zenoh-c

Release 0.6.0-beta.1

ZettaScale Zenoh team

Sep 30, 2022

CONTENTS

The *libzenoh-c* library provides a C client API for the zenoh protocol.

An introduction to zenoh and its concepts is available on zenoh.io.

CHAPTER

ONE

EXAMPLES

1.1 Publish

```c
#include <string.h>
#include "zenoh.h"

int main(int argc, char **argv) {
    z_owned_config_t config = z_config_default();
    z_owned_session_t s = z_open(z_move(config));

    char* value = "value";
    z_put(z_loan(s), z_keyexpr("key/expression"), (const uint8_t *)value, strlen(value),␣
↪NULL);

    z_close(z_move(s));
    return 0;
}
```

1.2 Subscribe

```c
#include <stdio.h>
#include "zenoh.h"

void data_handler(const z_sample_t *sample, const void *arg) {
    char *keystr = z_keyexpr_to_string(sample->keyexpr);
    printf(">> Received (%s, %.*s)\n",
        keystr, (int)sample->payload.len, sample->payload.start);
    free(keystr);
}

int main(int argc, char **argv) {
    z_owned_config_t config = z_config_default();
    z_owned_session_t s = z_open(z_move(config));

    z_owned_closure_sample_t callback = z_closure(data_handler);
    z_owned_subscriber_t sub = z_declare_subscriber(z_loan(s), z_keyexpr("key/expression
↪"), z_move(callback), NULL);
```

(continues on next page)

(continued from previous page)

```
    char c = 0;
    while (c != 'q') {
        c = fgetc(stdin);
    }

    z_undeclare_subscriber(z_move(sub));
    z_close(z_move(s));
    return 0;
}
```

1.3 Query

```
#include <stdio.h>
#include "zenoh.h"

int main(int argc, char** argv) {
    z_owned_config_t config = z_config_default();
    z_owned_session_t s = z_open(z_move(config));

    z_owned_reply_channel_t channel = z_reply_fifo_new(16);
    z_get(z_loan(s), z_keyexpr("key/expression"), "", z_move(channel.send), NULL);
    z_owned_reply_t reply = z_reply_null();
    for (z_call(channel.recv, &reply); z_check(reply); z_call(channel.recv, &reply))
    {
        if (z_reply_is_ok(&reply))
        {
            z_sample_t sample = z_reply_ok(&reply);
            char *keystr = z_keyexpr_to_string(sample.keyexpr);
            printf(">> Received ('%s': '%.*s')\n", keystr, (int)sample.payload.len,␣
↪sample.payload.start);
            free(keystr);
        }
    }

    z_drop(reply);
    z_drop(channel);
    z_close(z_move(s));
    return 0;
}
```

CHAPTER

TWO

API REFERENCE

2.1 Generic types

2.1.1 Bytes

struct **z_bytes_t**

An array of bytes.

bool **z_bytes_check**(const struct *z_bytes_t* *b)

Returns `true` if *b* is initialized.

2.2 Session

2.2.1 Session configuration

struct **z_owned_config_t**

An owned zenoh configuration.

Like most *z_owned_X_t* types, you may obtain an instance of *z_X_t* by loaning it using *z_X_loan(&val)*. The *z_loan(val)* macro, available if your compiler supports C11's *_Generic*, is equivalent to writing *z_X_loan(&val)*.

Like all *z_owned_X_t*, an instance will be destroyed by any function which takes a mutable pointer to said instance, as this implies the instance's inners were moved. To make this fact more obvious when reading your code, consider using *z_move(val)* instead of *&val* as the argument. After a move, *val* will still exist, but will no longer be valid. The destructors are double-drop-safe, but other functions will still trust that your *val* is valid.

To check if *val* is still valid, you may use *z_X_check(&val)* or *z_check(val)* if your compiler supports *_Generic*, which will return *true* if *val* is valid.

struct **z_owned_scouting_config_t**

struct *z_owned_config_t* **z_config_new**()

Return a new, zenoh-allocated, empty configuration.

Like most *z_owned_X_t* types, you may obtain an instance of *z_X_t* by loaning it using *z_X_loan(&val)*. The *z_loan(val)* macro, available if your compiler supports C11's *_Generic*, is equivalent to writing *z_X_loan(&val)*.

Like all *z_owned_X_t*, an instance will be destroyed by any function which takes a mutable pointer to said instance, as this implies the instance's inners were moved. To make this fact more obvious when reading your code, consider using *z_move(val)* instead of *&val* as the argument. After a move, *val* will still exist, but will no longer be valid. The destructors are double-drop-safe, but other functions will still trust that your *val* is valid.

To check if *val* is still valid, you may use *z_X_check(&val)* or *z_check(val)* if your compiler supports *_Generic*, which will return *true* if *val* is valid.

struct *z_owned_config_t* **z_config_default**()

Creates a default, zenoh-allocated, configuration.

struct *z_owned_config_t* **z_config_client**(const char *const *peers, uintptr_t n_peers)

Constructs a default, zenoh-allocated, client mode configuration. If *peer* is not null, it is added to the configuration as remote peer.

struct *z_owned_config_t* **z_config_peer**()

Constructs a default, zenoh-allocated, peer mode configuration.

struct *z_owned_config_t* **zc_config_from_file**(const char *path)

Constructs a configuration by parsing a file at *path*. Currently supported format is JSON5, a superset of JSON.

struct *z_owned_config_t* **zc_config_from_str**(const char *s)

Reads a configuration from a JSON-serialized string, such as '{mode:"client",connect:{endpoints:["tcp/127.0.0.1:7447"]}}'.

Passing a null-ptr will result in a gravestone value (*z_check(x)* == *false*).

int8_t **zc_config_insert_json**(struct z_config_t config, const char *key, const char *value)

Inserts a JSON-serialized *value* at the *key* position of the configuration.

Returns 0 if successful, a negative value otherwise.

char ***zc_config_get**(struct z_config_t config, const char *key)

Gets the property with the given path key from the configuration, returning an owned, null-terminated, JSON serialized string.

char ***zc_config_to_string**(struct z_config_t config)

Converts *config* into a JSON-serialized string, such as '{"mode":"client","connect":{"endpoints":["tcp/127.0.0.1:7447"]}}'.

struct z_config_t **z_config_loan**(const struct *z_owned_config_t* *s)

Returns a `z_config_t` loaned from *s*.

bool **z_config_check**(const struct *z_owned_config_t* *config)

Returns `true` if *config* is valid.

void **z_config_drop**(struct *z_owned_config_t* *config)

Frees *config*, invalidating it for double-drop safety.

2.2.2 Session management

2.2.2.1 Types

struct **z_session_t**

A loaned zenoh session.

struct **z_owned_session_t**

An owned zenoh session.

Like most *z_owned_X_t* types, you may obtain an instance of *z_X_t* by loaning it using *z_X_loan(&val)*. The *z_loan(val)* macro, available if your compiler supports C11's *_Generic*, is equivalent to writing *z_X_loan(&val)*.

Like all *z_owned_X_t*, an instance will be destroyed by any function which takes a mutable pointer to said instance, as this implies the instance's inners were moved. To make this fact more obvious when reading your code,

consider using *z_move(val)* instead of *&val* as the argument. After a move, *val* will still exist, but will no longer be valid. The destructors are double-drop-safe, but other functions will still trust that your *val* is valid.

To check if *val* is still valid, you may use *z_X_check(&val)* or *z_check(val)* if your compiler supports *_Generic*, which will return *true* if *val* is valid.

struct **z_owned_closure_zid_t**

A closure is a structure that contains all the elements for stateful, memory-leak-free callbacks:

void ***context**

a pointer to an arbitrary state.

void ***call**

the typical callback function. *context* will be passed as its last argument.

void ***drop**

allows the callback's state to be freed.

Closures are not guaranteed not to be called concurrently.

It is guaranteed that:

- *call* will never be called once *drop* has started.
- *drop* will only be called **once**, and **after every** *call* has ended.
- The two previous guarantees imply that *call* and *drop* are never called concurrently.

2.2.2.2 Functions

struct *z_owned_session_t* **z_open**(struct *z_owned_config_t* *config)

Opens a zenoh session. Should the session opening fail, *z_check* ing the returned value will return *false*.

int8_t **z_close**(struct *z_owned_session_t* *session)

Closes a zenoh session. This drops and invalidates *session* for double-drop safety.

struct *z_session_t* **z_session_loan**(const struct *z_owned_session_t* *s)

Returns a `z_session_t` loaned from *s*.

bool **z_session_check**(const struct *z_owned_session_t* *session)

Returns `true` if *session* is valid.

struct z_id_t **z_info_zid**(struct *z_session_t* session)

Returns the local Zenoh ID.

Unless the *session* is invalid, that ID is guaranteed to be non-zero. In other words, this function returning an array of 16 zeros means you failed to pass it a valid session.

int8_t **z_info_routers_zid**(struct *z_session_t* session, struct *z_owned_closure_zid_t* *callback)

Fetches the Zenoh IDs of all connected routers.

callback will be called once for each ID, is guaranteed to never be called concurrently, and is guaranteed to be dropped before this function exits.

Retuns 0 on success

int8_t **z_info_peers_zid**(struct *z_session_t* session, struct *z_owned_closure_zid_t* *callback)

Fetches the Zenoh IDs of all connected peers.

callback will be called once for each ID, is guaranteed to never be called concurrently, and is guaranteed to be dropped before this function exits.

Retuns 0 on success

void **z_closure_zid_call**(const struct *z_owned_closure_zid_t* *closure, const struct z_id_t *sample)

Calls the closure. Calling an uninitialized closure is a no-op.

void **z_closure_zid_drop**(struct *z_owned_closure_zid_t* *closure)

Drops the closure. Droping an uninitialized closure is a no-op.

2.3 Key expression

struct **z_keyexpr_t**

A loaned key expression.

Key expressions can identify a single key or a set of keys.

Examples :

- `"key/expression"`.
- `"key/ex*"`.

Using `z_declare_keyexpr()` allows zenoh to optimize a key expression, both for local processing and network-wise.

struct **z_owned_keyexpr_t**

A zenoh-allocated key expression.

Key expressions can identify a single key or a set of keys.

Examples :

- `"key/expression"`.
- `"key/ex*"`.

Key expressions can be mapped to numerical ids through `z_declare_expr()` for wire and computation efficiency.

A [key expression](https://github.com/eclipse-zenoh/roadmap/blob/main/rfcs/ALL/Key%20Expressions.md) can be eithe

- A plain string expression.
- A pure numerical id.
- The combination of a numerical prefix and a string suffix.

Like most *z_owned_X_t* types, you may obtain an instance of *z_X_t* by loaning it using *z_X_loan(&val)*. The *z_loan(val)* macro, available if your compiler supports C11's *_Generic*, is equivalent to writing *z_X_loan(&val)*.

Like all *z_owned_X_t*, an instance will be destroyed by any function which takes a mutable pointer to said instance, as this implies the instance's inners were moved. To make this fact more obvious when reading your code, consider using *z_move(val)* instead of *&val* as the argument. After a move, *val* will still exist, but will no longer be valid. The destructors are double-drop-safe, but other functions will still trust that your *val* is valid.

To check if *val* is still valid, you may use *z_X_check(&val)* or *z_check(val)* if your compiler supports *_Generic*, which will return *true* if *val* is valid.

struct *z_keyexpr_t* **z_keyexpr**(const char *name)

Constructs a `z_keyexpr_t` departing from a string. It is a loaned key expression that aliases *name*.

struct *z_keyexpr_t* **z_keyexpr_unchecked**(const char *name)

Constructs a `z_keyexpr_t` departing from a string without checking any of *z_keyexpr_t*'s assertions:

- *name* MUST be valid UTF8.
- *name* MUST follow the Key Expression specification, ie:
- MUST NOT contain *//*, MUST NOT start nor end with */*, MUST NOT contain any of the characters *?#$*.
- any instance of ** may only be lead or followed by */*.
- the key expression must have canon form.

It is a loaned key expression that aliases *name*.

char ***z_keyexpr_to_string**(struct *z_keyexpr_t* keyexpr)

Constructs a null-terminated string departing from a `z_keyexpr_t`. The user is responsible of droping the returned string using libc's *free*.

struct *z_bytes_t* **z_keyexpr_as_bytes**(struct *z_keyexpr_t* keyexpr)

Returns the key expression's internal string by aliasing it.

Currently exclusive to zenoh-c

int8_t **z_keyexpr_canonize**(char *start, uintptr_t *len)

Canonizes the passed string in place, possibly shortening it by modifying *len*.

Returns `0` upon success. Returns a negative value if canonization failed, which indicates that the passed string was an invalid key expression for reasons other than a non-canon form.

May SEGFAULT if *start* is NULL or lies in read-only memory (as values initialized with string litterals do).

int8_t **z_keyexpr_canonize_null_terminated**(char *start)

Canonizes the passed string in place, possibly shortening it by placing a new null-terminator.

Returns `0` upon success. Returns a negative value if canonization failed, which indicates that the passed string was an invalid key expression for reasons other than a non-canon form.

May SEGFAULT if *start* is NULL or lies in read-only memory (as values initialized with string litterals do).

int8_t **z_keyexpr_is_canon**(const char *start, uintptr_t len)

Returns `0` if the passed string is a valid (and canon) key expression.

bool **z_keyexpr_is_initialized**(const struct *z_keyexpr_t* *keyexpr)

Returns `true` if *keyexpr* is initialized.

struct *z_owned_keyexpr_t* **z_keyexpr_concat**(struct *z_keyexpr_t* left, const char *right_start, uintptr_t right_len)

Performs string concatenation and returns the result as a *z_owned_keyexpr_t*. In case of error, the return value will be set to its invalidated state.

You should probably prefer *z_keyexpr_join* as Zenoh may then take advantage of the hierachical separation it inserts.

To avoid odd behaviors, concatenating a key expression starting with * to one ending with * is forbidden by this operation, as this would extremely likely cause bugs.

struct *z_owned_keyexpr_t* **z_keyexpr_join**(struct *z_keyexpr_t* left, struct *z_keyexpr_t* right)

Performs path-joining (automatically inserting) and returns the result as a *z_owned_keyexpr_t*. In case of error, the return value will be set to its invalidated state.

int8_t **z_keyexpr_equals**(struct *z_keyexpr_t* left, struct *z_keyexpr_t* right)

Returns 1 if *left* and *right* define equal sets, `0` otherwise.

int8_t **z_keyexpr_includes**(struct *z_keyexpr_t* left, struct *z_keyexpr_t* right)

Returns 1 if the set defined by *left* contains every key belonging to the set defined by *right*, `0` if they don't. Returns negative values in case of error (if one of the key expressions is in an invalid state).

int8_t **z_keyexpr_intersects**(struct *z_keyexpr_t* left, struct *z_keyexpr_t* right)

Returns 1 if *left* and *right* define sets that have at least one key in common, `0` if they don't. Returns negative values in case of error (if one of the key expressions is in an invalid state).

struct *z_owned_keyexpr_t* **z_keyexpr_new**(const char *name)

Constructs a `z_keyexpr_t` departing from a string, copying the passed string.

struct *z_keyexpr_t* **z_keyexpr_loan**(const struct *z_owned_keyexpr_t* *keyexpr)

Returns a `z_keyexpr_t` loaned from `z_owned_keyexpr_t`.

bool **z_keyexpr_check**(const struct *z_owned_keyexpr_t* *keyexpr)

Returns `true` if *keyexpr* is valid.

void **z_keyexpr_drop**(struct *z_owned_keyexpr_t* *keyexpr)

Frees *keyexpr* and invalidates it for double-drop safety.

2.4 Encoding

struct **z_encoding_t**

The encoding of a payload, in a MIME-like format.

For wire and matching efficiency, common MIME types are represented using an integer as *prefix*, and a *suffix* may be used to either provide more detail, or in combination with the *Empty* prefix to write arbitrary MIME types.

z_encoding_prefix_t **prefix**

The integer prefix of this encoding.

z_bytes_t **suffix**

The suffix of this encoding. *suffix* MUST be a valid UTF-8 string.

struct **z_owned_encoding_t**

An owned payload encoding.

z_encoding_prefix_t **prefix**

The integer prefix of this encoding.

z_bytes_t **suffix**

The suffix of this encoding. *suffix* MUST be a valid UTF-8 string.

Like all *z_owned_X_t*, an instance will be destroyed by any function which takes a mutable pointer to said instance, as this implies the instance's inners were moved. To make this fact more obvious when reading your code, consider using *z_move(val)* instead of *&val* as the argument. After a move, *val* will still exist, but will no longer be valid. The destructors are double-drop-safe, but other functions will still trust that your *val* is valid.

To check if *val* is still valid, you may use *z_X_check(&val)* (or *z_check(val)* if your compiler supports *_Generic*), which will return *true* if *val* is valid.

struct *z_encoding_t* **z_encoding_default**()

Constructs a default *z_encoding_t*.

struct *z_encoding_t* **z_encoding_loan**(const struct *z_owned_encoding_t* *encoding)

Returns a *z_encoding_t* loaned from *encoding*.

bool **z_encoding_check**(const struct *z_owned_encoding_t* *encoding)

Returns `true` if *encoding* is valid.

void **z_encoding_drop**(struct *z_owned_encoding_t* *encoding)

Frees *encoding*, invalidating it for double-drop safety.

struct **z_encoding_prefix_t**

A *z_encoding_t* integer *prefix*.

- **Z_ENCODING_PREFIX_EMPTY**
- **Z_ENCODING_PREFIX_APP_OCTET_STREAM**
- **Z_ENCODING_PREFIX_APP_CUSTOM**
- **Z_ENCODING_PREFIX_TEXT_PLAIN**
- **Z_ENCODING_PREFIX_APP_PROPERTIES**
- **Z_ENCODING_PREFIX_APP_JSON**
- **Z_ENCODING_PREFIX_APP_SQL**
- **Z_ENCODING_PREFIX_APP_INTEGER**
- **Z_ENCODING_PREFIX_APP_FLOAT**
- **Z_ENCODING_PREFIX_APP_XML**
- **Z_ENCODING_PREFIX_APP_XHTML_XML**
- **Z_ENCODING_PREFIX_APP_X_WWW_FORM_URLENCODED**
- **Z_ENCODING_PREFIX_TEXT_JSON**
- **Z_ENCODING_PREFIX_TEXT_HTML**
- **Z_ENCODING_PREFIX_TEXT_XML**
- **Z_ENCODING_PREFIX_TEXT_CSS**
- **Z_ENCODING_PREFIX_TEXT_CSV**
- **Z_ENCODING_PREFIX_TEXT_JAVASCRIPT**
- **Z_ENCODING_PREFIX_IMAGE_JPEG**
- **Z_ENCODING_PREFIX_IMAGE_PNG**
- **Z_ENCODING_PREFIX_IMAGE_GIF**

2.5 Value

struct **z_value_t**

A zenoh value.

z_bytes_t **payload**

The payload of this zenoh value.

z_encoding_t **encoding**

The encoding of this zenoh value *payload*.

2.6 Sample

struct **z_sample_t**

A data sample.

A sample is the value associated to a given resource at a given point in time.

z_keyexpr_t **keyexpr**

The resource key of this data sample.

z_bytes_t **payload**

The value of this data sample.

z_encoding_t **encoding**

The encoding of the value of this data sample.

z_sample_kind_t **kind**

The kind of this data sample (PUT or DELETE).

z_timestamp_t **timestamp**

The timestamp of this data sample.

2.7 Publication

2.7.1 Types

struct **z_owned_publisher_t**

An owned zenoh publisher.

Like most *z_owned_X_t* types, you may obtain an instance of *z_X_t* by loaning it using *z_X_loan(&val)*. The *z_loan(val)* macro, available if your compiler supports C11's *_Generic*, is equivalent to writing *z_X_loan(&val)*.

Like all *z_owned_X_t*, an instance will be destroyed by any function which takes a mutable pointer to said instance, as this implies the instance's inners were moved. To make this fact more obvious when reading your code, consider using *z_move(val)* instead of *&val* as the argument. After a move, *val* will still exist, but will no longer be valid. The destructors are double-drop-safe, but other functions will still trust that your *val* is valid.

To check if *val* is still valid, you may use *z_X_check(&val)* or *z_check(val)* if your compiler supports *_Generic*, which will return *true* if *val* is valid.

struct **z_congestion_control_t**

The kind of congestion control.

- **BLOCK**
- **DROP**

struct **z_priority_t**

The priority of zenoh messages.

- **REAL_TIME**
- **INTERACTIVE_HIGH**
- **INTERACTIVE_LOW**
- **DATA_HIGH**
- **DATA**
- **DATA_LOW**
- **BACKGROUND**

struct **z_put_options_t**

Options passed to the *z_put()* function.

z_encoding_t **encoding**

The encoding of the payload.

z_congestion_control_t **congestion_control**

The congestion control to apply when routing this message.

z_priority_t **priority**

The priority of this message.

struct *z_put_options_t* **z_put_options_default**()

Constructs the default value for *z_put_options_t*.

struct **z_publisher_options_t**

Options passed to the *z_declare_publisher()* function.

z_congestion_control_t **congestion_control**

The congestion control to apply when routing messages from this publisher.

z_priority_t **priority**

The priority of messages from this publisher.

struct *z_publisher_options_t* **z_publisher_options_default**()

Constructs the default value for *z_publisher_options_t*.

struct **z_publisher_put_options_t**

Options passed to the *z_publisher_put()* function.

z_encoding_t **encoding**

The encoding of the payload.

2.7.2 Functions

int **z_put**(struct *z_session_t* session, struct *z_keyexpr_t* keyexpr, const uint8_t *payload, size_t len, const struct *z_put_options_t* *opts)

Put data.

The payload's encoding can be sepcified through the options.

Parameters

- **session** – The zenoh session.
- **keyexpr** – The key expression to put.
- **payload** – The value to put.
- **len** – The length of the value to put.
- **options** – The put options.

Returns `0` in case of success, 1 in case of failure.

struct *z_owned_publisher_t* **z_declare_publisher**(struct *z_session_t* session, struct *z_keyexpr_t* keyexpr, const struct *z_publisher_options_t* *options)

Declares a publisher for the given key expression.

Data can be put and deleted with this publisher with the help of the *z_publisher_put()* and *z_publisher_delete()* functions.

Parameters

- **session** – The zenoh session.
- **keyexpr** – The key expression to publish.
- **options** – additional options for the publisher.

Returns

A `z_owned_publisherr_t`.

To check if the publisher decalration succeeded and if the publisher is still valid, you may use *z_publisher_check(&val)* or *z_check(val)* if your compiler supports *_Generic*, which will return *true* if *val* is valid.

Like all *z_owned_X_t*, an instance will be destroyed by any function which takes a mutable pointer to said instance, as this implies the instance's inners were moved. To make this fact more obvious when reading your code, consider using *z_move(val)* instead of *&val* as the argument. After a move, *val* will still exist, but will no longer be valid. The destructors are double-drop-safe, but other functions will still trust that your *val* is valid.

Example

Declaring a publisher passing *NULL* for the options:

```
z_owned_publisher_t pub = z_declare_publisher(z_loan(s), z_keyexpr(expr), NULL);
```

is equivalent to initializing and passing the default publisher options:

```
z_publisher_options_t opts = z_publisher_options_default();
z_owned_publisher_t sub = z_declare_publisher(z_loan(s), z_keyexpr(expr), &opts);
```

int8_t **z_publisher_put**(struct z_publisher_t publisher, const uint8_t *payload, uintptr_t len, const struct *z_publisher_put_options_t* *options)

Sends a *PUT* message onto the publisher's key expression.

The payload's encoding can be sepcified through the options.

Parameters

- **session** – The zenoh session.
- **payload** – The value to put.
- **len** – The length of the value to put.
- **options** – The publisher put options.

Returns `0` in case of success, 1 in case of failure.

int8_t **z_publisher_delete**(struct z_publisher_t publisher, const struct z_publisher_delete_options_t *_options)

Sends a *DELETE* message onto the publisher's key expression.

Returns `0` in case of success, 1 in case of failure.

int8_t **z_undeclare_publisher**(struct *z_owned_publisher_t* *publisher)

Undeclares the given *z_owned_publisher_t*, droping it and invalidating it for double-drop safety.

2.8 Subscription

2.8.1 Types

struct **z_owned_subscriber_t**

An owned zenoh subscriber. Destroying the subscriber cancels the subscription.

Like most *z_owned_X_t* types, you may obtain an instance of *z_X_t* by loaning it using *z_X_loan(&val)*. The *z_loan(val)* macro, available if your compiler supports C11's *_Generic*, is equivalent to writing *z_X_loan(&val)*.

Like all *z_owned_X_t*, an instance will be destroyed by any function which takes a mutable pointer to said instance, as this implies the instance's inners were moved. To make this fact more obvious when reading your code, consider using *z_move(val)* instead of *&val* as the argument. After a move, *val* will still exist, but will no longer be valid. The destructors are double-drop-safe, but other functions will still trust that your *val* is valid.

To check if *val* is still valid, you may use *z_X_check(&val)* or *z_check(val)* if your compiler supports *_Generic*, which will return *true* if *val* is valid.

struct **z_owned_pull_subscriber_t**

An owned zenoh pull subscriber. Destroying the subscriber cancels the subscription.

Like most *z_owned_X_t* types, you may obtain an instance of *z_X_t* by loaning it using *z_X_loan(&val)*. The *z_loan(val)* macro, available if your compiler supports C11's *_Generic*, is equivalent to writing *z_X_loan(&val)*.

Like all *z_owned_X_t*, an instance will be destroyed by any function which takes a mutable pointer to said instance, as this implies the instance's inners were moved. To make this fact more obvious when reading your code, consider using *z_move(val)* instead of *&val* as the argument. After a move, *val* will still exist, but will no longer be valid. The destructors are double-drop-safe, but other functions will still trust that your *val* is valid.

To check if *val* is still valid, you may use *z_X_check(&val)* or *z_check(val)* if your compiler supports *_Generic*, which will return *true* if *val* is valid.

struct **z_owned_closure_sample_t**

A closure is a structure that contains all the elements for stateful, memory-leak-free callbacks.

void ***context**

a pointer to an arbitrary state.

void ***call**

the typical callback function. *context* will be passed as its last argument.

void ***drop**

allows the callback's state to be freed.

Closures are not guaranteed not to be called concurrently.

It is guaranteed that:

- *call* will never be called once *drop* has started.
- *drop* will only be called **once**, and **after every** *call* has ended.
- The two previous guarantees imply that *call* and *drop* are never called concurrently.

enum **z_reliability_t**

The subscription reliability.

- **Z_RELIABILITY_BEST_EFFORT**
- **Z_RELIABILITY_RELIABLE**

struct **z_subscriber_options_t**

Options passed to the *z_declare_subscriber()* or *z_declare_pull_subscriber()* function.

z_reliability_t **reliability**

The subscription reliability.

struct *z_subscriber_options_t* **z_subscriber_options_default**()

Constructs the default value for *z_subscriber_options_t*.

2.8.2 Functions

struct *z_owned_subscriber_t* **z_declare_subscriber**(struct *z_session_t* session, struct *z_keyexpr_t* keyexpr, struct *z_owned_closure_sample_t* *callback, const struct *z_subscriber_options_t* *opts)

Declare a subscriber for a given key expression.

Parameters

- **session** – The zenoh session.
- **keyexpr** – The key expression to subscribe.
- **callback** – The callback function that will be called each time a data matching the subscribed expression is received.
- **opts** – The options to be passed to describe the options to be passed to the subscriber declaration.

Returns

A *z_owned_subscriber_t*.

To check if the subscription succeeded and if the subscriber is still valid, you may use *z_subscriber_check(&val)* or *z_check(val)* if your compiler supports *_Generic*, which will return *true* if *val* is valid.

Like all *z_owned_X_t*, an instance will be destroyed by any function which takes a mutable pointer to said instance, as this implies the instance's inners were moved. To make this fact more obvious when reading your code, consider using *z_move(val)* instead of *&val* as the argument. After a move, *val* will still exist, but will no longer be valid. The destructors are double-drop-safe, but other functions will still trust that your *val* is valid.

Example

Declaring a subscriber passing *NULL* for the options:

```
z_owned_subscriber_t sub = z_declare_subscriber(z_loan(s), z_keyexpr(expr),␣
↪callback, NULL);
```

is equivalent to initializing and passing the default subscriber options:

```
z_subscriber_options_t opts = z_subscriber_options_default();
z_owned_subscriber_t sub = z_declare_subscriber(z_loan(s), z_keyexpr(expr),␣
↪callback, &opts);
```

Passing custom arguments to the **callback** can be done by defining a custom structure:

```
typedef struct {
  z_keyexpr_t forward;
  z_session_t session;
} myargs_t;

void callback(const z_sample_t sample, const void *arg)
{
  myargs_t *myargs = (myargs_t *)arg;
  z_put(myargs->session, myargs->forward, sample->value, NULL);
}

int main() {
  myargs_t cargs = {
    forward = z_keyexpr("forward"),
    session = s,
  };
  z_subscriber_options_t opts = z_subscriber_options_default();
  opts.cargs = (void *)&cargs;
  z_owned_subscriber_t sub = z_declare_subscriber(z_loan(s), z_keyexpr(expr),␣
↪callback, &opts);
}
```

bool **z_subscriber_check**(const struct *z_owned_subscriber_t* *sub)

Returns `true` if *sub* is valid.

int8_t **z_undeclare_subscriber**(struct *z_owned_subscriber_t* *sub)

Undeclares the given *z_owned_subscriber_t*, droping it and invalidating it for double-drop safety.

struct *z_owned_pull_subscriber_t* **z_declare_pull_subscriber**(struct *z_session_t* session, struct *z_keyexpr_t* keyexpr, struct *z_owned_closure_sample_t* *callback, const struct z_pull_subscriber_options_t *opts)

Declares a pull subscriber for a given key expression.

Parameters

- **session** – The zenoh session.
- **keyexpr** – The key expression to subscribe.
- **callback** – The callback function that will be called each time a data matching the subscribed expression is received.
- **opts** – additional options for the pull subscriber.

Returns

A *z_owned_subscriber_t*.

To check if the subscription succeeded and if the pull subscriber is still valid, you may use *z_pull_subscriber_check(&val)* or *z_check(val)* if your compiler supports *_Generic*, which will return *true* if *val* is valid.

Like all *z_owned_X_t*, an instance will be destroyed by any function which takes a mutable pointer to said instance, as this implies the instance's inners were moved. To make this fact more obvious when reading your code, consider using *z_move(val)* instead of *&val* as the argument. After a move, *val* will still exist, but will no longer be valid. The destructors are double-drop-safe, but other functions will still trust that your *val* is valid.

Example

Declaring a subscriber passing `NULL` for the options:

```
z_owned_subscriber_t sub = z_declare_pull_subscriber(z_loan(s), z_keyexpr(expr),␣
↪callback, NULL);
```

is equivalent to initializing and passing the default subscriber options:

```
z_subscriber_options_t opts = z_subscriber_options_default();
z_owned_subscriber_t sub = z_declare_pull_subscriber(z_loan(s), z_keyexpr(expr),␣
↪callback, &opts);
```

Passing custom arguments to the **callback** can be done by defining a custom structure:

```
typedef struct {
  z_keyexpr_t forward;
  z_session_t session;
} myargs_t;

void callback(const z_sample_t sample, const void *arg)
{
  myargs_t *myargs = (myargs_t *)arg;
  z_put(myargs->session, myargs->forward, sample->value, NULL);
}
```

(continues on next page)

(continued from previous page)

```
int main() {
  myargs_t cargs = {
    forward = z_keyexpr("forward"),
    session = s,
  };
  z_pull_subscriber_options_t opts = z_pull_subscriber_options_default();
  opts.cargs = (void *)&cargs;
  z_owned_pull_subscriber_t sub = z_declare_pull_subscriber(z_loan(s), z_
↪keyexpr(expr), callback, &opts);
}
```

int8_t **z_subscriber_pull**(struct z_pull_subscriber_t sub)

Pull data for *z_owned_pull_subscriber_t*. The pulled data will be provided by calling the **callback** function provided to the *z_declare_subscriber()* function.

Parameters

- **sub** – The *z_owned_pull_subscriber_t* to pull from.

bool **z_pull_subscriber_check**(const struct *z_owned_pull_subscriber_t* *sub)

Returns `true` if *sub* is valid.

int8_t **z_undeclare_pull_subscriber**(struct *z_owned_pull_subscriber_t* *sub)

Undeclares the given *z_owned_pull_subscriber_t*, droping it and invalidating it for double-drop safety.

void **z_closure_sample_call**(const struct *z_owned_closure_sample_t* *closure, const struct *z_sample_t* *sample)

Calls the closure. Calling an uninitialized closure is a no-op.

void **z_closure_sample_drop**(struct *z_owned_closure_sample_t* *closure)

Drops the closure. Droping an uninitialized closure is a no-op.

2.9 Query

2.9.1 Types

enum **z_query_target_t**

The Queryables that should be target of a *z_get()*.

- **BEST_MATCHING**: The nearest complete queryable if any else all matching queryables.
- **ALL_COMPLETE**: All complete queryables.
- **ALL**: All matching queryables.

struct **z_owned_closure_reply_t**

A closure is a structure that contains all the elements for stateful, memory-leak-free callbacks:

void ***context**

a pointer to an arbitrary state.

void ***call**

the typical callback function. *context* will be passed as its last argument.

void *drop

allows the callback's state to be freed.

Closures are not guaranteed not to be called concurrently.

It is guaranteed that:

- *call* will never be called once *drop* has started.
- *drop* will only be called **once**, and **after every** *call* has ended.
- The two previous guarantees imply that *call* and *drop* are never called concurrently.

enum z_consolidation_mode_t

Consolidation mode values.

- **Z_CONSOLIDATION_MODE_AUTO**

  Let Zenoh decide the best consolidation mode depending on the query selector If the selector contains time range properties, consolidation mode *NONE* is used. Otherwise the *LATEST* consolidation mode is used.

- **Z_CONSOLIDATION_MODE_NONE**

  No consolidation is applied. Replies may come in any order and any number.

- **Z_CONSOLIDATION_MODE_MONOTONIC**

  It guarantees that any reply for a given key expression will be monotonic in time w.r.t. the previous received replies for the same key expression. I.e., for the same key expression multiple replies may be received. It is guaranteed that two replies received at t1 and t2 will have timestamp ts2 > ts1. It optimizes latency.

- **Z_CONSOLIDATION_MODE_LATEST**

  It guarantees unicity of replies for the same key expression. It optimizes bandwidth.

type z_query_consolidation_t

The replies consolidation strategy to apply on replies to a *z_get()*.

- **AUTO**: Automatic query consolidation strategy selection.
- **MANUAL**: Manual query consolidation strategy selection.

struct *z_query_consolidation_t* z_query_consolidation_default()

Creates a default *z_query_consolidation_t* (consolidation mode AUTO).

struct *z_query_consolidation_t* z_query_consolidation_auto()

Automatic query consolidation strategy selection.

A query consolidation strategy will automatically be selected depending the query selector. If the selector contains time range properties, no consolidation is performed. Otherwise the *z_query_consolidation_latest()* strategy is used.

Returns Returns the constructed *z_query_consolidation_t*.

struct *z_query_consolidation_t* z_query_consolidation_none()

Disable consolidation.

struct *z_query_consolidation_t* z_query_consolidation_monotonic()

Monotonic consolidation.

struct *z_query_consolidation_t* z_query_consolidation_latest()

Latest value consolidation.

struct **z_owned_reply_t**

An owned reply to a *z_get()*.

Like all *z_owned_X_t*, an instance will be destroyed by any function which takes a mutable pointer to said instance, as this implies the instance's inners were moved. To make this fact more obvious when reading your code, consider using *z_move(val)* instead of *&val* as the argument. After a move, *val* will still exist, but will no longer be valid. The destructors are double-drop-safe, but other functions will still trust that your *val* is valid.

To check if *val* is still valid, you may use *z_X_check(&val)* (or *z_check(val)* if your compiler supports *_Generic*), which will return *true* if *val* is valid.

bool **z_reply_check**(const struct *z_owned_reply_t* *reply_data)

Returns `true` if *reply_data* is valid.

void **z_reply_drop**(struct *z_owned_reply_t* *reply_data)

Frees *reply_data*, invalidating it for double-drop safety.

2.9.2 Functions

bool **z_get**(struct *z_session_t* session, struct *z_keyexpr_t* keyexpr, const char *parameters, struct *z_owned_closure_reply_t* *callback, const struct z_get_options_t *options)

Query data from the matching queryables in the system. Replies are provided through a callback function.

Parameters

- **session** – The zenoh session.
- **keyexpr** – The key expression matching resources to query.
- **parameters** – The query's parameters, similar to a url's query segment.
- **callback** – The callback function that will be called on reception of replies for this query. Note that the *reply* parameter of the callback is passed by mutable reference, but **will** be dropped once your callback exits to help you avoid memory leaks. If you'd rather take ownership, please refer to the documentation of *z_reply_null()*
- **options** – additional options for the get.

bool **z_reply_is_ok**(const struct *z_owned_reply_t* *reply)

Returns `true` if the queryable answered with an OK, which allows this value to be treated as a sample.

If this returns `false`, you should use `z_check()` before trying to use *z_reply_err()* if you want to process the error that may be here.

struct *z_sample_t* **z_reply_ok**(const struct *z_owned_reply_t* *reply)

Yields the contents of the reply by asserting it indicates a success.

You should always make sure that *z_reply_is_ok()* returns `true` before calling this function.

struct *z_value_t* **z_reply_err**(const struct *z_owned_reply_t* *reply)

Yields the contents of the reply by asserting it indicates a failure.

You should always make sure that *z_reply_is_ok()* returns `false` before calling this function.

struct *z_owned_reply_t* **z_reply_null**()

Returns an invalidated *z_owned_reply_t*.

This is useful when you wish to take ownership of a value from a callback to *z_get()*:

- copy the value of the callback's argument's pointee,

- overwrite the pointee with this function's return value,
- you are now responsible for dropping your copy of the reply.

void **z_closure_reply_call**(const struct *z_owned_closure_reply_t* *closure, struct *z_owned_reply_t* *sample)

Calls the closure. Calling an uninitialized closure is a no-op.

void **z_closure_reply_drop**(struct *z_owned_closure_reply_t* *closure)

Drops the closure. Droping an uninitialized closure is a no-op.

2.10 Queryable

2.10.1 Types

struct **z_owned_queryable_t**

An owned zenoh queryable.

Like most *z_owned_X_t* types, you may obtain an instance of *z_X_t* by loaning it using *z_X_loan(&val)*. The *z_loan(val)* macro, available if your compiler supports C11's *_Generic*, is equivalent to writing *z_X_loan(&val)*.

Like all *z_owned_X_t*, an instance will be destroyed by any function which takes a mutable pointer to said instance, as this implies the instance's inners were moved. To make this fact more obvious when reading your code, consider using *z_move(val)* instead of *&val* as the argument. After a move, *val* will still exist, but will no longer be valid. The destructors are double-drop-safe, but other functions will still trust that your *val* is valid.

To check if *val* is still valid, you may use *z_X_check(&val)* or *z_check(val)* if your compiler supports *_Generic*, which will return *true* if *val* is valid.

struct **z_owned_closure_query_t**

A closure is a structure that contains all the elements for stateful, memory-leak-free callbacks:

void ***context**

a pointer to an arbitrary state.

void ***call**

the typical callback function. *context* will be passed as its last argument.

void ***drop**

allows the callback's state to be freed.

Closures are not guaranteed not to be called concurrently.

It is guaranteed that:

- *call* will never be called once *drop* has started.
- *drop* will only be called **once**, and **after every** *call* has ended.
- The two previous guarantees imply that *call* and *drop* are never called concurrently.

struct *z_keyexpr_t* **z_query_keyexpr**(const struct z_query_t *query)

Get a query's key by aliasing it.

struct *z_bytes_t* **z_query_parameters**(const struct z_query_t *query)

Get a query's [value selector](https://github.com/eclipse-zenoh/roadmap/tree/main/rfcs/ALL/Selectors) by aliasing it.

2.10.2 Functions

struct *z_owned_queryable_t* **z_declare_queryable**(struct *z_session_t* session, struct *z_keyexpr_t* keyexpr, struct *z_owned_closure_query_t* *callback, const struct z_queryable_options_t *options)

Creates a Queryable for the given key expression.

Parameters

- **session** – The zenoh session.
- **keyexpr** – The key expression the Queryable will reply to.
- **callback** – The callback function that will be called each time a matching query is received.
- **options** – Options for the queryable.

Returns The created *z_owned_queryable_t* or `null` if the creation failed.

void **z_query_reply**(const struct z_query_t *query, struct *z_keyexpr_t* key, const uint8_t *payload, uintptr_t len, const struct z_query_reply_options_t *options)

Send a reply to a query.

This function must be called inside of a Queryable callback passing the query received as parameters of the callback function. This function can be called multiple times to send multiple replies to a query. The reply will be considered complete when the Queryable callback returns.

Parameters

- **query** – The query to reply to.
- **key** – The key of this reply.
- **payload** – The value of this reply.
- **len** – The length of the value of this reply.
- **options** – The options of this reply.

bool **z_queryable_check**(const struct *z_owned_queryable_t* *qable)

Returns `true` if *qable* is valid.

int8_t **z_undeclare_queryable**(struct *z_owned_queryable_t* *qable)

Undeclares a *z_owned_queryable_t*, droping it and invalidating it for doube-drop safety.

Parameters

- **qable** – The *z_owned_queryable_t* to undeclare.

void **z_closure_query_call**(const struct *z_owned_closure_query_t* *closure, const struct z_query_t *query)

Calls the closure. Calling an uninitialized closure is a no-op.

void **z_closure_query_drop**(struct *z_owned_closure_query_t* *closure)

Drops the closure. Droping an uninitialized closure is a no-op.

INDEX

Z