ndn-python-repo

Zhaoning Kong

Jul 13, 2022

CONTENTS:

CHAPTER ONE

NDN-PYTHON-REPO

A Named Data Networking (NDN) Repo implementation using python-ndn.

Please see our documentation if you have any issues.

CHAPTER TWO

INSTALL AND RUN

2.1 Install

Install the latest release with pip:

```
$ /usr/bin/pip3 install ndn-python-repo
```

Optionally, you can install the latest development version from source:

```
$ git clone https://github.com/JonnyKong/ndn-python-repo.git
$ cd ndn-python-repo && /usr/bin/pip3 install -e .
```

2.2 Migrate from repo-ng

ndn-python-repo provides a script to migrate existing data from repo-ng:

```
$ ndn-python-repo-port -d <path-to-repo-ng-dbfile> \
                       -a <ndn-python-repo-ipaddr> \
                       -p <ndn-python-repo-port>
```

It takes as input a repo-ng database file, reads the Data packets and pipe them through TCP bulk insert into the new repo.

2.3 Instruction for developers

Setup virtual environment with editable installation:

```
$ python3 -m venv venv
$ . venv/bin/activate
$ pip3 install -e .
```

Run all tests:

```
$ pip3 install pytest
$ pytest
```

Compile the documentation with Sphinx:

```
$ cd docs && pip3 install -r requirements.txt
$ make html
$ open _build/html/index.html
```

Or one can use Pipenv to run it:

```
$ pipenv install --dev
$ pipenv run main   # This starts the repo as an app
$ pipenv run test   # Please nfd-start before running unit test
```

CHAPTER

THREE

CONFIGURATION

You can configure ndn-python-repo with a config file, by specifying the path to the file when starting a repo instance:

```
$ ndn-python-repo -c <config_file>
```

A sample config file is provided at `ndn_python_repo/ndn-python-repo.conf.sample`.

If no config file is given on the command line, this sample config file will be used by default.

3.1 Repo namespace

Specify the name of a repo in the config file. For example:

```
repo_config:
  # the repo's routable prefix
  repo_name: 'testrepo'
```

Another option is to specify the repo name when starting a repo on the command line. This overrides the repo name in the config file:

```
$ ndn-python-repo -r "/name_foo"
```

3.2 Repo prefix registration

By default, the repo registers the root prefix `/`.

Alternatively, you can configure repo such that it doesn't register the root prefix:

```
repo_config:
  register_root: False
```

If `register_root` is set to `False`, the client is responsible of telling the repo which prefix to register or unregister every time in `RepoCommandParameter`. See *Insert* and *Delete* for details.

3.3 Choose the backend database

The ndn-python-repo uses one of the three backend databases:

- SQLite3 (default)
- leveldb
- MongoDB

To use non-default databases, perform the following steps:

1. Install ndn-python-repo with additional database support that you need:

```
$ /usr/bin/pip3 install ndn-python-repo[leveldb]
$ /usr/bin/pip3 install ndn-python-repo[mongodb]
```

2. Specify the database selection and database file in the config file. For example:

```
db_config:
  # choose one among sqlite3, leveldb, and mongodb
  db_type: 'mongodb'

  # only the chosen db's config will be read
  mongodb:
    'db': 'repo'
    'collection': 'data'
```

3.4 TCP bulk insert

By default, the repo listens on `0.0.0.0:7376` for TCP bulk insert. You can configure in the config file which address the repo listens on. For example:

```
tcp_bulk_insert:
  'addr': '127.0.0.1'
  'port': '7377'
```

3.5 Logging

Repo uses the python logging module, and by default logs all messages of and above level `INFO` to `stdout`. You can override the default options in the config file. For example:

```
logging_config:
  'level': 'WARNING'
  'file': '/var/log/ndn/ndn-python-repo/repo.log'
```

3.6 systemd

To run ndn-python-repo with systemd on Linux, perform the following steps:

1. Run the provided script to install the systemd script to `/etc/systemd/system/`:

```
$ sudo ndn-python-repo-install
```

2. Then, start, stop, and monitor a repo instance with systemd:

```
$ sudo systemctl start ndn-python-repo
$ sudo systemctl stop ndn-python-repo
$ sudo systemctl status ndn-python-repo
```

3. Examine logs:

```
$ sudo journalctl -u ndn-python-repo.service
```

CHAPTER FOUR

SPECIFICATION

4.1 Encoding

Most repo commands and status reports are Data packets whose Content contains `RepoCommandParam` or `RepoCommandRes` structure. These Data are issued via Pub-Sub protocol. Each `RepoCommandParam` and `RepoCommandRes` contains multiple `ObjectParam` and `ObjectResult`, resp. These structures are defined as follows:

```
ObjectParam =
    Name
    [ForwardingHint]
    [StartBlockId]
    [EndBlockId]
    [RegisterPrefix]

ObjectResult =
    Name
    StatusCode
    [InsertNum]
    [DeleteNum]

RepoCommandParam =
    1 * (OBJECT-PARAM-TYPE TLV-LENGTH ObjectParam)

RepoCommandRes =
    StatusCode
    0 * (OBJECT-RESULT-TYPE TLV-LENGTH ObjectResult)

RepoStatQuery =
    RequestNo

ForwardingHint = FORWARDING-HINT-TYPE TLV-LENGTH Name

StartBlockId = START-BLOCK-ID-TYPE TLV-LENGTH NonNegativeInteger

EndBlockId = END-BLOCK-ID-TYPE TLV-LENGTH NonNegativeInteger

RegisterPrefix = REGISTER-PREFIX-TYPE TLV-LENGTH Name

StatusCode = STATUS-CODE-TYPE TLV-LENGTH NonNegativeInteger
```

(continues on next page)

(continued from previous page)

```
InsertNum = INSERT-NUM-TYPE TLV-LENGTH NonNegativeInteger

DeleteNum = DELETE-NUM-TYPE TLV-LENGTH NonNegativeInteger

RequestNo = REQUEST-NO-TYPE TLV-LENGTH 1*OCTET
```

The type number assignments are as follows:

type	Assigned number (decimal)	Assigned number (hexadecimal)
START-BLOCK-ID-TYPE	204	0xCC
END-BLOCK-ID-TYPE	205	0xCD
REQUEST-NO-TYPE	206	0xCE
STATUS-CODE-TYPE	208	0xD0
INSERT-NUM-TYPE	209	0xD1
DELETE-NUM-TYPE	210	0xD2
FORWARDING-HINT-TYPE	211	0xD3
REGISTER-PREFIX-TYPE	212	0xD4
OBJECT-PARAM-TYPE	301	0x12D
OBJECT-RESULT-TYPE	302	0x12E

4.1.1 Status Code Definition

The status codes are defined as follows:

Code name	Value	Explanation
ROGER	100	Command received but not been executed yet
COMPLETED	200	Command completed
IN-PROGRESS	300	Command working in progress
FAILED	400	Command or parts of it failed
MALFORMED	403	Command is malformed
NOT-FOUND	404	Queried command not found

4.2 Insert

Repo insertion process makes use of the *PubSub package*.

1. The repo subscribes to the topic `/<repo_name>/insert`.
2. The client publishes a message to the topic `/<repo_name>/insert`. The message payload is `RepoCommandParam` containing one or more `ObjectParam` with the following fields:
    - `name`: either a Data packet name, or a name prefix of segmented Data packets.
    - `start_block_id` (Optional): inclusive start segment number.
    - `end_block_id` (Optional): inclusive end segment number.
    - `forwarding_hint` (Optional): forwarding hint for Data fetching.

      This is useful in two scenarios:

    - The producer choose not to announce its name prefix, but only allow the repo to reach it via forwarding hint.
    - The name prefix is already announced by repo node(s), but the producer in another node wants to insert to the repo.
  - `register_prefix` (Optional): if repo doesn't register the root prefix (*Configuration* `register_root` is disabled), client can tell repo to register this prefix.

3. The repo fetches and inserts single or segmented Data packets according to given parameters.

- If neither `start_block_id` nor `end_block_id` are given, the repo fetches a single packet identified in `name` parameter. The insertion process succeeds when this packet is received.
- If only `end_block_id` is given, `start_block_id` is considered 0.
- If only `start_block_id` is given, `end_block_id` is auto detected, i.e. infinity.
- If both block ids are given, the command is considered as correct only if `end_block_id >= start_block_id`.
- Whenever the repo cannot fetch a segment, it will stop, no matter what `end_block_id` is.
- Segment numbers are encoded in accordance with NDN naming conventions rev2.

4.2.1 Insert status check

The client can use the *Check* protocol to check the progress of an insertion process. The insertion check response message payload is `RepoCommandRes` containing zero or more `ObjectResult` with the following fields:

- `status_code`: status code, as defined on *Check*. Both the command itself and objects has a status code.
- `name`: the name of object to insert.
- `insert_num`: number of Data packets received by the repo so far.
- The number of `ObjectResult` in the result should be either: * =0, which means the command is malformed or not allowed. * equals to the number of `ObjectParam` in the insertion command.

4.3 Delete

Repo deletion process makes use of the *PubSub package*.

1. The repo subscribes to the topic `/<repo_name>/delete`.
2. The client publishes a message to the topic `/<repo_name>/delete`. The message payload is `RepoCommandParam` containing one or more `ObjectParam` with the following fields:

- `name`: either a Data packet name, or a name prefix of Data packets.
- `start_block_id` (Optional): inclusive start segment number.
- `end_block_id` (Optional): inclusive end segment number.
- `register_prefix` (Optional): if repo doesn't register the root prefix (*Configuration* `register_root` is disabled), client can tell repo to unregister this prefix.

3. The repo deletes Data packets according to given parameters.

- If both `start_block_id` and `end_block_id` are omitted, the repo deletes a single packet identified in `name` parameter. The deletion process succeeds when this packet is deleted.

- If `start_block_id` is specified but `end_block_id` is omitted, the repo starts deleting segments starting from `/name/start_block_id`, and increments segment number after each packet. When a name query does not find an existing segment, the deletion process stops and is considered successful.
- Otherwise, the repo fetches all segments between `/name/start_block_id` and `/name/end_block_id`. If `start_block_id` is omitted, it defaults to 0. The deletion process succeeds when all packets are deleted.
- Segment numbers are encoded in accordance with NDN naming conventions rev2.

Warning: Please use exactly the same parameters as you inserted the Data to delete them. The current maintainer is not sure whether there will be problems if you provide a wrong `register_prefix` or only delete partial of the segments (i.e. provide different block ids). Also, using single packet deletion command to delete a segment Data object or vice versa will always fail, with `delete_num` being 0.

4.3.1 Delete status check

The client can use the *Check* protocol to check the progress of an deletion process. The deletion check response message payload is `RepoCommandRes` containing zero or more `ObjectResult` with the following fields:

- `status_code`: status code, as defined on *Check*. Both the command itself and objects has a status code.
- `name`: the name of object to delete.
- `delete_num`: number of Data packets deleted by the repo so far.
- The number of `ObjectResult` in the result should be either: * =0, which means the command is malformed or not allowed. * equals to the number of `ObjectParam` in the deletion command.

4.4 Check

The check protocol is used by clients to check the progress of a insertion or deletion process.

1. The check prefix for `<command>` is `/<repo_name>/<command> check`. For example, the check prefix for insertion is `/<repo_name>/insert check`, and deletion is `/<repo_name>/delete check`.
2. Status check Interests are send to the check prefix directly. No Pub-Sub is used here.
3. The check Interest should carry an application parameter `RepoStatQuery`, which contains the request number of the command. The request number of the command is always the SHA256 hash of the command data published in Pub-Sub.
4. After receiving the query Interest, the repo responds with a Data packet containing `RepoCommandRes`.
5. The status is only kept for 60s after the operation finishes. After that time, all queries will be responded with `NOT-FOUND`.

4.5 RepoCommandRes

- The `RepoCommandRes` Data contains a status code for the whole command, with the following rules:
  - `MALFORMED`: If the command cannot be parsed.
  - `NOT-FOUND`: If the given request no is not associated with a valid command. This is also returned when the repo has not finish fetching the command from Pub-Sub, or the command has finished for more than 60s.
  - `COMPLETED`: If all operations (for all objects) completed.
  - `IN-PROGRESS`: The command is received and being executed.
  - `FAILED`: If one or more operation in the command fails. If the is insertion, this means some or all objects requested to insert cannot be completely fetched. However, fetched objects or segments are still inserted into the repo. Only the objects with `insert_num=0` are not inserted.
- For each `ObjectResult` contained in the `RepoCommandRes`, the status code can be one of the following:
  - `ROGER`: The whole command is received and the operation on this object will be started in the future.
  - `MALFORMED`: If the object has wrong parameter.
  - `FAILED`: If the operation on this object failed to execute. For example, not all segments specified can be fetched. Note that even for a failed object, fetched segments are still put into the repo and can be fetched.
  - `COMPLETED`: If the operation on this object succeeded.

4.6 TCP bulk insert

CHAPTER

FIVE

MISCELLANEOUS PACKAGES

5.1 Client-side packages

5.1.1 Introduction

Application built with python-ndn can make use of the client packages provided.

There are four parts:

1. **PutfileClient**: insert files into the repo.
2. **GetfileClient**: get files from the repo.
3. **DeleteClient**: detele data packets from the repo.
4. **CommandChecker**: check process status from the repo.

The example programs in `examples/` illustrate how to use these packages.

Note that the type `Union[Iterable[Union[bytes, bytearray, memoryview, str]], str, bytes, bytearray, memoryview]` in the documentation is equivalent to the `ndn.name.NonStrictName` type.

5.1.2 Reference

class `ndn_python_repo.clients.putfile.`**`PutfileClient`**(*app*, *prefix*, *repo_name*)

A client to insert files into the repo.

Parameters

- **`app`** (`NDNApp`) – NDNApp.
- **`prefix`** (`Union[Iterable[Union[bytes, bytearray, memoryview, str]], str, bytes, bytearray, memoryview]`) – NonStrictName. The name of this client
- **`repo_name`** (`Union[Iterable[Union[bytes, bytearray, memoryview, str]], str, bytes, bytearray, memoryview]`) – NonStrictName. Routable name to remote repo.

async **`insert_file`**(*file_path*, *name_at_repo*, *segment_size*, *freshness_period*, *cpu_count*, *forwarding_hint=None*, *register_prefix=None*, *check_prefix=None*)

Insert a file to remote repo.

Parameters

- **`file_path`** (`str`) – Local FS path to file to insert.
- **`name_at_repo`** (`Union[Iterable[Union[bytes, bytearray, memoryview, str]], str, bytes, bytearray, memoryview]`) – NonStrictName. Name used to store file at repo.

- **segment_size** (`int`) – Max size of data packets.
- **freshness_period** (`int`) – Freshness of data packets.
- **cpu_count** (`int`) – Cores used for converting file to TLV format.
- **forwarding_hint** (`Union[Iterable[Union[bytes, bytearray, memoryview, str]], str, bytes, bytearray, memoryview, None]`) – NonStrictName. The forwarding hint the repo uses when fetching data.
- **register_prefix** (`Union[Iterable[Union[bytes, bytearray, memoryview, str]], str, bytes, bytearray, memoryview, None]`) – NonStrictName. If repo is configured with `register_root=False`, it registers `register_prefix` after receiving the insertion command.
- **check_prefix** (`Union[Iterable[Union[bytes, bytearray, memoryview, str]], str, bytes, bytearray, memoryview, None]`) – NonStrictName. The repo will publish process check messages under `<check_prefix>/check`. It is necessary to specify this value in the param, instead of using a predefined prefix, to make sure the subscriber can register this prefix under the NDN prefix registration security model. If not specified, default value is the client prefix.

Return type
`int`

Returns
Number of packets inserted.

`class ndn_python_repo.clients.getfile.`**`GetfileClient`**`(`*app*, *repo_name*`)`

This client fetches a file from the repo, and save it to working directory.

A client to retrieve files from the remote repo.

Parameters

- **app** (`NDNApp`) – NDNApp.
- **repo_name** – NonStrictName. Routable name to remote repo.

`async` **`fetch_file`**`(`*name_at_repo*, *local_filename=None*, *overwrite=False*`)`

Fetch a file from remote repo, and write to the current working directory.

Parameters

- **name_at_repo** (`Union[Iterable[Union[bytes, bytearray, memoryview, str]], str, bytes, bytearray, memoryview]`) – NonStrictName. The name with which this file is stored in the repo.
- **local_filename** (`Optional[str]`) – str. The filename of the retrieved file on the local file system.
- **overwrite** – If true, existing files are replaced.

`class ndn_python_repo.clients.delete.`**`DeleteClient`**`(`*app*, *prefix*, *repo_name*`)`

This client deletes data packets from the remote repo.

Parameters

- **app** (`NDNApp`) – NDNApp.
- **repo_name** (`Union[Iterable[Union[bytes, bytearray, memoryview, str]], str, bytes, bytearray, memoryview]`) – NonStrictName. Routable name to remote repo.

`async delete_file(prefix, start_block_id=0, end_block_id=None, register_prefix=None, check_prefix=None)`

Delete from repo packets between "<name_at_repo>/<start_block_id>" and "<name_at_repo>/<end_block_id>" inclusively.

Parameters

- **prefix** (`Union[Iterable[Union[bytes, bytearray, memoryview, str]], str, bytes, bytearray, memoryview]`) – NonStrictName. The name of the file stored in the remote repo.
- **start_block_id** (`int`) – int. Default value is 0.
- **end_block_id** (`Optional[int]`) – int. If not specified, repo will attempt to delete all data packets with segment number starting from *start_block_id* continously.
- **register_prefix** (`Union[Iterable[Union[bytes, bytearray, memoryview, str]], str, bytes, bytearray, memoryview, None]`) – If repo is configured with `register_root=False`, it unregisters `register_prefix` after receiving the deletion command.
- **check_prefix** (`Union[Iterable[Union[bytes, bytearray, memoryview, str]], str, bytes, bytearray, memoryview, None]`) – NonStrictName. The repo will publish process check messages under `<check_prefix>/check`. It is necessary to specify this value in the param, instead of using a predefined prefix, to make sure the subscriber can register this prefix under the NDN prefix registration security model. If not specified, default value is the client prefix.

Return type
`int`

Returns
Number of deleted packets.

`class ndn_python_repo.clients.command_checker.CommandChecker(app)`

This client sends check interests to the repo.

Parameters
app (`NDNApp`) – NDNApp.

`async check_delete(repo_name, request_no)`

Check the status of a delete process.

Parameters

- **repo_name** – NonStrictName. The name of the remote repo.
- **request_no** (`bytes`) – bytes. The request id of the process to check.

Return type
`RepoCommandRes`

Returns
The response from the repo.

`async check_insert(repo_name, request_no)`

Check the status of an insert process.

Parameters

- **repo_name** (`Union[Iterable[Union[bytes, bytearray, memoryview, str]], str, bytes, bytearray, memoryview]`) – NonStrictName. The name of the remote repo.

- **request_no** (`bytes`) – bytes. The request id of the process to check.

Return type
`RepoCommandRes`

Returns
The response from the repo.

5.2 `ConcurrentFetcher` package

5.2.1 Introduction

Fetch data packets in parallel using a fixed window size.

Note that the type `Union[Iterable[Union[bytes, bytearray, memoryview, str]], str, bytes, bytearray, memoryview]` in the documentation is equivalent to the `ndn.name.NonStrictName` type.

5.2.2 Reference

`async ndn_python_repo.utils.`**`concurrent_fetcher`**(*app*, *name*, *start_block_id*, *end_block_id*, *semaphore*, ***kwargs*)

An async-generator to fetch data packets between "*name/start_block_id*" and "*name/end_block_id*" concurrently.

Parameters

- **app** (`NDNApp`) – NDNApp.
- **name** (`Union[Iterable[Union[bytes, bytearray, memoryview, str]], str, bytes, bytearray, memoryview]`) – NonStrictName. Name prefix of Data.
- **start_block_id** (`int`) – int. The start segment number.
- **end_block_id** (`Optional[int]`) – Optional[int]. The end segment number. If not specified, continue fetching until an interest receives timeout or nack or 3 times.

Returns
Yield `(FormalName, MetaInfo, Content, RawPacket)` tuples in order.

5.3 `PubSub` package

5.3.1 Introduction

The `PubSub` package provides a pub-sub API with best-effort, at-most-once delivery guarantee.

If there are no subscribers reachable when a message is published, this message will not be re-transmitted.

If there are multiple subscribers reachable, the nearest subscriber will be notified of the published message in an any-cast style.

Note that the type `Union[Iterable[Union[bytes, bytearray, memoryview, str]], str, bytes, bytearray, memoryview]` in the documentation is equivalent to the `ndn.name.NonStrictName` type.

5.3.2 Process

Under the hood the PubSub module transmits a series of Interest and Data packets:

1. The subscriber calls `subscribe(topic, cb)`. This makes the subcriber listen on `"/<topic>/notify"`.

2. The publisher invokes `publish(topic, msg)`. This method sends an Interest with name `"/<topic>/notify"`, which will be routed to a subscriber. The interest carries the following fields in its application parameters:

   - Publisher prefix: used by the subscriber to reach the publisher in the next step
   - NotifyNonce: a random bytes string, used by the publisher to de-multiplex among different publications
   - Forwarding hint (optional): if publisher prefix is not announced in the routing system, publisher can provide a forwarding hint

   Meanwhile, `msg` is wrapped into a Data packet named `"/<pub_prefix>/msg/<topic>/<notify_nonce>"`. Here, the data name contains `topic` to establish a binding between topic and nonce, to prevent man-in-the-middle attacks that changes the topic.

3. The subscriber receives the notification interest, constructs a new Interest `"/<pub_prefix>/msg/<topic>/<notify_nonce>"` and send it to the publisher.

4. The publisher receives the interest `"/<pub_prefix>/msg/<topic>/<notify_nonce>"`, and returns the corresponding data.

5. The subscriber receives the data, and invokes `cb(data.content)` to hand the message to the application.

6. The publisher receives the acknowledgement Data packet, and erases the soft state.

5.3.3 Encoding

The notify Interest's application parameter is encoded as follows:

```
NotifyAppParam = DATA-TYPE TLV-LENGTH
    [PublisherPrefix]
    [NotifyNonce]
    [PublisherFwdHint]

PublisherPrefix = Name

NotifyNonce = NOTIFY-NONCE-TYPE TLV-LENGTH Bytes

PublisherFwdHint = PUBLISHER-FWD-HINT-TYPE TLV-LENGTH Name
```

The type number assignments are as follows:

type	Assigned number (decimal)	Assigned number (hexadecimal)
NOTIFY-NONCE-TYPE	128	0x80
PUBLISHER-FWD-HINT-TYPE	211	0xD3

5.3.4 Reference

class `ndn_python_repo.utils.PubSub`(*app*, *prefix=None*, *forwarding_hint=None*)

Initialize a `PubSub` instance with identity `prefix` and can be reached at `forwarding_hint`. TODO: support msg larger than MTU

Parameters

- **app** (`NDNApp`) – NDNApp.
- **prefix** (`Union[Iterable[Union[bytes, bytearray, memoryview, str]], str, bytes, bytearray, memoryview, None]`) – NonStrictName. The identity of this `PubSub` instance. The publisher needs a prefix under which can publish data. Note that you cannot initialize two `PubSub` instances with the same `prefix` on the same node, since it will cause double registration error.
- **forwarding_hint** (`Union[Iterable[Union[bytes, bytearray, memoryview, str]], str, bytes, bytearray, memoryview, None]`) – NonStrictName. When working as publisher, if `prefix` is not reachable, the subscriber can use `forwarding_hint` to reach the publisher.

async `publish`(*topic*, *msg*)

Publish `msg` to `topic`. Make several attempts until the subscriber returns a response.

Parameters

- **topic** (`Union[Iterable[Union[bytes, bytearray, memoryview, str]], str, bytes, bytearray, memoryview]`) – NonStrictName. The topic to publish `msg` to.
- **msg** (`bytes`) – bytes. The message to publish. The pub-sub API does not make any assumptions on the format of this message.

Returns

Return true if received response from a subscriber.

`set_base_prefix`(*prefix*)

Avoid registering too many prefixes, by registering `prefix` with NFD. All other prefixes under `prefix` will be registered with interest filters, and will not have to be registered with NFD. Need to be called before `_wait_for_ready()`.

Parameters

prefix (`Union[Iterable[Union[bytes, bytearray, memoryview, str]], str, bytes, bytearray, memoryview]`) – NonStrictName. The base prefix to register.

`set_publisher_prefix`(*prefix*)

Set the identify of the publisher after initialization. Need to be called before `_wait_for_ready()`.

Parameters

prefix (`Union[Iterable[Union[bytes, bytearray, memoryview, str]], str, bytes, bytearray, memoryview]`) – NonStrictName. The identity of this `PubSub` instance.

`subscribe`(*topic*, *cb*)

Subscribe to `topic` with `cb`.

Parameters

- **topic** (`Union[Iterable[Union[bytes, bytearray, memoryview, str]], str, bytes, bytearray, memoryview]`) – NonStrictName. The topic to subscribe to.
- **cb** (`callable`) – callable. A callback that will be called when a message under `topic` is received. This function takes one `bytes` argument.

unsubscribe(*topic*)

Unsubscribe from `topic`.

Parameters

topic (`Union[Iterable[Union[bytes, bytearray, memoryview, str]], str, bytes, bytearray, memoryview]`) – NonStrictName. The topic to unsubscribe from.

async wait_for_ready()

Need to be called to wait for pub-sub to be ready.

5.4 `Storage` package

ndn-python-repo supports 3 types of databases as backends. The `Storage` package provides a unified key-value storage API with the following features:

- Supports `MustBeFresh`
- Supports `CanBePrefix`
- Batched writes with periodic writebacks to improve performance

The `Storage` class provides an interface, and is implemented by:

- `SqliteStorage`
- `LevelDBStorage`
- `MongoDBStorage`

Note that the type `Union[Iterable[Union[bytes, bytearray, memoryview, str]], str, bytes, bytearray, memoryview]` in the documentation is equivalent to the `ndn.name.NonStrictName` type.

5.4.1 Reference

class `ndn_python_repo.storage.`**Storage**

Interface for a unified key-value storage API.

get_data_packet(*name*, *can_be_prefix=False*, *must_be_fresh=False*)

Get a data packet named `name`.

Parameters

- **name** (`Union[Iterable[Union[bytes, bytearray, memoryview, str]], str, bytes, bytearray, memoryview]`) – NonStrictName. The name of the data packet.
- **can_be_prefix** (`bool`) – bool. If true, use prefix match instead of exact match.
- **must_be_fresh** (`bool`) – bool. If true, ignore expired data.

Return type

`Optional[bytes]`

Returns

The value of the data packet.

put_data_packet(*name*, *data*)

Insert a data packet named `name` with value `data`. This method will parse `data` to get its freshnessPeriod, and compute its expiration time by adding the freshnessPeriod to the current time.

Parameters

- **name** (Union[Iterable[Union[bytes, bytearray, memoryview, str]], str, bytes, bytearray, memoryview]) – NonStrictName. The name of the data packet.
- **data** (bytes) – bytes. The value of the data packet.

`remove_data_packet`(*name*)

Remove a data packet named `name`.

Parameters
name (Union[Iterable[Union[bytes, bytearray, memoryview, str]], str, bytes, bytearray, memoryview]) – NonStrictName. The name of the data packet.

Return type
`bool`

Returns
True if a data packet is being removed.

class ndn_python_repo.storage.**SqliteStorage**(*db_path*)

Init table "data" with the attribute `key` being the primary key.

Parameters
db_path (str) – str. Path to database file.

class ndn_python_repo.storage.**LevelDBStorage**(*dir*)

Creates a LevelDB storage instance at disk location `str`.

Parameters
dir (str) – str. The disk location of the database directory.

class ndn_python_repo.storage.**MongoDBStorage**(*db*, *collection*)

Init a MongoDB storage with unique index on key.

Parameters

- **db** (str) – str. Database name.
- **collection** (str) – str. Collection name.

CHAPTER

SIX

EXAMPLES

TBD

CHAPTER SEVEN

AUTHORS

- Zhaoning Kong <https://jonnykong.com>
- Xinyu Ma <https://zjkmxy.github.io>
- Yufeng Zhang <https://yufengzh.io>
- Zhiyi Zhang <https://zhiyi-zhang.com>
- Davide Pesavento
- Susmit Shannigrahi
- Saurab Dulal
- Junxiao Shi <https://yoursunny.com>

CHAPTER

EIGHT

FUTURE PLANS

- Fix knowing bugs and problems
- Make better tests
- Make a Docker container for test, since installing nfd takes too long.

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX