zproc Documentation

dev aggarwal

Oct 28, 2018

Contents

CHAPTER 1

The User Guide

This part of the documentation, which is mostly prose, begins with some background information about ZProc, then focuses on step-by-step instructions for getting the most out of ZProc.

1.1 Introduction to ZProc

The whole architecture of zproc is built around a *State* object.

Context is provided as a convenient wrapper over *Process* and *State*.

It's the most obvious way to launch processes with zproc.

Each *Context* object is associated with a state; accessible by its processes.

Here's how you create a *Context*.

```
import zproc

ctx = zproc.Context()
```

1.1.1 Launching a Process

Lets launch a process that does nothing.

```
def my_process(state):
    pass

ctx.process(my_process)
```

The *process()* will launch a process, and provide it with `state`.

If you like to be cool, then you can use it as a decorator. (*process()* works both as a function, and decorator)

```
@ctx.process
def my_process(state):
    pass
```

The `state` is a *dict-like* object.

dict-like, because it's not exactly a dict. It provides a `dict` interface, but is actually just passing messages.

You *cannot* mutate the underlying `dict` directly. It's protected by a Process whose sole job is to manage it.

You can also access it from the *`Context`* itself using `ctx.state`.

```
state['apples'] = 5

state.get('apples')

state.setdefault('apples', 10)

...
```

1.1.2 Providing arguments to a Process

To provide some initial values to a Process, you can use use *args and **kwargs.

```
def my_process(state, num, exp):
    print(num, exp)  # 2, 4

ctx.process(my_process, args=[2], kwargs={'exp': 4})
```

1.1.3 Waiting for a Process

Once you've launched a Process, you can wait for it to complete, and get it's return value like this:

```
from time import sleep


@ctx.process
def my_process(state):
    sleep(5)
    return 'Hello There!'


print(my_process.wait())    # Hello There!
```

1.1.4 Process Factory

1.1.5 Process Map

Python's inbuilt `multiprocessing.Pool` let's you use the in-built *map()* function in a parallel way.

However, it gets quite finicky to use for anything serious.

That's why ZProc provides a more powerful construct, *`process_map()`* for mapping iterables to processes.

Listing 1: Works similar to `map()`

```
def square(num):
    return num * num

# [1, 4, 9, 16]
list(ctx.process_map(square, [1, 2, 3, 4]))
```

Listing 2: Common Arguments.

```
def power(num, exp):
    return num ** exp

# [0, 1, 8, 27, 64, ... 941192, 970299]
list(
     ctx.process_map(
        power,
        range(100),
        args=[3],
        count=10  # distribute among 10 workers.
     )
)
```

Listing 3: Mapped Positional Arguments.

```
def power(num, exp):
    return num ** exp

# [4, 9, 36, 256]
list(
    ctx.process_map(
        power,
        map_args=[(2, 2), (3, 2), (6, 2), (2, 8)]
    )
)
```

Listing 4: Mapped Keyword Arguments.

```
def my_thingy(seed, num, exp):
    return seed + num ** exp

# [1007, 3132, 298023223876953132, 736, 132, 65543, 8]
list(
    ctx.process_map(
        my_thingy,
        args=[7],
        map_kwargs=[
            {'num': 10, 'exp': 3},
            {'num': 5, 'exp': 5},
            {'num': 5, 'exp': 2},
            {'num': 9, 'exp': 3},
            {'num': 5, 'exp': 3},
            {'num': 4, 'exp': 8},
            {'num': 1, 'exp': 4},
        ],
        count=5
```

(continues on next page)

(continued from previous page)

```
    )
)
```

What's really cool about the process map is that it returns a generator.

The moment you call it, it will distribute the task to "count" number of workers.

It will then, return with a generator, which in-turn will do the job of pulling in the results from these workers, and arranging them in order.

```
>>> import zproc
>>> import time
```

```
>>> ctx = zproc.Context()
```

```
>>> def my_blocking_thingy(x):
...     time.sleep(5)
...
...     return x * x
...
```

```
>>> res = ctx.process_map(my_blocking_thingy, range(10))  # returns immediately
>>> res
<generator object Context._pull_results_for_task at 0x7fef735e6570>
```

```
>>> next(res)  # might block
0
>>> next(res)  # might block
1
>>> next(res)  # might block
4
>>> next(res)  # might block
9
>>> next(res)  # might block
16
...
```

It is noteworthy, that computation continues in the background while the main process is running.

As a result, the amount of time it takes for `next(res)` to return changes over time.

1.1.6 Reactive programming with zproc

This is the part where you really start to see the benefits of a smart state. The state knows when it's being mutated, and does the job of notifying everyone.

I like to call it *The magic of state watching*.

—

State watching allows you to react to some change in the state in an efficient way.

Lets say, you want to wait for the number of "cookies" to be "5".

Normally, you might do it with something like this:

```
while True:
    if cookies == 5:
        print('done!')
        break
```

But then you find out that this eats too much CPU, and put put some sleep.

```
from time import sleep

while True:
    if cookies == 5:
        print('done!')
        break
    sleep(1)
```

And from there on, you try to manage the time for which your application sleeps (to arrive at a sweet spot).

zproc provides an elegant, easy to use solution to this problem.

```
def my_process(state):
    state.get_when_equal('cookies', 5)
    print('done with zproc!')
```

This eats very little to no CPU, and is fast enough for almost everyone needs.

You must realise that this doesn't do any of that expensive "busy" waiting. Under the covers, it's actually just a socket waiting for a request.

If you want, you can even provide a function:

```
def my_process(state):
    state.get_when(lambda state: state.get('cookies') == 5)
```

The function you provide will get called on each state update, to check whether the return value is *truthy*.

Caution: You can't do things like this:

```
from time import time

t = time()
state.get_when(lambda state: time() > t + 5)  # wrong!
```

The State responds to *state* changes. Changing time doesn't signify a state update.

1.1.7 Mutating objects inside state

You must remember that you can't mutate (update) objects deep inside the state.

```
state['numbers'] = [1, 2, 3]  # works

state['numbers'].append(4)  # doesn't work
```

While this might look like a flaw of zproc (and it somewhat is), you can see this as a feature. It will avoid you from

1. over-complicating your state. (Keeping the state as flat as possible is generally a good idea).
2. avoiding race conditions. (Think about the atomicity of `state['numbers'].append(4)`).

The correct way to mutate objects inside the state, is to do them atomically, which is to say using the *atomic()* decorator.

```
@zproc.atomic
def add_a_number(state, to_add)
    state['numbers'].append(to_add)


def my_process(state):
    add_a_number(state, 4)
```

Read more about *Atomicity and race conditions*.

1.1.8 Something to keep in mind

Absolutely none of the the classes in ZProc are Process/Thread safe. You must never attempt to share a Context/State between multiple processes.

Create a new one for each Process/Thread. Communicate and synchronize using the State at all times.

This is also, in-general *very* good practice.

Never attempt to directly share python objects between Processes, and the multitasking gods will reward you :).

1.2 Atomicity and race conditions

When writing parallel code, one has to think about atomicity.

If an operation is atomic, then it means that the operation is indivisible, just like an atom.

If an operation can be divided into pieces, then processes might jump in and out between the pieces and try to meddle with each others' work, confusing everyone.

While zproc does provide mechanisms to avoid these kind of race conditions, it is ultimately up-to you to figure out whether an operation is atomic or not.

zproc **guarantees™** that a single method call on a `dict` is atomic.

This takes out a lot of guesswork in determining the atomicity of an operation.

Just think in terms of `dict` methods.

1.2.1 Example

```
def increment(state, step):
    state['count'] += step

increment(state, 5)
```

`increment()` might look like a single operation, but don't get fooled! (They're 2)

1. get `'count'`, a.k.a. `dict.__getitem__('count')`
2. set `'count'` to `count + 1`, a.k.a. `dict.__setitem__('count', count + 1)`

`dict.__getitiem__()` and `dict.__setitem__()` are **guarateed™** to be atomic on their own, but NOT in conjunction.

If these operations are not done atomically, it exposes the possibility of other Processes trying to do operations between "1" and "2"

---

zproc makes it dead simple to avoid such race conditions.

Let's make some changes to our example..

```
@zproc.atomic
def increment(state, step):
    state['count'] += step

increment(state, 5)
```

atomic() transforms any arbitrary function into an atomic operation on the state.

—

This is different from traditional locks. Locks are just flags. This on the other hand, is a hard restriction on the state.

Keep in mind, you still have to identify the critical points where a race condition can occur, and prevent it using *atomic()*.

However, this fundamental difference between locks and *atomic()* makes it easier to write safe and correct parallel code.

For what it's worth, If an error shall occur while the function is running, the state will remain *unaffected*.

Note: The first argument to the atomic function must be a *State* object.

Note: The `state` you get inside the atomic function is not a *State* object, but the complete underlying `dict` object.

SO, while you can't do cool stuff like state watching, you can freely mutate objects inside the state.

You're simply accessing the underlying `dict` object.

(This means, things like appending to a list will work)

<- full example

1.3 The magic of state watching

Watch the state for events, as-if you were watching a youtube video!

zproc allows you to *watch* the state using these methods, @ the *State* API.

- *get_when_change()*
- *get_when()*
- *get_when_equal()*
- *get_when_not_equal()*

- *get_when_none()*
- *get_when_not_none()*

For example, the following code will watch the state, and print out a message whenever the price of gold is below 40.

```
while True:
    snapshot = state.get_when(lambda state: state['gold_price'] < 40)

    print('"gold_price" is below 40!!:', snapshot['gold_price'])
```

—

There also these utility methods in *Context* that are just a wrapper over their counterparts in *State*.

- *call_when_change()*
- *call_when()*
- *call_when_equal()*
- *call_when_not_equal()*
- *call_when_none()*
- *call_when_not_none()*

For example, the function `want_pizza()` will be called every-time the `"num_pizza"` key in the state changes.

```
@ctx.call_when_change("num_pizza")
def want_pizza(snapshot, state):
    print("pizza be tasty!", snapshot['num_pizza'])
```

Note: All state-watchers are `KeyError` safe. That means, if the dict key you requested for isn't present, a `KeyError` won't be thrown.

1.3.1 Snapshots

All watchers provide return with a *snapshot* of the state, corresponding to the state-change for which the state watcher was triggered.

The *snapshot* is just a regular `dict` object.

In practice, this helps avoid race conditions – especially in cases where state keys are inter-dependent.

1.3.2 Duplicate-ness of events

1.3.3 Live-ness of events

zproc provides 2 different "modes" for watching the state.

By default, all state watchers will provide **buffered updates**.

Let us see what that exactly means, in detail.

Peanut generator

First, let us create a `Process` that will generate some peanuts, periodically.

```
from time import sleep
import zproc


ctx = zproc.Context()
state = ctx.state
state["peanuts"] = 0


@zproc.atomic
def inc_peanuts(state):
    state['peanuts'] += 1


@ctx.process
def peanut_gen(state):
    while True:
        inc_peanuts(state)
        sleep(1)
```

Live consumer

```
while True:
    num = state.get_when_change("peanuts", live=True)
    print("live consumer got:", num)

    sleep(2)
```

The above code will miss any updates that happen while it is sleeping (`sleep(2)`).

When consuming live updates, your code **can miss events**, if it's not paying attention.

like a live youtube video, you only see what's currently happening.

Buffered consumer

To modify this behaviour, you need to pass `live=False`.

```
while True:
    num = state.get_when_change("peanuts", live=False)
    print("non-live consumer got:", num)

    sleep(2)
```

This way, the events are stored in a *queue*, so that your code **doesn't miss any events**.

like a normal youtube video, where you won't miss anything, since it's buffering.

Hybrid consumer

But a live youtube video can be buffered as well!

Hence the need for a *go_live()* method.

It *clears* the outstanding queue (or buffer) – deleting all previous events.

That's somewhat like the "LIVE" button on a live stream, that skips ahead to the live broadcast.

```
while True:
    num = state.get_when_change("peanuts", live=False)
    print("hybrid consumer got:", num)

    state.go_live()

    sleep(2)
```

Note: *go_live()* only affects the behavior when `live` is set to `False`.

Has no effect when `live` is set to `True`.

A **live** state watcher is strictly **LIVE**.

A Full Example is available here.

Decision making

Its easy to decide whether you need live updates or not.

- If you don't care about missing an update or two, and want the most up-to date state, use live mode.
- If you care about each state update, at the cost of speed, and the recency of the updates, don't use live mode.

Live mode is obviously faster (potentially), since it can miss an update or two, which eventually trickles down to less computation.

1.3.4 Timeouts

You can also provide timeouts while watching the state, using `timeout` parameter.

If an update doesn't occur within the specified timeout, a `TimeoutError` is raised.

```
try:
    print(state.get_when_change(timeout=5))  # wait 5 seconds for an update
except TimeoutError:
    print('Waited too long!)
```

1.3.5 Button Press

Let's take an example, to put what we learned into real world usage.

Here, we want to watch a button press, and determine whether it was a long or a short press.

Some assumptions:

- If the value of `'button'` is `True`, the the button is pressed
- If the value of `'button'` is `False`, the button is not pressed.

- The `Reader` is any arbitrary source of a value, e.g. a GPIO pin or a socket connection, receiving the value from an IOT button.

```
@ctx.process
def reader(state):
    # reads the button value from a reader and stores it in the state

    reader = Reader()
    old_value = None

    while True:
        new_value = reader.read()

        # only update state when the value changes
        if old_value != new_value:
            state['button'] = new_value
            old_value = new_value



# calls handle_press() whenever button is pressed
@ctx.call_when_equal('button', True, live=True)
def handle_press(_, state):  # The first arg will be the value of "button". We don't␣
↪need that.

    print("button pressed")

    try:
        # wait 0.5 sec for a button to be released
        state.get_when_equal('button', False, timeout=0.5)

        print('its a SHORT press')

    # give up waiting
    except TimeoutError as e:

        print('its a LONG press')

        # wait infinitely for button to be released
        state.get_when_equal('button', False)

    print("button is released")
```

Here, passing `live=True` makes sense, since we don't care about a missed button press.

It makes the software respond to the button in real-time.

If `live=False` was passed, then it would not be real-time, and sometimes the application would lag behind the real world button state.

This behavior is undesirable when making Human computer interfaces, where keeping stuff responsive is a priority.

(The above code is simplified version of the code used in this project).

1.4 How ZProc talks

While you don't need to do any communication on your own, ZProc is actively doing it behind the covers, using zmq sockets.

Thanks to this, you take the same code and run it in a different environment, with very little to no modifications.

Furthermore, you can even take your existing code and scale it across multiple computers on your network.

This is the benefit of message passing parallelism. Your whole stack is built on communication, and hence, becomes extremely scalable and flexible when you need it to be.

1.4.1 The server address spec

An endpoint is a string consisting of two parts as follows: `<transport>://<address>`. The transport part specifies the underlying transport protocol to use. The meaning of the address part is specific to the underlying transport protocol selected.

The following transports may be used:

- **ipc** local inter-process communication transport, see zmq_ipc

  (`tcp://<address>:<port>`)

- **tcp** unicast transport using TCP, see zmq_tcp

  (`ipc://<file path>`)

Listing 5: Example

```
server_address='tcp://0.0.0.0:50001'

server_address='ipc:///home/username/my_endpoint'
```

1.4.2 IPC or TCP?

If you have a POSIX, and don't need to communicate across multiple computers, you are better off reaping the performance benefits of IPC.

For other use-cases, TCP.

By default, zproc will use IPC if it is available, else TCP.

1.4.3 Starting the server manually

When you create a *`Context`* object, ZProc will produce a random `server_address`, and start a server.

For advanced use-cases, you might want to use a well-known static address that all the services in your application are aware of.

This is quite useful when you want to access the same state across multiple nodes on a network, or in a different context on the same machine; anywhere communicating a "random" address would become an issue.

However, if you use a static address, *`Context`* won't start that server itself, and you have to do it manually, using *`start_server()`* (This behavior enables us to spawn multiple *`Context`* objects with the same address).

All the classes in ZProc take `server_address` *as their first argument.*

```
>>> import zproc
>>> ADDRESS = 'tcp://127.0.0.1:5000'
>>> zproc.start_server(ADDRESS)  # Important!
(<Process(Process-1, started)>, 'tcp://127.0.0.1:5000')
>>> zproc.Context(ADDRESS)
```

(continues on next page)

(continued from previous page)

```
<Context for State: {} to 'tcp://127.0.0.1:5000' at ...>
>>> zproc.State(ADDRESS)
<State: {} to 'tcp://127.0.0.1:5000' at ...>
```

The above example uses tcp, but ipc works just as well. (except across multiple machines)

Caution: Start the server *before* you access the *State* in *any* way; it solely depends on the server.

TLDR; You can start the server from anywhere you wish, and then access it though the address.

1.5 Security considerations

1.5.1 Cryptographic signing

Why?

Since un-pickling from an external source is considered dangerous, it becomes necessary to verify whether the other end is also a ZProc node, and not some attacker trying to exploit our application.

Hence, ZProc provides cryptographic signing support using itsdangerous.

Just provide the *secret_key* parameter to *Context*, and you should be good to go!

```
>>> import zproc
>>> ctx = zproc.Context(secret_key="muchsecret")
>>> ctx.secret_key
'muchsecret'
```

Similarly, *State* also takes the `secret_key` parameter.

By default, `secret_key` is set to `None`, which implies that no cryptographic signing is performed.

Yes, but why?

Here is an example demonstrating the usefulness of the this feature.

```
import zproc

home = zproc.Context(secret_key="muchsecret")
ADDRESS = home.server_address

home.state['gold'] = 5
```

An attacker somehow got to know our server's address. But since his secret key didn't match ours, their attempts to connect our server are futile.

```
attacker = zproc.Context(ADDRESS) # blocks forever
```

If however, you tell someone the secret key, then they are allowed to access the state.

```
friend = zproc.Context(ADDRESS, secret_key="muchsecret")
print(friend.state['gold'])  # 5
```

CHAPTER 2

The API Documentation / Guide

If you are looking for information on a specific function, class, or method, this part of the documentation is for you.

2.1 API

2.1.1 Functions

zproc.**ping**(*server_address: str*, ***, *timeout: Union[float, int, None] = None*, *sent_payload: Optional[bytes] = None*, *secret_key: str = None*) → Optional[int]

Ping the zproc server.

This can be used to easily detect if a server is alive and running, with the aid of a suitable `timeout`.

Parameters

- **server_address** – The zproc server's address.

  Please read *The server address spec* for a detailed explanation.

- **timeout** – The timeout in seconds.

  If this is set to `None`, then it will block forever, until the zproc server replies.

  For all other values, it will wait for a reply, for that amount of time before returning with a `TimeoutError`.

  By default it is set to `None`.

- **sent_payload** – payload that will be sent to the server.

  If it is set to None, then `os.urandom(56)` (56 random bytes) will be used.

  (No real reason for the `56` magic number.)

Returns

The zproc server's **pid** if the ping was successful, else `None`

If this returns `None`, then it probably means there is some fault in communication with the server.

`zproc.`**`atomic`**(*fn: Callable*) → Callable

Wraps a function, to create an atomic operation out of it.

No Process shall access the state while `fn` is running.

Note:

- The first argument to the wrapped function *must* be a *`State`* object.
- The wrapped function receives a frozen version (snapshot) of state;
  a `dict` object, not a *`State`* object.

Please read *Atomicity and race conditions* for a detailed explanation.

Parameters **`fn`** – The `function` to be wrapped, as an atomic function.

Returns

A wrapper `function`.

The "wrapper" `function` returns the value returned by the "wrapped" `function`.

```
>>> import zproc
>>>
>>> @zproc.atomic
... def increment(snapshot):
...     return snapshot['count'] + 1
...
>>>
>>> ctx = zproc.Context()
>>> ctx.state['count'] = 0
>>>
>>> increment(ctx.state)
1
```

`zproc.`**`start_server`**(*server_address: str = None*, *, *backend: Callable = <class 'multiprocessing.context.Process'>*, *secret_key: str = None*)

Start a new zproc server.

Parameters

- **`server_address`** – The zproc server's address.

  If it is set to `None`, then a random address will be generated.

  Please read *The server address spec* for a detailed explanation.

- **`backend`** – The backend to use for launching the server process.

  For example, you may use `threading.Thread` as the backend.

  Warning: Not guaranteed to work well with anything other than `multiprocessing.Process`.

Returns `tuple`, containing a `multiprocessing.Process` object for server and the server address.

zproc.**signal_to_exception**(*sig: signal.Signals*)
Convert a `signal.Signals` to a `SignalException`.

This allows for a natural, pythonic signal handing with the use of try-except blocks.

2.1.2 Exceptions

exception zproc.**ProcessWaitError**(*message*, *exitcode*, *process=None*)

exception zproc.**RemoteException**(*exc_info=None*)

exception zproc.**SignalException**(*sig*, *frame*)

2.1.3 Context

class zproc.**Context**(*server_address: str = None*, ***, *wait: bool = False*, *cleanup: bool = True*, *server_backend: Callable = <class 'multiprocessing.context.Process'>*, *namespace: str = 'default'*, *secret_key: str = None*, ***process_kwargs*)
Provides a high level interface to *State* and *Process*.

Primarily used to manage and launch processes.

All processes launched using a Context, share the same state.

Don't share a Context object between Processes / Threads. A Context object is not thread-safe.

Parameters

- **server_address** – The address of the server.

  If it is set to `None`, then a new server is started and a random address will be generated.

  Otherwise, it will connect to an existing server with the address provided.

  > **Caution:** If you provide a "server_address", be sure to manually start the server, as described here - *Starting the server manually*.

  Please read *The server address spec* for a detailed explanation.

- **wait** – Wait for all running process to finish their work before exiting.

  Alternative to manually calling *wait_all()* at exit.

- **cleanup** – Whether to cleanup the process tree before exiting.

  Registers a signal handler for `SIGTERM`, and an `atexit` handler.

- **server_backend** – Passed on to *start_server()* as `backend`.

- ****process_kwargs** – Keyword arguments that *Process* takes, except `server_address` and `target`.

  If provided, these will be used while creating processes using this Context.

Variables

- **state** – A *State* instance.
- **process_list** – A list of child `Process`(s) created under this Context.
- **worker_list** – A list of worker `Process`(s) created under this Context. Used for *Context.process_map()*.

- **server_process** – A `multiprocessing.Process` object for the server, or None.
- **server_address** – The server's address as a 2 element `tuple`.
- **namespace** – Passed on from the constructor. This is read-only.

process(*target: Optional[Callable] = None*, ***process_kwargs*) → Union[zproc.process.Process, Callable]

Produce a child process bound to this context.

Can be used both as a function and decorator:

Listing 1: Usage

```
@zproc.process()  # you may pass some arguments here
def my_process1(state):
        print('hello')


@zproc.process  # or not...
def my_process2(state):
    print('hello')


def my_process3(state):
    print('hello')

zproc.process(my_process3)  # or just use as a good ol' function
```

Parameters

- **target** – Passed on to the *Process* constructor.

  SHOULD be omitted when using this as a decorator.

- ****process_kwargs** – Keyword arguments that *Process* takes, except `server_address` and `target`.

  If provided, these will have a precedence over the one's provided in *Context*'s constructor.

Returns The *Process* instance produced.

process_factory(**targets*, *count: int = 1*, ***process_kwargs*)

Produce multiple child process(s) bound to this context.

Parameters

- ***targets** – Passed on to the *Process* constructor, one at a time.
- **count** – The number of processes to spawn for each item in `targets`.
- ****process_kwargs** – Keyword arguments that *Process* takes, except `server_address` and `target`.

  If provided, these will have a precedence over the one's provided in *Context*'s constructor.

Returns A `list` of the *Process* instance(s) produced.

pull_results_for_task(*task_detail: zproc.context.TaskDetail*) → Generator[[Any, None], None]

PULL "count" results from the process pool. Also arranges the results in-order.

process_map(*target: Callable*, *map_iter: Sequence[Any] = None*, ***, *map_args: Sequence[Sequence[Any]] = None*, *args: Sequence = None*, *map_kwargs: Sequence[Mapping[str, Any]] = None*, *kwargs: Mapping = None*, *count: int = None*, *stateful: bool = False*, *new: bool = False*, *return_task: bool = False*) → Union[zproc.context.TaskDetail, Generator[[Any, None], None]]

Functional equivalent of `map()` in-built function, but executed in a parallel fashion.

Distributes the iterables provided in the `map_*` arguments to `count` no of worker *Process*(s).

(Aforementioned worker processes are visible here: `Context.worker_list`)

The idea is to:

1. Split the the iterables provided in the `map_*` arguments into `count` number of equally sized chunks.
2. Send these chunks to `count` number of worker *Process*(s).
3. Wait for all these worker *Process*(s) to finish their task(s).
4. Combine the acquired results in the same sequence as provided in the `map_*` arguments.
5. Return the combined results.

Steps 3-5 are done lazily, on the fly with the help of a `generator`

Note: This function won't spawn new worker *Process*(s), each time it is called.

Existing workers will be used if a sufficient amount is available. If the workers are busy, then this will wait for them to finish up their current work.

Use the `new=True` Keyword Argument to spawn new workers, irregardless of existing ones.

You need not worry about shutting down workers. ZProc will take care of that automatically.

Note: This method doesn't have a way to pass Keyword Arguments to *Process*.

This was done, to prevent weird behavior due to the re-use of workers done by ZProc.

Use the *Context*'s constructor to workaround this problem.

Parameters

- **target** – The `Callable` to be invoked inside a *Process*.

  It is invoked with the following signature:

  ```
  target(state, map_iter[i], *map_args[i], *args,
  **map_kwargs[i], **kwargs)
  ```

  Where:

  - `state` is a *State* instance. (Disabled by default. Use the `stateful` Keyword Argument to enable)
  - `i` is the index of n^{th} element of the Iterable(s) provided in the `map_*` arguments.
  - `args` and `kwargs` are passed from the `**process_kwargs`.

  P.S. The `stateful` Keyword Argument of *Process* allows you to omit the `state` arg.

- **map_iter** – A sequence whose elements are supplied as the *first* positional argument (after `state`) to the `target`.

- **map_args** – A sequence whose elements are supplied as positional arguments (`*args`) to the `target`.
- **map_kwargs** – A sequence whose elements are supplied as keyword arguments (`**kwargs`) to the `target`.
- **args** – The argument tuple for `target`, supplied after `map_iter` and `map_args`.

  By default, it is an empty `tuple`.
- **kwargs** – A dictionary of keyword arguments for `target`.

  By default, it is an empty `dict`.
- **stateful** – Weather this process needs to access the state.

  If this is set to `False`, then the `state` argument won't be provided to the `target`.

  If this is set to `True`, then a *`State`* object is provided as the first Argument to the `target`.

  Unlike *`Process`* it is set to `False` by default. (To retain a similar API to in-built `map()`)
- **new** – Weather to spawn new workers.

  If it is set to `True`, then it will spawn new workers, irregardless of existing ones.

  If it is set to `False`, then `size - len(Context.worker_list)` will be spawned.

  Un-used workers are thrashed automatically.
- **count** – The number of worker *`Process`* (s) to use.

  By default, it is set to `multiprocessing.cpu_count()` (The number of CPU cores on your system)
- **return_task** – Return a `TaskDetail` namedtuple object, instead of a Generator that yields the results of the computation.

  The `TaskDetail` returned can be passed to *`Context.pull_results_for_task()`*, which will fetch the results for you.

  This is useful in situations where the results are required at a later time, and since a Generator object is not easily serializable, things get a little tricky. On the other hand, a namedtuple can be serialized to JSON, pretty easily.

Returns

The result is quite similar to `map()` in-built function.

It returns a `generator` whose elements are the return value of the `target` function, when applied to every item of the Iterables provided in the `map_*` arguments.

The actual "processing" starts as soon as you call this function.

The returned `generator` fetches the results from the worker processes, one-by-one.

Warning:

- If `len(map_iter) != len(maps_args) != len(map_kwargs)`, then the results will be cut-off at the shortest Sequence.

See *Process Map* for Examples.

call_when_change(**keys*, *exclude: bool = False*, *live: bool = False*, ***process_kwargs*)

Decorator version of *get_when_change()*.

Spawns a new *Process*, and then calls the wrapped function inside of that new process.

The wrapped function is run with the following signature:

```
target(snapshot, state, *args, **kwargs)
```

Where:

- `target` is the wrapped function.
- `snapshot` is a `dict` containing a copy of the state.

  Its serves as a *snapshot* of the state, corresponding to the state-change for which the wrapped function is being called.
- `state` is a *State* instance.
- `*args` and `**kwargs` are passed on from `**process_kwargs`.

Parameters

- ***keys** – Watch for changes on these keys in the state `dict`.

  If this is not provided, then all state-changes are respected. (default)
- **exclude** – Reverse the lookup logic i.e.,

  Watch for all changes in the state *except* in `*keys`.

  If `*keys` is not provided, then this has no effect. (default)
- **live** – Whether to get **live** updates.

  Please read *Live-ness of events* for a detailed explanation.
- **timeout** – Sets the timeout in seconds.

  If the value is `None`, it will block until an update is available.

  For all other values (`>=0`), it will wait for a state-change, for that amount of time before returning with a `TimeoutError`.
- **duplicate_okay** – Whether it's okay to process duplicate updates.

  Please read *Duplicate-ness of events* for a detailed explanation.
- ****process_kwargs** – Keyword arguments that *Process* takes, except `server_address` and `target`.

  If provided, these will have a precedence over the one's provided in *Context*'s constructor.

Returns

A decorator function

The decorator function will return the *Process* instance created.

Listing 2: Example

```
import zproc

ctx = zproc.Context()

@ctx.call_when_change('gold')
def test(snapshot, state):
    print(snapshot['gold'], state)
```

call_when(*test_fn: Callable*, *, *live: bool = False*, ***process_kwargs*)

Decorator version of *get_when()*.

Spawns a new *Process*, and then calls the wrapped function inside of that new process.

The wrapped function is run with the following signature:

```
target(snapshot, state, *args, **kwargs)
```

Where:

- `target` is the wrapped function.
- `snapshot` is a `dict` containing a copy of the state.

  Its serves as a *snapshot* of the state, corresponding to the state-change for which the wrapped function is being called.
- `state` is a *State* instance.
- `*args` and `**kwargs` are passed on from `**process_kwargs`.

Parameters

- **test_fn** – A `Callable`, which is called on each state-change.
- **live** – Whether to get **live** updates.

  Please read *Live-ness of events* for a detailed explanation.
- **timeout** – Sets the timeout in seconds.

  If the value is `None`, it will block until an update is available.

  For all other values (`>=0`), it will wait for a state-change, for that amount of time before returning with a `TimeoutError`.
- **duplicate_okay** – Whether it's okay to process duplicate updates.

  Please read *Duplicate-ness of events* for a detailed explanation.
- ****process_kwargs** – Keyword arguments that *Process* takes, except `server_address` and `target`.

  If provided, these will have a precedence over the one's provided in *Context*'s constructor.

Returns

A decorator function

The decorator function will return the *Process* instance created.

Listing 3: Example

```
import zproc

ctx = zproc.Context()

@ctx.get_state_when(lambda state: state['trees'] == 5)
def test(snapshot, state):
    print(snapshot['trees'], state)
```

call_when_equal(*key: collections.abc.Hashable*, *value: Any*, ***, *live: bool = False*, ***process_kwargs*)

Decorator version of *get_when_equal()*.

Spawns a new *Process*, and then calls the wrapped function inside of that new process.

The wrapped function is run with the following signature:

```
target(snapshot, state, *args, **kwargs)
```

Where:

- `target` is the wrapped function.
- `snapshot` is a `dict` containing a copy of the state.

  Its serves as a *snapshot* of the state, corresponding to the state-change for which the wrapped function is being called.
- `state` is a *State* instance.
- `*args` and `**kwargs` are passed on from `**process_kwargs`.

Parameters

- **key** – Some key in the state `dict`.
- **value** – The value corresponding to the `key` in state `dict`.
- **live** – Whether to get **live** updates.

  Please read *Live-ness of events* for a detailed explanation.
- **timeout** – Sets the timeout in seconds.

  If the value is `None`, it will block until an update is available.

  For all other values (`>=0`), it will wait for a state-change, for that amount of time before returning with a `TimeoutError`.
- **duplicate_okay** – Whether it's okay to process duplicate updates.

  Please read *Duplicate-ness of events* for a detailed explanation.
- ****process_kwargs** – Keyword arguments that *Process* takes, except `server_address` and `target`.

  If provided, these will have a precedence over the one's provided in *Context*'s constructor.

Returns

A decorator function

The decorator function will return the *Process* instance created.

Listing 4: Example

```
import zproc

ctx = zproc.Context()

@ctx.call_when_equal('oranges', 5)
def test(snapshot, state):
    print(snapshot['oranges'], state)
```

call_when_not_equal(*key: collections.abc.Hashable*, *value: Any*, *, *live: bool = False*, ***process_kwargs*)

Decorator version of *get_when_not_equal()*.

Spawns a new *Process*, and then calls the wrapped function inside of that new process.

The wrapped function is run with the following signature:

```
target(snapshot, state, *args, **kwargs)
```

Where:

- `target` is the wrapped function.
- `snapshot` is a `dict` containing a copy of the state.

  Its serves as a *snapshot* of the state, corresponding to the state-change for which the wrapped function is being called.
- `state` is a *State* instance.
- `*args` and `**kwargs` are passed on from `**process_kwargs`.

Parameters

- **key** – Some key in the state `dict`.
- **value** – The value corresponding to the `key` in state `dict`.
- **live** – Whether to get **live** updates.

  Please read *Live-ness of events* for a detailed explanation.
- **timeout** – Sets the timeout in seconds.

  If the value is `None`, it will block until an update is available.

  For all other values (`>=0`), it will wait for a state-change, for that amount of time before returning with a `TimeoutError`.
- **duplicate_okay** – Whether it's okay to process duplicate updates.

  Please read *Duplicate-ness of events* for a detailed explanation.
- ****process_kwargs** – Keyword arguments that *Process* takes, except `server_address` and `target`.

  If provided, these will have a precedence over the one's provided in *Context*'s constructor.

Returns

A decorator function

The decorator function will return the *Process* instance created.

Listing 5: Example

```
import zproc

ctx = zproc.Context()

@ctx.call_when_not_equal('apples', 5)
def test(snapshot, state):
    print(snapshot['apples'], state)
```

call_when_none(*key: collections.abc.Hashable*, *, *live: bool = False*, ***process_kwargs*)

Decorator version of *get_when_none()*.

Spawns a new *Process*, and then calls the wrapped function inside of that new process.

The wrapped function is run with the following signature:

```
target(snapshot, state, *args, **kwargs)
```

Where:

- `target` is the wrapped function.
- `snapshot` is a `dict` containing a copy of the state.

  Its serves as a *snapshot* of the state, corresponding to the state-change for which the wrapped function is being called.
- `state` is a *State* instance.
- `*args` and `**kwargs` are passed on from `**process_kwargs`.

Parameters

- **key** – Some key in the state `dict`.
- **value** – The value corresponding to the `key` in state `dict`.
- **live** – Whether to get **live** updates.

  Please read *Live-ness of events* for a detailed explanation.
- **timeout** – Sets the timeout in seconds.

  If the value is `None`, it will block until an update is available.

  For all other values (`>=0`), it will wait for a state-change, for that amount of time before returning with a `TimeoutError`.
- **duplicate_okay** – Whether it's okay to process duplicate updates.

  Please read *Duplicate-ness of events* for a detailed explanation.
- ****process_kwargs** – Keyword arguments that *Process* takes, except `server_address` and `target`.

  If provided, these will have a precedence over the one's provided in *Context*'s constructor.

Returns

A decorator function

The decorator function will return the *Process* instance created.

call_when_not_none(*key: collections.abc.Hashable*, *, *live: bool = False*, ***process_kwargs*)
Decorator version of *get_when_not_none()*.

Spawns a new *Process*, and then calls the wrapped function inside of that new process.

The wrapped function is run with the following signature:

```
target(snapshot, state, *args, **kwargs)
```

Where:

- `target` is the wrapped function.
- `snapshot` is a `dict` containing a copy of the state.

  Its serves as a *snapshot* of the state, corresponding to the state-change for which the wrapped function is being called.
- `state` is a *State* instance.
- `*args` and `**kwargs` are passed on from `**process_kwargs`.

Parameters

- **key** – Some key in the state `dict`.
- **value** – The value corresponding to the `key` in state `dict`.
- **live** – Whether to get **live** updates.

  Please read *Live-ness of events* for a detailed explanation.
- **timeout** – Sets the timeout in seconds.

  If the value is `None`, it will block until an update is available.

  For all other values (`>=0`), it will wait for a state-change, for that amount of time before returning with a `TimeoutError`.
- **duplicate_okay** – Whether it's okay to process duplicate updates.

  Please read *Duplicate-ness of events* for a detailed explanation.
- ****process_kwargs** – Keyword arguments that *Process* takes, except `server_address` and `target`.

  If provided, these will have a precedence over the one's provided in *Context*'s constructor.

Returns

A decorator function

The decorator function will return the *Process* instance created.

call_when_available(*key: collections.abc.Hashable*, *, *live: bool = False*, ***process_kwargs*)
Decorator version of *get_when_available()*.

Spawns a new *Process*, and then calls the wrapped function inside of that new process.

The wrapped function is run with the following signature:

```
target(snapshot, state, *args, **kwargs)
```

Where:

- `target` is the wrapped function.

- `snapshot` is a `dict` containing a copy of the state.

  Its serves as a *snapshot* of the state, corresponding to the state-change for which the wrapped function is being called.

- `state` is a *`State`* instance.
- `*args` and `**kwargs` are passed on from `**process_kwargs`.

Parameters

- **`key`** – Some key in the state `dict`.
- **`value`** – The value corresponding to the `key` in state `dict`.
- **`live`** – Whether to get **live** updates.

  Please read *Live-ness of events* for a detailed explanation.

- **`timeout`** – Sets the timeout in seconds.

  If the value is `None`, it will block until an update is available.

  For all other values (`>=0`), it will wait for a state-change, for that amount of time before returning with a `TimeoutError`.

- **`duplicate_okay`** – Whether it's okay to process duplicate updates.

  Please read *Duplicate-ness of events* for a detailed explanation.

- **`**process_kwargs`** – Keyword arguments that *`Process`* takes, except `server_address` and `target`.

  If provided, these will have a precedence over the one's provided in *`Context`*'s constructor.

Returns

A decorator function

The decorator function will return the *`Process`* instance created.

`wait_all`(*timeout: Union[float, int, None] = None*, *safe: bool = False*) → List[Union[Any, Exception]]

Call *`wait()`* on all the child processes of this Context. (Excluding the worker processes)

Retains the same order as `Context.process_list`.

Parameters

- **`timeout`** – Same as *`wait()`*.

  This parameter controls the timeout for all the Processes combined, not a single *`wait()`* call.

- **`safe`** – Suppress any errors that occur while waiting for a Process.

  The return value of failed *`wait()`* calls are substituted with the `Exception` that occurred.

Returns A `list` containing the values returned by child Processes of this Context.

`start_all`()

Call *`start()`* on all the child processes of this Context

Ignores if a Process is already started, unlike *`start()`*, which throws an `AssertionError`.

stop_all()
Call *stop()* on all the child processes of this Context

Retains the same order as `Context.process_list`.

Returns A `list` containing the exitcodes of the child Processes of this Context.

ping(***kwargs*)
Ping the zproc server.

Parameters ****kwargs** – Keyword arguments that *ping()* takes, except `server_address`.

Returns Same as *ping()*

close()
Close this context and stop all processes associated with it.

Once closed, you shouldn't use this Context again.

2.1.4 Process

class zproc.**Process**(*target: Callable*, *server_address: str*, ***, *stateful: bool = True*, *pass_context: bool = False*, *args: Sequence = None*, *kwargs: Mapping = None*, *retry_for: Sequence[Union[signal.Signals, Exception]] = ()*, *retry_delay: Union[int, float] = 5*, *max_retries: Optional[bool] = None*, *retry_args: Optional[tuple] = None*, *retry_kwargs: Optional[dict] = None*, *start: bool = True*, *backend: Callable = <class 'multiprocessing.context.Process'>*, *namespace: str = 'default'*, *secret_key: Optional[str] = None*)
Provides a higher level interface to `multiprocessing.Process`.

Please don't share a Process object between Processes / Threads. A Process object is not thread-safe.

Parameters

- **server_address** – The address of zproc server.

  If you are using a *Context*, then this is automatically provided.

  Please read *The server address spec* for a detailed explanation.

- **target** – The Callable to be invoked inside a new process.

  The ''target'' is invoked with the following signature:

  ```
  target(state, *args, **kwargs)
  ```

  Where:

  – `state` is a *State* instance.

  – `args` and `kwargs` are passed from the constructor.

- **pass_context** – Weather to pass a *Context* to this process.

  If this is set to `True`, then the first argument to `target` will be a *Context* object in-place of the default - *State*.

  In other words, The `target` is invoked with the following signature:

  ```
  target(ctx, *args, **kwargs)
  ```

  Where:

  – `ctx` is a *Context* object.

– `args` and `kwargs` are passed from the constructor.

Note: The *`Context`* object provided here, will be different than the one used to create this process.

The new *`Context`* object can be used to create nested processes that share the same state.

- **`stateful`** – Weather this process needs to access the state.

  If this is set to `False`, then the `state` argument won't be provided to the `target`.

  In other words, The `target` is invoked with the following signature:

  `target(*args, **kwargs)`

  Where:

  – `args` and `kwargs` are passed from the constructor.

  Has no effect if `pass_context` is set to `True`.
- **`start`** – Automatically call *`start()`* on the process.
- **`retry_for`** – Retry only when one of these `Exception`/`signal.Signals` is raised.

Listing 6: Example

```
import signal

# retry if a ConnectionError, ValueError or signal.SIGTERM is␣
↪received.
ctx.process(
    my_process,
    retry_for=(ConnectionError, ValueError, signal.SIGTERM)
)
```

To retry for *any* Exception - `retry_for=(Exception, )`

The items of this sequence MUST be a subclass of `BaseException` or of type `signal.Signals`.

- **`retry_delay`** – The delay in seconds, before retrying.
- **`max_retries`** – Give up after this many attempts.

  A value of `None` will result in an *infinite* number of retries.

  After "max_tries", any Exception / Signal will exhibit default behavior.
- **`args`** – The argument tuple for `target`.

  By default, it is an empty `tuple`.
- **`kwargs`** – A dictionary of keyword arguments for `target`.

  By default, it is an empty `dict`.
- **`retry_args`** – Used in place of `args` when retrying.

  If set to `None`, then it has no effect.
- **`retry_kwargs`** – Used in place of `kwargs` when retrying.

  If set to `None`, then it has no effect.

- **backend** – The backend to use for launching the process(s).

  For example, you may use `threading.Thread` as the backend.

  > **Warning:** Not guaranteed to work well with anything other than `multiprocessing.Process`.

Variables

- **child** – A `multiprocessing.Process` instance for the child process.
- **server_address** – Passed on from the constructor.
- **target** – Passed on from the constructor.
- **namespace** – Passed on from the constructor. This is read-only.

start()
Start this Process

If the child has already been started once, it will return with an `AssertionError`.

Returns the process PID

stop()
Stop this process.

Once closed, it should not, and cannot be used again.

Returns *exitcode*.

wait(*timeout: Union[float, int, None] = None*)
Wait until this process finishes execution, then return the value returned by the `target`.

Parameters **timeout** – The timeout in seconds.

If the value is `None`, it will block until the zproc server replies.

For all other values, it will wait for a reply, for that amount of time before returning with a `TimeoutError`.

Returns The value returned by the `target` function.

If the child finishes with a non-zero exitcode, or there is some error in retrieving the value returned by the `target`, a *ProcessWaitError* is raised.

is_alive
Whether the child process is alive.

Roughly, a process object is alive; from the moment the *start()* method returns, until the child process is stopped manually (using *stop()*) or naturally exits

pid
The process ID.

Before the process is started, this will be None.

exitcode
The child's exit code.

This will be None if the process has not yet terminated. A negative value `-N` indicates that the child was terminated by signal `N`.

2.1.5 State

class zproc.**State**(*server_address: str*, ***, *namespace: str = 'default'*, *secret_key: Optional[str] = None*)

Allows accessing state stored on the zproc server, through a dict-like API.

Communicates to the zproc server using the ZMQ sockets.

Please don't share a State object between Processes/Threads. A State object is not thread-safe.

Boasts the following `dict`-like members, for accessing the state:

- **Magic methods:** `__contains__()`, `__delitem__()`, `__eq__()`, `__getitem__()`, `__iter__()`, `__len__()`, `__ne__()`, `__setitem__()`
- **Methods:** `clear()`, `copy()`, `get()`, `items()`, `keys()`, `pop()`, `popitem()`, `setdefault()`, `update()`, `values()`

Parameters **server_address** – The address of zproc server.

If you are using a *Context*, then this is automatically provided.

Please read *The server address spec* for a detailed explanation.

Variables **server_address** – Passed on from constructor.

fork(*server_address: Optional[str] = None*, ***, *namespace: Optional[str] = None*, *secret_key: Optional[str] = None*) → zproc.state.State

"Forks" this State object.

Takes the same args as the *State* constructor, except that they automatically default to the values provided during the creation of this State object.

If no args are provided to this function, then it shall create a new *State* object that follows the exact same semantics as this one.

This is preferred over copying a *State* object.

Useful when one needs to access 2 or more namespaces on the same server.

set(*value: dict*)

Set the state, completely over-writing the previous value.

copy()

Return a deep-copy of the state.

Unlike the shallow-copy returned by `dict.copy()`.

go_live()

Clear the outstanding queue (or buffer), thus clearing any past events that were stored.

Internally, this re-opens a socket, which in-turn clears the queue.

Please read *Live-ness of events* for a detailed explanation.

get_raw_update(*live: bool = False*, *timeout: Union[float, int, None] = None*, *duplicate_okay: bool = False*) → Tuple[dict, dict, bool]

A low-level hook that emits each and every state update.

Parameters

- **live** – Whether to get **live** updates.

  Please read *Live-ness of events* for a detailed explanation.

- **timeout** – Sets the timeout in seconds.

  If the value is `None`, it will block until an update is available.

  For all other values (`>=0`), it will wait for a state-change, for that amount of time before returning with a `TimeoutError`.

- **duplicate_okay** – Whether it's okay to process duplicate updates.

  Please read *Duplicate-ness of events* for a detailed explanation.

get_when_change(**keys*, *exclude: bool = False*, *live: bool = False*, *timeout: Union[float, int, None] = None*, *duplicate_okay: bool = False*) → dict

Block until a change is observed, and then return a copy of the state.

Parameters

- ***keys** – Watch for changes on these keys in the state `dict`.

  If this is not provided, then all state-changes are respected. (default)

- **exclude** – Reverse the lookup logic i.e.,

  Watch for all changes in the state *except* in `*keys`.

  If `*keys` is not provided, then this has no effect. (default)

- **live** – Whether to get **live** updates.

  Please read *Live-ness of events* for a detailed explanation.

- **timeout** – Sets the timeout in seconds.

  If the value is `None`, it will block until an update is available.

  For all other values (`>=0`), it will wait for a state-change, for that amount of time before returning with a `TimeoutError`.

Returns

A `dict` containing a copy of the state.

This copy serves as a *snapshot* of the state, corresponding to the state-change for which this state watcher was triggered.

get_when(*test_fn*, ***, *live: bool = False*, *timeout: Union[float, int, None] = None*, *duplicate_okay: bool = False*) → dict

Block until `test_fn(snapshot)` returns a "truthy" value, and then return a copy of the state.

Where-

`snapshot` is a `dict`, containing a copy of the state.

Parameters

- **test_fn** – A `Callable`, which is called on each state-change.

- **live** – Whether to get **live** updates.

  Please read *Live-ness of events* for a detailed explanation.

- **timeout** – Sets the timeout in seconds.

  If the value is `None`, it will block until an update is available.

  For all other values (`>=0`), it will wait for a state-change, for that amount of time before returning with a `TimeoutError`.

- **duplicate_okay** – Whether it's okay to process duplicate updates.

  Please read *Duplicate-ness of events* for a detailed explanation.

Returns

A `dict` containing a copy of the state.

This copy serves as a *snapshot* of the state, corresponding to the state-change for which this state watcher was triggered.

get_when_equal(*key: collections.abc.Hashable*, *value: Any*, ***, *live: bool = False*, *timeout: Union[float, int, None] = None*, *duplicate_okay: bool = False*) → dict

Block until `state[key] == value`, and then return a copy of the state.

Parameters

- **key** – Some key in the state `dict`.
- **value** – The value corresponding to the `key` in state `dict`.
- **live** – Whether to get **live** updates.

  Please read *Live-ness of events* for a detailed explanation.

- **timeout** – Sets the timeout in seconds.

  If the value is `None`, it will block until an update is available.

  For all other values (`>=0`), it will wait for a state-change, for that amount of time before returning with a `TimeoutError`.

- **duplicate_okay** – Whether it's okay to process duplicate updates.

  Please read *Duplicate-ness of events* for a detailed explanation.

Returns

A `dict` containing a copy of the state.

This copy serves as a *snapshot* of the state, corresponding to the state-change for which this state watcher was triggered.

get_when_not_equal(*key: collections.abc.Hashable*, *value: Any*, ***, *live: bool = False*, *timeout: Union[float, int, None] = None*, *duplicate_okay: bool = False*) → dict

Block until `state[key] != value`, and then return a copy of the state.

Parameters

- **key** – Some key in the state `dict`.
- **value** – The value corresponding to the `key` in state `dict`.
- **live** – Whether to get **live** updates.

  Please read *Live-ness of events* for a detailed explanation.

- **timeout** – Sets the timeout in seconds.

  If the value is `None`, it will block until an update is available.

  For all other values (`>=0`), it will wait for a state-change, for that amount of time before returning with a `TimeoutError`.

- **duplicate_okay** – Whether it's okay to process duplicate updates.

  Please read *Duplicate-ness of events* for a detailed explanation.

Returns

A `dict` containing a copy of the state.

This copy serves as a *snapshot* of the state, corresponding to the state-change for which this state watcher was triggered.

get_when_none(*key: collections.abc.Hashable*, ***, *live: bool = False*, *timeout: Union[float, int, None] = None*, *duplicate_okay: bool = False*) → dict

Block until `state[key] is None`, and then return a copy of the state.

Parameters

- **key** – Some key in the state `dict`.
- **value** – The value corresponding to the `key` in state `dict`.
- **live** – Whether to get **live** updates.

  Please read *Live-ness of events* for a detailed explanation.
- **timeout** – Sets the timeout in seconds.

  If the value is `None`, it will block until an update is available.

  For all other values (`>=0`), it will wait for a state-change, for that amount of time before returning with a `TimeoutError`.
- **duplicate_okay** – Whether it's okay to process duplicate updates.

  Please read *Duplicate-ness of events* for a detailed explanation.

Returns

A `dict` containing a copy of the state.

This copy serves as a *snapshot* of the state, corresponding to the state-change for which this state watcher was triggered.

get_when_not_none(*key: collections.abc.Hashable*, ***, *live: bool = False*, *timeout: Union[float, int, None] = None*, *duplicate_okay: bool = False*) → dict

Block until `state[key] is not None`, and then return a copy of the state.

Parameters

- **key** – Some key in the state `dict`.
- **value** – The value corresponding to the `key` in state `dict`.
- **live** – Whether to get **live** updates.

  Please read *Live-ness of events* for a detailed explanation.
- **timeout** – Sets the timeout in seconds.

  If the value is `None`, it will block until an update is available.

  For all other values (`>=0`), it will wait for a state-change, for that amount of time before returning with a `TimeoutError`.
- **duplicate_okay** – Whether it's okay to process duplicate updates.

  Please read *Duplicate-ness of events* for a detailed explanation.

Returns

A `dict` containing a copy of the state.

This copy serves as a *snapshot* of the state, corresponding to the state-change for which this state watcher was triggered.

get_when_available(*key: collections.abc.Hashable*, ***, *live: bool = False*, *timeout: Union[float, int, None] = None*, *duplicate_okay: bool = False*)

Block until `key in state`, and then return a copy of the state.

Parameters

- **key** – Some key in the state `dict`.
- **value** – The value corresponding to the `key` in state `dict`.
- **live** – Whether to get **live** updates.

  Please read *Live-ness of events* for a detailed explanation.
- **timeout** – Sets the timeout in seconds.

  If the value is `None`, it will block until an update is available.

  For all other values (`>=0`), it will wait for a state-change, for that amount of time before returning with a `TimeoutError`.
- **duplicate_okay** – Whether it's okay to process duplicate updates.

  Please read *Duplicate-ness of events* for a detailed explanation.

Returns

A `dict` containing a copy of the state.

This copy serves as a *snapshot* of the state, corresponding to the state-change for which this state watcher was triggered.

ping(***kwargs*)

Ping the zproc server corresponding to this State's Context

Parameters **kwargs** – Keyword arguments that *ping()* takes, except `server_address`.

Returns Same as *ping()*

close()

Close this State and disconnect with the Server.

CHAPTER 3

Indicies

- genindex
- modindex
- search

Index