zenoh-python

Release 0.10.1-rc

ZettaScale Zenoh team, ⟨zenoh@zettascale.tech⟩

Feb 28, 2024

CONTENTS

[Zenoh](https://zenoh.io) /zeno/ is a stack that unifies data in motion, data at rest and computations. It elegantly blends traditional pub/sub with geo distributed storage, queries and computations, while retaining a level of time and space efficiency that is well beyond any of the mainstream stacks.

Before delving into the examples, we need to introduce few **Zenoh** concepts. First off, in Zenoh you will deal with **Resources**, where a resource is made up of a key and a value. The other concept you'll have to familiarize yourself with are **key expressions**, such as `robot/sensor/temp`, `robot/sensor/*`, `robot/**`, etc. As you can gather, the above key expression denotes set of keys, while the * and ** are wildcards representing respectively (1) a single chunk (non-empty sequence of characters that doesn't contain /), and (2) any amount of chunks (including 0).

Below are some examples that highlight these key concepts and show how easy it is to get started with.

QUICK START EXAMPLES:

1.1 Publish a key/value pair onto Zenoh

```
>>> import zenoh
>>> z = zenoh.open()
>>> z.put('/demo/example/hello', 'Hello World!')
```

1.2 Subscribe to a set of keys with Zenoh

```
>>> import zenoh, time
>>> def listener(sample):
>>>     print(f"{sample.key_expr} => {sample.payload.decode('utf-8')}")
>>>
>>> z = zenoh.open()
>>> subscriber = z.subscribe('/demo/example/**', listener)
>>> time.sleep(60)
>>> subscriber.undeclare()
```

1.3 Get keys/values from zenoh

```
>>> import zenoh
>>> z = zenoh.open()
>>> for response in z.get('/demo/example/**', zenoh.Queue()):
>>>     response = response.ok
>>>     print(f"{response.key_expr} => {response.payload.decode('utf-8')}")
```

1.3.1 module zenoh

zenoh.**init_logger**()

> Initialize the logger used by the Rust implementation of this API.
>
> Once initialized, you can configure the logs displayed by the API using the RUST_LOG environment variable. For instance, start python with the *debug* logs available:

```
$ RUST_LOG=debug python
```

> More details on the RUST_LOG configuration on https://docs.rs/env_logger/latest/env_logger

zenoh.**open**(**args*, ***kwargs*)

> Open a Zenoh session.
>
> > **Parameters**
> > **config** (Config) – The configuration of the Zenoh session
> >
> > **Return type**
> > *Session*
> >
> > **Example**

```
>>> import zenoh
>>> s = zenoh.open(zenoh.Config())
```

zenoh.**scout**(*handler=None*, *what=None*, *config=None*, *timeout=None*)

> Scout for routers and/or peers.
>
> This spawns a task that periodically sends scout messages for a specified duration and returns a list of received *Hello* messages.
>
> > **Parameters**
> >
> > - **what** (*str | None*) – The kind of zenoh process to scout for
> >
> > - **config** (Config *| None*) – The configuration to use for scouting
> >
> > - **timeout** – the duration of scout (in seconds)
> >
> > - **handler** (IHandler*[Hello, Any, Any] |* IClosure*[Hello, Any] | Tuple[IClosure, Any] | Tuple[Callable[[Hello], Any], Callable[[], None], Any] | Tuple[Callable[[Hello], Any], Callable[[], None]] | Callable[[Hello], Any] | None*) –
> >
> > **Return type**
> > list of *Hello*
> >
> > **Example**

```
>>> import zenoh
>>> for hello in zenoh.scout(what = "peer|router", timeout=1.0).receiver():
...     print(hello)
```

Hello

class zenoh.**Hello**

Represents a single Zenoh node discovered through scouting.

> **property locators: List[str]**
>
> > The locators through which this node may be adressed.
>
> **property whatami: str**
>
> > The node's type, returning either None, 'peer', 'router', or 'client'.
>
> **property zid:** *ZenohId*
>
> > The node's Zenoh UUID.

Config

class zenoh.**Config**

> **static from_file**(*filename*)
>
> > Reads the configuration from a file. The file's extension must be json, json5 or yaml.
> >
> > > **Parameters**
> > > **filename** (*str*) –
>
> **static from_json5**(*json*)
>
> > Reads the configuration from a JSON5 string.
> >
> > JSON5 is a superset of JSON, so any JSON string is a valid input for this function.
> >
> > > **Parameters**
> > > **json** (*str*) –
>
> **static from_obj**(*obj*)
>
> > Reads the configuration from `obj` as if it was a JSON file.
>
> **get_json**(*path*)
>
> > Returns the part of the configuration at `path`, in a JSON-serialized form.
> >
> > > **Parameters**
> > > **path** (*str*) –
> > >
> > > **Return type**
> > > str
>
> **insert_json5**(*path*, *value*)
>
> > Inserts the provided value (read from JSON) at the given path in the configuration.
> >
> > > **Parameters**
> > >
> > > - **path** (*str*) –
> > >
> > > - **value** (*str*) –
> > >
> > > **Return type**
> > > str

Session

class zenoh.**Session**(*config=None*)

> A Zenoh Session, the core interaction point with a Zenoh network.
>
> Note that most applications will only need a single instance of `Session`. You should _never_ construct one session per publisher/subscriber, as this will significantly increase the size of your Zenoh network, while preventing potential locality-based optimizations.
>
> > **Parameters**
> > **config** ([Config](#) | *Any*) –
>
> **close**()
>
> > Attempts to close the Session.
> >
> > The session will only be closed if all publishers, subscribers and queryables based on it have been undeclared, and there are no more python references to it.
>
> **config**()
>
> > Returns a configuration object that can be used to alter the session's configuration at runtime.
> >
> > Note that in Python specifically, the config you passed to the session becomes the result of this function if you passed one, letting you keep using it.
> >
> > > **Return type**
> > > [Config](#)
>
> **declare_keyexpr**(*keyexpr*)
>
> > Informs Zenoh that you intend to use the provided Key Expression repeatedly.
> >
> > This function returns an optimized representation of the passed `keyexpr`.
> >
> > It is generally not needed to declare key expressions, as declaring a subscriber, a queryable, or a publisher will also inform Zenoh of your intent to use their key expressions repeatedly.
> >
> > > **Parameters**
> > > **keyexpr** ([KeyExpr](#) | *_KeyExpr* | *str*) –
> > >
> > > **Return type**
> > > [KeyExpr](#)
>
> **declare_publisher**(*keyexpr*, *priority=None*, *congestion_control=None*)
>
> > Declares a publisher, which may be used to send values repeatedly onto a same key expression.
> >
> > Written resources that match the given key will only be sent on the network if matching subscribers exist in the system.
> >
> > > **Parameters**
> > > - **keyexpr** ([KeyExpr](#) | *_KeyExpr* | *str*) – The key expression to publish to
> > > - **priority** ([Priority](#) | *None*) – The priority to use when routing the published data
> > > - **congestion_control** ([CongestionControl](#) | *None*) – The congestion control to use when routing the published data
> > >
> > > **Return type**
> > > *[Publisher](#)*
> >
> > **Examples**

```
>>> import zenoh
>>> s = zenoh.open({})
>>> pub = s.declare_publisher('key/expression')
>>> pub.put('value')
```

declare_pull_subscriber(*keyexpr*, *handler*, *reliability=None*)

Declares a pull-mode subscriber, which will receive a single published sample with a key expression intersecting `keyexpr` any time its `pull` method is called.

These samples are passed to the *handler*'s closure as instances of the *Sample* class. The *handler* can typically be a queue or a callback. The *handler*'s receiver is returned as the *receiver* field of the returned *PullSubscriber*.

> **Parameters**
>
> - **keyexpr** (KeyExpr | _KeyExpr | str) – The key expression to subscribe to
>
> - **handler** (IHandler[Sample, Any, Any] | IClosure[Sample, Any] | Tuple[IClosure, Any] | Tuple[Callable[[Sample], Any], Callable[[], None], Any] | Tuple[Callable[[Sample], Any], Callable[[], None]] | Callable[[Sample], Any]) –
>
> - **reliability** (Reliability | None) – the reliability to use when routing the subscribed samples
>
> **Return type**
> *PullSubscriber*
>
> **Examples**

```
>>> import zenoh
>>> s = zenoh.open({})
>>> sub = s.declare_pull_subscriber('key/expression', lambda sample:
...     print(f"Received '{sample.key_expr}': '{sample.payload.decode('utf-8')}'
→"))
...
>>> sub.pull()
```

declare_queryable(*keyexpr*, *handler*, *complete=None*)

Declares a queryable, which will receive queries intersecting with `keyexpr`.

These queries are passed to the *handler* as instances of the *Query* class. The *handler* can typically be a queue or a callback. The *handler*'s receiver is returned as the *receiver* field of the returned *Queryable*. The replies can be sent back by calling the *reply `function of the `Query*.

> **Examples**
> **Parameters**
>
> - **keyexpr** (KeyExpr | _KeyExpr | str) –
>
> - **handler** (IHandler[Query, Any, Any] | IClosure[Query, Any] | Tuple[IClosure, Any] | Tuple[Callable[[Query], Any], Callable[[], None], Any] | Tuple[Callable[[Query], Any], Callable[[], None]] | Callable[[Query], Any]) –
>
> - **complete** (bool | None) –

Using a callback:

```
>>> import zenoh
>>> s = zenoh.open({})
>>> qabl = s.declare_queryable('key/expression', lambda query:
...     query.reply(zenoh.Sample('key/expression', 'value')))
```

Using a queue:

```
>>> import zenoh
>>> s = zenoh.open({})
>>> qabl = s.declare_queryable('key/expression', zenoh.Queue())
>>> while True:
...     query = qabl.receiver.get()
...     query.reply(zenoh.Sample('key/expression', 'value'))
...     del query
```

IMPORTANT: due to how RAII and Python work, you MUST bind this function's return value to a variable in order for it to function as expected. This is because as soon as a value is no longer referenced in Python, that value's destructor will run, which will undeclare your queryable, stopping it immediately.

declare_subscriber(*keyexpr*, *handler*, *reliability=None*)

Declares a subscriber, which will receive any published sample with a key expression intersecting `keyexpr`.

These samples are provided to the *handler* as instances of the *Sample* class. The *handler* can typically be a queue or a callback. The *handler*'s receiver is returned as the *receiver* field of the returned *Subscriber*.

> **Parameters**
>
> * **keyexpr** (*KeyExpr | _KeyExpr | str*) – The key expression to subscribe to
> * **handler** (*IHandler[Sample, Any, Any] | IClosure[Sample, Any] | Tuple[IClosure, Any] | Tuple[Callable[[Sample], Any], Callable[[], None], Any] | Tuple[Callable[[Sample], Any], Callable[[], None]] | Callable[[Sample], Any]*) –
> * **reliability** (*Reliability | None*) – the reliability to use when routing the subscribed samples
>
> **Return type**
>   *Subscriber*
>
> **Examples**

Using a callback:

```
>>> import zenoh
>>> s = zenoh.open({})
>>> sub = s.declare_subscriber('key/expression', lambda sample:
...     print(f"Received '{sample.key_expr}': '{sample.payload.decode('utf-8')}'
↪")
```

Using a queue:

```
>>> import zenoh
>>> s = zenoh.open({})
>>> sub = s.declare_subscriber('key/expression', zenoh.Queue())
>>> for sample in sub.receiver:
>>>     print(f"{sample.key_expr}: {sample.payload.decode('utf-8')}")
```

IMPORTANT: due to how RAII and Python work, you MUST bind this function's return value to a variable in order for it to function as expected. This is because as soon as a value is no longer referenced in Python, that value's destructor will run, which will undeclare your subscriber, deactivating the subscription immediately.

delete(*keyexpr*, *priority=None*, *congestion_control=None*)

Deletes the values associated with the keys included in `keyexpr`.

This uses the same mechanisms as `session.put`, and will be received by subscribers. This operation is especially useful with storages.

> **Parameters**
>
> - **keyexpr** (`KeyExpr` | `_KeyExpr` | `str`) – The key expression to publish
> - **priority** (`Priority` | `None`) – The priority to use when routing the delete
> - **congestion_control** (`CongestionControl` | `None`) – The congestion control to use when routing the delete
>
> **Examples**

```
>>> import zenoh
>>> s = zenoh.open({})
>>> s.delete('key/expression')
```

get(*selector*, *handler*, *consolidation=None*, *target=None*, *value=None*)

Emits a query, which queryables with intersecting selectors will be able to reply to.

The replies are provided to the given *handler* as instances of the *Reply* class. The *handler* can typically be a queue, a single callback or a pair of callbacks. The *handler*'s receiver is returned by the *get* function.

> **Parameters**
>
> - **selector** (`Selector` | `_Selector` | `KeyExpr` | `_KeyExpr` | `str`) – The selection of keys to query
> - **handler** (`IHandler[Reply, Any, Receiver]` | `IClosure[Reply, Any]` | `Tuple[IClosure, Receiver]` | `Tuple[Callable[[Reply], Any]`, `Callable[[], None], Receiver]` | `Tuple[Callable[[Reply], Any]`, `Callable[[], None]]` | `Callable[[Reply], Any]`) –
> - **consolidation** (`QueryConsolidation` | `None`) – The consolidation to apply to replies
> - **target** (`QueryTarget` | `None`) – The queryables that should be target to this query
> - **value** (`IValue` | `bytes` | `str` | `int` | `float` | `object` | `None`) – An optional value to attach to this query
>
> **Returns**
> > The receiver of the handler
>
> **Return type**
> > Receiver
>
> **Examples**

Using a queue:

```
>>> import zenoh
>>> s = zenoh.open({})
>>> for reply in s.get('key/expression', zenoh.Queue()):
...     try:
...         print(f"Received '{reply.ok.key_expr}': '{reply.ok.payload.decode(
→'utf-8')}'")
...     except:
...         print(f"Received ERROR: '{reply.err.payload.decode('utf-8')}'")
```

Using a single callback:

```
>>> s.get('key/expression', lambda reply:
...     print(f"Received '{reply.ok.key_expr}': '{reply.ok.payload.decode('utf-8
→')}'")
...     if reply.ok is not None else print(f"Received ERROR: '{reply.err.
→payload.decode('utf-8')}'"))
```

Using a reply callback and a termination callback:

```
>>> s.get('key/expression', (
...     lambda reply:
...         print(f"Received '{reply.ok.key_expr}': '{reply.ok.payload.decode(
→'utf-8')}'")
...         if reply.ok is not None else print(f"Received ERROR: '{reply.err.
→payload.decode('utf-8')}'"),
...     lambda:
...         print("No more replies")))
```

info()

> Returns an accessor for informations about this Session

put(*keyexpr*, *value*, *encoding=None*, *priority=None*, *congestion_control=None*, *sample_kind=None*)

> Sends a value over Zenoh.
>
> Subscribers on an expression that intersect with keyexpr will receive the sample. Storages will store the value if keyexpr is non-wild, or update the values for all known keys that are included in keyexpr if it is wild.
>
> > **Parameters**
> >
> > - **keyexpr** (KeyExpr | _KeyExpr | str) – The key expression to publish
> > - **value** (IValue | bytes | str | int | float | object) – The value to send
> > - **priority** (Priority | None) – The priority to use when routing the published data
> > - **congestion_control** (CongestionControl | None) – The congestion control to use when routing the published data
> > - **sample_kind** (SampleKind | None) – The kind of sample to send
> >
> > **Examples**

```
>>> import zenoh
>>> s = zenoh.open({})
>>> s.put('key/expression', 'value')
```

Info

class zenoh.**Info**(*session*)

> **Parameters**
> > **session**(*_Session*) –

peers_zid()

> Returns the neighbooring peers' identifiers
>
> > **Return type**
> > > *List*[ZenohId]

routers_zid()

> Returns the neighbooring routers' identifiers
>
> > **Return type**
> > > *List*[ZenohId]

zid()

> Returns this Zenoh Session's identifier
>
> > **Return type**
> > > ZenohId

KeyExpr

class zenoh.**KeyExpr**(*expr*)

> Zenoh's address space is designed around keys which serve as the names of ressources.
>
> Keys are slash-separated lists of non-empty UTF8 strings. They may not contain the following characters: $*#?.
>
> Zenoh's operations are executed on key expressions, a small language that allows the definition of sets of keys via the use of wildcards:
>
> - * is the single-chunk wildcard, and will match any chunk: a/*/c will match a/b/c, a/hello/c, etc…
>
> - ** is the 0 or more chunks wildcard: a/**/c matches a/c, a/b/c, a/b/hello/c, etc…
>
> - $* is the subchunk wildcard, it will match any amount of non-/ characters: a/b$* matches a/b, a/because, a/blue… but not a/c nor a/blue/c
>
> To allow for better performance and gain the property that two key expressions define the same set if and only if they are the same string, the rules of canon form are mandatory for a key expression to be propagated by a Zenoh network:
>
> - **/** may not exist, as it could always be replaced by the shorter **,
>
> - **/* may not exist, and must be written as its equivalent */** instead,
>
> - $* may not exist alone in a chunk, as it must be written * instead.
>
> The KeyExpr.autocanonize constructor exists to correct eventual infrigements of the canonization rules.
>
> A KeyExpr is a string that has been validated to be a valid Key Expression.
>
> > **Parameters**
> > > **expr** (KeyExpr | _KeyExpr | str) –

static **autocanonize**(*expr*)

This alternative constructor for key expressions will attempt to canonize the passed expression before checking if it is valid.

Raises a zenoh.ZError exception if `expr` is not a valid key expression.

> **Parameters**
> **expr** (`str`) –
>
> **Return type**
> KeyExpr

includes(*other*)

This method returns `True` if all of the keys defined by `other` also belong to the set defined by `self`.

> **Parameters**
> **other** (KeyExpr *|* _KeyExpr *|* str) –
>
> **Return type**
> bool

intersects(*other*)

This method returns `True` if there exists at least one key that belongs to both sets defined by `self` and `other`.

> **Parameters**
> **other** (KeyExpr *|* _KeyExpr *|* str) –
>
> **Return type**
> bool

undeclare(*session*)

Undeclares a key expression previously declared on the session.

> **Parameters**
> **session** (Session) –

Sample

class zenoh.**Sample**(*key*, *value*, *kind=None*, *timestamp=None*)

A KeyExpr-Value pair, annotated with the kind (PUT or DELETE) of publication used to emit it and a timestamp.

> **Parameters**
>
> - **key** (KeyExpr *|* _KeyExpr *|* str) –
>
> - **value** (IValue *|* bytes *|* str *|* int *|* float *|* object) –
>
> - **kind** (SampleKind) –
>
> - **timestamp** (Timestamp) –

property **encoding**: *Encoding*

A shortcut to `self.value.encoding`

property **key_expr**: *KeyExpr*

The sample's key expression

property **kind**: *SampleKind*

The sample's kind

---

property payload: bytes

> A shortcut to `self.value.payload`

property timestamp: *Timestamp* **| None**

> The sample's timestamp. May be None.

property value: *Value*

> The sample's value

SampleKind

class zenoh.**SampleKind**(*inner*)

> Similar to an HTTP METHOD: only PUT and DELETE are currently supported.
>
> > **Parameters**
> > **inner** (*_SampleKind*) –

static DELETE()

> > **Return type**
> > SampleKind

static PUT()

> > **Return type**
> > SampleKind

Value

class zenoh.**Value**(*payload*, *encoding=None*)

> A Value is a pair of a binary payload, and a mime-type-like encoding string.
>
> When constructed with `encoding==None`, the encoding will be selected depending on the payload's type.
>
> > **Parameters**
> >
> > - **payload** (*IValue | bytes | str | int | float | object*) –
> > - **encoding** (*Encoding*) –

static autoencode(*value*)

> Automatically encodes the value based on its type
>
> > **Parameters**
> > **value** (*IValue | bytes | str | int | float | object*) –
> >
> > **Return type**
> > Value

Encoding

class zenoh.**Encoding**(*inner*)

> **Parameters**
> **inner** (*_Encoding*) –

> **static APP_CUSTOM()**
>
> > **Return type**
> > Encoding

> **static APP_FLOAT()**
>
> > **Return type**
> > Encoding

> **static APP_INTEGER()**
>
> > **Return type**
> > Encoding

> **static APP_JSON()**
>
> > **Return type**
> > Encoding

> **static APP_OCTET_STREAM()**
>
> > **Return type**
> > Encoding

> **static APP_PROPERTIES()**
>
> > **Return type**
> > Encoding

> **static APP_SQL()**
>
> > **Return type**
> > Encoding

> **static APP_XHTML_XML()**
>
> > **Return type**
> > Encoding

> **static APP_XML()**
>
> > **Return type**
> > Encoding

> **static APP_X_WWW_FORM_URLENCODED()**
>
> > **Return type**
> > Encoding

> **static EMPTY()**
>
> > **Return type**
> > Encoding

static IMAGE_GIF()

> **Return type**
>> [Encoding](#)

static IMAGE_JPEG()

> **Return type**
>> [Encoding](#)

static IMAGE_PNG()

> **Return type**
>> [Encoding](#)

static TEXT_CSS()

> **Return type**
>> [Encoding](#)

static TEXT_CSV()

> **Return type**
>> [Encoding](#)

static TEXT_HTML()

> **Return type**
>> [Encoding](#)

static TEXT_JAVASCRIPT()

> **Return type**
>> [Encoding](#)

static TEXT_JSON()

> **Return type**
>> [Encoding](#)

static TEXT_PLAIN()

> **Return type**
>> [Encoding](#)

static TEXT_XML()

> **Return type**
>> [Encoding](#)

append(*s*)

> **Parameters**
>> **s** (*str*) –

static from_str(*s*)

> **Parameters**
>> **s** (*str*) –
>
> **Return type**
>> [Encoding](#)

Publisher

class zenoh.**Publisher**(*p*)

Use `Publisher` (constructed with `Session.declare_publisher`) when you want to send values often for the same key expression, as declaring them informs Zenoh that this is you intent, and optimizations will be set up to do so.

> **Parameters**
>
> > **p** (*_Publisher*) –

delete()

> An optimised version of `session.delete(self.key_expr)`

property key_expr: *KeyExpr*

> This `Publisher`'s key expression

put(*value*, *encoding=None*)

> An optimised version of `session.put(self.key_expr, value, encoding=encoding)`
>
> > **Parameters**
> >
> > - **value** (*IValue | bytes | str | int | float | object*) –
> > - **encoding** (*Encoding | None*) –

undeclare()

> Stops the publisher.

CongestionControl

class zenoh.**CongestionControl**(*inner*)

Defines the network's behaviour regarding a message when heavily congested.

> **Parameters**
>
> > **inner** (*_CongestionControl*) –

static BLOCK()

> Prevents the message from being dropped at all cost. In the face of heavy congestion on a part of the network, this could result in your publisher node blocking.
>
> > **Return type**
> >
> > > CongestionControl

static DROP()

> Allows the message to be dropped if all buffers are full.
>
> > **Return type**
> >
> > > CongestionControl

Priority

class zenoh.**Priority**(*inner*)

>   The priority of a sending operation.
>
>   They are ordered à la Linux priority: `Priority.REAL_TIME()` < `Priority.INTERACTIVE_HIGH()` < `Priority.INTERACTIVE_LOW()` < `Priority.DATA()` < `Priority.BACKGROUND()`
>
>   > **Parameters**
>   >   **inner** (*_SampleKind*) –
>
>   **static BACKGROUND**()
>
>   >   > **Return type**
>   >   >   Priority
>
>   **static DATA**()
>
>   >   > **Return type**
>   >   >   Priority
>
>   **static DATA_HIGH**()
>
>   >   > **Return type**
>   >   >   Priority
>
>   **static DATA_LOW**()
>
>   >   > **Return type**
>   >   >   Priority
>
>   **static INTERACTIVE_HIGH**()
>
>   >   > **Return type**
>   >   >   Priority
>
>   **static INTERACTIVE_LOW**()
>
>   >   > **Return type**
>   >   >   Priority
>
>   **static REAL_TIME**()
>
>   >   > **Return type**
>   >   >   Priority

Subscriber

class zenoh.**Subscriber**(*s*, *receiver=None*)

>   A handle to a subscription.
>
>   Its main purpose is to keep the subscription active as long as it exists.
>
>   When constructed through `Session.declare_subscriber(session, keyexpr, handler)`, it exposes handler's receiver through `self.receiver`.
>
>   > **Parameters**
>   >   **s** (*_Subscriber*) –
>
>   **undeclare**()
>
>   >   Undeclares the subscription

PullSubscriber

class zenoh.**PullSubscriber**(*s*, *receiver=None*)

A handle to a pull subscription.

Its main purpose is to keep the subscription active as long as it exists.

When constructed through `Session.declare_pull_subscriber(session, keyexpr, handler)`, it exposes `handler`'s receiver through `self.receiver`.

Calling `self.pull()` will prompt the Zenoh network to send a new sample when available.

> **Parameters**
> **s** (*_PullSubscriber*) –

pull()

Prompts the Zenoh network to send a new sample if available. Note that this sample will not be returned by this function, but provided to the handler's callback.

undeclare()

Undeclares the subscription

Reliability

class zenoh.**Reliability**(*inner*)

Used by subscribers to inform the network of the reliability it wishes to obtain.

> **Parameters**
> **inner** (*_Reliability*) –

static **BEST_EFFORT**()

Informs the network that dropping some messages is acceptable

> **Return type**
> CongestionControl

static **RELIABLE**()

Informs the network that this subscriber wishes for all publications to reliably reach it.

Note that if a publisher puts a sample with the `CongestionControl.DROP()` option, this reliability requirement may be infringed to prevent slow readers from blocking the network.

> **Return type**
> CongestionControl

Query

class zenoh.**Query**(*inner*)

> **Parameters**
> **inner** (*_Query*) –

decode_parameters()

Decodes the value selector into a dictionary.

Raises a ZError if duplicate keys are found, as they might otherwise be used for HTTP Parameter Pollution like attacks.

> **Return type**
>> *Dict*[str, str]

property key_expr: *KeyExpr*

> The query's targeted key expression

property parameters: str

> The query's value selector. If you'd rather not bother with parsing it yourself, use `self.decode_parameters()` instead.

reply(*sample*)

> Allows you to reply to a query. You may send any amount of replies to a single query, including 0.
>
>> **Parameters**
>>> **sample** (Sample) –

reply_err(*value*)

> Allows you to reply to a query with an error. You may send any amount of replies to a single query, including 0. Sending error responses does not exclude sending other responses.
>
>> **Parameters**
>>> **value** (IValue | *bytes* | *str* | *int* | *float* | *object*) –

property selector: *Selector*

> The query's selector as a whole.

property value: *Value* | None

> The query's value.

Selector

class zenoh.**Selector**(*selector*)

> A selector is the combination of a [Key Expression](crate::prelude::KeyExpr), which defines the set of keys that are relevant to an operation, and a `parameters`, a set of key-value pairs with a few uses:
>
> - specifying arguments to a queryable, allowing the passing of Remote Procedure Call parameters
>
> - filtering by value,
>
> - filtering by metadata, such as the timestamp of a value,
>
> When in string form, selectors look a lot like a URI, with similar semantics:
>
> - the `key_expr` before the first ? must be a valid key expression.
>
> - the `parameters` after the first ? should be encoded like the query section of a URL:
>
>   - key-value pairs are separated by &,
>
>   - the key and value are separated by the first =,
>
>   - in the absence of =, the value is considered to be the empty string,
>
>   - both key and value should use percent-encoding to escape characters,
>
>   - defining a value for the same key twice is considered undefined behavior.
>
> Zenoh intends to standardize the usage of a set of keys. To avoid conflicting with RPC parameters, the Zenoh team has settled on reserving the set of keys that start with non-alphanumeric characters.
>
> This document will summarize the standardized keys for which Zenoh provides helpers to facilitate coherent behavior for some operations.

Queryable implementers are encouraged to prefer these standardized keys when implementing their associated features, and to prefix their own keys to avoid having conflicting keys with other queryables.

Here are the currently standardized keys for Zenoh:

- `_time`: used to express interest in only values dated within a certain time range, values for this key must be readable by the [Zenoh Time DSL](zenoh_util::time_range::TimeRange) for the value to be considered valid.

- `_filter`: *TBD* Zenoh intends to provide helper tools to allow the value associated with this key to be treated as a predicate that the value should fulfill before being returned. A DSL will be designed by the Zenoh team to express these predicates.

    Parameters
        selector (`Selector` | `_Selector` | `KeyExpr` | `_KeyExpr` | `str`) –

decode_parameters()

Decodes the value selector part of the selector.

Raises a ZError if some keys were duplicated: duplicated keys are considered undefined behaviour, but we encourage you to refuse to process incoming messages with duplicated keys, as they might be attempting to use HTTP Parameter Pollution like exploits.

    Return type
        Dict[str, str]

property key_expr: *KeyExpr*

The key expression part of the selector.

property parameters

The value selector part of the selector.

property set_parameters

The value selector part of the selector.

QueryTarget

class zenoh.**QueryTarget**(*inner*)

    Parameters
        inner (*_QueryTarget*) –

QueryConsolidation

class zenoh.**QueryConsolidation**(*inner*)

    Parameters
        inner (*_QueryConsolidation*) –

Reply

class zenoh.**Reply**(*inner*)

A reply to a query (`Session.get`).

A single query can result in multiple replies from multiple queryables.

> **Parameters**
> **inner** (*_Reply*) –

property err: *Value*

The reply's error value.

Raises a ZError if the `self` is actually an `ok` reply.

property is_ok: **bool**

Checks if the reply is *ok*.

Returns *True* if the reply is *ok*, *False* otherwise

property ok: *Sample*

The reply's inner data sample.

Raises a ZError if the `self` is actually an `err` reply.

property replier_id: *ZenohId*

The reply's sender's id.

Queryable

class zenoh.**Queryable**(*inner*, *receiver*)

A handle to a queryable.

Its main purpose is to keep the queryable active as long as it exists.

When constructed through `Session.declare_queryable(session, keyexpr, handler)`, it exposes handler's receiver through `self.receiver`.

> **Parameters**
> **inner** (*_Queryable*) –

undeclare()

Stops the queryable.

ZenohId

class zenoh.**ZenohId**

A Zenoh UUID

Timestamp

class zenoh.**Timestamp**

A timestamp taken from the Zenoh HLC (Hybrid Logical Clock).

These timestamps are guaranteed to be unique, as each machine annotates its perceived time with a UUID, which is used as the least significant part of the comparison operation.

property get_time: int

Returns the time part, as generated by the Zenoh HLC in NTP64 format (See https://datatracker.ietf.org/doc/html/rfc5905#section-6).

property seconds_since_unix_epoch: float

Returns the number of seconds since the Unix Epoch.

Considering the large number of seconds since the Unix Epoch, the precision of the resulting f64 is in the order of microseconds. Therefore, it should not be used for comparison. Directly comparing Timestamp objects is preferable.

class zenoh.**Closure**(*closure*, *type_adaptor=None*, *prevent_direct_calls=False*)

A Closure is a pair of a `call` function that will be used as a callback, and a `drop` function that will be called when the closure is destroyed.

property call: Callable[[In], Out]

Returns the closure's call function as a lambda.

property drop: Callable[[], None]

Returns the closure's destructor as a lambda.

class zenoh.**Handler**(*input*, *type_adaptor=None*, *prevent_direct_calls=True*)

A Handler is a value that may be converted into a callback closure for zenoh to use on one side, while possibly providing a receiver for the data that zenoh would provide through that callback.

Note that the values will be piped onto a `Queue` before being sent to your handler by another Thread unless either:

> a) `input` is already an instance of `Closure` or `Handler` where `input.closure` is an instance of `Closure`
>
> b) `prevent_direct_calls` is set to `False`

property closure: *IClosure*[In, Out]

The part of the handler that should be passed as a callback to a zenoh function.

property receiver: Receiver

The part of the handler that should be used as the receiver when the handler is channel-like.

class zenoh.**IClosure**(**args*, ***kwds*)

A Closure is a pair of a `call` function that will be used as a callback, and a `drop` function that will be called when the closure is destroyed.

abstract property call: Callable[[In], Out]

Returns the closure's call function as a lambda.

abstract property drop: Callable[[], None]

Returns the closure's destructor as a lambda.

class zenoh.**IHandler**(**args*, ***kwds*)

A Handler is a value that may be converted into a callback closure for zenoh to use on one side, while possibly providing a receiver for the data that zenoh would provide through that callback.

abstract property closure: *IClosure*[In, Out]

> The part of the handler that should be passed as a callback to a zenoh function.

abstract property receiver: Receiver

> The part of the handler that should be used as the receiver when the handler is channel-like.

class zenoh.**IValue**

The IValue interface exposes how to recover a value's payload in a binary-serialized format, as well as that format's encoding.

abstract property encoding: *Encoding*

> The value's encoding

abstract property payload: bytes

> The value itself, as an array of bytes

class zenoh.**ListCollector**(*timeout=None*)

A simple collector that aggregates values into a list.

When used as a handler, it provides a callback that appends elements to a list, and provides a function that will await the closing of the callback before returning said list.

property closure

> The part of the handler that should be passed as a callback to a zenoh function.

property receiver

> The part of the handler that should be used as the receiver when the handler is channel-like.

class zenoh.**Queue**(*bound=None*)

A binding for a Rust multi-producer, single-consumer queue implementation.

When used as a handler, it provides itself as the receiver, and will provide a callback that appends elements to the queue.

Can be bounded by passing a maximum size as bound.

property closure: *IClosure*[In, None]

> The part of the handler that should be passed as a callback to a zenoh function.

get(*timeout=None*)

> Gets one element from the queue.
>
> Raises a StopIteration exception if the queue was closed before the timeout ran out, this allows using the Queue as an iterator in for-loops. Raises a TimeoutError if the timeout ran out.
>
> > **Parameters**
> > > **timeout** (*float | None*) –

get_remaining(*timeout=None*)

> Awaits the closing of the queue, returning the remaining queued values in a list. The values inserted into the queue up until this happens will be available through get.
>
> Raises a TimeoutError if the timeout in seconds provided was exceeded before closing, whose args[0] will contain the elements that were collected before timing out.
>
> > **Parameters**
> > > **timeout** (*float | None*) –
> >
> > **Return type**
> > > *List*[*In*]

put(*value*)

> Puts one element on the queue.
>
> Raises a `PyBrokenPipeError` if the Queue has been closed.

property receiver: *Queue*

> The part of the handler that should be used as the receiver when the handler is channel-like.

PYTHON MODULE INDEX

Z

Z

zenoh
    module, 4, 22
ZenohId (*class in zenoh*), 21
zid (*zenoh.Hello property*), 5
zid() (*zenoh.Info method*), 11