hop-client Documentation

SCiMMA

Sep 16, 2020

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, you must use the channel from the SCiMMA Anaconda organization:

```
conda install --channel scimma 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 configuration settings from `config.toml`. The path to this configuration can be found by running `hop configure locate`. One can initialize this configuration with default settings by running `hop configure setup`. To disable authentication in the CLI client, one can run `--no-auth`.

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 until an end of stream (EOS) is received.

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)
```

This will listen to the Hop broker, listening to new messages and printing them to stdout as they arrive until there are no more messages in the stream. 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)
```

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

The `persist` option allows one to listen to messages forever and keeps the connection open after an end of stream (EOS) is received. 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)
```

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`
- `persist`: Whether to keep a long-live connection to the client beyond EOS

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)
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, 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://[groupid@]broker/topic[,topic2[,...]]
```

The broker takes the form `hostname[:port]` and gives the URL to connect to a Kafka broker. Optionally, a `groupid` is provided which is used to keep track of which messages have been read from a topic with a given group ID. This allows a long-lived process reading messages to pick up where they left off after a restart, for example. 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://mygroup@hostname:port/topic1", "r") as s:
    for message, metadata in s.read(metadata=True, autocommit=False):
        print(message, metadata.topic)
        s.mark_done(metadata)
```

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 configure` 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 configure setup`, which prompts the user 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 in the configuration file generation as `hop configure setup --import <CREDENTIALS_FILE>`

The default location for the configuration file can be found with `hop configure locate`, which points by default to `${HOME}/.config/hop/config.toml`, but can be configured by setting the `XDG_CONFIG_PATH` variable.

1.4.2 Using Credentials

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

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"})
```

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 `configure` 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 messages can be sent directly to an open `Stream` instance and will be serialized appropriately. Any python object that can be JSON serialized can be sent. Examples include a dictionary, a byte64 encoded string, and a list.

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` 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

    @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)
```

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

...

@plugins.register
def get_models():
    return {
        "donut": Donut,
    }
```

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 Commands

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

- `hop configure`: Authentication utilities
- `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`

1.6.1 hop configure

This command allows a user to handle auth-based configuration.

```
usage: hop configure [-h] <command> ...

Configuration utilities.

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

commands:
    locate            display configuration path
    setup             set up configuration
```

1.6.2 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 byte-encoded or JSON-serializable data.

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

```
usage: hop publish [-h] [--no-auth] [-f {VOEVENT,CIRCULAR,BLOB}]
                   URL [MESSAGE [MESSAGE ...]]

Publish messages.

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

optional arguments:
  -h, --help            show this help message and exit
  --no-auth             If set, disable authentication.
  -f {VOEVENT,CIRCULAR,BLOB}, --format {VOEVENT,CIRCULAR,BLOB}
                        Specify the message format. Defaults to BLOB for an
                        unstructured message.
```

1.6.3 hop subscribe

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

```
usage: hop subscribe [-h] [--no-auth] [-s {EARLIEST,LATEST}] [-p] [-j] URL

Subscribe to messages.

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.
  -s {EARLIEST,LATEST}, --start-at {EARLIEST,LATEST}
                        Set the message offset offset to start at. Default:
                        LATEST.
  -p, --persist         If set, persist or listen to messages indefinitely.
                        Otherwise, will stop listening when EOS is received.
  -j, --json            Request message output as raw json
```

1.6.4 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
```

CHAPTER

TWO

API REFERENCE

2.1 hop-client API

2.1.1 hop.auth

class hop.auth.**Auth**(*user*, *password*, *ssl=True*, *method=<SASLMethod.SCRAM_SHA_512: 3>*, ***kwargs*)

Attach SASL-based authentication to a client.

Returns client-based auth options when called.

user [*str*] Username to authenticate with.

password [*str*] Password to authenticate with.

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.

hop.auth.**load_auth**(*config_file='/home/docs/.config/hop/config.toml'*)

Configures an Auth instance given a configuration file.

Args:

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

Returns: A configured Auth instance.

Raises: KeyError: An error occurred parsing the configuration file.

2.1.2 hop.cli

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

Add general client options to an argument parser.

Args: parser: An ArgumentParser instance to add client options to.

2.1.3 hop.configure

hop.configure.**get_config_path**()
Determines the default location for auth configuration.

Returns: The path to the authentication configuration file.

hop.configure.**set_up_configuration**(*config_file*, *csv_file*)
Set up configuration file.

Args: config_file: Configuration file path csv_file: Path to csv credentials file

hop.configure.**write_config_file**(*config_file*, *username*, *password*)
Write configuration file for the given username and password.

Args: config_file: configuration file path username: username at hopskotch password: password at hopskotch

2.1.4 hop.io

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

class hop.io.**Metadata**(*topic: str*, *partition: int*, *offset: int*, *timestamp: int*, *key: Union[str, bytes]*, *_raw: cimpl.Message*)
Broker-specific metadata that accompanies a consumed message.

class hop.io.**Stream**(*auth=True*, *start_at=<ConsumerStartPosition.LATEST: 2>*, *persist=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.

Args:

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. persist: Whether to listen to new messages forever or stop

when EOS is received in read mode. Defaults to False.

open(*url*, *mode='r'*)
Opens a connection to an event stream.

Args: url: Sets the broker URL to connect to.

Kwargs: mode: Read ('r') or write ('w') from the stream.

Returns: An open connection to the client, either an adc Producer instance in write mode or an adc Consumer instance in read mode.

Raises:

ValueError: If the mode is not set to read/write or if more than one topic is specified in write mode.

2.1.5 hop.publish

2.1.6 hop.subscribe

hop.subscribe.**print_message**(*message_model*, *json_dump=False*)
Print the content of a message.

Args: message_model: dataclass model object for a message json_dump: boolean indicating whether to print as raw json

Returns: None

2.1.7 hop.models

class hop.models.**Blob**(*content: Union[str, int, float, bool, None, Dict[str, Any], List[Any]]*, *missing_schema: bool = False*)
Defines an unformatted message blob.

asdict()
Represents the message as a dictionary.

Returns: The dictionary representation of the message.

classmethod load(*blob_input*)
Create a blob message from input text.

Args: blob_input: The unstructured message text or file object.

Returns: The Blob.

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.

Args: email_input: A file object or string.

Returns: The GCNCircular.

serialize()
Wrap the message with its format and content.

Returns: A dictionary with "format" and "content" key-value pairs.

class hop.models.**MessageModel**
An abstract message model.

asdict()
Represents the message model as a dictionary.

abstract classmethod load(*input_*)
Create a new message model from a file object or string.

Args: **input_**: A file object or string.

Returns: The message model.

classmethod load_file(*filename*)
Create a new message model from a file.

Args: 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.**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.

Args: 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.

Args: filename: Name of the VOEvent file.

Returns: The VOEvent.

2.1.8 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.

2.1.9 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