hop-client Documentation

SCiMMA

Jun 28, 2022

CONTENTS

`hop-client` is a pub-sub client library for Multimessenger Astrophysics.

CHAPTER

ONE

USER'S GUIDE

1.1 Installation

You can install hop-client either via pip, conda, or from source.

To install with pip:

```
pip install -U hop-client
```

To install with conda:

```
conda install -c conda-forge hop-client
```

To install from source:

```
tar -xzf hop-client-x.y.z.tar.gz
cd hop-client-x.y.z
python setup.py install
```

1.2 Quickstart

1.2.1 Using the CLI

By default, authentication is enabled, reading in credentials from `auth.toml`. The path to this configuration can be found by running `hop auth locate`. One can initialize this configuration with default settings by running `hop auth add`. To disable authentication in the CLI client, one can use the `--no-auth` option.

Publish messages

```
hop publish kafka://hostname:port/gcn -f CIRCULAR example.gcn3
```

Example messages are provided in `tests/data` including:

- A GCN circular (`example.gcn3`)
- A VOEvent (`example_voevent.xml`)

Consume messages

```
hop subscribe kafka://hostname:port/gcn -s EARLIEST
```

This will read messages from the gcn topic from the earliest offset and read messages as they arrive. By default this will listen to messages until the user stops the program (Ctrl+C to stop).

View Available Topics

```
hop list-topics kafka://hostname:port/
```

This will list all of the topics on the given server which you are currently authorized to read or write.

1.2.2 Using the Python API

Publish messages

Using the python API, we can publish various types of messages, including structured messages such as GCN Circulars and VOEvents:

```
from hop import stream
from hop.models import GCNCircular

# read in a GCN circular
with open("path/to/circular.gcn3", "r") as f:
    circular = GCNCircular.load(f)

with stream.open("kafka://hostname:port/topic", "w") as s:
    s.write(circular)
```

In addition, we can also publish unstructured messages as long as they are JSON serializable:

```
from hop import stream

with stream.open("kafka://hostname:port/topic", "w") as s:
    s.write({"my": "message"})
```

By default, authentication is enabled for the Hop broker, reading in configuration settings from `config.toml`. In order to modify various authentication options, one can configure a `Stream` instance and pass in an `Auth` instance with credentials:

```
from hop import Stream
from hop.auth import Auth

auth = Auth("my-username", "my-password")
stream = Stream(auth=auth)

with stream.open("kafka://hostname:port/topic", "w") as s:
    s.write({"my": "message"})
```

To explicitly disable authentication, one can set `auth` to `False`.

Consume messages

One can consume messages through the python API as follows:

```
from hop import stream

with stream.open("kafka://hostname:port/topic", "r") as s:
    for message in s:
        print(message.content)
```

This will listen to the Hop broker, listening to new messages and printing them to stdout as they arrive. By default, this will only process new messages since the connection was opened. The `start_at` option lets you control where in the stream you can start listening from. For example, if you'd like to listen to all messages stored in a topic, you can do:

```
from hop import stream
from hop.io import StartPosition

stream = Stream(start_at=StartPosition.EARLIEST)

with stream.open("kafka://hostname:port/topic", "r") as s:
    for message in s:
        print(message.content)
```

1.3 Streaming

1.3.1 The Stream Object

The `Stream` object allows a user to connect to a Kafka broker and read in a variety of alerts, such as GCN circulars. It also allows one to specify default settings used across all streams opened from the Stream instance.

Let's open up a stream and show the `Stream` object in action:

```
from hop import Stream

stream = Stream(until_eos=True)
with stream.open("kafka://hostname:port/topic", "r") as s:
    for message in s:
        print(message.content)
```

The `until_eos` option allows one to listen to messages until the no more messages are available (EOS or end of stream). By default the connection is kept open indefinitely. This is to allow long-lived connections where one may set up a service to process incoming GCNs, for example.

A common use case is to not specify any defaults ahead of time, so a shorthand is provided for using one:

```
from hop import stream

with stream.open("kafka://hostname:port/topic", "r") as s:
    for message in s:
        print(message.content)
```

A complete list of configurable options in `Stream` are:

- `auth`: A *bool* or `auth.Auth` instance to provide authentication
- `start_at`: The message offset to start at, by passing in an `io.StartPosition`
- `until_eos`: Whether to stop processing messages after an EOS is received

One doesn't have to use the context manager protocol (`with` block) to open up a stream as long as the stream is explicitly closed afterwards:

```
from hop import stream

s = stream.open("kafka://hostname:port/topic", "r")
for message in s:
    print(message.content)
s.close()
```

So far, all examples have shown the iterator interface for reading messages from an open stream. But one can instead call `s.read()` directly or in the case of more specialized workflows, may make use of extra keyword arguments to

configure an open stream. For example, the `metadata` option allows one to retrieve Kafka message metadata as well as the message itself, such as the Kafka topic, key, timestamp and offset. This may be useful in the case of listening to multiple topics at once:

```
from hop import stream

with stream.open("kafka://hostname:port/topic1,topic2", "r") as s:
    for message, metadata in s.read(metadata=True):
        print(message.content, metadata.topic)
```

1.3.2 Anatomy of a Kafka URL

Both the CLI and python API take a URL that describes how to connect to various Kafka topics, and takes the form:

```
kafka://[username@]broker/topic[,topic2[,...]]
```

The broker takes the form `hostname[:port]` and gives the URL to connect to a Kafka broker. Optionally, a `username` can be provided, which is used to select among available credentials to use when communicating with the broker. Finally, one can publish to a topic or subscribe to one or more topics to consume messages from.

1.3.3 Committing Messages Manually

By default, messages that are read in by the stream are marked as read immediately after returning them from an open stream instance for a given group ID. This is suitable for most cases, but some workflows have more strict fault tolerance requirements and don't want to lose messages in the case of a failure while processing the current message. We can instead commit messages after we are done processing them so that in the case of a failure, a process that is restarted can get the same message back and finish processing it before moving on to the next. This requires returning broker-specific metadata as well as assigning yourself to a specific group ID. A workflow to do this is shown below:

```
from hop import stream

with stream.open("kafka://hostname:port/topic1", "r", "mygroup") as s:
    for message, metadata in s.read(metadata=True, autocommit=False):
        print(message.content, metadata.topic)
        s.mark_done(metadata)
```

1.3.4 Attaching Metadata to Messages

Apache Kafka supports headers to associate metadata with messages, separate from the message body, and the hop python API supports this feature as well. Headers should generally be *small* and ideally optional information; most of a message's content should be in its body.

Each header has a string key, and a binary or unicode value. A collection of headers may be provided either as a dictionary or as a list of (key, value) tuples. Duplicate header keys are permitted; the list representation is necessary to utilize this allowance.

It is important to note that Hopskotch reserves all header names starting with an underscore (_) for internal use; users should not set their own headers with such names.

Sending messages with headers and viewing the headers attached to received messages can be done as shown below:

```
from hop import stream

with stream.open("kafka://hostname:port/topic1", "w") as s:
    s.write({"my": "message"}, headers={"priority": "1", "sender": "test"})
    s.write({"my": "other message"}, headers=[("priority", "2"), ("sender", "test")])
```

```
from hop import stream

with stream.open("kafka://hostname:port/topic1", "r") as s:
    for message, metadata in s.read(metadata=True):
        print(message, metadata.headers)
```

1.4 Authentication

1.4.1 Configuration

Since connections to the Hopskotch server require authentication, there are several utilities exposed to generate and provide credentials for both the CLI and python API. `hop auth` provides command line options to generate a configuration file with proper credentials needed to authenticate.

In order to generate a configuration file, one can run `hop auth add`, which prompts for a username and password to connect to Hopskotch to publish or subscribe to messages. If you have the credentials csv file, you can use it directly with `hop auth add <CREDENTIALS_FILE>`.

The default location for the authentication data file can be found with `hop auth locate`, which points by default to `${XDG_CONFIG_HOME}/hop/auth.toml` or `${HOME}/.config/hop/auth.toml` if the `XDG_CONFIG_HOME` variable is not set.

1.4.2 Using Credentials

Authentication is enabled by default and will read credentials from the path resolved by `hop auth locate`.

Multiple credentials may be stored together using this mechanism. Additional credentials may be added using `hop auth add`, while the currently available credentials may be displayed with `hop auth list` and unwanted credentials can be removed with `hop auth remove`. Credentials can be added either interactively or from CSV files. For removal, credentials are specified by username, or `<username>@<hostname>` in case of ambiguity.

When using the *hop* CLI to connect to connect to a kafka server, a credential will be selected according to the following rules:

1. A credential with a matching hostname will be selected, unless no stored credential has a matching hostname, in which case a credential with no specific hostname can be selected.
2. If a username is specified as part of the authority component of the URL (e.g. `kafka://username@example.com/topic`) only credentials with that username will be considered.

3. If no username is specified and there is only one credential, which is not specifically associated with any hostname, it will be used for all hosts.

For the python API, one can modify various authentication options by passing in an `Auth` instance with credentials to a `Stream` instance. This provides a similar interface to authenticating as with the requests library.

```
from hop import Stream
from hop.auth import Auth

auth = Auth("my-username", "my-password")
stream = Stream(auth=auth)

with stream.open("kafka://hostname:port/topic", "w") as s:
    s.write({"my": "message"})
```

A list of multiple `Auth` instance may also be passed, in which case the best match for the connection being opened will be selected as described above.

In order to disable authentication in the command line interface, you can pass `--no-auth` for various CLI commands. For the python API, you can set `auth` to `False`.

1.5 Message Formats

The hop client provides a few in-memory representations for common message types for easy access to various message properties, as well as loading messages from their serialized forms or from disk. These message formats, or models, can be sent directly to an open `Stream` to provide seamless serialization of messages through Hopskotch.

1.5.1 Structured Messages

Currently, the structured messages available through the hop client are `VOEvent` and `GCNCircular`. To give an example of its usage:

```
from hop import Stream
from hop.auth import load_auth
from hop.models import VOEvent

xml_path = "/path/to/voevent.xml"
voevent = VOEvent.load_file(xml_path)

stream = Stream(auth=load_auth())
with stream.open("kafka://hostname:port/topic", "w") as s:
    s.write(voevent)
```

1.5.2 Unstructured Messages

Unstructured (or less structured messages) can be sent directly to an open `Stream` instance. Any python object that can be JSON serialized can be sent. Examples include a dictionary, a string, and a list. At an even more raw level, bytes objects can be sent without any further encoding or interpretation.

On the more structured end of the spectrum, the hop client understands some general data formats, and can automatically encode and decode them. These include JSON with the `JSONBlob` class and Apache Avro with the `AvroBlob` class. Using `JSONBlob` is equivalent to simply writing an unstructured but JSON-encodable python object. `AvroBlob` supports efficiently including bytes subobjects, as well as schemas. If no schema is supplied, it will create a schema to describe the object(s) it is given, but a deliberately designed schema may also be used.

```
from hop import Stream
from hop.auth import load_auth
from hop.models import JSONBlob, AvroBlob
import fastavro

stream = Stream(auth=load_auth())
with stream.open("kafka://hostname:port/topic", "w") as s:

    # Writing simple, unstructured messages
    s.write("a string message")
    s.write(["some", "data", "with", "numbers:", 5, 6, 7])
    s.write({"priority": 1, "payload": "data"})
    s.write(b'\x02Binary data\x1DMessage ends\x03')

    # Explicitly writing a partially-structured message as JSON
    s.write(JSONBlob({"priority": 1, "payload": "data"}))

    # Write an Avro message with an ad-hoc schema
    # Avro may contain arbitrarily many records,
    # so it always expects a list or records to be written
    s.write(AvroBlob([{"priority": 1, "payload": b'\x02Binary data\x03'}]))

    # Write an Avro message with a specific schema
    schema = fastavro.load_schema("my_schema.avsc")
    s.write(AvroBlob([{priority: 1, payload: b'\x02Binary data\x03'}],
                     schema=schema))
```

All unstructured messages are unpacked by the hop client back into message model objects containing python objects equivalent to what was sent when they are read from a stream. The decoded objects are available from each of the unstructured message model types as `content` property. Some model types also make additional information available, for example, the `AvroBlob` also has a `schema` property which contains the schema with which the message was sent.

Please note that the `AvroBlob` message model serializes using the Avro container format, not the Avro variant of the Confluent wire format.

1.5.3 Register External Message Models

Sometimes it may be useful to use custom structured messages that aren't currently available in the stock client. For instance, sending specialized messages between services that are internal to a specific observatory. The hop client provides a mechanism in which to register custom message types that are discoverable within hop when publishing and subscribing for your own project. This requires creating an external python library and setting up an entry point so that hop that discover it upon importing the client.

There are three steps involved in creating and registering a custom message model:

1. Define the message model.
2. Register the message model.
3. Set up an entry point within your package.

Define a message model

To do this, you need to define a dataclass that subclasses `hop.models.MessageModel`, choose an identifier (name) that will be used to refer to your model, and implement functionality to load your message mode via the `load()` class method. As an example, assuming the message is represented as JSON on disk:

```
from dataclasses import dataclass
import json

from hop.models import MessageModel

@dataclass
class Donut(MessageModel):

    category: str
    flavor: str
    has_filling: bool

    format_name: "donut"  # optional

    @classmethod
    def load(cls, input_):
        # input_ is a file object
        if hasattr(donut_input, "read"):
            donut = json.load(input_)
        # serialized input_
        else:
            donut = json.loads(input_)

        # unpack the JSON dictionary and return the model
        return cls(**donut)
```

If you do not explicitly define the format name for your model, as a string property named `format_name`, the class name, converted to all lower case, will be used.

By default, the base `MessageModel` class will provide serialization and deserialization of the fields defined in your model to and from JSON. If you want greater control over how these processes work, your model class can define its own `serialize` and `deserialize` methods. If you choose to implement these methods yourself, `serialize` must return a dictionary with two keys: *"format"* which maps to your model's identifier string, and *"content"* which maps to the encoded form of the model instance's data, as a `bytes` object. Using `hop.models.format_name` is the

recommended way to determine the value for the *"format"* key, as it will automatically follow the standard convention. `deserialize` must be a class method which accepts encoded data (as `bytes`) and produces an instance of your model after decoding. It is also possible to customize the `load_file` convenience class method, which normally just attempts to open the specified path as a file for reading and passes the resulting file object to `load`; the most common reason to customize this method is for models which need to ensure that input files are opened in binary mode.

For more information on dataclasses, see the Python Docs.

Register a message model

Once you have defined your message model, registering the message model involves defining a function with the `hop.plugins.register` decorator with key-value pairs mapping a message model name and the model:

```
from hop import plugins
from hop.models import format_name

...

@plugins.register
def get_models():
    model_classes = [Donut]
    return {format_name(cls): cls for cls in model_classes}
```

Using `hop.models.format_name` to compose the keys is recommended because it means that you only need to define the format name once, as part of the class definition.

Set up entry points within your package

After registering your model, you'll need to set up an entry point to your package named `hop_plugin` as that entry point is explicitly used to auto-discover new plugins. The module used for the entry point is wherever you registered your model.

Setting up entry points may be different depending on how your package is set up. Below we'll give an example for setuptools and setup.py. In setup.py:

```
from setuptools import setup

...

setup(
    ...

    entrypoints = {"hop_plugin": ["donut-plugin = my.custom.module"]}
)
```

Some further resources on entry points:

- https://setuptools.readthedocs.io/en/latest/setuptools.html#dynamic-discovery-of-services-and-plugins

1.6 Robust Publication

1.6.1 The RobustProducer

In some situations, it may be critical to ensure that messages are sent to the Kafka broker, even when the intervening network may be unreliable, the sending process may be killed unexpectedly, and so on. The *RobustProducer* object extends the capabilities of the simple *Producer* to provide this functionality.

The two main mechanisms used by the *RobustProducer* are to maintain a local journal of messages which are queued to be sent or are in flight, and to listen for confirmation messages from the Kafka broker that messages have been received. The use of receipt confirmations enables the resending of messages which are lost in the network or if the broker fails unexpectedly, while use of the journal ensures that even if the sending program is stopped suddenly, it can resend any messages whose receipt was not yet confirmed. Implications of this are that local disk space is required for the message journal (and the amount of space used will be at least that of the sum of sizes of all messages in flight at the same time), and that at-least-once delivery is guaranteed, but that in providing that guarantee, messages may be duplicated. For example, duplication of messages on the broker will occur if the producer sends the mesage, but the broker's confirmation is lost in the network, so the producer is forced to assume that the message did not go through and resends it. Clients should be prepared to handle duplicate messages appropriately.

1.6.2 Usage

The simplest way to use the *RobustProducer* is as a context manager:

```
from hop.robust_publisher import RobustProducer

with RobustProducer("kafka://hostname:port/topic") as publisher:
    for message in messages:
        publisher.write(message)
```

To control the location where the message journal is stored, one may set the `journal_path` option when constructing the *RobustProducer*; the default is `"publisher.journal"` which will place it the script's current working directory.

Message sending is asynchronous, so *RobustProducer.write* will return almost immediately, as it only queues the message for sending. The *RobustProducer* blocks internally until all messages are successfully sent, so there can be a noticeable delay after all messages have been queued while they complete sending. In the event of a network or broker failure, this delay may extend indefinitely.

The *RobustProducer constructor* also accepts an `auth` argument for specifying the credentials with which it should connect, and will pass through any extra keyword arguments to *io.Stream.open*.

For more advanced uses, *RobustProducer* can also be used directly without being treated as a context manager:

```
from hop.robust_publisher import RobustProducer

publisher = RobustProducer("kafka://hostname:port/topic")
publisher.start()
```

(continues on next page)

(continued from previous page)

```
#. . .
publisher.write(some_message)

#. . .
publisher.stop()
```

When used in this way, it is necessary to call *RobustProducer.start* before sending any messages, and *RobustProducer.stop* after all messages have been sent to shut down the *RobustProducer*'s internal background worker thread. It is important to note that the user should *not* call *RobustProducer.run*, as this method is exposed only as a part of the python `threading.Thread` interface, and will block whatever thread calls it, indefinitely. Once stopped, a *RobustProducer* object cannot be restarted.

1.6.3 Miscellaneous Details

The message journal is intended to protect against disruption of the sending program, but at this time does not include meaningful protection against sudden failure of the machine on which the program is running; in particular, it does not ensure that data written to it is definitely flushed through filesystem or hardware caching layers. As a result, issues like power failures can lead to data loss. The journal does contain checksumming and other sanity checking which enable detecting most forms of data corruption, although truncation of the journal exactly at a boundary between entries currently cannot be detected. Currently, corruption of the journal will trigger an error and block (re)starting the *RobustProducer*.

Messages are written to the journal essentially in plain text, so users whose data is sensitive should take into account that the journal file must be suitably protected.

Currently, *RobustProducer.write* takes over the `delivery_callback` option for *Producer.write* for its own use, so end users are not able to register their own delivery callback handlers.

1.7 Commands

hop-client provides a command line interface for various tasks:

- `hop auth`: Authentication utilities
- `hop list-topics`: SHow accessible Kafka topics
- `hop publish`: Publish messages such as GCN circulars and notices
- `hop subscribe`: Listen to messages such as GCN circulars and notices
- `hop version`: Show version dependencies of `hop-client`

1.7.1 hop auth

This command allows a user to configure credentials for authentication.

```
usage: hop auth [-h] [-q | -v] <command> ...

Authentication configuration utilities.

optional arguments:
  -h, --help     show this help message and exit
  -q, --quiet    If set, only display warnings and errors.
  -v, --verbose  If set, display additional logging messages.

commands:
  <command>
    locate       display configuration path
    list         Display all stored credentials
    add          Load a credential, specified either via a CSV file or
                 interactively
    remove       Delete a stored credential

No valid credential data found
You can get a credential from https://my.hop.scimma.org
To load your credential, run `hop auth add`
```

1.7.2 hop list-topics

This command allows a user to view the topics that are available for subscribing or publishing on a given Hopskotch server.

Note that other topics may exist which the current user does not have permission to access.

```
usage: hop list-topics [-h] [--no-auth] URL

List available topics.

positional arguments:
  URL         Sets the URL (kafka://host[:port]/topic) to publish messages to.

optional arguments:
  -h, --help  show this help message and exit
  --no-auth   If set, disable authentication.

No valid credential data found
You can get a credential from https://my.hop.scimma.org
To load your credential, run `hop auth add`
```

1.7.3 hop publish

This command allows a user to publish various structured and unstructured messages, including:

- RFC 822 formatted GCN circular
- An XML formatted GCN/VOEvent notice
- Unstructured messages such as JSON-serializable data.

Structured messages such as GCN circulars and VOEvents are published as JSON-formatted text.

Unstructured messages may be piped to this command to be published. This mode of operation requires JSON input with individual messages separated by newlines, and the Blob format (*-f BLOB*) to be selected.

```
usage: hop publish [-h] [--no-auth] [-q | -v]
                   [-f {VOEVENT,CIRCULAR,BLOB,JSON,AVRO}] [-t]
                   URL [MESSAGE [MESSAGE ...]]

Publish messages.

positional arguments:
  URL                   Sets the URL (kafka://host[:port]/topic) to publish
                        messages to.
  MESSAGE               File of messages to publish. (standard input is used
                        if omitted)

optional arguments:
  -h, --help            show this help message and exit
  --no-auth             If set, disable authentication.
  -q, --quiet           If set, only display warnings and errors.
  -v, --verbose         If set, display additional logging messages.
  -f {VOEVENT,CIRCULAR,BLOB,JSON,AVRO}, --format {VOEVENT,CIRCULAR,BLOB,JSON,AVRO}
                        Specify the message format. Defaults to BLOB for an
                        unstructured message.
  -t, --test            Mark messages as test messages by adding a header with
                        key '_test'.

No valid credential data found
You can get a credential from https://my.hop.scimma.org
To load your credential, run `hop auth add`
```

1.7.4 hop subscribe

This command allows a user to subscribe to messages and print them to stdout.

```
usage: hop subscribe [-h] [--no-auth] [-q | -v] [-s {EARLIEST,LATEST}] [-e]
                     [-g GROUP_ID] [-j] [-t]
                     URL

Subscribe to messages.

positional arguments:
  URL                   Sets the URL (kafka://host[:port]/topic) to publish
                        messages to.
```

(continues on next page)

(continued from previous page)

```
optional arguments:
  -h, --help            show this help message and exit
  --no-auth             If set, disable authentication.
  -q, --quiet           If set, only display warnings and errors.
  -v, --verbose         If set, display additional logging messages.
  -s {EARLIEST,LATEST}, --start-at {EARLIEST,LATEST}
                        Set the message offset offset to start at. Default:
                        LATEST.
  -e, --until-eos       If set, only subscribe until EOS is received (end of
                        stream). Otherwise, listen to messages indefinitely.
  -g GROUP_ID, --group-id GROUP_ID
                        Consumer group ID. If unset, a random ID will be
                        generated.
  -j, --json            Request message output as raw json
  -t, --test            Process test messages instead of ignoring them.

No valid credential data found
You can get a credential from https://my.hop.scimma.org
To load your credential, run `hop auth add`
```

1.7.5 hop version

This command prints all the versions of the dependencies

```
usage: hop version [-h]

List all the dependencies' versions.

optional arguments:
  -h, --help  show this help message and exit

No valid credential data found
You can get a credential from https://my.hop.scimma.org
To load your credential, run `hop auth add`
```

CHAPTER

TWO

API REFERENCE

2.1 hop-client API

2.1.1 hop.auth

class hop.auth.**Auth**(*user*, *password*, *host=''*, *ssl=True*, *method=SASLMethod.SCRAM_SHA_512*, ***kwargs*)

Attach SASL-based authentication to a client.

Returns client-based auth options when called.

Parameters

- **user** (*str*) – Username to authenticate with.
- **password** (*str*) – Password to authenticate with.
- **host** (*str*, optional) – The name of the host for which this authentication is valid.
- **ssl** (*bool*, optional) – Whether to enable SSL (enabled by default).
- **method** (*SASLMethod*, optional) – The SASL method to authenticate, default = SASLMethod.SCRAM_SHA_512. See valid SASL methods in SASLMethod.
- **ssl_ca_location** (*str*, optional) – If using SSL via a self-signed cert, a path/location to the certificate.

property hostname

The hostname with which this creential is associated, or the empty string if the credential did not contain this information

property mechanism

The authentication mechanism to use

property password

The password for this credential

property protocol

The communication protocol to use

property ssl

Whether communication should be secured with SSL

property ssl_ca_location

The location of the Certfificate Authority data used for SSL, or None if SSL is not enabled

property username

The username for this credential

hop.auth.**add_credential**(*args*)

Load a new credential and store it to the persistent configuration.

Parameters

args – Command line options/arguments object. args.cred_file is taken as the path to a CSV file to import, or if None the user is prompted to enter a credential directly. args.force controls whether an existing credential with an identical name will be overwritten.

hop.auth.**delete_credential**(*name: str*)

Delete a credential from the persistent configuration.

Parameters

- **name** – The username, or username and hostname separated by an '@' character of the credential
- **delete.**(*to*) –

Raises

RuntimeError – If no credentials or more than one credential matches the specified name, making the operation impossible or ambiguous.

hop.auth.**list_credentials**()

Display a list of all configured credentials.

hop.auth.**load_auth**(*config_file=None*)

Configures an Auth instance given a configuration file.

Parameters

config_file – Path to a configuration file, loading from the default location if not given.

Returns

A list of configured Auth instances.

Raises

- **RuntimeError** – The config file exists, but has unsafe permissions and will not be read until they are corrected.
- **KeyError** – An error occurred parsing the configuration file.
- **FileNotFoundError** – The configuration file, either as specified explicitly or found automatically, does not exist

hop.auth.**prune_outdated_auth**(*config_file=None*)

Remove auth data from a general configuration file.

This can be needed when updating auth data which was read from the general config for backwards compatibility, but is then written out to the correct new location in a separate auth config, as is now proper. With no further action, this would leave a vestigial copy from before the update in the general config file, which would not be rewritten, so this function exists to perform the necessary rewrite.

Parameters

config_file – Path to a configuration file, rewriting the default location if not given.

Raises

RuntimeError – The config file is malformed.

hop.auth.**read_new_credential**(*csv_file=None*)

Import a credential from a CSV file or obtain it interactively from the user.

Parameters
csv_file – Path to a file from which to read credential data in CSV format. If unspecified, the user will be prompted to enter data instead.

Returns
A configured *Auth* object containing the new credential.

Raises

- **FileNotFoundError** – If csv_file is not None and refers to a nonexistent path.
- **KeyError** – If csv_file is not None and the specified file does not contain either a username or password field.
- **RuntimeError** – If csv_file is None and the interactively entered username or passwod is empty.

hop.auth.**select_matching_auth**(*creds*, *hostname*, *username=None*)

Selects the most appropriate credential to use when attempting to contact the given host.

Parameters

- **creds** – A list of configured Auth objects. These can be obtained from *load_auth()*.
- **hostname** – The name of the host for which to select suitable credentials.
- **username** – *str*, optional The name of the credential to use.

Returns
A single Auth object which should be used to authenticate.

Raises
RuntimeError – Too many or too few credentials matched.

hop.auth.**write_auth_data**(*config_file*, *credentials*)

Write configuration file for the set of credentials.

Creates containing directories as needed.

Parameters

- **config_file** – configuration file path
- **credentials** – list of *Auth* objects representing credentials to be stored

2.1.2 hop.cli

hop.cli.**add_client_opts**(*parser*)

Add general client options to an argument parser.

Parameters
parser – An ArgumentParser instance to add client options to.

hop.cli.**add_logging_opts**(*parser*)

Add logging client options to an argument parser.

Parameters
parser – An ArgumentParser instance to add client options to.

hop.cli.**get_log_level**(*args*)

Determine the log level from logging options.

Parameters
args – The parsed argparse arguments.

Returns
the logging log level.

hop.cli.**set_up_logger**(*args*)

Set up common logging settings for CLI usage.

Parameters
args – The parsed argparse arguments.

2.1.3 hop.configure

hop.configure.**get_config_path**(*type: str = 'general'*)

Determines the default location for auth configuration.

Parameters
type – The type of configuration data for which the path should be looked up. Recognized types are 'general' and 'auth'.

Returns
The path to the requested configuration file.

Raises
ValueError – Unrecognized config type requested.

2.1.4 hop.io

class hop.io.**Consumer**(*group_id*, *broker_addresses*, *topics*, *ignoretest=True*, ***kwargs*)

An event stream opened for reading one or more topics. Instances of this class should be obtained from *Stream.open()*.

close()

End all subscriptions and shut down.

static is_test(*message*)

True if message is a test message (contains '_test' as a header key).

Parameters
message – The message to test.

mark_done(*metadata*)

Mark a message as fully-processed.

Parameters
metadata – A Metadata instance containing broker-specific metadata.

read(*metadata=False*, *autocommit=True*, ***kwargs*)

Read messages from a stream.

Parameters

- **metadata** – Whether to receive message metadata alongside messages.

- **autocommit** – Whether messages are automatically marked as handled via *mark_done* when the next message is yielded. Defaults to True.
- **batch_size** – The number of messages to request from Kafka at a time. Lower numbers can give lower latency, while higher numbers will be more efficient, but may add latency.
- **batch_timeout** – The period of time to wait to get a full batch of messages from Kafka. Similar to batch_size, lower numbers can reduce latency while higher numbers can be more efficient at the cost of greater latency. If specified, this argument should be a datetime.timedelta object.

class hop.io.**Deserializer**(*value*)

An enumeration.

class hop.io.**Metadata**(*topic: str*, *partition: int*, *offset: int*, *timestamp: int*, *key: Union[str, bytes]*, *headers: List[Tuple[str, bytes]]*, *_raw: Message*)

Broker-specific metadata that accompanies a consumed message.

class hop.io.**Producer**(*broker_addresses*, *topic*, ***kwargs*)

An event stream opened for writing to a topic. Instances of this class should be obtained from *Stream.open()*.

close()

Wait for enqueued messages to be written and shut down.

flush()

Request that any messages locally queued for sending be sent immediately.

static pack(*message*, *headers=None*, *test=False*)

Pack and serialize a message.

This is an advanced interface, which most users should not need to call directly, as *Producer.write* uses it automatically.

Parameters

- **message** – The message to pack and serialize.
- **headers** – The set of headers requested to be sent with the message, either as a mapping, or as a list of 2-tuples. In either the mapping or the list case, all header keys must be strings and and values should be either string-like or bytes-like objects.
- **test** – Message should be marked as a test message by adding a header with key '_test'.

Returns: A tuple containing the serialized message and the collection of headers which should be sent with it.

write(*message*, *headers=None*, *delivery_callback=<function raise_delivery_errors>*, *test=False*)

Write messages to a stream.

Parameters

- **message** – The message to write.
- **headers** – The set of headers requested to be sent with the message, either as a mapping, or as a list of 2-tuples. In either the mapping or the list case, all header keys must be strings and and values should be either string-like or bytes-like objects.
- **delivery_callback** – A callback which will be called when each message is either delivered or permenantly fails to be delivered.

- **test** – Message should be marked as a test message by adding a header with key '_test'.

write_raw(*packed_message*, *headers=None*, *delivery_callback=<function raise_delivery_errors>*)

Write a pre-encoded message to the stream.

This is an advanced interface; for most purposes it is preferrable to use *Producer.write* instead.

Parameters

- **packed_message** – The message to write, which must already be correctly encoded by *Producer.pack*
- **headers** – Any headers to attach to the message, either as a dictionary mapping strings to strings, or as a list of 2-tuples of strings.
- **delivery_callback** – A callback which will be called when each message is either delivered or permenantly fails to be delivered.

class hop.io.**Stream**(*auth=True*, *start_at=ConsumerStartPosition.LATEST*, *until_eos=False*)

Defines an event stream.

Sets up defaults used within the client so that when a stream connection is opened, it will use defaults specified here.

Parameters

- **auth** – A *bool* or *Auth* instance. Defaults to loading from *auth.load_auth* if set to True. To disable authentication, set to False.
- **start_at** – The message offset to start at in read mode. Defaults to LATEST.
- **until_eos** – Whether to listen to new messages forever (False) or stop when EOS is received in read mode (True). Defaults to False.

open(*url*, *mode='r'*, *group_id=None*, *ignoretest=True*, ***kwargs*)

Opens a connection to an event stream.

Parameters

- **url** – Sets the broker URL to connect to.
- **mode** – Read ('r') or write ('w') from the stream.
- **group_id** – The consumer group ID from which to read. Generated automatically if not specified.
- **ignoretest** – When True, read mode will silently discard test messages.

Returns

An open connection to the client, either a *Producer* instance in write mode or a *Consumer* instance in read mode.

Raises

ValueError – If the mode is not set to read/write, if more than one topic is specified in write mode, or if more than one broker is specified

hop.io.**list_topics**(*url: str*, *auth: Union[bool,* Auth*] = True*)

List the accessible topics on the Kafka broker referred to by url.

Parameters

- **url** – The Kafka broker URL. Only one broker may be specified. Topics may be specified, in which case only topics in the intersection of the set specified by the URL and actually present on the broker will be returned. If a userinfo component is present in the URL and auth is True, it will be treated as a hint to the automatic auth lookup.

- **auth** – A *bool* or *Auth* instance. Defaults to loading from *auth.load_auth* if set to True. To disable authentication, set to False. If a username is specified as part of url but auth is a *Auth* instance the url information will be ignored.

Returns
A dictionary mapping topic names to `confluent_kafka.admin.TopicMetadata` instances.

Raises
ValueError – If more than one broker is specified.

2.1.5 hop.robust_publisher

class hop.robust_publisher.**PublicationJournal**(*journal_path='publisher.journal'*)

An object which tracks the state of messages which are being sent, persists that state to disk, and enables it to be restored if the program stops unexpectedly.

__init__(*journal_path='publisher.journal'*)

Prepare a journal, including loading any data previously persisted to disk.

Parameters
journal_path – The filesystem path from/to which the journal data should be read/written.

Raises

- **PermissionError** – If existing journal file does not have suitable permissions.
- **RuntimeError** – If existing journal data cannot be read.

class NullLock

A trivial context manager-compatible class which can be used in place of a lock when no locking is needed.

static error_callback(*kafka_error: KafkaError*)

A safe callback handler for reporting Kafka errors.

get_delivery_callback(*seq_num*, *lock=<hop.robust_publisher.PublicationJournal.NullLock object>*)

Construct a callback handler specific to a particular message which will either mark it successfully sent or requeue it to send again.

The callback which is produced will take two arguments: A confluent_kafka.KafkaError describing any error in sending the message, and confluent_kafka.Message containing the message itself.

Parameters

- **seq_num** – The sequence number of the message in question, previously returned by *get_next_message_to_send()*.
- **lock** – An optional reference to a lock object which the callback should hold when invoked, e.g. to protect concurrent access to the journal.

get_next_message_to_send()

Fetch the next message which should be sent

Returns
The next message in the form of a tuple of (seqeunce number, message, message headers), or None if there are no messages currently needing to be sent.

has_messages_in_flight()

Check whether there are messages for which a sending attempt has been started, but has not yet conclusively succeeded or failed

has_messages_to_send()

Check whether there are messages queued for sending (which have either not been sent at all, or for which all sending attempts so far have failed, causing them to be requeued).

mark_message_sent(*sequence_number*)

Mark a message as successfully sent, and removes it from further consideration.

Truncates and restarts the backing journal file if the number of messages in-flight and waiting to be sent falls to zero, and restarts the sequence number assignment sequence.

Raises

RuntimeError – If no message with the specifed sequence number is currently recorded as being in-flight.

queue_message(*message: bytes*, *headers=None*)

Record to the journal a message which is to be sent.

Parameters

- **message** – A message to send, encoded as a bytes-like object.
- **headers** – Headers to be sent with the message, as a list of 2-tuples of bytes-like objects.

Returns

The sequence number assigned to the message. Sequence numbers are unique among all messages which are 'live' at the same time, but will otherwise be recycled.

Raises

- **RuntimeError** – If appending the new message to the on-disk journal fails.
- **TypeError** – If the message is not a suitable type (bytes)

requeue_message(*sequence_number*)

Record a message send attempt as having failed by moving the message back from the in-flight pool to the queue of messages needing to be sent.

Raises

RuntimeError – If no message with the specifed sequence number is currently recorded as being in-flight.

class hop.robust_publisher.**RobustProducer**(*url*, *auth=True*, *journal_path='publisher.journal'*, *poll_wait=0.0001*, ***kwargs*)

__init__(*url*, *auth=True*, *journal_path='publisher.journal'*, *poll_wait=0.0001*, ***kwargs*)

Construct a publisher which will retry sending messages if it does not receive confirmation that they have arrived, including if it is itself taken offline (i.e. crashes) for some reason.

This is intended to provide *at least once* delivery of messages: If a message is confirmed received by the broker, it will not be sent again, but if any disruption of the network or the publisher itself prevents it from receiving that confirmation, even if the message was actually received by the broker, the publisher will assume the worst and send the message again. Users of this class (and more generally consumers of data published with it) should be prepared to discard duplicate messages.

Parameters

- **url** – The URL for the Kafka topci to which messages will be published.
- **auth** – A *bool* or *Auth* instance. Defaults to loading from *auth.load_auth* if set to True. To disable authentication, set to False.

- **journal_path** – The path on the filesystem where the messages being sent should be recorded until they are known to have been successfully received. This path should be located somewhere that will survive system restarts, and if messages contain sensitive data it should be noted that they will be written unencrypted to this path. The journal size is generally limited to the sum of sizes of messages queued for sending or in flight at the same time, plus some small (few tens of bytes per message) bookkeeping overhead. Note that this size can become large is a lengthy network disruption prev ents messages from being sent; enough disk spacec should be available to cover this possibility for the expected message rate and duration of disruptions which may need to be handled.
- **poll_wait** – The time the publisher should spend checking for receipt of each message directly after sending it. Tuning this parameter controls a tradeoff between low latency discovery of successful message delivery and throughput. If the time between sending messages is large compared to the latency for a message to be sent and for a confirmation of receipt to return, it is useful to increase this value so that the publisher will wait to discover that each message has been sent (in the success case) instead of sleeping and waiting for another message to send. If this value is 'too low' (much smaller than both the time for a message to be sent and acknowledged and the time for the next message to be ready for sending), the publisher will waste CPU time entering and exiting the internal function used to receive event notifications. If this value is too large (larger than or similar in size to the time between messages needing to be sent) throughput will be lost as time will be spent waiting to see if the previous message has been acknowledged which could be better spent getting the next message sent out. When in doubt, it is probably best to err on the side of choosing a small value.
- **kwargs** – Any additional arguments to be passed to `hop.io.open`.

Raises

- **OSError** – If a journal file exists but cannot be read.
- **Runtime Error** – If the contents of the journal file are corrupted.

run()

This method is not part of the public interface of this class, and should not be called directly by users.

start()

Start the background communication thread used by the publisher to send messages. This should be called prior to any calls to *RobustProducer.write*. This method should not be called more than once.

stop()

Stop the background communication thread used by the publisher to send messages. This method will block until the thread completes, which includes sending all queued messages. *RobustProducer.write* should not be called after this method has been called. This method should not be called more than once.

write(*message*, *headers=None*)

Queue a message to be sent. Message sending occurs asynchronously on a background thread, so this method returns immediately unless an error occurs queuing the message. *RobustProducer.start* must be called prior to calling this method.

Parameters

- **message** – A message to send.
- **headers** – Headers to be sent with the message, as a list of 2-tuples of strings.

Raises

- **RuntimeError** – If appending the new message to the on-disk journal fails.
- **TypeError** – If the message is not a suitable type.

2.1.6 hop.publish

2.1.7 hop.subscribe

2.1.8 hop.models

class hop.models.**AvroBlob**(*content: List[Union[str, int, float, bool, None, Dict[str, Any], List[Any]]]*, *schema: Optional[dict] = None*)

Defines an unformatted message blob.

classmethod deserialize(*data*)

Unwrap a message produced by serialize() (the "content" value).

Returns
An instance of the model class.

classmethod load(*blob_input*)

Create a blob message from input avro data.

Parameters
blob_input – The encoded Avro data or file object.

Returns
The Blob.

classmethod load_file(*filename*)

Create a new message model from a file.

Parameters
filename – The path to a file.

Returns
The message model.

serialize()

Wrap the message with its format and content.

Returns
A dictionary with "format" and "content" keys.

class hop.models.**Blob**(*content: bytes*)

Defines an opaque message blob.

classmethod deserialize(*data*)

Unwrap a message produced by serialize() (the "content" value).

Returns
An instance of the model class.

classmethod load(*blob_input*)

Create a blob message from input data.

Parameters
blob_input – The unstructured message data (bytes) or file object.

Returns
The Blob.

serialize()

Wrap the message with its format and content.

Returns
A dictionary with "format" and "content" keys. The value stored under "format" is the format label. The value stored under "content" is the actual encoded data.

class hop.models.**GCNCircular**(*header: dict*, *body: str*)

Defines a GCN Circular structure.

The parsed GCN circular is formatted as a dictionary with the following schema:

{'headers': {'title': ..., 'number': ..., ...}, 'body': ...}

classmethod load(*email_input*)

Create a new GCNCircular from an RFC 822 formatted circular.

Parameters
email_input – A file object or string.

Returns
The GCNCircular.

class hop.models.**JSONBlob**(*content: Union[str, int, float, bool, None, Dict[str, Any], List[Any]]*)

Defines an unformatted message blob.

classmethod deserialize(*data*)

Unwrap a message produced by serialize() (the "content" value).

Returns
An instance of the model class.

classmethod load(*blob_input*)

Create a blob message from input text.

Parameters
blob_input – The unstructured message text or file object.

Returns
The Blob.

serialize()

Wrap the message with its format and content.

Returns
A dictionary with "format" and "content" keys. The value stored under "format" is the format label. The value stored under "content" is the actual encoded data.

class hop.models.**MessageModel**

An abstract message model.

classmethod deserialize(*data*)

Unwrap a message produced by serialize() (the "content" value).

Returns
An instance of the model class.

abstract classmethod load(*input_*)

Create a new message model from a file object or string. This base implementation has no functionality and should not be called.

Parameters
input – A file object or string.

Returns
The message model.

`classmethod load_file(filename)`

Create a new message model from a file.

Parameters
filename – The path to a file.

Returns
The message model.

`serialize()`

Wrap the message with its format and content.

Returns
A dictionary with "format" and "content" keys. The value stored under "format" is the format label. The value stored under "content" is the actual encoded data.

`class hop.models.VOEvent(ivorn: str, role: str = 'observation', version: str = '2.0', Who: dict = <factory>, What: dict = <factory>, WhereWhen: dict = <factory>, How: dict = <factory>, Why: dict = <factory>, Citations: dict = <factory>, Description: dict = <factory>, Reference: dict = <factory>)`

Defines a VOEvent 2.0 structure.

Implements the schema defined by:
http://www.ivoa.net/Documents/VOEvent/20110711/

`classmethod load(xml_input)`

Create a new VOEvent from an XML-formatted VOEvent.

Parameters
xml_input – A file object, string, or generator.

Returns
The VOEvent.

`classmethod load_file(filename)`

Create a new VOEvent from an XML-formatted VOEvent file.

Parameters
filename – Name of the VOEvent file.

Returns
The VOEvent.

2.1.9 hop.plugins

`hop.plugins.get_models()`

This plugin spec is used to return message models in the form:
{"type": Model}

where the type refers to a specific message model.

2.1.10 hop.version

hop.version.**get_packages**()

Returns the package dependencies used within hop-client.

hop.version.**print_packages_versions**()

Print versions for the passed packages.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

h

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

O

P

Q

R

S

U

V

W