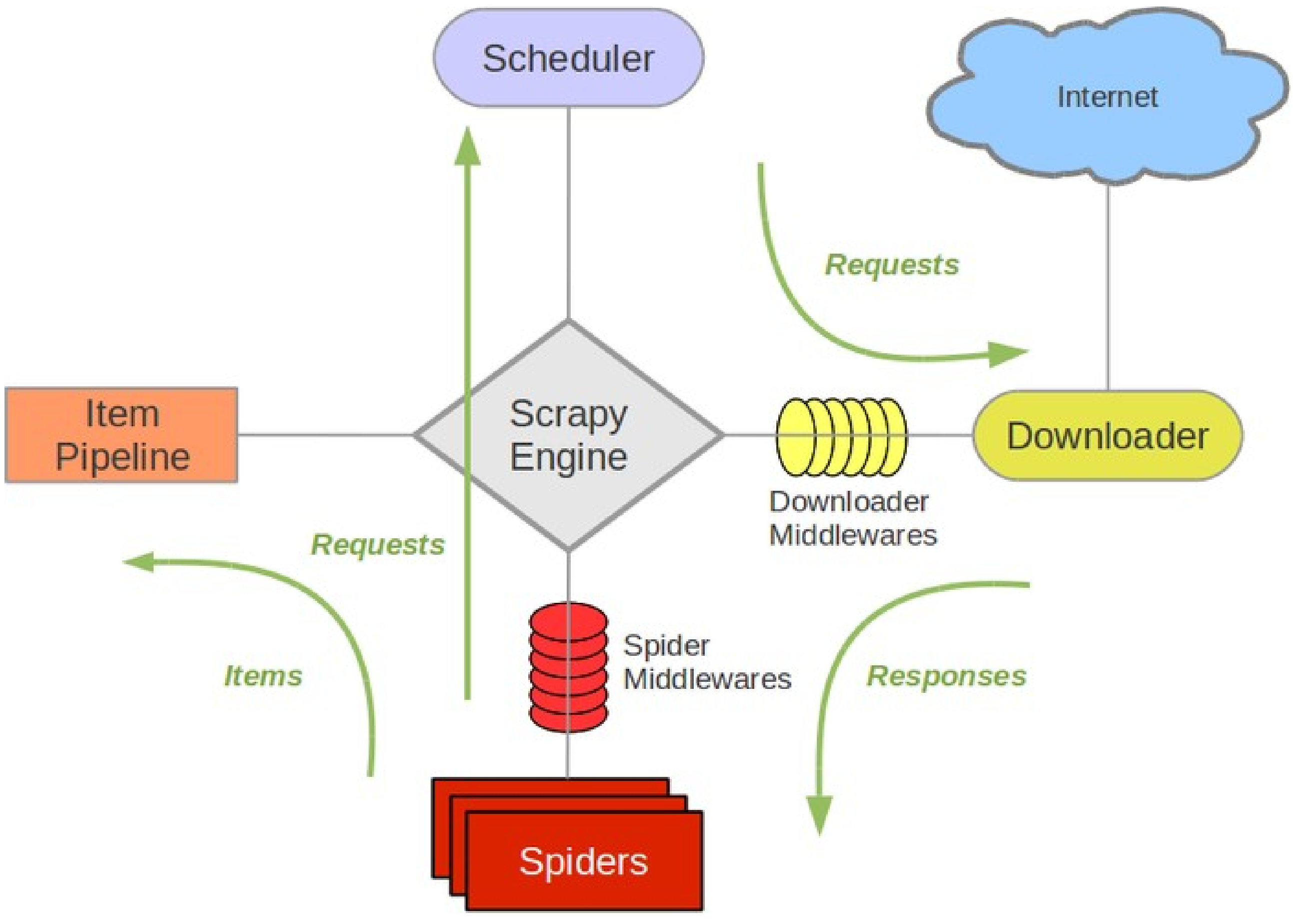


Figure 1: An overview of Scrapy architecture and data flow

Below is the description of the different components specified in the above image.

- Spiders are user-defined classes that inherit from the base spider class defined in the Scrapy framework. These classes define the behaviour and property of the web crawler. Each spider class handles a specific domain (or a group of domains). They are used to parse and extract information from each response object they get from the Scrapy engine and also, possibly crawl additional URLs if specified.

- Scrapy engine helps in controlling the data flow between the different components and also triggering of events when certain actions occur.
- The scheduler is used to en-queue requests obtained from the engine and later provide it to the engine when requested.
- The downloader is responsible for fetching of web pages based on the requests provided by the engine and also returning the responses to the engine. The engine feeds these responses to the spiders.
- The item pipeline is responsible for the processing of the data extracted by the spiders. It typically is used for cleansing, validation and persistence of data.
- The spider and downloader middleware sit between the engine and the spider and downloader respectively. They provide flexibility of extending the Scrapy functionality by allowing to plug in custom code.

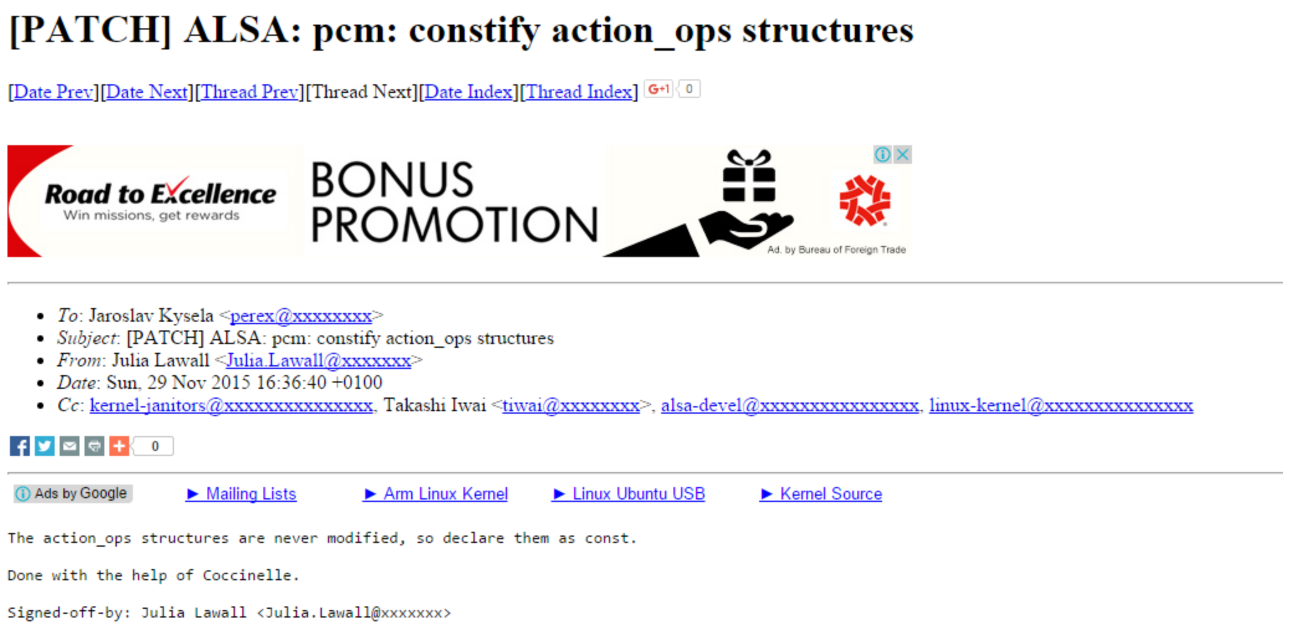
Every mail has a header which contains a lot of meta-data on the mail including details like the timestamps and SMTP ids of the intermediate mail servers that received and forwarded the mail. Out of this, we require the following details –

- From address
- To address(s)
- In-reply-to address(s)
- Cc address(s)
- Message-Id
- A list of references to all the ancestor mails in a thread

The mail parser is provided with the URL to a page in the archive, which contains links to the mails received. It follows the links in the page to get to the web page containing the header details of each mail. Using XPath selectors to parse out the response object, the parser then extracts the required information which is then finally stored in a json file.

Problems in the archive parser

The mailing list archive on which data was obtained from is that of Linux kernel. The source of this archive was initially considered from spinic.net. But the main problem associated with this approach was that the header of each mail was not completely available and hence this version of the mailing list archive could not be used for extraction of the required details.


[PATCH] ALSA: pcm: constify action_ops structures

[Date Prev][Date Next][Thread Prev][Thread Next][Date Index][Thread Index]

- *To*: Jaroslav Kysela <perex@xxxxxxxx>
- *Subject*: [PATCH] ALSA: pcm: constify action_ops structures
- *From*: Julia Lawall <Julia.Lawall@xxxxxxx>
- *Date*: Sun, 29 Nov 2015 16:36:40 +0100
- *Cc*: kernel-janitors@xxxxxxxxxxxxxxx, Takashi Iwai <tiwai@xxxxxxxx>, alsa-devel@xxxxxxxxxxxxxxxx, linux-kernel@xxxxxxxxxxxxxxx

```
The action_ops structures are never modified, so declare them as const.

Done with the help of Coccinelle.

Signed-off-by: Julia Lawall <Julia.Lawall@xxxxxxx>
```

Figure 2: A screenshot of a sample mail from Linux kernel mailing list archive available at spinics.net

The above problem prompted the change to lkml.org as the source for the archive, where the complete header details of each mail were available. Here, the mail parser was provided with the URL to the page containing links to all the mails received on that day. The parser would then crawl all these links and extract all the header details available in the header page of each mail.

The Scrapy engine, by default, schedules 100 request to be sent to the server concurrently. However, it is not always the case that responses for each of the 100 requests are obtained from the server. While, for the cases where a response object is present, extraction of the required header details is possible, in other cases it is not, leading to loss in data.

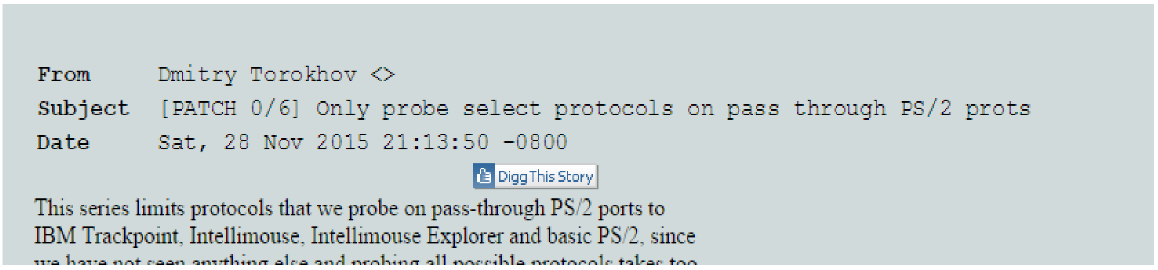
Linux Kernel Mail Parser

This section deals with the mail parser which uses a mail account subscribed to the Linux kernel mailing list, to obtain the mail headers and parses them to extract the required information.

Reasons for subscriptions to Linux kernel mailing list

There are two major reason as to why the initial approach (scraping data from online archive and further parsing of it) was dropped and a subscription was made to the kernel mailing list -

- Changes in the lkml.org archive: While initially, the archive made the header details of each mail available, these were later abstracted. Of the details that were abstracted, some of them were the ones required for the analysis. Hence, the web crawler-parser could not be used to retrieve the required information form the site.



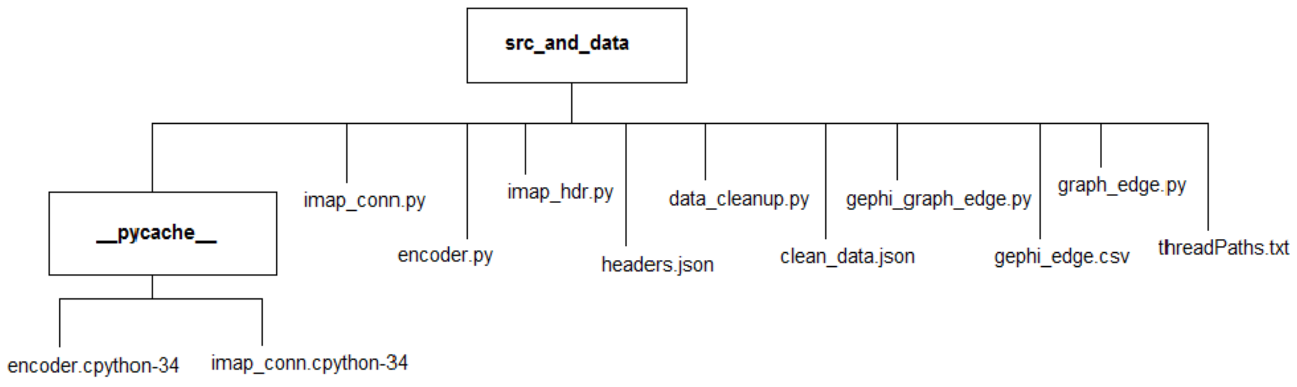
```
From      Dmitry Torokhov <>
Subject   [PATCH 0/6] Only probe select protocols on pass through PS/2 prots
Date      Sat, 28 Nov 2015 21:13:50 -0800
```

This series limits protocols that we probe on pass-through PS/2 ports to
IBM Trackpoint, Intellimouse, Intellimouse Explorer and basic PS/2, since

Figure 3: A screen shot of the headers page of a mail currently available at lkml.org archive

- Extraction of data: The engine of the Scrapy framework sends multiple request to the server concurrently. However, not all requests are accepted leading to loss of some data. This is not seen in the mail parser as it access to the mail account to download many messages can be handled by the mail server. Also the parser does not send concurrent request unlike the web crawler -parser.
- Easier access: Subscription to the mailing list implies that we have an unconstrained access to all the mails received there on and there is no need to keep track of when the mail was received. As for the web crawler-parser, it only retrieves information for all the mails received each day. For the web crawler-parser to access any mails from a previous data information on when it was received must also be maintained.

Directory structure



__pycache__: This is a directory created by the python interpreter. It contains the byte code – compiled and optimized byte code-complied versions of program modules that are used elsewhere in other program files.

src_and_data: This directory contains all the source program files and the data files along with the __pycache__ directory.

Code Overview

imap_conn.py

This file contains a function to establish connection to the imap server of Gmail, in order to access the inbox of the mail account subscribed to the Linux kernel mailing list. A point to be noted is that the IMAP service must be enabled in the account in order to obtain connection to the said account. Another point is that Google uses OAuth2 to facilitate authentication to an account. This means that a normal IMAP SSL connection will be refused by the server when trying to log into the account. Hence, the security setting of the account must be lowered to be able to obtain connection to the mail account.

- open_connection (verbose=False): This function is used to open a SSL connection to the imap mail server. A configuration file contains details of the server hostname, username and password required to login to the mail account. This functions reads all such details from the configuration file and uses it to open a connection. It then returns the connection object thus created.

encoder.py

This file contains two classes, namely – NoIndent and MyEncoder. These classes override the default encoder of the json module. They are used for pretty printing of lists in the json objects onto the .json file.

- NoIndent: The purpose of the NoIndent class is to convert the value of its object (if it is a list) to a string of comma separated elements of the list. This it defined in its __repr__ () method.
- MyEncoder: This class inherits the JSONEncoder class defined in the json module. It overrides the default () method. This method is used when writing json objects on to a file. It uses the default representation as present in the default () method as present in the parent class for objects of all types except those of type NoIndent. In case a NoIndent object is passed, then it uses the representation as specified in the __repr__ () method in the NoIndent class.

imap_hdr.py

This files contains the parser component required to extract information from mail headers. The parser uses the connection object returned by the open_connection () function present in the imap_conn.py file. The returned object is used to select the 'INBOX' folder of the mail account, after which a search for all mails in the inbox, having UIDs greater than 2000 is performed. UID is a unique number that the mail client associates with each mail. This search returns a list of mail UIDs present in the inbox, from 2000 to the latest mail in the inbox. Looping through the list, we fetch the header of the mails associated with UID, one at each cycle of iteration. From the header obtained, we extract the required information specified before and stored in a dictionary before writing onto the file.

A point of importance is that the message-Id and the list of references obtained from each header is a tokenizable string. However, since strings are difficult to handle for any sort of numerical analysis, each message-id and the message-ids in the references list had to be associated with a unique number. Hence, a map was created, associating each message-id with the UID of the mail header it first occurred in. Information on the headers of all mails sent in the time period before the subscription to the mailing list would not be made available. Hence, for any such mails, it would not be possible to associate a UID from the list of UIDs obtained to its message-id. In such a case, we associate the value of 0 to the message-id string.

For every message-id present the list of references, the same is done. If the message-id has been associated with a UID, then it is replaced with it else, it is replaced with 0. This list is then packaged in a NoIndent object. This object become the value for 'References' key in the dictionary. This is done so that, while writing of the dictionary onto the json file, the __repr__ () method of the NoIndent class would be used to obtain a representation of the list contained within a line and help with the pretty printing of the dictionary onto the file. All of the header details are stored in the headers.json file while the map associating the UID of message-id strings is written onto another file as well.

data_cleanup.py

Since we have access to the mails after the time of subscription to the mailing list, if any of the mails contain a reference to mails before subscription (in the form of a 0 in its list of references), then they represent some sort of incomplete information and cannot be included for analysis. Hence, the data present in headers.json file must be cleaned before proceeding further.

Here, one json object at a time is read from the headers.json file. This is achieved by the lines_per_n () function. This function takes as arguments the file object and the number of lines to be read and combined into a single string. This string is then returned by it and is used by the json.loads () function to load the json object.

For each json object thus obtained, the presence of a 0-reference is checked for. The message-ids of all such mails are added to a set and the object are then discarded. For json objects that do not have a 0-reference in its list of references, we check it if it has any messaged-ids that are present in the above set. If such is the case, then this json object is discarded, else, they are written on to another json file called clean_data.json. Another case is when the list of references of the json object of the mail points to none. Such mails are indicative of being the root of a thread of discussion. Such objects are also written onto clean_data.json

gephi_graph_edge.py

This file contains code to convert the information in each json object in the clean_data.json to a listing of edges and save it onto a csv file. This csv file can later be imported onto gephi for visualizations purposes. For every mail in a thread, the last message-id in the list of references indicates the message-id of the immediate ancestor (parent) of the mail in the thread. Wherever the list is present, the last element is extracted from it. This along with the message-id of the current thread is written onto the file in the following format:

<Parent message-id> ;< message-id>

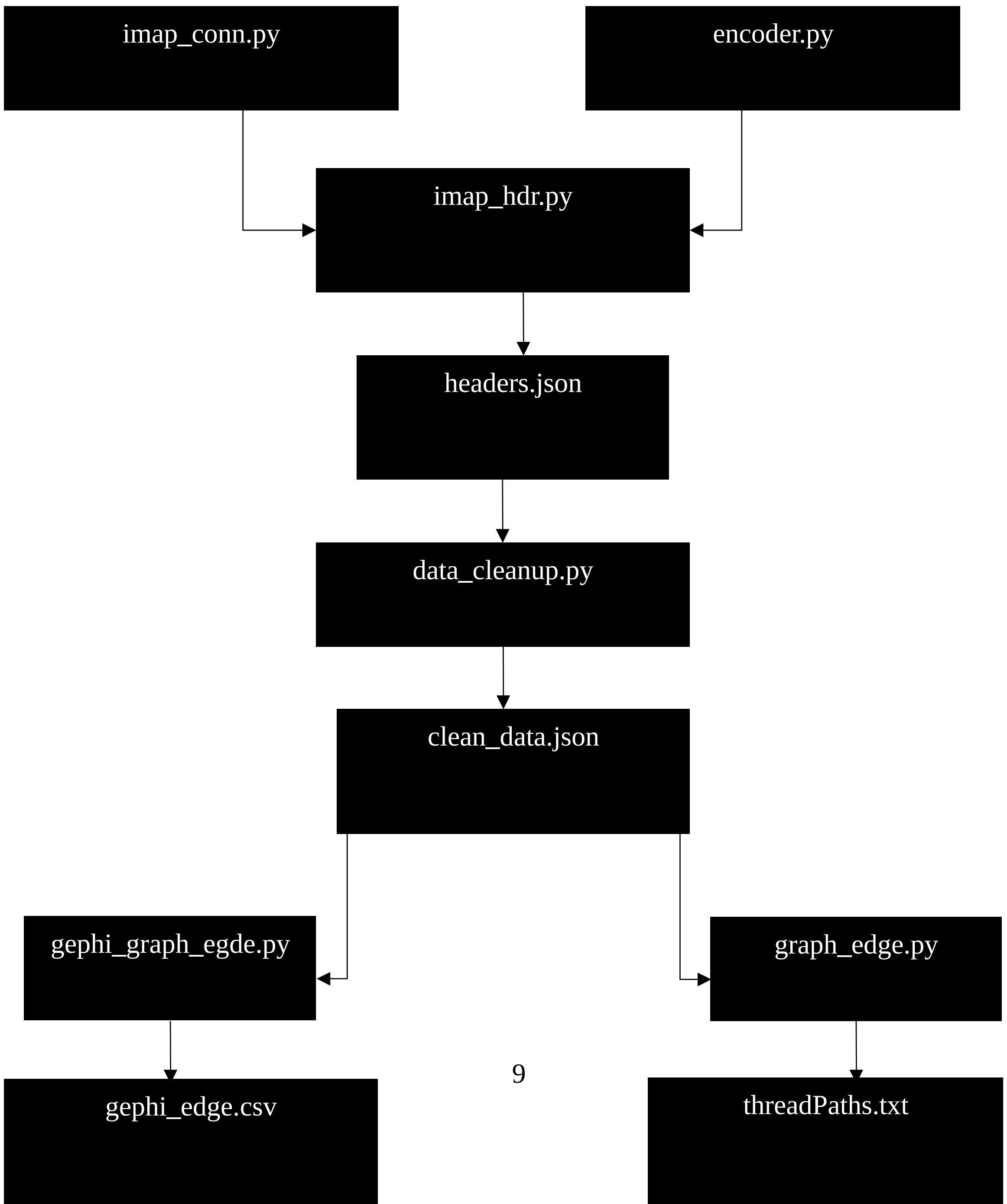
graph_edge.py

Leaf mails are mails whose message-ids are not present in the list of references of any other mail collected thus far. This python file prints the list of references of all leaf mails. The list of references along with the message-id of the leaf mails give all the paths in all the threads of mails. A set of leaf message-id and a map of message-id to the list of references is maintained. Each time a json object is read from the clean_data.json file, the message-id of the mail represented by this object is initially added to the set of message-ids. Then, this set is compared with the list of references of present in the object to remove any message-ids that are not those of leaf mail. So for every object read, the set is updated. The set obtained after reading all the objects in the clean_data.json file contains all the leaf mails present in the dataset collected until now. Using the map of message-ids to reference list, the paths in all the threads are printed in the following format:

<message-id> : <reference list>

The output is copied and stored in the files threadPaths.txt.

Below is a diagrammatic representation of the different stages, namely - data extraction, clean-up and generation of edges.



Support Modules

The process of extraction and parsing of data from headers, clean-up of the collected data and generation of edges and paths made use of the following modules/libraries available in python –

- json: The json module provides an API for converting in-memory Python objects to a serialized representation known as JavaScript Object. JSON has the benefit of having implementations in many languages (especially JavaScript), making it suitable for inter-application communication. JSON is probably most widely used for communicating between the web server and client in an AJAX application, but is not limited to that problem domain.

- imaplib: imaplib implements a client for communicating with Internet Message Access Protocol (IMAP) version 4 servers. The IMAP protocol defines a set of commands sent to the server and the responses delivered back to the client. Most of the commands are available as methods of the IMAP4 object used to communicate with the server.

- itertools: The itertools module includes a set of functions for working with iterable (sequence-like) data sets. The functions provided are inspired by similar features of the “lazy functional programming language” Haskell and SML. They are intended to be fast and use memory efficiently, but also to be hooked together to express more complicated iteration-based algorithms.

- email: The email package is a library for managing email messages, including MIME and other RFC 2822-based message documents. The primary distinguishing feature of the email package is that it splits the parsing and generating of email messages from the internal object model representation of email. Applications using this package deal primarily with objects; you can add sub-objects to messages, remove sub-objects from messages, completely re-arrange the contents, etc. There is a separate parser and a separate generator which handles the transformation from flat text to the object model, and then back to flat text again.

- pprint: pprint contains a “pretty printer” for producing aesthetically pleasing representations of your data structures. The formatter produces representations of data structures that can be parsed correctly by the interpreter, and are also easy for a human to read. The output is kept on a single line, if possible, and indented when split across multiple lines.

- configparser: This module is used to manage user-editable configuration files for an application. The configuration files are organized into sections, and each section can contain name-value pairs for configuration data. Value interpolation using Python formatting strings is also supported, to build values that depend on one another (this is especially handy for URLs and message strings).

References

- imaplib : https://pymotw.com/2/imaplib/index.html
- configparser : https://pymotw.com/2/ConfigParser/index.html
- email : https://docs.python.org/3/library/email.html
- itertools : https://docs.python.org/3/library/itertools.html
- json : https://docs.python.org/3/library/json.html ; https://pymotw.com/2/json/index.html
- RFC 3501 (RFC on IMAP)
- RFC 2822, RFC 822 (RFC for Internet Message Format)