zenoh-python

Release 1.9.0

ZettaScale Zenoh team, <zenoh@zettascale.tech>

May 26, 2026

CONTENTS:

Zenoh /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.

The Zenoh protocol allows nodes to form a graph with an arbitrary topology, such as a mesh, a star, or a clique. The zenoh routers keeps the network connected and routes the messages between the nodes.

This documentation provides an overview of the Zenoh concepts and components and a reference of the Zenoh python API. For more information about Zenoh, please visit the documentation section on the Zenoh website. It's useful to consult also the Zenoh Rust API reference since the Python API is a binding over the Rust implementation.

All examples presented in this documentation can be found in the examples/ directory of the Zenoh Python GitHub repository.

CHAPTER

ONE

DOCUMENTATION CONTENTS

1.1 Quick Start Examples

Below are some examples that highlight these key concepts and show how easy it is to get started with. The more detailed documentation is available in the other sections.

1.1.1 Publish a key/value pair onto Zenoh

```
# Publish a key/value pair onto Zenoh
with zenoh.open(zenoh.Config()) as session:
    session.put("demo/example/hello", "Hello World!")
```

1.1.2 Subscribe to a set of keys with Zenoh

```
# Subscribe to a set of keys with Zenoh
with zenoh.open(zenoh.Config()) as session:
    with session.declare_subscriber("demo/example/**") as subscriber:
        for sample in subscriber:
            print(f"{sample.key_expr} => {sample.payload.to_string()}")
```

1.1.3 Get keys/values from zenoh

```
# Get keys/values from zenoh
with zenoh.open(zenoh.Config()) as session:
    for response in session.get("demo/example/**"):
        response = response.ok
        print(f"{response.key_expr} => {response.payload.to_string()}")
```

1.2 Components and Concepts

1.2.1 Session and Config

Zenoh supports two paradigms of communication: *Publish/Subscribe* and *Query/Reply*. The entities that perform communication (for example, publishers, subscribers, queriers, and queryables) are declared through a *zenoh.Session*. A session is created by the *zenoh.open()* function, which takes a *zenoh.Config* as an argument.

The configuration is stored in a JSON file and can be read with *zenoh.Config.from_file()*. The file format is documented in the Zenoh Rust API Config reference.

Important: The recommended way to create a session is using a context manager (`with` statement). If a session is not explicitly closed or managed with a context manager, on exit object finalizers may be called when the library thread has already been killed, which can cause the script to hang.

Either use a context manager (recommended) or explicitly call *zenoh.Session.close()* before your script exits. See examples in the *Quick Start Examples* section.

Example: Creating a session with context manager

```
# Recommended: Using context manager
# The session is automatically closed when exiting the 'with' block
with zenoh.open(zenoh.Config()) as session:
    # Use the session
    session.put("demo/example/hello", "Hello World!")
```

Example: Creating a session with explicit close

```
# Alternative: Explicit open and close
# You must explicitly close the session before script exit
session = zenoh.open(zenoh.Config())
try:
    # Use the session
    session.put("demo/example/hello", "Hello World!")
finally:
    # Always close the session
    session.close()
```

1.2.2 Key Expressions

Key expressions are Zenoh's address space.

In Zenoh, data is associated with keys in the form of a slash-separated path, e.g., `robot/sensor/temp`. The requesting side uses key expressions to address the data of interest. Key expressions can contain wildcards:

- `*` matches any chunk (a chunk is a sequence of characters between `/` separators)
- `**` matches any number of chunks (including zero chunks)

For example:

- `robot/sensor/*` matches `robot/sensor/temp`, `robot/sensor/humidity`, etc.
- `robot/**` matches `robot/sensor/temp`, `robot/actuator/motor`, `robot/status`, etc.

The *zenoh.KeyExpr* class provides validation and operations on key expressions. The *zenoh.KeyExpr* constructor validates the syntax of the provided string and raises a *zenoh.ZError* exception if the syntax is invalid (e.g., it contains spaces, other illegal characters, or has empty chunks like `foo//bar` or `/foo`).

The *zenoh.KeyExpr* constructor raises an exception for key expressions that are valid but not in canonical form. For example, `robot/sensor/**/*` is valid, but its canonical form is `robot/sensor/*/**`. The *zenoh.KeyExpr.autocanonize()* method can accept such key expressions and convert them to their canonical form.

Example: Validating key expressions

```
try:
    # Valid key expressions
    valid_ke = KeyExpr("robot/sensor/temperature")
    assert str(valid_ke) == "robot/sensor/temperature"
    canonized_ke = KeyExpr.autocanonize("robot/sensor/**/*/**/**")
    assert str(canonized_ke) == "robot/sensor/*/**"

    # Invalid key expression (empty segment)
    invalid_ke = KeyExpr("robot/sensor//*")
    assert True, "This line should not be reached"
except zenoh.ZError as e:
    print(f"Validation error: {e}")
```

Key expressions support operations such as intersection and inclusion (see *zenoh.KeyExpr.intersects()* and *zenoh.KeyExpr.includes()*), which help determine how different expressions relate to each other.

Example: Performing operations on key expressions

```
# Create a key expression with validation
sensor_ke = KeyExpr("robot/sensor")
assert str(sensor_ke) == "robot/sensor"

# Join with another segment
temp_ke = sensor_ke.join("temp")
assert str(temp_ke) == "robot/sensor/temp"

# Create a wildcard expression
all_sensors = sensor_ke.join("**")
assert str(all_sensors) == "robot/sensor/**"

# Check inclusion
assert all_sensors.includes(temp_ke)
assert not temp_ke.includes(all_sensors)

# Check intersection
assert all_sensors.intersects(temp_ke)
assert not sensor_ke.intersects(KeyExpr("robot/actuator"))
```

Key expressions can also be declared with the session to optimize routing and network usage:

Example: Declaring key expressions

```
# Declare a key expression for optimized routing
declared_ke = session.declare_keyexpr("robot/sensor/temperature")

# Use the declared key expression
publisher = session.declare_publisher(declared_ke)
```

1.2.3 Publish/Subscribe

Data is published via a *zenoh.Publisher*, which is declared using *zenoh.Session.declare_publisher()*. The publisher exposes two primary operations: *zenoh.Publisher.put()* and *zenoh.Publisher.delete()*. Publishing can also be performed directly from the session via *zenoh.Session.put()* and *zenoh.Session.delete()*.

Published data is received as *zenoh.Sample* instances by a *zenoh.Subscriber*, which is declared using *zenoh.Session.declare_subscriber()*. The samples are delivered to the callback or channel (*Channels and callbacks*).

Publishing can express two different semantics:

- producing a sequence of values
- updating a single value associated with a key expression

In the second case, it is necessary to indicate that a key is no longer associated with any value; the *zenoh.Publisher.delete()* operation is used for this.

On the receiving side, the subscriber distinguishes between *zenoh.SampleKind.PUT* and *zenoh.SampleKind.DELETE* using the *zenoh.Sample.kind* field in the *zenoh.Sample* structure.

The delete operation allows a subscriber to work with a *zenoh.Queryable* that caches the values associated with key expressions.

Example: Declaring a publisher and publishing data

```
# Declare a publisher and publish data
publisher = session.declare_publisher("key/expression")
publisher.put("value")
```

Example: Declaring a subscriber and receiving data

```
# Declare a subscriber and receive data
subscriber = session.declare_subscriber("key/expression")
for sample in subscriber:
    print(f">> Received {sample.payload.to_string()}")
```

Example: Using session methods directly

```
# Direct put operation
session.put("key/expression", "value")

# Direct delete operation
session.delete("key/expression")
```

1.2.4 Query/Reply

In the query/reply paradigm, data is made available by a *zenoh.Queryable* and requested by a *zenoh.Querier* or directly via *zenoh.Session.get()*.

A *zenoh.Queryable* is declared using *zenoh.Session.declare_queryable()*. It serves *zenoh.Query* requests via a callback or channel (*Channels and callbacks*).

The *zenoh.Query* provides the *zenoh.Query.reply()* method to reply with a data sample of the *zenoh.SampleKind.PUT* kind, and *zenoh.Query.reply_del()* to send a *zenoh.SampleKind.DELETE* reply. See *Publish/Subscribe* for more details on the difference between the two sample kinds. There is also the *zenoh.Query.reply_err()* method which can be used to send a reply containing error information.

Data is requested from queryables via *zenoh.Session.get()* or via a *zenoh.Querier* object. Each request returns zero or more *zenoh.Reply* structures — one per queryable that matches the request. Each reply contains either a *zenoh.Sample* from *reply* and *reply_del* or a *zenoh.ReplyError* from *reply_err*.

Example: Declaring a queryable

```
# Queryable that replies with temperature data for a given day
queryable = session.declare_queryable("room/temperature/history")
query_count = 0
for query in queryable:
    if "day" in query.selector.parameters:
        day = query.selector.parameters["day"]
        if day in temperature_data:
            query.reply("room/temperature/history", temperature_data[day])
        else:
            query.reply_del("room/temperature/history")
    else:
        query.reply_err("missing day parameter")
```

Example: Requesting data using Session.get

```
# Request temperature for a specific day
replies = session.get("room/temperature/history?day=2023-03-15")
for reply in replies:
    if reply.ok:
        print(f">> Temperature is {reply.ok.payload.to_string()}")
    else:
        print(f">> Error: {reply.err.payload.to_string()}")
```

Example: Using a Querier

```
# Declare a querier for multiple queries
querier = session.declare_querier("room/temperature/history")

# Send a query with parameters
replies = querier.get(parameters="?day=2023-03-15")
for reply in replies:
    if reply.ok:
        print(f">> Temperature is {reply.ok.payload.to_string()}")
    else:
        print(f">> Error: {reply.err.payload.to_string()}")
```

1.2.5 Query Parameters

The query/reply API allows specifying additional parameters for the request. A *zenoh.Selector* object is passed to the *zenoh.Session.get()* operation. It combines a key expression with optional parameters and can be constructed from these elements or by parsing a selector string. The selector string has a syntax similar to a URL: it is a key expression followed by a question mark and a list of parameters in the format "name=value", separated by `;`. For example: `key/expression?param1=value1;param2=value2`.

Alternatively, parameters can be constructed programmatically using the *zenoh.Parameters* class, which accepts a dictionary, and then combined with a key expression to create a *zenoh.Selector*.

On the receiving side, queryables can access these parameters via *zenoh.Query.parameters*.

Example: Constructing a Selector from dictionary

```
# Create parameters from a dictionary
params = zenoh.Parameters({"day": "2023-03-15", "format": "celsius"})

# Create a selector from key expression and parameters
selector = zenoh.Selector("room/temperature/history", params)

# Request data using the selector
replies = session.get(selector)
for reply in replies:
    if reply.ok:
        print(f">> {reply.ok.payload.to_string()}")
```

1.2.6 Data representation

Data is received as *zenoh.Sample* objects, which contain the *zenoh.Sample.payload* and associated metadata like *zenoh.Sample.timestamp*, *zenoh.Sample.encoding*, and *zenoh.Sample.kind*. Additionally, optional user-defined metadata can be attached via *zenoh.Sample.attachment*.

Both *zenoh.Sample.payload* and *zenoh.Sample.attachment* are of type *zenoh.ZBytes*, which represents raw byte data.

Example: Using `zenoh.ZBytes`

```
# Raw bytes
payload = zenoh.ZBytes(b"Hello, World!")
data = payload.to_bytes()
assert isinstance(data, bytes)
assert data == b"Hello, World!"

# String data
payload = zenoh.ZBytes("Hello, World!")
text = payload.to_string()
assert isinstance(text, str)
assert text == "Hello, World!"
```

Serialization and deserialization of basic types and structures is provided in the *zenoh.ext* module via *zenoh.ext.z_serialize()* and *zenoh.ext.z_deserialize()*.

Example: Data serialization

```
# Using zenoh.ext for serialization
from zenoh.ext import z_deserialize, z_serialize

# Serialize a dictionary
data = {"temperature": 25.5, "humidity": 60.0}
payload = z_serialize(data)
assert isinstance(payload, zenoh.ZBytes)

# Deserialize back
received = z_deserialize(dict[str, float], payload)
assert isinstance(received, dict)
assert received == {"temperature": 25.5, "humidity": 60.0}
```

1.2.7 Encoding

Zenoh uses *zenoh.Encoding* to indicate how data should be interpreted by the application. An encoding has a similar role to Content-Type in HTTP and is represented as a string in MIME-like format: `type/subtype[;schema]`.

To optimize network usage, Zenoh internally maps some predefined encoding strings to integer identifiers. These encodings are provided as class attributes of the *zenoh.Encoding* class, such as *zenoh.Encoding.ZENOH_BYTES*, *zenoh.Encoding.APPLICATION_JSON*, etc. This internal mapping is not exposed to the application layer, but using these predefined encodings is more efficient than custom strings.

The Zenoh protocol does not impose any encoding value and does not operate on it. It can be seen as optional metadata that is carried over by Zenoh, allowing applications to perform different operations depending on the encoding value.

Additionally, a schema can be associated with the encoding. The convention is to use the `;` separator if an encoding is created from a string. Alternatively, *zenoh.Encoding.with_schema()* can be used to add a schema to one of the predefined class attributes.

Example: Creating an `zenoh.Encoding` from a string and vice versa

```
encoding = zenoh.Encoding("text/plain")
text = str(encoding)
assert text == "text/plain"
```

Example: Using the schema

```
encoding1 = zenoh.Encoding("text/plain;utf-8")
encoding2 = zenoh.Encoding.TEXT_PLAIN.with_schema("utf-8")
assert encoding1 == encoding2
assert str(encoding1) == "text/plain;utf-8"
assert str(encoding2) == "text/plain;utf-8"
```

1.2.8 Scouting

Scouting is the process of discovering Zenoh nodes on the network. The scouting process depends on the transport layer and the Zenoh configuration. Note that it is not necessary to explicitly discover other nodes to publish, subscribe, or query data.

Scouting is performed using the *zenoh.scout()* function, which returns a *zenoh.Scout* object that yields *zenoh.Hello* messages for each discovered Zenoh node.

Scouting is different from *liveliness* requesting and monitoring. Liveliness works on the Zenoh protocol logical level and allows getting information about resources in terms of *key expressions*. On the other hand, *scouting* is about discovering Zenoh nodes visible to the local node on the network. The result of scouting is a list of *zenoh.Hello* messages, each containing information about a discovered Zenoh node:

- unique node identifier (*zenoh.Hello.zid*)
- node type (*zenoh.Hello.whatami*)
- list of node's network addresses (*zenoh.Hello.locators*)

See more details at scouting documentation.

Example: Scouting for Zenoh nodes

```
scout = zenoh.scout(what="peer|router")
threading.Timer(1.0, lambda: scout.stop()).start()
for hello in scout:
    print(hello)
```

1.2.9 Liveliness

Zenoh supports liveliness monitoring to notify when a specified resource appears or disappears on the network.

Sometimes it is necessary to know whether a Zenoh node is available. This can be achieved by declaring special publishers and queryables, but the dedicated liveliness API is more convenient and efficient.

The *zenoh.Liveliness* object is created by calling *zenoh.Session.liveliness()*. It allows a node to declare a *zenoh.LivelinessToken* associated with a key expression. To declare the token, use *zenoh.Liveliness.declare_token()*.

Other nodes can query this key expression using *zenoh.Liveliness.get()*. They can also subscribe using *zenoh.Liveliness.declare_subscriber()* to be notified when the token appears or disappears.

The *history* parameter of *zenoh.Liveliness.declare_subscriber()* allows immediate receipt of tokens that are already present on the network.

Example: Declaring a liveliness token

```
# Declare a liveliness token
token = session.liveliness().declare_token("node/A")
```

Example: Getting currently present liveliness tokens

```
# Get currently present liveliness tokens
replies = session.liveliness().get("node/A", timeout=5)
for reply in replies:
    if reply.ok:
        print(f"Alive token ('{reply.ok.key_expr}')")
    else:
        print(f"Received (ERROR: '{reply.err.payload.to_string()}')")
```

Example: Checking if a liveliness token is present and subscribing to changes

```
# Check if a liveliness token is present and subscribe to changes
subscriber = session.liveliness().declare_subscriber("node/A", history=True)
for sample in subscriber:
    if sample.kind == zenoh.SampleKind.PUT:
        print(f"Alive token ('{sample.key_expr}')")
    elif sample.kind == zenoh.SampleKind.DELETE:
        print(f"Dropped token ('{sample.key_expr}')")
```

1.2.10 Matching

The matching API lets the active side of communication (publisher or querier) learn whether there are interested parties on the other side (subscriber or queryable). This information can save bandwidth and CPU resources.

Declare a *zenoh.MatchingListener* via *zenoh.Publisher.declare_matching_listener()* or *zenoh.Querier.declare_matching_listener()*.

The matching listener behaves like a subscriber, but instead of producing data samples it yields *zenoh.MatchingStatus* instances whenever the matching status changes — for example, when the first matching subscriber or queryable appears or when the last one disappears.

Example: Declaring a matching listener for a publisher

```
# Declare a matching listener for a publisher
publisher = session.declare_publisher("key/expression")
listener = publisher.declare_matching_listener()
for status in listener:
    if status.matching:
        print(">> Publisher has at least one matching subscriber")
    else:
        print(">> Publisher has no matching subscribers")
```

Example: Declaring a matching listener for a querier

```
# Declare a matching listener for a querier
querier = session.declare_querier("service/endpoint")
listener = querier.declare_matching_listener()
for status in listener:
    if status.matching:
        print(">> Querier has at least one matching queryable")
    else:
        print(">> Querier has no matching queryables")
```

1.2.11 Channels and callbacks

There are two ways to receive sequential data from Zenoh primitives (for example, a series of *zenoh.Sample* objects from a *zenoh.Subscriber* or *zenoh.Reply* objects from a *zenoh.Query*): by channel or by callback.

This behavior is controlled by the `handler` parameter of the declare methods (for example, *zenoh.Session.declare_subscriber()* and *zenoh.Session.declare_querier()*). The parameter can be either a callable (a function or a method) or a channel type (blocking *zenoh.handlers.FifoChannel* or non-blocking *zenoh.handlers.RingChannel*). By default, the `handler` parameter is `None`, which uses *zenoh.handlers.DefaultHandler* (a FIFO channel with default capacity).

Channels

When constructed with a *zenoh.handlers.FifoChannel* or *zenoh.handlers.RingChannel* as `handler` (or using the default one), the returned object is iterable and can be used in a `for` loop to receive data sequentially. It also provides explicit methods such as *zenoh.Subscriber.recv()* to wait for data and *zenoh.Subscriber.try_recv()* to attempt a non-blocking receive. The subscriber (or queryable) is automatically undeclared when the object goes out of scope or when *zenoh.Subscriber.undeclare()* is explicitly called.

```
# Default channel
subscriber_default = session.declare_subscriber("key/expr")

# Explicit FIFO channel with custom capacity
subscriber_fifo = session.declare_subscriber(
    "key/expr", zenoh.handlers.FifoChannel(100)
)

# Ring channel (drops oldest when full)
subscriber_ring = session.declare_subscriber("key/expr", zenoh.handlers.RingChannel(50))
```

Callbacks

Caution: Calling Zenoh API functions, as well as performing any blocking operations from within a callback is disallowed. Even if this works in some particular cases, it's unsafe and may lead to deadlocks or crashes at any moment or with the future updates of the library.

It is possible to pass a callable object as `handler`. This callable is invoked for each received *zenoh.Sample* or *zenoh.Reply*. This also means the subscriber or queryable runs in **background mode**, i.e., it remains active even if the returned object goes out of scope. This allows declaring a subscriber without managing the returned object's lifetime.

```
def on_sample(sample):
    print(sample.payload.to_string())


# Subscriber runs in background mode
subscriber = session.declare_subscriber("key/expr", on_sample)
# The subscriber remains active even if 'subscriber' variable is not used
```

For more advanced callback handling, you can use *zenoh.handlers.Callback* to create a callback handler with cleanup functionality.

```
def on_sample(sample):
    print(sample.payload.to_string())


def on_cleanup():
    print("Subscriber undeclared")


callback = zenoh.handlers.Callback(on_sample, drop=on_cleanup)
```

(continues on next page)

(continued from previous page)

```
subscriber = session.declare_subscriber("key/expr", callback)
# The subscriber remains active even if 'subscriber' variable is not used
```

Custom channel implementation

For advanced use cases, you can implement your own custom channel in Python and pass it in the tuple form `(callback, handler)` where `callback` is a callable and `handler` is your custom Python object. This solution has the same performance penalties as the callback API, but it can be useful in some scenarios.

The callback is invoked for each received item and stores the data in the custom channel, which is accessible via the *zenoh.Subscriber.handler()* property, in the same way as with built-in channels.

Custom channel with priority queue

```
class PriorityChannel:
    def __init__(self, maxsize=100):
        self.queue: queue.PriorityQueue = queue.PriorityQueue(maxsize)
        # Counter to preserve FIFO order for samples with same priority
        self._counter = 0

    def recv(self) -> zenoh.Sample:
        return self.queue.get()[2]

    def send(self, sample: zenoh.Sample):
        self.queue.put((sample.priority, self._counter, sample))
        self._counter += 1
```

Usage of the custom channel

```
with zenoh.open(zenoh.Config()) as session:
    channel = PriorityChannel(maxsize=50)
    subscriber = session.declare_subscriber("key/expression", (channel.send, channel))
    sample = subscriber.handler.recv()
    print(f">> Received: {sample.payload.to_string()}")
```

1.3 API Reference

1.3.1 module zenoh

final exception zenoh.**ZError**

Exception raised for Zenoh-related errors.

This exception is raised by various Zenoh operations when they encounter errors, such as invalid *KeyExpr* instances or connection failures.

To handle ZError, wrap Zenoh operations in try-except blocks:

```
try:
    ke = KeyExpr("invalid/key")
```

(continues on next page)

(continued from previous page)

```
except ZError as e:
    print(f"Error: {e}")  # Get error message
```

The error message can be accessed via str(e) or by printing the exception directly.

final class zenoh.**CancellationToken**

Cancellation token that can be used for interrupting GET queries. .. warning:: This API has been marked as unstable: it works as advertised, but it may be changed in a future release.

Return type
Self

cancel()

Interrupt all associated GET queries. If the direct query callback is being executed, the call blocks until execution of callback finishes and its corresponding drop method returns (if any).

Once token is cancelled, all new associated GET queries will cancel automatically.

property is_cancelled: bool

Return true if token was cancelled, false otherwise.

final class zenoh.**Config**

The main configuration structure for Zenoh.

Zenoh configuration can be loaded from various sources:

- From a specific file path using *from_file()*.
- From a file specified by the ZENOH_CONFIG environment variable using *from_env()*.
- From a JSON5 string using *from_json5()*.

Configuration values can be retrieved as JSON using *get_json()* and modified using *insert_json5()*.

For detailed format information, see: https://docs.rs/zenoh/latest/zenoh/config/struct.Config.html

Return type
Self

classmethod from_env()

Load configuration from the file path specified in the ZENOH_CONFIG environment variable.

Return type
Self

classmethod from_file(*path*)

Load configuration from the file at path.

Parameters
path (*str | Path*) –

Return type
Self

classmethod from_json5(*json*)

Load configuration from the JSON5 string json.

Parameters
json (*str*) –

Return type
Self

get_json(*key*)

Returns a JSON string containing the configuration at key.

Parameters
key (*str*) –

Return type
Any

insert_json5(*key*, *value*)

Inserts configuration value value at key.

Parameters

- **key** (*str*) –
- **value** (*Any*) –

final enum zenoh.**CongestionControl**(*value*)

Congestion control strategy.

This parameter controls how messages are handled when a node's transmission queue is full. Set this when declaring publishers/queriers or in put, get, delete, and reply operations to specify congestion behavior. It can also be retrieved from *Sample* and *Publisher* objects.

See also:

- Parameters in: *Session.declare_publisher()*, *Session.declare_querier()*, *Session.put()*, *Session.delete()*, *Session.get()*, *Query.reply()*, *Query.reply_del()*
- Properties in: *Sample.congestion_control*, *Publisher.congestion_control*

Valid values are as follows:

DROP = <CongestionControl.DROP: 1>

When transmitting a message in a node with a full queue, the node may drop the message.

BLOCK = <CongestionControl.BLOCK: 2>

When transmitting a message in a node with a full queue, the node will wait for queue to progress.

BLOCK_FIRST = <CongestionControl.BLOCK_FIRST: 3>

When transmitting a message in a node with a full queue, the node will wait for queue to progress, but only for the first message sent with this strategy; other messages will be dropped.

final enum zenoh.**ConsolidationMode**(*value*)

The kind of consolidation to apply to a query.

Consolidation determines how multiple samples for the same key are handled during query operations, balancing between latency and bandwidth efficiency.

See also:

- Parameters in: *Session.get()*, *Session.declare_querier()*

Valid values are as follows:

AUTO = <ConsolidationMode.AUTO: 1>

Apply automatic consolidation based on queryable's preferences.

NONE = <ConsolidationMode.NONE: 2>

No consolidation applied: multiple samples may be received for the same key-timestamp.

MONOTONIC = <ConsolidationMode.MONOTONIC: 3>

Monotonic consolidation immediately forwards samples, except if one with an equal or more recent timestamp has already been sent with the same key.

This optimizes latency while potentially reducing bandwidth. Note that this doesn't cause re-ordering, but drops the samples for which a more recent timestamp has already been observed with the same key.

LATEST = <ConsolidationMode.LATEST: 4>

Holds back samples to only send the set of samples that had the highest timestamp for their key.

final class zenoh.**Encoding**(*encoding=None*)

Encoding information for Zenoh payloads.

Represents how data should be interpreted by the application, similar to HTTP Content-type. Encodings use MIME-like format: `type/subtype[;schema]`.

Predefined class attributes are provided for common encodings (e.g., *ZENOH_BYTES*, *APPLICATION_JSON*). Using these is more efficient than custom strings.

See also: *Encoding*

Parameters
encoding (*str | None*) –

Return type
Self

with_schema(*schema*)

Set a schema to this encoding. Zenoh does not define what a schema is and its semantics are left to the implementer. E.g. a common schema for text/plain encoding is utf-8.

Parameters
schema (*str*) –

Return type
Self

APPLICATION_CBOR: Self

A Concise Binary Object Representation (CBOR)-encoded data.

Constant alias for string: "application/cbor".

APPLICATION_CDR: Self

A Common Data Representation (CDR)-encoded data.

Constant alias for string: "application/cdr

APPLICATION_COAP_PAYLOAD: Self

Constrained Application Protocol (CoAP) data intended for CoAP-to-HTTP and HTTP-to-CoAP proxies.

Constant alias for string: "application/coap-payload".

APPLICATION_JAVA_SERIALIZED_OBJECT: Self

A Java serialized object.

Constant alias for string: "application/java-serialized-object".

APPLICATION_JSON: Self

JSON data intended to be consumed by an application.

Constant alias for string: "application/json".

APPLICATION_JSONPATH: Self

A JSONPath defines a string syntax for selecting and extracting JSON values from within a given JSON value.

Constant alias for string: "application/jsonpath".

APPLICATION_JSON_PATCH_JSON: Self

Defines a JSON document structure for expressing a sequence of operations to apply to a JSON document.

Constant alias for string: "application/json-patch+json".

APPLICATION_JSON_SEQ: Self

A JSON text sequence consists of any number of JSON texts, all encoded in UTF-8.

Constant alias for string: "application/json-seq".

APPLICATION_JWT: Self

A JSON Web Token (JWT).

Constant alias for string: "application/jwt".

APPLICATION_MP4: Self

An application-specific MPEG-4 encoded data, either audio or video.

Constant alias for string: "application/mp4".

APPLICATION_OCTET_STREAM: Self

An application-specific stream of bytes.

Constant alias for string: "application/octet-stream".

APPLICATION_OPENMETRICS_TEXT: Self

An openmetrics text data representation, commonly used by Prometheus.

Constant alias for string: "application/openmetrics-text".

APPLICATION_PROTOBUF: Self

An application-specific protobuf-encoded data.

Constant alias for string: "application/protobuf".

APPLICATION_PYTHON_SERIALIZED_OBJECT: Self

A Python object serialized using pickle.

Constant alias for string: "application/python-serialized-object".

APPLICATION_SOAP_XML: Self

A SOAP 1.2 message serialized as XML 1.0.

Constant alias for string: "application/soap+xml".

APPLICATION_SQL: Self

An application-specific SQL query.

Constant alias for string: "application/sql".

APPLICATION_XML: Self

An XML file intended to be consumed by an application.

Constant alias for string: "application/xml".

APPLICATION_X_WWW_FORM_URLENCODED: Self

An encoded a list of tuples, each consisting of a name and a value.

Constant alias for string: "application/x-www-form-urlencoded".

APPLICATION_YAML: Self

YAML data intended to be consumed by an application.

Constant alias for string: "application/yaml".

APPLICATION_YANG: Self

A YANG-encoded data commonly used by the Network Configuration Protocol (NETCONF).

Constant alias for string: "application/yang".

AUDIO_AAC: Self

A MPEG-4 Advanced Audio Coding (AAC) media.

Constant alias for string: "audio/aac".

AUDIO_FLAC: Self

A Free Lossless Audio Codec (FLAC) media.

Constant alias for string: "audio/flac".

AUDIO_MP4: Self

An audio codec defined in MPEG-1, MPEG-2, MPEG-4, or registered at the MP4 registration authority.

Constant alias for string: "audio/mp4".

AUDIO_OGG: Self

An Ogg-encapsulated audio stream.

Constant alias for string: "audio/ogg".

AUDIO_VORBIS: Self

A Vorbis-encoded audio stream.

Constant alias for string: "audio/vorbis".

IMAGE_BMP: Self

A BitMap (BMP) image.

Constant alias for string: "image/bmp".

IMAGE_GIF: Self

A Graphics Interchange Format (GIF) image.

Constant alias for string: "image/gif".

IMAGE_JPEG: Self

A Joint Photographic Experts Group (JPEG) image.

Constant alias for string: "image/jpeg".

IMAGE_PNG: Self

A Portable Network Graphics (PNG) image.

Constant alias for string: "image/png".

IMAGE_WEBP: Self

A Web Protable (WebP) image.

Constant alias for string: "image/webp

TEXT_CSS: Self

A CSS file.

Constant alias for string: "text/css".

TEXT_CSV: Self

A CSV file.

Constant alias for string: "text/csv".

TEXT_HTML: Self

An HTML file.

Constant alias for string: "text/html".

TEXT_JAVASCRIPT: Self

A JavaScript file.

Constant alias for string: "text/javascript".

TEXT_JSON: Self

JSON data intended to be human readable.

Constant alias for string: "text/json".

TEXT_JSON5: Self

JSON5 encoded data intended to be human readable.

Constant alias for string: "text/json5".

TEXT_MARKDOWN: Self

A MarkDown file.

Constant alias for string: "text/markdown".

TEXT_PLAIN: Self

A textual file.

Constant alias for string: "text/plain".

TEXT_XML: Self

An XML file that is human readable.

Constant alias for string: "text/xml".

TEXT_YAML: Self

YAML data intended to be human readable.

Constant alias for string: "text/yaml".

VIDEO_H261: Self

An h261-encoded video stream.

Constant alias for string: "video/h261".

VIDEO_H263: Self

An h263-encoded video stream.

Constant alias for string: "video/h263".

VIDEO_H264: Self

An h264-encoded video stream.

Constant alias for string: "video/h264".

VIDEO_H265: Self

An h265-encoded video stream.

Constant alias for string: "video/h265".

VIDEO_H266: Self

An h266-encoded video stream.

Constant alias for string: "video/h266".

VIDEO_MP4: Self

A video codec defined in MPEG-1, MPEG-2, MPEG-4, or registered at the MP4 registration authority.

Constant alias for string: "video/mp4".

VIDEO_OGG: Self

An Ogg-encapsulated video stream.

Constant alias for string: "video/ogg".

VIDEO_RAW: Self

An uncompressed, studio-quality video stream.

Constant alias for string: "video/raw".

VIDEO_VP8: Self

A VP8-encoded video stream.

Constant alias for string: "video/vp8".

VIDEO_VP9: Self

A VP9-encoded video stream.

Constant alias for string: "video/vp9".

ZENOH_BYTES: Self

Just some bytes.

Constant alias for string: "zenoh/bytes".

ZENOH_SERIALIZED: Self

Zenoh serialized data.

Constant alias for string: "zenoh/serialized".

ZENOH_STRING: Self

A UTF-8 string.

Constant alias for string: "zenoh/string".

final class zenoh.**EntityGlobalId**

The ID globally identifying an entity in a zenoh system.

> **Warning:** This API has been marked as unstable: it works as advertised, but it may be changed in a future release.

property eid: int

Returns the *EntityId* used to identify the entity in a Zenoh session.

property zid: *ZenohId*

Returns the *ZenohId*, i.e. the Zenoh session, this ID is associated to.

final class zenoh.**Hello**

A zenoh Hello message.

A Hello message is returned in the ::ref::*scouting* process for each found Zenoh node on the network. It contains information about the node's identity and its addresses in locators format.

property locators: list[str]

Get the locators (network addresses) of the Zenoh node.

property whatami: *WhatAmI*

Get the *WhatAmI* type of the Zenoh node.

property zid: *ZenohId*

Get the *ZenohId* of the Zenoh node.

final class zenoh.**KeyExpr**(*key_expr*)

Key expressions are Zenoh's address space for data routing and selection.

A KeyExpr represents a set of keys in a hierarchical namespace using slash-separated paths with wildcard support (* and **). It may carry optimizations for use with a Session that has declared it.

For detailed information about key expressions, see *Key Expressions*.

Parameters
key_expr (*str*) –

Return type
Self

classmethod autocanonize(*key_expr*)

Canonizes the passed value before returning it as a KeyExpr. Raises *ZError* if the passed value isn't a valid key expression despite canonization.

Parameters
key_expr (*str*) –

Return type
Self

concat(*other*)

Performs string concatenation and returns the result as a KeyExpr. Raises *ZError* if the result is not a valid key expression. You should probably prefer *join()* as Zenoh may then take advantage of the hierachical separation it inserts.

Parameters
other (*str*) –

Return type
KeyExpr

`includes`(*other*)

Returns true if self includes other, i.e. the set defined by self contains every key belonging to the set defined by other.

Parameters
other (KeyExpr | *str*) –

Return type
bool

`intersects`(*other*)

Returns true if the keyexprs intersect, i.e. there exists at least one key which is contained in both of the sets defined by self and other.

Parameters
other (KeyExpr | *str*) –

Return type
bool

`join`(*other*)

Joins both sides, inserting a / in between them. This should be your prefered method when concatenating path segments.

Parameters
other (*str*) –

Return type
KeyExpr

`relation_to`(*other*)

Returns the relation between self and other from self's point of view (*SetIntersectionLevel.INCLUDES* signifies that self includes other).
Note that this is slower than *intersects()* and *includes()*, so you should favor these methods for most applications.

> **Warning:** This API has been marked as unstable: it works as advertised, but it may be changed in a future release.

Parameters
other (KeyExpr | *str*) –

Return type
SetIntersectionLevel

`final class zenoh.Link`

Information about a Zenoh link within a transport.

A Link represents a single network connection within a transport. Transports may have multiple links for redundancy or different network paths.

`property auth_identifier: str | None`

The authentication identifier for this link, if any.

property dst: str

The destination locator of this link.

property group: str | None

The multicast group this link belongs to, if any.

property interfaces: list[str]

The network interfaces used by this link.

property is_streamed: bool

Whether this link uses a streamed protocol (e.g., TCP) or datagram (e.g., UDP).

property mtu: int

The Maximum Transmission Unit (MTU) of this link.

property priorities: tuple[int, int] | None

The priority range (min, max) for this link, if configured.

property reliability: *Reliability* | None

The reliability setting for this link, if configured.

property src: str

The source locator of this link.

property zid: *ZenohId*

The *ZenohId* of the remote node.

final class zenoh.**LinkEvent**

An event indicating a link was added or removed.

LinkEvent is emitted by *LinkEventsListener* when a link within a transport is established or terminated.

property kind: *SampleKind*

The kind of event: *SampleKind.PUT* for added, *SampleKind.DELETE* for removed.

property link: *Link*

The *Link* that was added or removed.

final class zenoh.**LinkEventsListener**

A listener that receives notifications when links are added or removed.

The listener is created using *SessionInfo.declare_link_events_listener()* and yields *LinkEvent* instances when links within transports are established or terminated.

recv()

Receive a *LinkEvent*, blocking until one is available.

Parameters
self (LinkEventsListener[Handler[LinkEvent]]) –

Return type
LinkEvent

try_recv()

Try to receive a *LinkEvent* without blocking.

Parameters
self (LinkEventsListener[Handler[LinkEvent]]) –

Return type
LinkEvent | None

undeclare()

Stop listening for link events.

property handler: _H

The handler associated with this LinkEventsListener instance.

See *Channels and callbacks* for more information on handlers.

final class zenoh.**Liveliness**

A structure with functions to declare a *LivelinessToken*, query existing *LivelinessToken*s and subscribe to liveliness changes.

A *LivelinessToken* is a token whose liveliness is tied to the Zenoh *Session* and can be monitored by remote applications.

The Liveliness structure can be obtained with the *Session.liveliness()* method of the *Session* class.

For more information, see *Liveliness*.

declare_subscriber(*key_expr*, *handler=None*, ***, *history=None*)

Create a *Subscriber* for liveliness changes matching the given key expression.

Parameters

- **key_expr** (KeyExpr | *str*) – The *LivelinessToken* key expression.
- **handler** (DefaultHandler[Sample] | FifoChannel[Sample] | RingChannel[Sample] | *tuple[Callable[[Sample], Any], Any]* | *Callable[[Sample], Any]* | *None*) – The handler for receiving liveliness samples (see *Channels and callbacks*).
- **history** (*bool* | *None*) – If True, the already present liveliness tokens will be reported upon declaration.

Return type
Subscriber

declare_token(*key_expr*)

Create a *LivelinessToken* for the given key expression.

Parameters
key_expr (KeyExpr | *str*) –

Return type
LivelinessToken

get(*key_expr*, *handler=None*, ***, *timeout=None*, *cancellation_token=None*)

Query *LivelinessToken* with matching key expressions.

Parameters

- **key_expr** (KeyExpr | *str*) –
- **handler** (DefaultHandler[Reply] | FifoChannel[Reply] | RingChannel[Reply] | *tuple[Callable[[Reply], Any], Any]* | *Callable[[Reply], Any]* | *None*) –
- **timeout** (*float* | *int* | *None*) –
- **cancellation_token** (CancellationToken | *None*) –

Return type
Handler

final class zenoh.**LivelinessToken**

A token whose liveliness is tied to the Zenoh *Session* and can be monitored by remote applications using the *Liveliness* structure. The token is declared using *Liveliness.declare_token()* with a specific key expression.

undeclare()

Undeclare the *LivelinessToken*.

final enum zenoh.**Locality**(*value*)

The locality of samples/queries to be received by subscribers/queryables or targeted by publishers/queriers.

This enum controls whether data is exchanged only with local entities (in the same session), only with remote entities, or with both. It is used in the following settings:

- *Session.declare_subscriber()* (allowed_origin)
- *Session.declare_queryable()* (allowed_origin)
- *Session.declare_publisher()* (allowed_destination)
- *Session.declare_querier()* (allowed_destination)

Valid values are as follows:

SESSION_LOCAL = <Locality.SESSION_LOCAL: 1>

Request/serve data only to entities in the same session.

REMOTE = <Locality.REMOTE: 2>

Request/serve data only to remote entities (not in the same session).

ANY = <Locality.ANY: 3>

Request/serve data to both local and remote entities.

final class zenoh.**MatchingListener**

A listener that sends notifications when the *MatchingStatus* of a corresponding Zenoh entity changes.

The matching listener allows publishers and queriers to detect when there are matching subscribers or queryables on the network, enabling efficient resource usage.

See also: *Matching*

recv()

Parameters
self (MatchingListener[Handler[MatchingStatus]]) –

Return type
MatchingStatus

try_recv()

Parameters
self (MatchingListener[Handler[MatchingStatus]]) –

Return type
MatchingStatus | None

undeclare()

Close a Matching listener. Matching listeners are automatically closed when dropped, but you may want to use this function to handle errors or close the Matching listener asynchronously.

property handler: _H

The handler associated with this MatchingListener instance.

See *Channels and callbacks* for more information on handlers.

final class zenoh.**MatchingStatus**

A struct that indicates if there exist entities matching the key expression.

See also: *Matching*

property matching: bool

Return true if there exist entities matching the target (i.e either Subscribers matching Publisher's key expression or Queryables matching Querier's key expression and target).

final class zenoh.**NTP64**(*seconds*, *nanos*)

A NTP 64-bits format as specified in RFC-5909 <https://tools.ietf.org/html/rfc5905#section-6>

The 1st 32-bits part is the number of second since the EPOCH of the physical clock, and the 2nd 32-bits part is the fraction of second.

Parameters

- **seconds** (*int*) –
- **nanos** (*int*) –

Return type
Self

as_nanos()

Returns the total duration converted to nanoseconds.

Return type
int

as_secs()

Returns the 32-bits seconds part.

Return type
int

as_secs_f64()

Returns this NTP64 as a f64 in seconds.

The integer part of the f64 is the NTP64's Seconds part. The decimal part of the f64 is the result of a division of NTP64's Fraction part divided by 2^32. Considering the probable large number of Seconds (for a time relative to UNIX_EPOCH), the precision of the resulting f64 might be in the order of microseconds. Therefore, it should not be used for comparison. Directly comparing [NTP64] objects is preferable.

Return type
float

classmethod parse_rfc3339(*s*)

Parse a RFC3339 time representation into a NTP64.

Parameters
s (*str*) –

Return type
Self

subsec_nanos()

Returns the 32-bits fraction of second part converted to nanoseconds.

Return type
int

to_datetime()

Returns this NTP64 as a datetime.

Return type
datetime

to_string_rfc3339_lossy()

Convert to a RFC3339 time representation with nanoseconds precision.

e.g.: *"2024-07-01T13:51:12.129693000Z"* `

Return type
str

final class zenoh.**Parameters**(*parameters=None*)

A collection of key-value parameters used in query operations.

Parameters allow attaching additional metadata to queries. They can be constructed from dictionaries, strings, or built programmatically. When combined with a key expression, they form a *Selector* for query operations.

See also: *Query Parameters*

Parameters
parameters (*dict[str, str] | str | None*) –

extend(*parameters*)

Extend these properties with other properties.

Parameters
parameters (*Parameters | dict[str, str] | str*) –

get(*key*, *default=None*)

Returns the value corresponding to the key.

Parameters

- **key** (*str*) –
- **default** (*str | None*) –

Return type
str | None

insert(*key*, *value*)

Inserts a key-value pair into the map. If the map did not have this key present, None` is returned. If the map did have this key present, the value is updated, and the old value is returned.

Parameters

- **key** (*str*) –
- **value** (*str*) –

is_empty()

Returns true if properties does not contain anything.

Return type
bool

is_ordered()

Returns *true* if all keys are sorted in alphabetical order.

Return type
bool

remove(*key*)

Removes a key from the map, returning the value at the key if the key was previously in the properties.

Parameters
key (*str*) –

values(*key*)

Returns the list of values corresponding to the key.

Parameters
key (*str*) –

Return type
list[str]

final enum zenoh.**Priority**(*value*)

The priority of Zenoh messages.

Priority determines the transmission priority of messages, with higher priority messages being delivered before lower priority ones when network congestion occurs.

See also: https://docs.rs/zenoh/latest/zenoh/qos/enum.Priority.html

Used in:

- *Query.reply()*
- *Query.reply_del()*
- *Session.put()*
- *Session.delete()*
- *Session.get()*

Valid values are as follows:

REAL_TIME = <Priority.REAL_TIME: 1>

INTERACTIVE_HIGH = <Priority.INTERACTIVE_HIGH: 2>

INTERACTIVE_LOW = <Priority.INTERACTIVE_LOW: 3>

DATA_HIGH = <Priority.DATA_HIGH: 4>

DATA = <Priority.DATA: 5>

DATA_LOW = <Priority.DATA_LOW: 6>

BACKGROUND = <Priority.BACKGROUND: 7>

final class zenoh.**Publisher**

A publisher that allows sending data through a stream.

Publishers are automatically undeclared when dropped.

A publisher is created using *zenoh.Session.declare_publisher()* and is used to publish data to *Subscriber* instances matching the publisher's key expression.

Publishers can declare *MatchingListener* instances to monitor if subscribers matching the publisher's key expression are present in the network.

For more information about publish/subscribe operations, see *Publish/Subscribe*.

declare_matching_listener(*handler=None*)

Create a *MatchingListener*. It will send notifications each time the matching status of this publisher changes.

Parameters

handler (*DefaultHandler[MatchingStatus] | FifoChannel[MatchingStatus] | RingChannel[MatchingStatus] | tuple[Callable[[MatchingStatus], Any], Any] | Callable[[MatchingStatus], Any] | None*) –

Return type

MatchingListener

delete(*, *attachment=None*, *timestamp=None*, *source_info=None*)

Declare that data associated with this publisher's key expression is deleted.

Subscriber instances will receive a *zenoh.Sample* with *zenoh.SampleKind.DELETE* kind, indicating that the data is no longer associated with the key expression.

Parameters

- **attachment** (*Any | None*) –
- **timestamp** (*Timestamp | None*) –
- **source_info** (*SourceInfo | None*) –

put(*payload*, *, *encoding=None*, *attachment=None*, *timestamp=None*, *source_info=None*)

Publish data to *Subscriber* instances matching this publisher's key expression.

Subscribers will receive the data as a *zenoh.Sample* with *zenoh.SampleKind.PUT* kind.

Parameters

- **payload** (*Any*) –
- **encoding** (*Encoding | str | None*) –
- **attachment** (*Any | None*) –
- **timestamp** (*Timestamp | None*) –
- **source_info** (*SourceInfo | None*) –

undeclare()

Undeclare the publisher, informing the network that it needn't optimize publications for its key expression anymore.

property congestion_control: *CongestionControl*

The congestion control strategy applied when routing data.

property encoding: *Encoding*

The encoding used when publishing data.

property id: *EntityGlobalId*

The global ID of this publisher. .. warning:: This API has been marked as unstable: it works as advertised, but it may be changed in a future release.

property key_expr: *KeyExpr*

The key expression this publisher publishes to.

property matching_status: bool

Whether there are subscribers matching this publisher's key expression.

property priority: *Priority*

The priority of the published data.

property reliability: *Reliability*

The reliability applied when routing data. .. warning:: This API has been marked as unstable: it works as advertised, but it may be changed in a future release.

final class zenoh.**Querier**

A querier that allows sending queries to a *Queryable*.

The querier is a preconfigured object that can be used to send multiple queries to a given key expression. It is declared using *Session.declare_querier()*. Queriers are automatically undeclared when dropped.

See *Query/Reply* for more information on the query/reply paradigm.

declare_matching_listener(*handler=None*)

Returns a *MatchingListener* for this Querier.

The *MatchingListener* will send a notification each time the *MatchingStatus* of the Querier changes.

Parameters

handler (DefaultHandler[MatchingStatus] | FifoChannel[MatchingStatus] | RingChannel[MatchingStatus] | tuple[Callable[[MatchingStatus], Any], Any] | Callable[[MatchingStatus], Any] | None) –

Return type

MatchingListener

get(*handler=None*, ***, *parameters=None*, *payload=None*, *encoding=None*, *attachment=None*, *source_info=None*, *cancellation_token=None*)

Sends a query and returns a channel for processing replies.

See *Channels and callbacks* for more information on handlers.

Parameters

- **handler** (DefaultHandler[Reply] | FifoChannel[Reply] | RingChannel[Reply] | tuple[Callable[[Reply], Any], Any] | Callable[[Reply], Any] | None) –
- **parameters** (Parameters | dict[str, str] | str | None) –
- **payload** (Any | None) –
- **encoding** (Encoding | str | None) –
- **attachment** (Any | None) –
- **source_info** (SourceInfo | None) –
- **cancellation_token** (CancellationToken | None) –

Return type

Handler

undeclare()

Undeclares the Querier, informing the network that it needn't optimize queries for its key expression anymore.

property accept_replies: *ReplyKeyExpr*

Returns the *ReplyKeyExpr* setting of this querier.

property id: *EntityGlobalId*

Returns the *EntityGlobalId* of this Querier. .. warning:: This API has been marked as unstable: it works as advertised, but it may be changed in a future release.

property key_expr: *KeyExpr*

Returns the *KeyExpr* this querier sends queries on.

property matching_status: bool

Returns true if there are *Queryable*s matching the Querier's key expression and target, false otherwise.

final class zenoh.**Query**

A request received by a *Queryable*.

It contains the key expression, parameters, payload, and attachment sent by a querier via *Session.get()* or *Querier.get()*. Use its methods to send replies.

Note: The Query's *key_expr* is **not** the key expression which should be used as the parameter of *reply()*, because it may contain globs. The *Queryable*'s key expression is the one that should be used.

This parameter is not set automatically because *Queryable* itself may serve glob key expressions and send replies on different concrete key expressions matching this glob. For example, a *Queryable* serving `foo/*` may receive a *Query* with key expression `foo/bar` and another one with `foo/baz`, and it should reply respectively on `foo/bar` and `foo/baz`.

Note: By default, queries only accept replies whose key expression intersects with the query's. I.e. it's not possible to send reply with key expression `foo/bar` to a query with key expression `baz/*`. To allow disjoint replies, use the `accept_replies` parameter with *ReplyKeyExpr.ANY* in *Session.get()* or *Session.declare_querier()*. Alternatively, the query may contain special parameter `_anyke` which also enables disjoint replies. See the *Selector* documentation for more information about this parameter.

See *Query/Reply* for more information on the query/reply paradigm.

accepts_replies()

Returns the *ReplyKeyExpr* setting of this query, indicating whether replies must match the query's key expression or can use any key expression.

Return type
ReplyKeyExpr

drop()

Drop the instance of a query. The query will only be finalized when all query instances (one per queryable matched) are dropped. Finalization is required to not have query hanging on the querier side.

This method should be called after handling the query, as Python finalizers are not reliable, especially when it comes to loop variables. It is also possible, and advised, to use query context manager, which calls *drop* on exit. Once a query is dropped, it's no more possible to use it, and its methods will raise an exception.

reply(*key_expr*, *payload*, *, *encoding=None*, *congestion_control=None*, *priority=None*, *express=None*, *attachment=None*, *timestamp=None*)

Sends a *Sample* of kind *SampleKind.PUT* as a reply to this query.

Note: See the class documentation for important details about which key expression to use for replies.

Deprecated since version The: `congestion_control` and `priority` parameters are deprecated and will be ignored. Response QoS now automatically matches the original query's QoS to avoid priority inversion.

Parameters

- **key_expr** (KeyExpr | *str*) –
- **payload** (*Any*) –
- **encoding** (Encoding | *str* | *None*) –
- **congestion_control** (CongestionControl | *None*) –
- **priority** (Priority | *None*) –
- **express** (*bool* | *None*) –
- **attachment** (*Any* | *None*) –
- **timestamp** (Timestamp | *None*) –

reply_del(*key_expr*, *, *congestion_control=None*, *priority=None*, *express=None*, *attachment=None*, *timestamp=None*)

Sends a *Sample* of kind *SampleKind.DELETE* as a reply to this query.

By default, queries only accept replies whose key expression intersects with the query's. Unless the query has enabled disjoint replies (you can check this through *accepts_replies()*), replying on a disjoint key expression will result in an error when resolving the reply.

Note: See the class documentation for important details about which key expression to use for replies.

Deprecated since version The: `congestion_control` and `priority` parameters are deprecated and will be ignored. Response QoS now automatically matches the original query's QoS to avoid priority inversion.

Parameters

- **key_expr** (KeyExpr | *str*) –
- **congestion_control** (CongestionControl | *None*) –
- **priority** (Priority | *None*) –
- **express** (*bool* | *None*) –
- **attachment** (*Any* | *None*) –
- **timestamp** (Timestamp | *None*) –

reply_err(*payload*, *, *encoding=None*)

Sends a *ReplyError* as a reply to this query.

Parameters

- **payload** (*Any*) –
- **encoding** (Encoding | *str* | *None*) –

property attachment: *ZBytes* | None

The attachment of this query, if any.

property encoding: *Encoding* | None

The encoding of this query's payload, if any.

property key_expr: *KeyExpr*

The key expression part of this query.

property parameters: *Parameters*

The selector parameters of this query.

property payload: *ZBytes* | None

The payload of this query, if any.

property selector: *Selector*

The full *Selector* of this query.

property source_info: *SourceInfo* | None

Gets info on the source of this Query. .. warning:: This API has been marked as unstable: it works as advertised, but it may be changed in a future release.

final class zenoh.**QueryConsolidation**(*mode*, */*)

The reply consolidation strategy to apply to replies to a get.

By default, the consolidation strategy is *QueryConsolidation.AUTO*, which lets the implementation choose the best strategy depending on the query parameters and the number of responders. Other strategies can be selected by using a specific *ConsolidationMode* as a parameter of the *Session.declare_querier()* or *Session.get()* methods.

See the documentation of *ConsolidationMode* for more details about each strategy.

Parameters
mode (ConsolidationMode) –

Return type
Self

AUTO: Self

DEFAULT: Self

property mode: *ConsolidationMode*

final enum zenoh.**QueryTarget**(*value*)

The Queryables to which a query from *Session.get()* or *Session.declare_querier()* is delivered.

QueryTarget.ALL makes the query be delivered to all the matching queryables. *QueryTarget.ALL_COMPLETE* makes the query be delivered to all the matching queryables which are marked as "complete". *QueryTarget.BEST_MATCHING* (default) makes the data to be requested from the queryable(s) selected by zenoh to get the fastest and most complete reply.

It is set by the target parameter of *Session.get()* or *Session.declare_querier()* methods.

Valid values are as follows:

BEST_MATCHING = <QueryTarget.BEST_MATCHING: 1>

Let Zenoh find the best matching queryable capable of serving the query.

ALL = <QueryTarget.ALL: 2>

Deliver the query to all queryables matching the query's key expression.

ALL_COMPLETE = <QueryTarget.ALL_COMPLETE: 3>

Deliver the query to all queryables matching the query's key expression that are declared as complete.

final class zenoh.**Queryable**

A Queryable is an entity that implements *Query/Reply* pattern.

It is declared by the *Session.declare_queryable()* method and provides Query`es using callback or channel. The Queryable receives :class:`Query requests from *Querier.get()* or *Session.get()* and sends back replies with the methods of *Query.reply()*, *Query.reply_err()*, or *Query.reply_del()*.

recv()

Receive a *Query* from the handler, blocking if necessary.

Parameters
self (Queryable[Handler[Query]]) –

Return type
Query

try_recv()

Try to receive a *Query* from the handler without blocking.

Parameters
self (Queryable[Handler[Query]]) –

Return type
Query | None

undeclare()

Undeclare the Queryable.

property handler: _H

The handler associated with this Queryable instance.

See *Channels and callbacks* for more information on handlers.

property id: *EntityGlobalId*

Returns the *EntityGlobalId* of this Queryable. .. warning:: This API has been marked as unstable: it works as advertised, but it may be changed in a future release.

property key_expr: *KeyExpr*

Returns the *KeyExpr* this queryable responds to.

final enum zenoh.**Reliability**(*value*)

Reliability guarantees for message delivery.

Used when declaring publishers with *Session.declare_publisher()* and accessible via the *Publisher.reliability* property.

> **Warning:** This API has been marked as unstable: it works as advertised, but it may be changed in a future release.

Valid values are as follows:

BEST_EFFORT = <Reliability.BEST_EFFORT: 1>

Messages may be lost.

RELIABLE = <Reliability.RELIABLE: 2>

Messages are guaranteed to be delivered.

final class zenoh.**Reply**

An answer received from a *Queryable*.

Contains the result of a request to a *Queryable* by *Session.get()* or *Querier.get()*. May be either a successful result with a *Sample* or an error with a *ReplyError*.

property err: *ReplyError* | None

Returns the error if this reply failed, *None* otherwise.

property ok: *Sample* | None

Returns the successful result if this reply is successful, *None* otherwise.

property replier_id: *EntityGlobalId* | None

Returns the ID of the zenoh instance that answered this reply. .. warning:: This API has been marked as unstable: it works as advertised, but it may be changed in a future release.

property result: *Sample* | *ReplyError*

Gets the result of this reply which may be either a successful *Sample* or an error *ReplyError*.

final class zenoh.**ReplyError**

An error reply received from a *Queryable* and available in the *Reply* structure.

property encoding: *Encoding*

Gets the encoding of this *ReplyError*.

property payload: *ZBytes*

Gets the payload of this *ReplyError*, usually an error message.

final enum zenoh.**ReplyKeyExpr**(*value*)

Controls whether replies to a query must match the query's key expression.

ReplyKeyExpr.MATCHING_QUERY (default) means that replies must have a key expression matching the query's key expression. *ReplyKeyExpr.ANY* allows replies with any key expression, even if it doesn't match the query.

It is set by the `accept_replies` parameter of *Session.get()* or *Session.declare_querier()* methods.

Valid values are as follows:

ANY = <ReplyKeyExpr.ANY: 1>

Accept replies whose key expressions may not match the query key expression.

MATCHING_QUERY = <ReplyKeyExpr.MATCHING_QUERY: 2>

Accept replies whose key expressions match the query key expression.

final class zenoh.**Sample**

The Sample structure is the data unit received by *Subscriber*, or by *Querier* or *Session.get()* as part of the *Reply*.

It contains the payload and all metadata associated with the data.

property attachment: *ZBytes* | None

Gets the sample attachment: a map of key-value pairs.

property congestion_control: *CongestionControl*

Gets the congestion control of this Sample.

property encoding: *Encoding*

Gets the encoding of this sample.

property express: **bool**

Gets the express flag value.

If true, the message is not batched during transmission, in order to reduce latency.

property key_expr: *KeyExpr*

Gets the key expression on which this Sample was published.

property kind: *SampleKind*

Gets the kind of this Sample.

property payload: *ZBytes*

Gets the payload of this Sample.

property priority: *Priority*

Gets the priority of this Sample.

property source_info: *SourceInfo* | **None**

Gets info on the source of this Sample. .. warning:: This API has been marked as unstable: it works as advertised, but it may be changed in a future release.

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

Gets the timestamp of this Sample.

final enum zenoh.**SampleKind**(*value*)

The kind of a *Sample*, indicating whether it contains data or indicates deletion.

Valid values are as follows:

PUT = <SampleKind.PUT: 1>

A *PUT* sample containing data.

DELETE = <SampleKind.DELETE: 2>

A *DELETE* sample indicating data removal.

final class zenoh.**Scout**

A Scout object that yields *zenoh.Hello* messages for discovered Zenoh nodes on the network.

Scout is returned by the *zenoh.scout()* function and is used to discover Zenoh nodes (routers and peers) on the network. It yields *zenoh.Hello* messages containing information about each discovered node.

See *Scouting* for more details on the scouting process.

recv()

Receive a *zenoh.Hello* message, blocking until one is available.

Parameters
self (Scout[Handler[Hello]]) –

Return type
Hello

stop()

Stop the scouting process.

try_recv()

Try to receive a *zenoh.Hello* message without blocking. Returns None if no message is available.

Parameters
self (Scout[Handler[Hello]]) –

Return type
Hello | None

property handler: _H

The handler associated with this Scout instance.

See *Channels and callbacks* for more information on handlers.

final class zenoh.**Selector**(*arg*, */*, *parameters=None*)

A selector is the combination of a *KeyExpr*, which defines the set of keys that are relevant to an operation, and a set of *Parameters* with a few intended uses.

Creating a Selector

A *Selector* can be created from a key expression and optional parameters:

```
selector = zenoh.Selector("key/expression", parameters)
```

Or from a complete selector string:

```
selector = zenoh.Selector("key/expression?param1=value1;param2=value2")
```

If first parameter is already a complete selector string, the *parameters* should be omitted.

A selector is the combination of a *KeyExpr*, which defines the set of keys that are relevant to an operation, and a set of *Parameters* with a few intended 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
- specifying arguments to zenoh when using the REST API

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:
  - parameters are separated by `;`
  - the parameter name and value are separated by the first `=`
  - in the absence of `=`, the parameter value is considered to be the empty string
  - both name and value should use percent-encoding (URL-encoding) to escape characters
  - defining a value for the same parameter name twice is considered undefined behavior, with the encouraged behaviour being to reject operations when a duplicate parameter is detected

Zenoh intends to standardize the usage of a set of parameter names. To avoid conflicting with RPC parameters, the Zenoh team has settled on reserving the set of parameter names that start with non-alphanumeric characters.

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

Here are the currently standardized parameters for Zenoh (check the specification page, for the exhaustive list):

- `[unstable] _time`: used to express interest in only values dated within a certain time range, values for this parameter must be readable by the Zenoh Time DSL for the value to be considered valid.
- `_anyke`: used in queries to express interest in replies coming from any key expression. By default, only replies whose key expression match query's key expression are accepted. `_anyke` disables the query-reply key expression matching check. See also *ReplyKeyExpr.ANY* as the preferred API for this functionality.

See also: *Key Expressions*, *Query Parameters*

Parameters

- **arg** (*_IntoKeyExpr | str*) –
- **parameters** (*_IntoParameters | None*) –

property key_expr: *KeyExpr*

The key expression part of this selector.

property parameters: *Parameters*

The parameters part of this selector.

final class zenoh.**Session**

The Session is the main component of Zenoh. It holds the zenoh runtime object, which maintains the state of the connection of the node to the Zenoh network.

The session allows declaring other zenoh entities like *Publisher*, *Subscriber*, *Querier*, *Queryable*, and obtaining *Liveliness* instances, and keeps them functioning. Closing the session will undeclare all objects declared by it.

A Zenoh session is instantiated using *open()* with parameters specified in the *Config* object.

close()

Close the zenoh Session.

Every *Subscriber* and *Queryable* declared will stop receiving data, and further attempts to publish or query will result in an error. Undeclaring an entity after session closing is a no-op. Session state can be checked with *is_closed()*.

Sessions are automatically closed when all their instances are dropped. But it can be useful to close the session explicitly.

declare_keyexpr(*key_expr*)

Informs Zenoh that you intend to use the provided key_expr multiple times and that it should optimize its transmission.

Parameters

key_expr (KeyExpr | *str*) –

declare_publisher(*key_expr*, *, *encoding=None*, *congestion_control=None*, *priority=None*, *express=None*, *reliability=None*, *allowed_destination=None*)

Create a *Publisher* for the given key expression.

Parameters

- **key_expr** (KeyExpr | *str*) –
- **encoding** (Encoding | *str* | *None*) –
- **congestion_control** (CongestionControl | *None*) –
- **priority** (Priority | *None*) –

- **express** (*bool | None*) –
- **reliability** (Reliability | *None*) –
- **allowed_destination** (Locality | *None*) –

Return type
Publisher

declare_querier(*key_expr*, ***, *target=None*, *consolidation=None*, *accept_replies=None*, *timeout=None*, *congestion_control=None*, *priority=None*, *express=None*, *allowed_destination=None*)

Create a *Querier* for the given key expression.

Parameters

- **key_expr** (KeyExpr | *str*) –
- **target** (QueryTarget | *None*) –
- **consolidation** (ConsolidationMode | *None*) –
- **accept_replies** (ReplyKeyExpr | *None*) –
- **timeout** (*float | int | None*) –
- **congestion_control** (CongestionControl | *None*) –
- **priority** (Priority | *None*) –
- **express** (*bool | None*) –
- **allowed_destination** (Locality | *None*) –

Return type
Querier

declare_queryable(*key_expr*, *handler=None*, ***, *complete=None*, *allowed_origin=None*)

Create a *Queryable* for the given key expression.

Parameters

- **key_expr** (KeyExpr | *str*) –
- **handler** (DefaultHandler[Query] | FifoChannel[Query] | RingChannel[Query] | *tuple[Callable[[Query], Any], Any]* | *Callable[[Query], Any]* | *None*) –
- **complete** (*bool | None*) –
- **allowed_origin** (Locality | *None*) –

Return type
Queryable

declare_subscriber(*key_expr*, *handler=None*, ***, *allowed_origin=None*)

Create a *Subscriber* for the given key expression.

Parameters

- **key_expr** (KeyExpr | *str*) –
- **handler** (DefaultHandler[Sample] | FifoChannel[Sample] | RingChannel[Sample] | *tuple[Callable[[Sample], Any], Any]* | *Callable[[Sample], Any]* | *None*) –
- **allowed_origin** (Locality | *None*) –

Return type
Subscriber

delete(*key_expr*, *, *congestion_control=None*, *priority=None*, *express=None*, *attachment=None*, *timestamp=None*, *allowed_destination=None*, *source_info=None*)

Publish a delete sample directly from the session.

This is a shortcut for declaring a *Publisher* and calling delete on it.

Parameters

- **key_expr** (KeyExpr | *str*) –
- **congestion_control** (CongestionControl | *None*) –
- **priority** (Priority | *None*) –
- **express** (*bool* | *None*) –
- **attachment** (*Any* | *None*) –
- **timestamp** (Timestamp | *None*) –
- **allowed_destination** (Locality | *None*) –
- **source_info** (SourceInfo | *None*) –

get(*selector*, *handler=None*, *, *target=None*, *consolidation=None*, *accept_replies=None*, *timeout=None*, *congestion_control=None*, *priority=None*, *express=None*, *payload=None*, *encoding=None*, *attachment=None*, *allowed_destination=None*, *source_info=None*, *cancellation_token=None*)

Query data from the matching queryables in the system.

This is a shortcut for declaring a *Querier* and calling get on it.

Parameters

- **selector** (Selector | KeyExpr | *str*) –
- **handler** (DefaultHandler[Reply] | FifoChannel[Reply] | RingChannel[Reply] | *tuple[Callable[[Reply], Any], Any]* | *Callable[[Reply], Any]* | *None*) –
- **target** (QueryTarget | *None*) –
- **consolidation** (ConsolidationMode | *None*) –
- **accept_replies** (ReplyKeyExpr | *None*) –
- **timeout** (*float* | *int* | *None*) –
- **congestion_control** (CongestionControl | *None*) –
- **priority** (Priority | *None*) –
- **express** (*bool* | *None*) –
- **payload** (*Any*) –
- **encoding** (Encoding | *str* | *None*) –
- **attachment** (*Any* | *None*) –
- **allowed_destination** (Locality | *None*) –
- **source_info** (SourceInfo | *None*) –
- **cancellation_token** (CancellationToken | *None*) –

Return type
Handler

`is_closed()`

Check if the session has been closed.

Return type
bool

`liveliness()`

Obtain a *Liveliness* instance tied to this Zenoh session.

Return type
Liveliness

`new_timestamp()`

Get a new *Timestamp* from a Zenoh session.

The returned timestamp has the current time, with the session's runtime *ZenohId* as the unique identifier. This ensures that timestamps from different sessions are unique even when created at the same time.

Returns:
A new *Timestamp* with current time and session's unique ID.

Return type
Timestamp

`put(key_expr, payload, *, encoding=None, congestion_control=None, priority=None, express=None, attachment=None, timestamp=None, allowed_destination=None, source_info=None)`

Publish data directly from the session.

This is a shortcut for declaring a *Publisher* and calling put on it.

Parameters

- **key_expr** (KeyExpr | *str*) –
- **payload** (*Any*) –
- **encoding** (Encoding | *str* | *None*) –
- **congestion_control** (CongestionControl | *None*) –
- **priority** (Priority | *None*) –
- **express** (*bool* | *None*) –
- **attachment** (*Any* | *None*) –
- **timestamp** (Timestamp | *None*) –
- **allowed_destination** (Locality | *None*) –
- **source_info** (SourceInfo | *None*) –

`undeclare(obj)`

Undeclare a zenoh entity declared by the session.

Parameters
obj (KeyExpr) –

zid()

Returns the identifier of the current session.

Return type
ZenohId

property id: *EntityGlobalId*

Returns the global identifier of the session object. .. warning:: This API has been marked as unstable: it works as advertised, but it may be changed in a future release.

property info: *SessionInfo*

Get information about the session: the session id, the connected nodes.

final class zenoh.**SessionInfo**

Struct returned by *Session.info()* that allows access to information about the current zenoh *Session*.

This information includes the *ZenohId* identifier of the current session, and the identifiers of the connected routers and peers (see also *WhatAmI* for more information about peers and routers).

declare_link_events_listener(*handler=None*, *, *history=None*)

Declare a listener for link events (links being added/removed).

Parameters

- **handler** (DefaultHandler[LinkEvent] | FifoChannel[LinkEvent] | RingChannel[LinkEvent] | tuple[Callable[[LinkEvent], Any], Any] | Callable[[LinkEvent], Any] | None) – The handler for receiving link events (see *Channels and callbacks*).
- **history** (*bool | None*) – If True, existing links will be reported upon declaration.

Returns
A *LinkEventsListener* that yields *LinkEvent* instances.

Return type
LinkEventsListener

declare_transport_events_listener(*handler=None*, *, *history=None*)

Declare a listener for transport events (connections opening/closing).

Parameters

- **handler** (DefaultHandler[TransportEvent] | FifoChannel[TransportEvent] | RingChannel[TransportEvent] | tuple[Callable[[TransportEvent], Any], Any] | Callable[[TransportEvent], Any] | None) – The handler for receiving transport events (see *Channels and callbacks*).
- **history** (*bool | None*) – If True, existing transports will be reported upon declaration.

Returns
A *TransportEventsListener* that yields *TransportEvent* instances.

Return type
TransportEventsListener

links()

Return the list of *Link* instances for currently open links.

Return type
list[Link]

peers_zid()

Return the *ZenohId* of the zenoh peers this process is currently connected to.

Return type
list[ZenohId]

routers_zid()

Return the *ZenohId* of the zenoh routers this process is currently connected to.

Return type
list[ZenohId]

transports()

Return the list of *Transport* instances for currently open transports.

Return type
list[Transport]

zid()

Return the *ZenohId* of the current zenoh Session.

Return type
ZenohId

final enum zenoh.**SetIntersectionLevel**(*value*)

The possible relations between two sets of key expressions defined by glob patterns.

Each glob key expression defines a set of possible concrete key expressions that it matches. This enum describes how two such sets relate to each other.

Returned by *KeyExpr.relation_to()*.

Note that *EQUALS* implies *INCLUDES*, which itself implies *INTERSECTS*.

You can check for intersection with *level* >= *SetIntersectionLevel.INTERSECTS* and for inclusion with *level* >= *SetIntersectionLevel.INCLUDES*.

Warning: This API has been marked as unstable: it works as advertised, but it may be changed in a future release.

Valid values are as follows:

DISJOINT = <SetIntersectionLevel.DISJOINT: 1>

The sets have no key expressions in common. Example: `foo/*` and `bar/*` - no overlap.

INTERSECTS = <SetIntersectionLevel.INTERSECTS: 2>

The sets have some key expressions in common, but neither fully contains the other. Example: `foo/*` and `*/bar` - `foo/bar` matches both.

INCLUDES = <SetIntersectionLevel.INCLUDES: 3>

The first set fully contains the second set. Example: `foo/**` includes `foo/*` (where ** matches any number of sections).

EQUALS = <SetIntersectionLevel.EQUALS: 4>

The sets are identical. Example: `foo/*` and `foo/*`.

final class zenoh.**SourceInfo**(*source_id*, *source_sn*)

Information on the source of a zenoh Sample.

Contains metadata about the origin of a data sample, including the source entity's global identifier and sequence number.

Warning: This API has been marked as unstable: it works as advertised, but it may be changed in a future release.

Parameters

- **source_id** (EntityGlobalId) –
- **source_sn** (*SourceSn*) –

Return type
Self

property source_id: *EntityGlobalId*

The EntityGlobalId of the zenoh entity that published the Sample in question.

property source_sn: int

The sequence number of the Sample from the source.

final class zenoh.**Subscriber**

A subscriber that receives data from *Publisher* instances matching its key expression.

Subscribers are automatically undeclared when dropped.

A subscriber is created using *zenoh.Session.declare_subscriber()* and is used to receive data from *Publisher* instances matching the subscriber's key expression.

For more information about publish/subscribe operations, see *Publish/Subscribe*.

recv()

Receive a *Sample*, blocking until one is available.

Parameters
self (Subscriber[Handler[Sample]]) –

Return type
Sample

try_recv()

Try to receive a *Sample* without blocking.

Returns the sample if available, or None if no sample is ready.

Parameters
self (Subscriber[Handler[Sample]]) –

Return type
Sample | None

undeclare()

Close a Subscriber. Subscribers are automatically closed when dropped, but you may want to use this function to handle errors or close the Subscriber asynchronously.

property handler: _H

The handler associated with this Subscriber instance.

See *Channels and callbacks* for more information on handlers.

property id: *EntityGlobalId*

The global ID of this subscriber. .. warning:: This API has been marked as unstable: it works as advertised, but it may be changed in a future release.

property key_expr: *KeyExpr*

The key expression this subscriber subscribes to.

final class zenoh.**Timestamp**(*time*, *id*)

A timestamp consisting of an NTP64 time and a unique identifier.

Timestamps are used to provide temporal ordering and uniqueness in Zenoh operations. They combine a high-precision NTP64 timestamp with a unique identifier to ensure causality and ordering in distributed systems.

Timestamps can be created using *Session.new_timestamp()*, which returns a timestamp with the current time and the session's unique runtime identifier.

String Representations:

Timestamps support two string formats:

- **Default format**: "<ntp64_time>/<id_hex>" (e.g., "7386690599959157260/33") This is a lossless, machine-readable format.
- **RFC3339 format**: "<rfc3339_time>/<id_hex>" (e.g., "2024-07-01T13:51:12.129693000Z/33") This is a human-readable format with nanosecond precision, but may lose some precision due to rounding when converting fractional seconds to nanoseconds.

For detailed information about Timestamp, see: https://docs.rs/zenoh/latest/zenoh/time/struct.Timestamp.html

Parameters

- **time** (*datetime | NTP64*) –
- **id** (*_IntoTimestampId*) –

Return type
Self

get_diff_duration(*other*)

Returns the duration difference between this timestamp and another.

Args:
other: The timestamp to compare against.

Returns:
A timedelta representing the time difference.

Parameters
other (Timestamp) –

Return type
timedelta

get_id()

Returns the unique identifier component of the timestamp as a *TimestampId*.

Return type
TimestampId

get_time()

Returns the time component of the timestamp as a datetime object.

Return type
datetime

`get_time_as_ntp64()`

Returns the time component of the timestamp as a datetime object.

Return type
NTP64

`classmethod parse_rfc3339(s)`

Parse a RFC3339 time representation into a Timestamp.

Args:
s: A string in RFC3339 format with timestamp ID (e.g., "2024-07-01T13:51:12.129693000Z/33").

Returns:
A new Timestamp object.

Raises:
ZError: If the string cannot be parsed.

Parameters
s (*str*) –

Return type
Self

`to_string_rfc3339_lossy()`

Convert to a RFC3339 time representation with nanoseconds precision.

This format is human-readable but may lose precision due to rounding when converting fractional seconds to nanoseconds.

Returns:
A string in RFC3339 format (e.g., "2024-07-01T13:51:12.129693000Z/33").

Note:
This conversion is not bijective - converting to string and back may result in a slightly different timestamp due to precision loss.

Return type
str

`final class zenoh.TimestampId(bytes)`

A unique identifier used in *Timestamp*.

TimestampId represents a unique identifier that is part of every *Timestamp*. It is typically derived from a *ZenohId* (session identifier) to ensure that timestamps from different sessions remain unique even when created simultaneously.

The identifier is stored as a fixed-size byte array and provides methods for comparison, hashing, and conversion to/from bytes.

Used in:

- `Timestamp.__new__()` - accepts `_IntoTimestampId` (bytearray, bytes, or TimestampId)
- *Timestamp.get_id()* - returns a TimestampId
- *Session.new_timestamp()* - creates timestamps with session's ZenohId as TimestampId

Parameters
bytes (*bytearray | bytes*) –

Return type
Self

final class zenoh.**Transport**

Information about a Zenoh transport connection.

A Transport represents a connection to another Zenoh node (peer or router). It provides information about the remote node and the transport characteristics.

property is_multicast: bool

Whether this is a multicast transport.

property is_qos: bool

Whether this transport supports QoS (Quality of Service).

property whatami: *WhatAmI*

The *WhatAmI* type of the remote node.

property zid: *ZenohId*

The *ZenohId* of the remote node.

final class zenoh.**TransportEvent**

An event indicating a transport connection was opened or closed.

TransportEvent is emitted by *TransportEventsListener* when a transport connection to another Zenoh node is established or terminated.

property kind: *SampleKind*

The kind of event: *SampleKind.PUT* for opened, *SampleKind.DELETE* for closed.

property transport: *Transport*

The *Transport* that was opened or closed.

final class zenoh.**TransportEventsListener**

A listener that receives notifications when transport connections open or close.

The listener is created using *SessionInfo.declare_transport_events_listener()* and yields *TransportEvent* instances when connections to other Zenoh nodes are established or terminated.

recv()

Receive a *TransportEvent*, blocking until one is available.

Parameters
self (*TransportEventsListener[Handler[TransportEvent]]*) –

Return type
TransportEvent

try_recv()

Try to receive a *TransportEvent* without blocking.

Parameters
self (*TransportEventsListener[Handler[TransportEvent]]*) –

Return type
TransportEvent | None

undeclare()

Stop listening for transport events.

property handler: _H

The handler associated with this TransportEventsListener instance.

See *Channels and callbacks* for more information on handlers.

final enum zenoh.**WhatAmI**(*value*)

The type of the node in the Zenoh network.

The zenoh application can work in three different modes: router, peer, and client.

For detailed format information, see: https://docs.rs/zenoh/latest/zenoh/config/enum.WhatAmI.html

Valid values are as follows:

ROUTER = <WhatAmI.ROUTER: 1>

Router mode: Used to run a zenoh router, which is a node that maintains a predefined zenoh network topology. Unlike peers, routers do not discover other nodes by themselves, but rely on static configuration.

PEER = <WhatAmI.PEER: 2>

Peer mode: The application searches for other nodes and establishes direct connections with them. This can work using multicast discovery and by getting gossip information from the initial entry points. The peer mode is the default mode.

CLIENT = <WhatAmI.CLIENT: 3>

Client mode: The application remains connected to a single connection point, which serves as a gateway to the rest of the network. This mode is useful for constrained devices that cannot afford to maintain multiple connections.

final class zenoh.**WhatAmIMatcher**(*matcher=None*)

A helper type that allows matching combinations of WhatAmI values in scouting.

WhatAmIMatcher can be created from a string specification like "peer|router" or "client", or built programmatically using methods like *router()*, *peer()*, and *client()*.

The *scout()* function accepts a WhatAmIMatcher to filter the nodes of the specified types.

Parameters
matcher (*str | None*) –

Return type
Self

client()

Adds *WhatAmI.CLIENT* to the matcher.

Return type
Self

classmethod empty()

Creates an empty matcher that matches no node types.

Return type
Self

is_empty()

Returns True if the matcher matches no node types.

Return type
bool

matches(*whatami*)

Returns True if the given WhatAmI value matches this matcher.

Parameters
whatami (WhatAmI) –

Return type
bool

peer()

Adds *WhatAmI.PEER* to the matcher.

Return type
Self

router()

Adds *WhatAmI.ROUTER* to the matcher.

Return type
Self

final class zenoh.**ZBytes**(*bytes=None*)

ZBytes represents raw bytes data that can be interpreted as strings or byte arrays.

ZBytes is the fundamental data container in Zenoh for all payload and attachment data. It provides flexible access to the underlying bytes, allowing conversion to strings, byte arrays, or direct byte access.

The Zenoh protocol treats ZBytes as opaque binary data and is completely unaware of its content or structure. This allows users to employ any data representation format of their choice, including JSON, protobuf, flatbuffers, MessagePack, or custom formats.

For convenience, Zenoh provides built-in serialization functions *zenoh.ext.z_serialize()* and *zenoh.ext.z_deserialize()* in the *zenoh.ext* module, which support serialization of standard Python types (primitives, lists, dicts, etc.) using Zenoh's native binary format. However, these are not mandatory - users are encouraged to use any serialization approach that fits their needs.

Parameters
bytes (*bytearray | bytes | str |* shm.ZShm *|* shm.ZShmMut *| None*) –

Return type
Self

as_shm()

> **Warning:** This API has been marked as unstable: it works as advertised, but it may be changed in a future release.

Return type
ZShm | None

to_bytes()

Return the underlying data as bytes.

Returns:
bytes: The raw byte data contained in this ZBytes instance.

Return type
bytes

to_string()

Return the underlying data as a UTF-8 decoded string.

Returns:
str: The string representation of the byte data, decoded as UTF-8.

Raises:
ValueError: If the byte data cannot be decoded as valid UTF-8.

Return type
str

final class zenoh.**ZenohId**

The global unique id of a zenoh peer.

zenoh.**init_log_from_env_or**(*level*)

Redirect zenoh logs to stdout, according to the *RUST_LOG* environment variable.

For example, *RUST_LOG=debug* will set the log level to DEBUG. If *RUST_LOG* is not set, then logging is set to the provided level.

Parameters
level (*str*) –

zenoh.**open**(*config*)

Open a zenoh *zenoh.Session*.

For more information about sessions and configuration, see *Session and Config*.

Parameters
config (Config) –

Return type
Session

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

Scout for routers and/or peers.

scout spawns a task that periodically sends scout messages and waits for *zenoh.Hello* replies. The scouting process can be stopped by calling *zenoh.Scout.stop()* on the returned *zenoh.Scout* object, or by letting the *zenoh.Scout* object go out of scope (dropping it).

Args:
handler: Optional handler for processing received *zenoh.Hello* messages. what: Optional *zenoh.WhatAmIMatcher* or string specifying which node types to scout for (e.g., "peer|router"). If None, scouts for all node types. config: Optional *zenoh.Config* for the scouting session.

For more information about scouting, see *Scouting*.

Parameters

- **handler** (DefaultHandler[Hello] | FifoChannel[Hello] | RingChannel[Hello] | *tuple[Callable[[Hello], Any], Any]* | *Callable[[Hello], Any]* | *None*) –
- **what** (WhatAmIMatcher | *str* | *None*) –
- **config** (Config | *None*) –

Return type
Scout

zenoh.**try_init_log_from_env**()

Redirect zenoh logs to stdout, according to the *RUST_LOG* environment variable.

For example, *RUST_LOG=debug* will set the log level to DEBUG. If *RUST_LOG* is not set, then logging is not enabled.

1.3.2 module zenoh.handlers

final class zenoh.handlers.**Callback**(*callback*, *drop=None*, *, *indirect=True*)

A callback handler that invokes a user-defined function for each received item.

The *Callback* class provides a way to handle asynchronous data reception by calling a user-provided callback function for each item received. It can also optionally call a drop function when the associated Zenoh primitive (*zenoh.Subscriber*, *zenoh.Querier*, etc.) is undeclared.

When a *Callback* handler is used, the associated Zenoh primitive runs in background mode, meaning the callback continues to execute even if the object goes out of scope. For more information about channels and callbacks, see *Channels and callbacks*.

Args:

callback:
A callable that will be invoked for each received item.

drop:
An optional callable that will be invoked when the associated Zenoh primitive is undeclared and the callback handler is being cleaned up.

indirect:
This feature is unstable and may change or be removed in future releases.

Controls the threading behavior of callback execution. If *True* (default), the callback is executed in a separate Python thread. If *False*, the callback is executed directly in the same thread that receives the data (the zenoh network thread).

Parameters

- **callback** (*Callable[[_T], Any]*) –
- **drop** (*Callable[[], Any] | None*) –
- **indirect** (*bool*) –

Return type
Self

property callback: Callable[[_T], Any]

The callback function that will be invoked for each received item.

property drop: Callable[[], Any] | None

The optional drop function that will be invoked when the handler is cleaned up.

property indirect: bool

Unstable Whether the callback executes in a separate thread (True) or same thread (False).

final class zenoh.handlers.**DefaultHandler**

The default handler type used by Zenoh when no explicit handler is provided.

DefaultHandler serves as an opaque wrapper around *FifoChannel* with default settings. When no channel or callback is specified for subscribers or queries, Zenoh automatically uses this handler.

This type provides API stability by allowing the underlying default handler implementation to change without breaking existing code. Currently, it wraps a FIFO queue implementation.

For more information about channels and callbacks, see *Channels and callbacks*.

final class zenoh.handlers.**FifoChannel**(*capacity*)

A handler implementing FIFO semantics.

FifoChannel provides a bounded FIFO (First-In-First-Out) queue for handling received data. When the channel reaches its capacity, pushing additional items will block until space becomes available.

Note: A slow consumer can block the underlying Zenoh thread if it doesn't empty the FifoChannel fast enough. For applications where dropping old samples is preferable to blocking, consider using `RingChannel` instead.

For more information about channels and callbacks, see *Channels and callbacks*.

Args:
capacity: The maximum number of items the channel can hold.

Parameters
capacity (*int*) –

Return type
Self

final class zenoh.handlers.**Handler**

Provides access to received Zenoh data.

Handler instances are returned by Zenoh operations that receive data asynchronously. Each instance provides methods to access the received data items of type _T.

Handler serves as a common interface for different channel implementations: `DefaultHandler`, `FifoChannel`, and `RingChannel`. Regardless of which channel type is used, *Handler* provides the same methods for data access.

Handler instances are returned by several Zenoh operations:

- `zenoh.Session.get()` returns *Handler[Reply]* for accessing query replies
- `zenoh.Querier.get()` returns *Handler[Reply]* for accessing querier replies
- `zenoh.Session.declare_subscriber()` returns *Subscriber[Handler[Sample]]* for accessing received samples

Handler provides both blocking and non-blocking methods to receive data, as well as iteration support. The underlying implementation determines the specific behavior (FIFO blocking, ring buffer dropping, etc.).

recv()

Receive an item, blocking if necessary.

Waits until an item is available and returns it. This method will block the calling thread until data arrives.

Returns:
The next available item.

Return type
_T

try_recv()

Attempt to receive an item without blocking.

Returns the next available item if one is ready, otherwise returns None. This method never blocks and is useful for polling or non-blocking loops.

Returns:
The next item if available, None otherwise.

Return type
_T | None

final class zenoh.handlers.**RingChannel**(*capacity*)

A synchronous ring channel with a limited size that allows users to keep the last N data items.

RingChannel implements FIFO semantics with a dropping strategy when full. When the channel reaches its capacity, the oldest elements are dropped to make room for newer ones, ensuring that only the most recent data is kept.

This makes *RingChannel* ideal for applications that need to maintain a sliding window of recent data without blocking the producer.

For applications where data loss is unacceptable and blocking is preferable to dropping old samples, consider using *FifoChannel* instead.

For more information about channels and callbacks, see *Channels and callbacks*.

Args:
capacity: The maximum number of items the channel can hold.

Parameters
capacity (*int*) –

Return type
Self

1.3.3 module zenoh.ext

exception zenoh.ext.**ZDeserializeError**

Exception raised when deserialization with *zenoh.ext.z_deserialize()* fails.

final class zenoh.ext.**AdvancedPublisher**

An extension to Publisher providing advanced functionalities.

Advanced publishers are created via *declare_advanced_publisher()* and works alongside *AdvancedSubscriber*. Its features include:

- **Caching** - Store last published samples for retrieval via subscriber history mechanisms. Configure via *CacheConfig*.
- **Sample miss detection** - Identify gaps in publications to detect missed samples. Configure via *MissDetectionConfig*. Subscribers can monitor misses via *AdvancedSubscriber.sample_miss_listener()*.
- **Publisher detection** - Assert presence through liveliness tokens. Subscribers can detect publishers via *AdvancedSubscriber.detect_publishers()*.

Warning: This API has been marked as unstable: it works as advertised, but it may be changed in a future release.

delete(*, *attachment=None*, *timestamp=None*)

Delete the value associated with the key expression. See *zenoh.Publisher.delete()*.

Parameters

- **attachment** (*Any | None*) –
- **timestamp** (*Timestamp | None*) –

put(*payload*, *, *encoding=None*, *attachment=None*, *timestamp=None*)

Publish data to the key expression. See *zenoh.Publisher.put()*.

Parameters

- **payload** (*Any*) –
- **encoding** (*Encoding | str | None*) –
- **attachment** (*Any | None*) –
- **timestamp** (*Timestamp | None*) –

undeclare()

Undeclare the AdvancedPublisher. See *zenoh.Publisher.undeclare()*.

property congestion_control: *CongestionControl*

The congestion control policy applied to published data. See *zenoh.Publisher.congestion_control()*.

property encoding: *Encoding*

The encoding used for published data. See *zenoh.Publisher.encoding()*.

property id: *EntityGlobalId*

The globally unique id of this AdvancedPublisher. See *zenoh.Publisher.id()*. .. warning:: This API has been marked as unstable: it works as advertised, but it may be changed in a future release.

property key_expr: *KeyExpr*

The key expression this AdvancedPublisher publishes to. See *zenoh.Publisher.key_expr()*.

property priority: *Priority*

The priority level of published data. See *zenoh.Publisher.priority()*.

final class zenoh.ext.**AdvancedSubscriber**

An extension to Subscriber providing advanced functionalities.

AdvancedSubscriber is created with *declare_advanced_subscriber()* and works alongside *AdvancedPublisher*. Its features include:

- missing samples detection using periodic queries or heartbeat subscription configurable via *RecoveryConfig*. Notification about missed samples is done via *AdvancedSubscriber.sample_miss_listener()*.
- recovering missing samples with max age, sample count and late joiner detection configurable via *HistoryConfig*.
- matching publishers detection using liveliness mechanisms. Use *AdvancedSubscriber.detect_publishers()* to find active publishers.

Warning: This API has been marked as unstable: it works as advertised, but it may be changed in a future release.

detect_publishers(*handler=None*, ***, *history=None*)

Declare a listener to detect matching publishers.

Only *AdvancedPublisher* instances that enable *publisher_detection* in the *declare_advanced_publisher()* can be detected. This uses *Liveliness* to track publisher presence.

Parameters

- **handler** (*DefaultHandler[Sample] | FifoChannel[Sample] | RingChannel[Sample] | tuple[Callable[[Sample], Any], Any] | Callable[[Sample], Any] | None*) – Optional handler for receiving publisher detection events.
- **history** (*bool | None*) – If *True*, the already present publishers will be reported upon declaration. Uses *history* feature of *zenoh.Liveliness.declare_subscriber()*.

Return type
Subscriber

recv()

Receive a sample, blocking until one is available. See *zenoh.Subscriber.recv()*.

Parameters
self (*AdvancedSubscriber[Handler[Sample]]*) –

Return type
Sample

sample_miss_listener(*handler=None*)

Declare a listener to detect missed samples.

Missed samples can only be detected from *AdvancedPublisher* instances that enable *sample_miss_detection*. The listener will receive *Miss* notifications indicating the source and number of missed samples.

Parameters
handler (*DefaultHandler[Miss] | FifoChannel[Miss] | RingChannel[Miss] | tuple[Callable[[Miss], Any], Any] | Callable[[Miss], Any] | None*) –

Return type
SampleMissListener

try_recv()

Try to receive a sample without blocking. See *zenoh.Subscriber.try_recv()*.

Parameters
self (*AdvancedSubscriber[Handler[Sample]]*) –

Return type
Sample | None

undeclare()

Undeclare the AdvancedSubscriber. See *zenoh.Subscriber.undeclare()*.

property handler: _H

The handler used to process received samples. See *zenoh.Subscriber.handler()*.

property id: *EntityGlobalId*

The globally unique id of this AdvancedSubscriber. See *zenoh.Subscriber.id()*. .. warning:: This API has been marked as unstable: it works as advertised, but it may be changed in a future release.

property key_expr: *KeyExpr*

The key expression this AdvancedSubscriber subscribes to. See *zenoh.Subscriber.key_expr()*.

final class zenoh.ext.**CacheConfig**(*max_samples=None*, *, *replies_config=None*)

Configure caching behavior for an *AdvancedPublisher*.

param max_samples
specify how many samples to keep for each resource, default to 1

param replies_config
the QoS to apply to replies

> **Warning:** This API has been marked as unstable: it works as advertised, but it may be changed in a future release.

Parameters

- **max_samples** (*int | None*) –
- **replies_config** (RepliesConfig | *None*) –

Return type
Self

class zenoh.ext.**Float32**(*x=0*, */*)

float subclass enabling to (de)serialize 32bit floating point numbers.

class zenoh.ext.**Float64**(*x=0*, */*)

float subclass enabling to (de)serialize 64bit floating point numbers.

final class zenoh.ext.**HistoryConfig**(*, *detect_late_publishers=None*, *max_samples=None*, *max_age=None*)

Configure history retrieval behavior for an *AdvancedSubscriber*.

param detect_late_publishers
enable detection of late joiner publishers and query for their historical data; late joiner detection can only be achieved for *AdvancedPublisher* that enable *publisher_detection* history can only be retransmitted by *AdvancedPublisher* that enable *cache*

param max_samples
specify how many samples to query for each resource

param max_age
specify the maximum age of samples to query in seconds

> **Warning:** This API has been marked as unstable: it works as advertised, but it may be changed in a future release.

Parameters

- **detect_late_publishers** (*bool | None*) –
- **max_samples** (*int | None*) –
- **max_age** (*float | int | None*) –

Return type
Self

class zenoh.ext.**Int128**

int subclass enabling to (de)serialize 128bit signed integer.

class zenoh.ext.**Int16**

int subclass enabling to (de)serialize 16bit signed integer.

class zenoh.ext.**Int32**

int subclass enabling to (de)serialize 32bit signed integer.

class zenoh.ext.**Int64**

int subclass enabling to (de)serialize 64bit signed integer.

class zenoh.ext.**Int8**

int subclass enabling to (de)serialize 8bit signed integer.

final class zenoh.ext.**Miss**

Notification about missed samples detected by an *AdvancedSubscriber*.

A Miss indicates that one or more samples from an *AdvancedPublisher* were not received by the subscriber. This can occur due to network congestion, packet loss, or when the subscriber cannot keep up with the publication rate.

Miss detection requires the publisher to enable *sample_miss_detection* in *declare_advanced_publisher()* and the subscriber to have a *SampleMissListener* via *AdvancedSubscriber.sample_miss_listener()*.

Warning: This API has been marked as unstable: it works as advertised, but it may be changed in a future release.

property **nb**: **int**

The number of consecutive samples that were missed from this source.

property **source**: *EntityGlobalId*

The globally unique identifier of the *AdvancedPublisher* that published the missed samples.

final class zenoh.ext.**MissDetectionConfig**(*, *heartbeat*, *sporadic_heartbeat*)

Configure miss detection behavior for an *AdvancedPublisher*.

param heartbeat
period in seconds, allow last sample miss detection through periodic heartbeat; periodically send the last published *zenoh.Sample*'s sequence number to allow last sample recovery. *zenoh.ext.AdvancedSubscriber* can only recover the last sample with the *heartbeat* option enabled.

This option can not be enabled simultaneously with `sporadic_heartbeat`.

param sporadic_heartbeat
period in seconds, allow last sample miss detection through sporadic heartbeat; each period, the last published *zenoh.Sample*'s sequence number is sent with *zenoh.CongestionControl.Block* but only if it has changed since the last period. *zenoh.ext.AdvancedSubscriber* can only recover the last sample with the *heartbeat* option enabled.

This option can not be enabled simultaneously with `heartbeat`.

Warning: This API has been marked as unstable: it works as advertised, but it may be changed in a future release.

Parameters

- **heartbeat** (*float | int | None*) –
- **sporadic_heartbeat** (*float | int | None*) –

Return type
Self

final class zenoh.ext.**RecoveryConfig**(*, *periodic_queries*, *heartbeat*)

Configure recovery behavior for an *AdvancedSubscriber*.

param periodic_queries
enable periodic queries for not yet received Samples and specify their period; it allows retrieving the last Sample(s) if the last Sample(s) is/are lost, so it is useful for sporadic publications but useless for periodic publications with a period smaller or equal to this period. Retransmission can only be achieved by *AdvancedPublisher* that enable *cache* and *sample_miss_detection*.

This option can not be enabled simultaneously with `heartbeat`.

param heartbeat
subscribe to heartbeats of *AdvancedPublisher*; it allows receiving the last published Sample's sequence number and check for misses. Heartbeat subscriber must be paired with *AdvancedPublisher* that enable *cache* and *sample_miss_detection* with *heartbeat* or *sporadic_heartbeat*.

This option can not be enabled simultaneously with `periodic_queries`.

Warning: This API has been marked as unstable: it works as advertised, but it may be changed in a future release.

Parameters

- **periodic_queries** (*float | int | None*) –
- **heartbeat** (*Literal[True] | None*) –

Return type
Self

final class zenoh.ext.**RepliesConfig**(*, *congestion_control=None*, *priority=None*, *express=None*)

Configure QoS settings for replies from a *AdvancedSubscriber*. Parameter in `zenoh.CacheConfig`.

Warning: This API has been marked as unstable: it works as advertised, but it may be changed in a future release.

Parameters

- **congestion_control** (*CongestionControl | None*) –
- **priority** (*Priority | None*) –
- **express** (*bool | None*) –

Return type
Self

`final class zenoh.ext.SampleMissListener`

Listener for detecting missed samples from an *AdvancedSubscriber*.

Instances are created via *AdvancedSubscriber.sample_miss_listener()*.

This listener receives *Miss* notifications when gaps are detected in the sequence of samples from *AdvancedPublisher* instances. This works for publishers that enable *sample_miss_detection* in *declare_advanced_publisher()*.

Warning: This API has been marked as unstable: it works as advertised, but it may be changed in a future release.

`recv()`

Receive a miss notification, blocking until one is available.

Parameters
self (*SampleMissListener[Handler[Miss]]*) –

Return type
Miss

`try_recv()`

Try to receive a miss notification without blocking.

Parameters
self (*SampleMissListener[Handler[Miss]]*) –

Return type
Miss | None

`undeclare()`

Undeclare the SampleMissListener.

`class zenoh.ext.UInt128`

int subclass enabling to (de)serialize 128bit unsigned integer.

`class zenoh.ext.UInt16`

int subclass enabling to (de)serialize 16bit unsigned integer.

class zenoh.ext.**UInt32**

int subclass enabling to (de)serialize 32bit unsigned integer.

class zenoh.ext.**UInt64**

int subclass enabling to (de)serialize 64bit unsigned integer.

class zenoh.ext.**UInt8**

int subclass enabling to (de)serialize 8bit unsigned integer.

zenoh.ext.**declare_advanced_publisher**(*session*, *key_expr*, *, *encoding=None*, *congestion_control=None*, *priority=None*, *express=None*, *reliability=None*, *allowed_destination=None*, *cache=None*, *sample_miss_detection=None*, *publisher_detection=None*)

Declare an *AdvancedPublisher* for the given key expression. .. warning:: This API has been marked as unstable: it works as advertised, but it may be changed in a future release.

Parameters

- **session** (Session) –
- **key_expr** (KeyExpr | *str*) –
- **encoding** (Encoding | *str* | *None*) –
- **congestion_control** (CongestionControl | *None*) –
- **priority** (Priority | *None*) –
- **express** (*bool* | *None*) –
- **reliability** (Reliability | *None*) –
- **allowed_destination** (Locality | *None*) –
- **cache** (CacheConfig | *None*) –
- **sample_miss_detection** (MissDetectionConfig | *None*) –
- **publisher_detection** (*bool* | *None*) –

Return type

AdvancedPublisher

zenoh.ext.**declare_advanced_subscriber**(*session*, *key_expr*, *handler=None*, *, *allowed_origin=None*, *history=None*, *recovery=None*, *subscriber_detection=None*)

Declare an *AdvancedSubscriber* for the given key expression.

Parameters

- **session** (Session) –
- **key_expr** (KeyExpr | *str*) –
- **handler** (DefaultHandler[Sample] | FifoChannel[Sample] | RingChannel[Sample] | *tuple[Callable[[Sample], Any], Any]* | *Callable[[Sample], Any]* | *None*) –
- **allowed_origin** (Locality | *None*) –
- **history** (HistoryConfig | *None*) –
- **recovery** (RecoveryConfig | *None*) –
- **subscriber_detection** (*bool* | *None*) –

Return type
AdvancedSubscriber

zenoh.ext.**z_deserialize**(*tp*, *zbytes*)

Deserialize into an object of supported type according to the Zenoh serialization format.

Supported types are:

- UInt8, UInt16, Uint32, UInt64, UInt128, Int8, Int16, Int32, Int64, Int128, int (handled as int32), Float32, Float64, float (handled as Float64), bool;
- Str, Bytes, ByteArray;
- List, Dict, Set, FrozenSet and Tuple of supported types.

Parameters

- **tp** (*type[_T]*) –
- **zbytes** (ZBytes) –

Return type
_T

zenoh.ext.**z_serialize**(*obj*)

Serialize an object of supported type according to the Zenoh serialization format.

Supported types are:

- UInt8, UInt16, Uint32, UInt64, UInt128, Int8, Int16, Int32, Int64, Int128, int (handled as int32), Float32, Float64, float (handled as Float64), bool;
- Str, Bytes, ByteArray;
- List, Dict, Set, FrozenSet and Tuple of supported types.

Parameters
obj (*Any*) –

Return type
ZBytes

1.3.4 module zenoh.shm

final class zenoh.shm.**AllocAlignment**(*pow*)

alignment in powers of 2: 0 == 1-byte alignment, 1 == 2byte, 2 == 4byte, 3 == 8byte etc .. warning:: This API has been marked as unstable: it works as advertised, but it may be changed in a future release.

Parameters
pow (*int*) –

Return type
Self

align_size(*size*)

Align size according to inner alignment. This call may extend the size

Parameters
size (*int*) –

Return type
int

`get_alignment_value`()
Get alignment in normal units (bytes)

Return type
int

`ALIGN_1_BYTE: Self`

`ALIGN_2_BYTE: Self`

`ALIGN_4_BYTE: Self`

`ALIGN_8_BYTE: Self`

`final class` zenoh.shm.`BlockOn`(*inner_policy=<zenoh.shm.JustAlloc object>*)

Warning: This API has been marked as unstable: it works as advertised, but it may be changed in a future release.

Parameters
`inner_policy` (*_AllocPolicy*) –

Return type
Self

`final class` zenoh.shm.`Deallocate`(*inner_policy=<zenoh.shm.JustAlloc object>*, *alt_policy=<zenoh.shm.JustAlloc object>*)

Deallocating policy.
Forcibly deallocate up to N buffers until allocation succeeds.

Warning: This API has been marked as unstable: it works as advertised, but it may be changed in a future release.

Parameters

- `inner_policy` (*_AllocPolicy*) –
- `alt_policy` (*_AllocPolicy*) –

Return type
Self

`final class` zenoh.shm.`Defragment`(*inner_policy=<zenoh.shm.JustAlloc object>*, *alt_policy=<zenoh.shm.JustAlloc object>*)

Warning: This API has been marked as unstable: it works as advertised, but it may be changed in a future release.

Parameters

- `inner_policy` (*_AllocPolicy*) –

- **alt_policy** (*_AllocPolicy*) –

Return type
Self

final class zenoh.shm.**GarbageCollect**(*inner_policy=<zenoh.shm.JustAlloc object>*, *alt_policy=<zenoh.shm.JustAlloc object>*, ***, *safe=True*)

> **Warning:** This API has been marked as unstable: it works as advertised, but it may be changed in a future release.

Parameters

- **inner_policy** (*_AllocPolicy*) –
- **alt_policy** (*_AllocPolicy*) –
- **safe** (*bool*) –

Return type
Self

final class zenoh.shm.**JustAlloc**

Just try to allocate .. warning:: This API has been marked as unstable: it works as advertised, but it may be changed in a future release.

final class zenoh.shm.**MemoryLayout**(*size*, *alignment*)

Memory layout representation: alignment and size aligned for this alignment .. warning:: This API has been marked as unstable: it works as advertised, but it may be changed in a future release.

Parameters

- **size** (*int*) –
- **alignment** (AllocAlignment) –

Return type
Self

property alignment: *AllocAlignment*

property size: int

final class zenoh.shm.**ShmProvider**

A generalized interface for shared memory data sources .. warning:: This API has been marked as unstable: it works as advertised, but it may be changed in a future release.

alloc(*layout*, *policy=<zenoh.shm.JustAlloc object>*)

Rich interface for making allocations

Parameters

- **layout** (MemoryLayout | *tuple[int, AllocAlignment]* | *int*) –
- **policy** (JustAlloc | BlockOn | Defragment | GarbageCollect) –

Return type
ZShmMut

classmethod default_backend(*layout*)

Set the default backend

Parameters
layout (MemoryLayout | tuple[int, AllocAlignment] | int) –

Return type
Self

defragment()

Defragment memory

garbage_collect()

Try to collect free chunks. Returns the size of largest collected chunk

Return type
int

garbage_collect_unsafe()

Try to collect free chunks. Returns the size of largest collected chunk

Return type
int

property available: int

Bytes available for use

final class zenoh.shm.**ZShm**

A SHM buffer .. warning:: This API has been marked as unstable: it works as advertised, but it may be changed in a future release.

is_valid()

Return type
bool

final class zenoh.shm.**ZShmMut**

A mutable SHM buffer .. warning:: This API has been marked as unstable: it works as advertised, but it may be changed in a future release.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

Z

INDEX

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z