DANE

Jun 23, 2023

Contents:

CHAPTER 1

Introduction

The Distributed Annotation 'n' Enrichment (DANE) system handles compute task assignment and file storage for the automatic annotation of content.

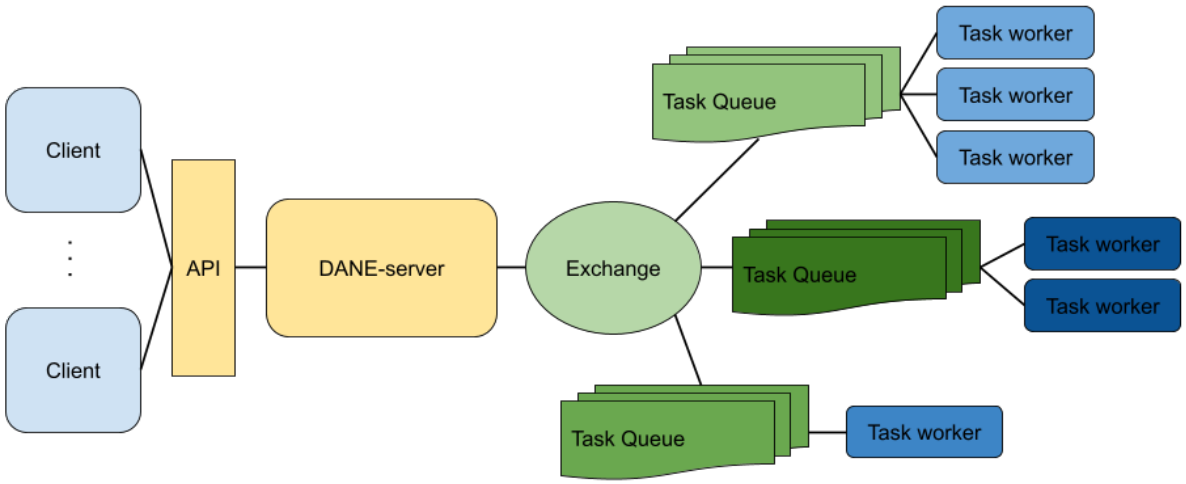
The use-case for which DANE was designed centres around the issue that the compute resources, and the collection of source media are not on the same device. Due to limited resources or policy choices it might not be possible or desirable to bulk transfer all source media to the compute resources, alternatively the source collection might be continuously growing or require on-demand processing.

In all cases, the computation depends on several stages which can all bottleneck (i.e., introduce delays) to the overall process. By subdividing the work into individual tasks, which can be scheduled separately DANE is capable of more optimally using the available resources. For example, in the case of intensive video analysis of a large archive, it is not feasible to move the entire video archive in a single pass to the compute server. By designing specific tasks for data transfer between servers, analysis, and post-hoc clean-up, DANE can be used to schedule these tasks such that they can be performed in parallel.

1.1 Usage

In essence the DANE ecosystem consists of three parts, (1) The back-end (DANE-server), (2) The compute workers, (3) A client that submits the tasks.

The format of the communication between these components follows the *document specification* format which details the source material to process, the *task specification* format that details the tasks to be performed, and the *result specification* with information on task results.



Once a document is submitted to the DANE-server tasks can be assigned to, which in turn will be assigned to a worker. As such, a worker relies on a DANE-server instance for its task assignment. To use DANE, one thus needs all three parts, namely an instance of DANE-server, some compute workers, and some client or process to submit tasks. Examples of workers and clients can be found *here*, whereas DANE-server is documented in its repository.

1.2 Configuration

The configuration of DANE components is done through the DANE.config module, which builds on top of YACS. The DANE.config specifies some default options, with default values, but it is mainly meant to be extended with component specific options. YACS makes it possible to specify configurations in a yaml format, and in code, here is a yaml example with some of the default config options:

```
DANE:
  HOST: '0.0.0.0'
  PORT: 5500
  API_URL: 'http://localhost:5500/DANE/'
RABBITMQ:
  HOST: 'localhost'
  PORT: 5672
  EXCHANGE: 'DANE-exchange'
```

Here, we have specified that the host that the DANE server listens on is *0.0.0.0* with port *5500*, additionally, the url at which the API is reachable is given by the *API_URL* field. Similarly, we specify a number of options for the RabbitMQ queueing system.

To deviate from the default options there are two options, 1) the system-wide DANE config file, and 2) the component specific config file. To best illustrate how these are used we will first demonstrate how to get access to the config. The DANE.config module has an cfg object, which is a YACS config node, which we can get access to by importing it as follows:

```
from DANE.config import cfg
```

We now have access to the config, and then we can pass it for example to a worker (as shown in the *Examples*):

```
fsw = filesize_worker(cfg)
```

or we can retrieve specific values from the config.

```
print('The DANE API is available at', cfg.DANE.API_URL)
```

During the loading of the config module, the default configuration will be constructed. Once the default config is setup it will, in order, search for a system-wide config, a component specific 'base_config', and a runtime specific config. By loading these in this order, the most specific options will be used (i.e., system-wide overrides defaults, and component specific overrides both the system-wide and defaults settings). DANE.config will look for the system-wide config at `$HOME/.dane/config.yml` (or `$DANE_HOME/config.yml` if available).

For the component specific config DANE.config looks in the directory of file that is importing it for a *base_config.yml*, thus if the module which uses DANE.config is at *$PYTHONLIB/site-packages/mymodule* then it will look in that same directory for the *base_config.yml*.

Lastly, the config module will look for the component specific config (*config.yml*) in the current working directory. For a worker, simply consist of a directory of code, and which is not installed, the directory structure might thus look like this:

```
filesize_worker/
    filesize_worker.py
    base_config.yml
    config.yml
```

A nice feature of YACS is that it is not necessary to overwrite all default configuration options, we only need to specific the ones we would like to change or add. For the filesize_worker, the base_config.yml might thus look like this:

```
FILESIZE_WORKER:
    UNIT: 'KB'
    PRECISION: 2
```

Defining new (non-functional) options for the worker, namely the units in which the filesize should be expressed, and the number of decimals we want shown in the output. It also gives a default value for this option. Subsequently, we can define an instance specific config.yml (which shouldn't be committed to GIT), which contains the following options.

```
DANE:
  API_URL: 'http://somehost.ext:5500/DANE/'
FILESIZE_WORKER:
    UNIT: 'MB'
```

This indicates that the API can be found at a different URL than the default one, and that we want the file size expressed in MB, for all other config options we rely on the previously defined defaults.

However, in some cases it might be necessary that the user always overwrites the base config, for instance when it contains paths that might be environment specific. In this case we can require that a *config.yml* is found by including the following in the base config:

```
CONFIG:
  REQUIRED: True
```

If no *config.yml* is found but the base config has indicated its required the config module will raise a *DANE.errors.ConfigRequiredError*.

1.3 Task states

Once a DANE worker has completed a task, or task progression has been interrupted due to an error, it should return a JSON object consisting of a *state* and a *message*. The message is expected to be an informative, and brief, indication of what went wrong, this message is not intended for automatic processing.

The state returned by a worker is used for automatic processing in DANE, based on this state it is determined whether a task is completed, in progress, requires retrying, or requires manual intervention. The state is one of the numerical HTTP Status codes with the aim of trying to adhere to the semantics of what the status code represents. For example, the state 200 indicates that the task has been successfully handled, whereas 102 indicates it is still in progress. Below we provide an overview of all used state codes and how they are handled by DANE.

1.3.1 State overview

- *102*: Task has been sent to a queue, it might be being worked on or held in queue.
- *200*: Task completed successfully.
- *201*: Task is registered, but has not been acted upon.
- *205*: Task reset state, typically after manual intervention
- *400*: Malformed request, typically the document or task description.
- *403*: Access denied to underlying source material.
- *404*: Underlying source material not found.
- *412*: Task has a dependency which has not completed yet.
- *422*: If a task cannot be routed to a queue, this state is returned.
- *500*: Error occurred during processing, details should be given in message.
- *502*: Worker received invalid or partial input.
- *503*: Worker received an error response from a remote service it depends on.

Tasks with state 205, 412, 502, or 503, can be retried automatically. Whereas states 400, 403, 404, 422, and 500 require manual intervention. Once a manual intervention has taken place the task can be resumed.

CHAPTER 2

Installation

DANE is available through PyPi: *pypi.org/project/DANE <https://pypi.org/project/DANE/>*. To install simply run

```
pip install DANE
```

In order to use DANE it is, in most cases, necessary to specify a configuration. How the DANE configuration works is specified *Here*.

CHAPTER 3

Developer API

API stuff

3.1 Document

class DANE.**Document**(*target*, *creator*, *api=None*, *_id=None*, *created_at=None*, *updated_at=None*)

This is a class representation of a document in DANE, it holds both data and some logic.

Parameters

- **target** (*dict*) – Dict containing *id*, *url*, and *type* keys to described the target document.
- **creator** (*dict*) – Dict containing *id*, and *type* keys to describe the document owner/creator.
- **api** (base_classes.base_handler, optional) – Reference to a class:*base_classes.base_handler* which is used to communicate with the server.
- **_id** (*int, optional*) – ID of the document, assigned by DANE-server
- **created_at** – Creation date
- **updated_at** – Last modified date

delete()

Delete this document. Requires an API to be set.

static from_json(*json_str*)

Constructs a *DANE.Document* instance from a JSON string

Parameters **json_str** (*str or dict*) – Serialised *DANE.Document*

Returns JSON string of the document

Return type *DANE.Document*

getAssignedTasks(*task_key=None*)
Retrieve tasks assigned to this document. Accepts an optional task_key to filter for a specific type of tasks. Requires an API to be set.

Parameters **task_key** (*string, optional*) – Key of task type to filter for

Returns list of dicts with task keys and ids.

register()
Register this document in DANE, this will assign an _id to the document. Requires an API to be set.

Returns self

set_api(*api*)
Set the API for the document

Parameters **api** (base_classes.base_handler, optional) – Reference to a base_classes.base_handler which is used to communicate with the database, and queueing system.

Returns self

to_json(*indent=None*)
Returns this document serialised as JSON, excluding the API reference.

Returns JSON string of the document

Return type str

3.2 Task

class DANE.**Task**(*key*, *priority=1*, *_id=None*, *api=None*, *state=None*, *msg=None*, *created_at=None*, *updated_at=None*, ***kwargs*)
Class representation of a task, contains task information and has logic for interacting with DANE-server through a base_classes.base_handler

Parameters

- **key** (*str*) – Key of the task, should match a binding key of a worker
- **priority** (*int*) – Priority to give to this task in queue. Defautls to 1.
- **_id** (*int, optional*) – id assigned by DANE-server to this task
- **api** (base_classes.base_handler, optional) – Reference to a base_classes.base_handler which is used to communicate with the database, and queueing system.
- **state** (*int, optional*) – Status code representing task state
- **msg** (*str, optional*) – Textual message accompanying the state
- **created_at** – Creation date
- **updated_at** – Last modified date
- ****kwargs** – Arbitrary keyword arguments. Will be stored in task.args

apply(*fn*)
Applies *fn* to self

Parameters **fn** (*function*) – Function handle in the form *fn(task)*

Returns self

assign(*document_id*)
Assign a task to a document, this will set an _id for the task and run it. Requires an API to be set.

Parameters **document_id** – id of document to assign this task to.

Returns self

assignMany(*document_ids*)
Assign this task to multiple documents and run it. Requires an API to be set.

delete()
Delete this task, requires it to be registered

Returns bool

static from_json(*task_str*)
Constructs a *DANE.Task* instance from a JSON string

Parameters **task_str** (*str or dict*) – Serialised *DANE.Task*

Returns An initialised Task

Return type *DANE.Task*

isDone()
Check if this task has been completed.

A task is completed if it's *state* equals 200. This will consult the API if the state isn't set.

Returns Task doneness

Return type bool

refresh()
Retrieves the latest information for task state and msg which might have changed their values since the creation of this task. Requires an API to be set

Returns self

reset()
Reset the task state to *201*

This can be used to force tasks to re-run after a preceding task has completed. Typically, the preceding task will be retried with *force=True*.

Returns self

retry(*force=False*)
Try to run this task again. Unlike *run()* this will attempt to run even after an error state was encountered.

Parameters **force** (*bool, optional*) – Force task to rerun regardless of previous state

Returns self

run()
Run this task, requires it to be registered

Returns self

set_api(*api*)
Set the API for this task

Parameters **api** (base_classes.base_handler, optional) – Reference to a base_classes.base_handler which is used to communicate with the database, and queueing system.

Returns self

state()
Get task state of this task.

Returns Task state

Return type int

to_json(*indent=None*)
Returns this task serialised as JSON

Returns JSON serialisation of the task

Return type str

3.3 Result

class DANE.**Result**(*generator*, *payload={}*, *_id=None*, *api=None*)
Class representation of a analysis result, containing the outcome and logic for interacting with DANE-server through a `base_classes.base_handler`

Parameters

- **generator** (*dict*) – Details of analysis that generated this result, requires *id*, *name*, *type*, and *homepage* fields.
- **payload** (*dict*) – The actual result(s) to be stored
- **_id** (*int, optional*) – id assigned by DANE-server to this task

delete()
Delete this result.

static from_json(*json_str*)
Constructs a *DANE.Result* instance from a JSON string

Parameters **task_str** (*str or dict*) – Serialised *DANE.Result*

Returns An initialised Result

Return type *DANE.Result*

save(*task_id*)
Save this result, this will set an _id for the result

Parameters **task_id** – id of the task that generated this result

Returns self

to_json(*indent=None*)
Returns this result serialised as JSON

Returns JSON string of the result

Return type str

3.4 Handlers

class DANE.handlers.**ESHandler**(*config*, *queue=None*)

assignTask(*task*, *document_id*)
Assign a task to a document and run it.

Parameters

- **task** (*DANE.Task*) – the task to assign
- **document_id** (*int*) – id of the document this task belongs to

Returns task_id

Return type int

assignTaskToMany(*task*, *document_ids*)
Assign a task to a document and run it.

Parameters

- **task** (*DANE.Task*) – the task to assign
- **document_id** (*[int]*) – list of ids of the documents to assign this to

Returns task_ids

Return type [int]

callback(*task_id*, *response*)
Function that is called once a task gives back a response.

This updates the state and response of the task in the database, and then tries to run the other tasks assigned to the document.

Parameters

- **task_id** (*int*) – The id of a task
- **response** (*dict*) – Task response, should contain at least the *state* and a *message*

deleteDocument(*document*)
Delete a document and its underlying tasks from the database

Parameters **document** (*DANE.Document*) – The document

deleteResult(*result*)
Delete a result

Parameters **result** (*DANE.Result*) – The result to delete

Returns bool

deleteTask(*task*)
Delete a task.

Parameters **task** (*DANE.Task*) – the task to delete

Returns bool

documentFromDocumentId(*document_id*)
Construct and return a *DANE.Document* given a document_id

Parameters **document_id** (*int*) – The id for the document

Returns The document

Return type *DANE.Document*

documentFromTaskId(*task_id*)
Construct and return a *DANE.Document* given a task_id

Parameters **task_id** (*int*) – The id of a task

Returns The document

Return type *DANE.Document*

getAssignedTasks(*document_id*, *task_key=None*)
Retrieve tasks assigned to a document. Accepts an optional task_key to filter for a specific type of tasks.

Parameters

- **document_id** – document to of interest
- **task_key** (*string, optional*) – Key of task type to filter for

Returns list of dicts with task ids, keys, and states.

getTaskKey(*task_id*)
Retrieve task_key for a given task_id

Parameters **task_id** (*int*) – id of the task

Returns task_key

Return type str

getTaskState(*task_id*)
Retrieve state for a given task_id

Parameters **task_id** (*int*) – id of the task

Returns task_state

Return type int

getUnfinished(*only_runnable=False*)
Returns tasks which are not finished, i.e., tasks that dont have state *200*

Parameters **only_runnable** – Return only tasks that can be *run()*

Returns ids of found tasks

Return type dict

registerDocument(*document*)
Register a document in the database

Parameters **document** (*DANE.Document*) – The document

Returns document_id

Return type int

registerDocuments(*documents*)
Register list of documents in the database

Parameters **document** (*DANE.Document*) – The document

Returns two lists with successfully and failed documents, as tuple

registerResult(*result*, *task_id*)
Save a result for a task

Parameters

- **result** (*DANE.Result*) – The result
- **task_id** – id of the task that generated this result

Returns self

resultFromResultId(*result_id*)
Construct and return a *DANE.Result* given a result_id

Parameters **result_id** (*int*) – The id of a result

Returns The result

Return type *DANE.Result*

retry(*task_id*, *force=False*)
Retry the task with this id.

Attempts to run a task which previously might have crashed. Defaults to skipping tasks with state 200, or 102, unless Force is specified, then it should rerun regardless of previous state.

Parameters

- **task_id** (*int*) – The id of a task
- **force** (*bool, optional*) – Force task to rerun regardless of previous state

run(*task_id*)
Run the task with this id, and change its task state to *102*.

Running a task involves submitting it to a queue, so results might only be available much later. Expects a task to have state *201*, and it may retry tasks with state *502* or *503*.

Parameters **task_id** (*int*) – The id of a task

search(*target_id*, *creator_id*, *page=1*)
Returns documents matching target_id and creator_id

Parameters

- **target_id** – The id of the target
- **creator_id** – The id of the creator

Returns list of found documents

searchResult(*document_id*, *task_key*)
Search for a result of a task with task_key applied to a specific document

Parameters

- **document_id** – id of the document the task should be applied to
- **task_key** – key of the task that was applied

Returns List of initialised *DANE.Result*

taskFromTaskId(*task_id*)
Retrieve task for a given task_id

Parameters **task_id** (*int*) – id of the task

Returns the task, or error if it doesnt exist

Return type *DANE.Task*

updateTaskState(*task_id*, *state*, *message*)
Update the state, message, and last updated of a task.

Parameters

- **task_id** (*int, required*) – The id of a task

- **state** (*int, required*) – The new task state
- **message** (*string, required*) – The new task message

3.5 Base classes

class DANE.base_classes.**base_worker**(*queue*, *binding_key*, *config*, *depends_on=[]*, *auto_connect=True*, *no_api=False*)

Abstract base class for a worker.

This class contains most of the logic of dealing with DANE-server, classes (workers) inheriting from this class only need to specific the callback method.

Parameters

- **queue** (*str*) – Name of the queue for this worker
- **binding_key** (*str or list*) – A string following the format as explained here: https://www.rabbitmq.com/tutorials/tutorial-five-python.html or a list of such strings
- **config** (*dict*) – Config settings of the worker
- **depends_on** (*list, optional*) – List of task_keys that need to have been performed on the document before this task can be run
- **auto_connect** (*bool, optional*) – Connect to AMQ on init, set to false to debug worker as a standalone class.
- **no_api** (*bool, optional*) – Disable ESHandler, mainly for debugging.

callback(*task*, *document*)

Function containing the core functionality that is specific to a worker.

Parameters

- **task** (*DANE.Task*) – Task to be executed
- **document** (*DANE.Document*) – Document the task is applied to

Returns Task response with the *message*, *state*, and optional additional response information

Return type dict

connect()

Connect the worker to the AMQ. Called by init if autoconnecting.

getDirs(*document*)

This function returns the TEMP and OUT directories for this job creating them if they do not yet exist output should be stored in response['SHARED']

Parameters **job** (DANE.Job) – The job

Returns Dict with keys *TEMP_FOLDER* and *OUT_FOLDER*

Return type dict

run()

Start listening for tasks to be executed.

stop()

Stop listening for tasks to be executed.

3.6 Utils

3.7 Errors and exceptions

exception DANE.errors.**APIRegistrationError**
Raised when registering the API fails.

exception DANE.errors.**ConfigRequiredError**
Error to indicate that the base_config.yml is declared abstract, and that it requires a config.yml.

exception DANE.errors.**DANException**
Wrapper for DANE exception.

exception DANE.errors.**DocumentExistsError**
Raised when document does (not) exists.

exception DANE.errors.**MissingEndpointError**
Raised when an action fails due to lack of API.

exception DANE.errors.**RefuseJobException**
Exception for workers to throw when they want to refuse a job at this point in time.

This will result in a nack (no ack) being sent back to the queue, causing the job to be requeued (at the or close to the head of the queue).

exception DANE.errors.**ResourceConnectionError**
Raised when a component cant connect to a resource it depends on.

Used for catching resource specific errors, and wrapping them in a soft blanket of custom error handling.

exception DANE.errors.**ResultExistsError**
Raised when result does (not) exists.

exception DANE.errors.**TaskAssignedError**
Raised when task is already/not yet assigned.

exception DANE.errors.**TaskExistsError**
Raised when task does (not) exists.

exception DANE.errors.**UnregisteredError**
Raised when DANE object does not have an _id.

CHAPTER 4

Examples

All code examples can be found here.

4.1 Examples

To explore the options of DANE we've create a jupyter notebook for you to experiment with creating a DANE documents and tasks yourself.

https://github.com/CLARIAH/DANE/blob/master/examples/dane_example.ipynb

Instead of performing any interaction with a DANE server, the examples work with a DummyHandler which implements all the required handler functionality. The DummyHandler simply stores all information in variables, as such there is no persistence of the data, but it does allow for experimenting with DANE.

4.2 An example worker

DANE workers have been designed such that very little boilerplate code is necessary. Nonetheless, some boilerplate is required to ensure we can rely on the logic defined in the *DANE.base_classes.base_worker*.

In this example we will break down the code on how to construct a worker which returns the file size of a source file.

We'll start by defining a new class, which we have appropriately named *filesize_worker*. We ensure that it inherits from *DANE.base_classes.base_worker*, and then we're ready to start adding logic.

```
class filesize_worker(DANE.base_classes.base_worker):
    __queue_name = 'filesize_queue'
    __binding_key = '#.FILESIZE'

    def __init__(self, config):
        super().__init__(queue=self.__queue_name,
                binding_key=self.__binding_key, config=config)
```

First, we define two class constants with the name of the queue the worker should use, and the binding key. We want all workers of the same type to share the same queue name, so if we start multiple workers they can divide the work.

The binding key follows the pattern `<document type>.<task key>`, where the document type can be * for any type of source material, or optionally we can build a worker which only processes a specific type of document. The a task object uses a key to specify that we mean this type of worker.

In theory, multiple different workers can have the same key, while having a different queue name. This could be use for example to do logging. However, this can be risky in that if the queue for the intended task is not initialised, the task might never be assigned to the correct queue.

Up next is the __init__ function. In order to properly set up the worker we need to call the init of the base_worker class, provide the queue name, binding key, and the config parameters to connect to the RabbitMQ instance. If the worker requires any set up, the init can be extended to include this as well.

Besides any setting up logic which might be in the init, the majority of worker specific logic is contained in the callback. This function is called whenever a new task is read from the queue.

The base_worker contains all the code for interacting with the queue, so in the callback we can focus on actually doing the work.

```
def callback(self, task, doc):
    if exists(doc.target['url']):

        fs = getsize(doc.target['url'])

        return json.dumps({'state': 200,
            'message': 'Success')
    else:
        return json.dumps({'state': 404,
            'message': 'No file found at source_url'})
```

The callback receives a task and a document. For the file size worker we are only interested in the source material. We assume that the source material is a local file, so we can rely on functionality from `os.path`.

The first step is to check if the source material actually exists. In general, any input verification and validity checking is relegated to the workers themselves. If the file exists, we retrieve its size and return a JSON serialised dict containing the success state (200), a message detailing that we have succeeded If we want to store the retrieved file size, or make it available to later tasks we can store it in a *DANE.Result*.

For the else clause, we can simply return a 404 state, and a descriptive message to indicate that the source material was not found. In all cases a task **must** return a **state** and a **message**. For more on states see *Task states*.

Lastly, we need some code to start the worker.

```
if __name__ == '__main__':

    fsw = filesize_worker(cfg)
    print(' # Initialising worker. Ctrl+C to exit')

    try:
        fsw.run()
    except KeyboardInterrupt:
        fsw.stop()
```

To start a worker, we first initialise it with a config file. By default a worker only needs access to the ElasticSearch and RabbitMQ details provided by the DANE.config, such that it can store and read data, as well as set up a queue and listen to work to perform. However, this can be extended with worker specific configuration options. More details on how to work with the configuration can be found in the *Usage* guide.

After having initialised the worker we can simply call the *DANE.base_classes.base_worker.run()* method to start listening for work. As this starts a blocking process, we have added a way in which we can (slightly) more elegantly interrupt it. Namely, once Ctrl+C is pressed, this will trigger the KeyBoardInterrupt exception, which we catch with the try-except block, and then we call the stop method.

To test this worker it is necessary to have access to a RabbitMQ instance. However, to simulate task requests we have constructed a generator which can be run without having to set up the other components of a DANE server.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index

S

T

U