isc Documentation

Release 0.8

Andrew Dunai

Jan 24, 2018

Contents

CHAPTER 1

Quick start

1.1 Supported versions

- Python 2.7
- Python 3.5+

1.2 Requirements

- `pika`
- `gevent` (server only)

1.3 Installation

```
pip install isclib
```

CHAPTER 2

Usage

2.1 Comprehensive example

server.py

```
#!/usr/bin/env python3.6

from isc.server import Node, expose, on, local_timer
from time import sleep
import logging


class ExampleService(object):
    name = 'example'

    def __init__(self):
        self.tracker = None

    @expose
    def add(self, a, b, wait=0):
        sleep(wait)
        return str(a + b) * 8000

    @expose
    def raise_error(self):
        raise Exception('testing')

    def private_method(self):
        return 'Cannot call me!'

    @on('boom')
    def do_stuff(self, arg):
        print('Got stuff:', arg)

    @expose
```

```
    def slow_method(self):
        sleep(3)

    @expose
    def start_tracking(self):
        self.tracker = tracker.SummaryTracker()

    @expose
    def get_summary(self):
        return list(self.tracker.format_diff())

    # @local_timer(timeout=3)
    # def print_stats(self):
    #     print('Stats: foobar')


service = ExampleService()
node = Node(exchange='isctest')
node.set_logging_level(logging.DEBUG)
node.register_service(service)

if __name__ == '__main__':
    try:
        node.run()
    except KeyboardInterrupt:
        node.stop()
```

client.py

```
#!/usr/bin/env python3.6

from isc.client import Client, RemoteException, TimeoutException

client = Client(exchange='isctest')
client.start()

import time
time.sleep(2)

client.notify('boom', dict(foo='bar'))
a
assert client.example.add(2, 3) == '5' * 8000
assert client.invoke('example', 'add', 2, 3) == '5' * 8000

try:
    client.example.add(2, '3')
except RemoteException:
    pass
else:
    assert False

try:
    client.example.raise_error()
except RemoteException:
    pass
else:
    assert False
```

```
try:
    client.example.private_method()
except RemoteException:
    pass
else:
    assert False

try:
    client.example.unexisting_method()
except RemoteException:
    pass
else:
    assert False


try:
    client.set_invoke_timeout(1)
    client.example.slow_method()
except TimeoutException:
    pass
else:
    assert False

client.stop()
```

2.2 Testing

The tests can be executed using py.test:

```
make test
```

2.3 Caveats

If you import anything from `isc.server`, keep in mind that this library uses `gevent` to patch built-in libraries. This doesn't apply to `isc.client` though.

CHAPTER 3

Simple server & client

```
# users_service.py

from isc.server import Node, expose

class UserService(object):
    name = 'users'

    @expose
    def get_user(self, id):
        if id == 1:
            return 'Andrew'
        elif id == 2:
            return 'Victoria'
        return None

node = Node()
node.register_service(UserService())
try:
    node.run()
except KeyboardInterrupt:
    node.stop()
```

```
# app.py

from isc.client import Client

client = Client()

print(client.users.get_user(1))  # prints 'Andrew'
print(client.users.get_user(2))  # prints 'Victoria'
print(client.users.get_user(3))  # prints 'None'
```

CHAPTER 4

Two services

```python
# users_service.py

from isc.server import Node, expose, on
from superapp.models import User

class UserService(object):
    name = 'users'

    @expose
    def get_user(self, id):
        # Let's use some ORM to retrieve the user from DB
        user = User.objects.filter(id=id).first()
        if user:
            # User not found!
            return {'username': user.username}
        return None

    @on('new_message')
    def on_new_message(self, username, message):
        print('New message for user {}: {}'.format(username, message))

node = Node()
node.register_service(UserService())
try:
    node.run()
except KeyboardInterrupt:
    node.stop()
```

```python
# messages_service.py

from isc.server import Node, expose
from isc.client import Client
from superapp.models import Message
```

```
client = Client()

class MessageService(object):
    name = 'messages'

    @expose
    def send_message(self, body, receipt):
        user = client.users.get_user(receipt)
        if not user:
            # User not found!
            raise Exception('Cannot send message: user not found')
        Message.objects.create(receipt=receipt, message=body)

        # Broadcast to all instances
        client.notify('new_message', user['username'], message)

node = Node()
node.register_service(MessageService())
try:
    node.run()
except KeyboardInterrupt:
    node.stop()
```

```
# app.py

from isc.client import Client

client = Client()

# ...

try:
    client.messages.send_message('Hello!', some_user_id)
except RemoteException as e:
    print('Failed to send message, error was: {}'.format(str(e)))
else:
    print('Message send!')
```

CHAPTER 5

Timers

```
# tick_service.py

from isc.server import Node, expose, local_timer

class TickerService(object):
    name = 'ticker'

    def __init__(self):
        """
        WARNING:
        Do NOT do this in real projects. (I'm speaking about local state
        which is represented by `self.ticks` attribute here.)

        Services MUST be stateless.
        This dirty trick right here is used just to demonstrate
        how the timer works without involving any external storage.

        In real project:
        - ALWAYS database or any other external storage instead of `self`
        - NEVER mutate service object.

        So in real project you would have done something like this:
        self.db_conn = SomeDatabaseConnection()
        # ...
        spam = next(self.db_conn.query('SELECT spam;').fetchone(), None)
        """

        self.ticks = 0

    @expose
    def get_ticks(self, id):
        return self.ticks

    @expose
    def reset_ticks(self):
```

```
        self.ticks = 0

    @local_timer(timeout=5)
    def local_timer(self):
        self.ticks += 1


node = Node()
node.register_service(TickerService())
try:
    node.run()
except KeyboardInterrupt:
    node.stop()
```

```
# app.py

from isc.client import Client
from time import sleep

client = Client()

client.ticker.reset_ticks()
print(client.ticker.get_ticks())  # prints 0

sleep(10)
print(client.ticker.get_ticks())  # prints 2

sleep(10)
print(client.ticker.get_ticks())  # prints 4

client.ticker.reset_ticks()
print(client.ticker.get_ticks())  # prints 0

sleep(10)
print(client.ticker.get_ticks())  # prints 2
```

CHAPTER 6

Server bindings

CHAPTER 7

Client bindings

class isc.client.**Client**(*host='amqp://guest:guest@127.0.0.1:5672/'*, *exchange='isc'*, *codec=None*, *connect_timeout=2*, *reconnect_timeout=3*, *invoke_timeout=20*)

Represents a single low-level connection to the ISC messaging broker. Thread-safe.

force_json()

Force to use JSON codec only.

invoke(*service*, *method*, **args*, ***kwargs*)

Call a remote method and wait for a result. Blocks until a result is ready.

invoke_async(*service*, *method*, **args*, ***kwargs*)

Serialize & publish method call request.

notify(*event*, *data*)

Serialize & publish notification.

set_codec(*codec*)

Set codec.

set_invoke_timeout(*timeout*)

Sets timeout for waiting for results on this client.

set_logging_level(*level*)

Set logging level.

start()

Start connection & publisher threads.

stop()

Stops the client and waits for its termination.

class isc.client.**ConsumerThread**(*hostname*, *exchange_name*, *connect_timeout*, *reconnect_timeout*, *codec*)

Internal class. Represents connection & message consuming thread.

class isc.client.**FutureResult**(*cannonical_name*, ***extra*)

Encapsulates future result. Provides interface to block until future data is ready. Thread-safe.

is_ready()
Checks if this result has been resolved or rejected.

reject(*exception*)
Rejects this promise with exception and sets "ready" event.

resolve(*value*)
Resolves this promise with result and sets "ready" event.

wait(*timeout=5*)
Blocks until data is ready.

class isc.client.**MethodProxy**(*client*, *service_name*, *method_name*)
Convenience wrapper for method.

It allows you to perform attribute chaining (e. g. `client.example.add(2, 3)`)

call_async(**args*, ***kwargs*)
Finalizes the chain & performs actual RPC invocation. Does not block.

Returns *FutureResult*.

This is same as calling *invoke_async()*

class isc.client.**PublisherThread**(*consumer*)
Internal class. Represents message publishing thread.

class isc.client.**QueuedInvocation**(*codec*, *service*, *method*, *args*, *kwargs*)
Internal class. Represents pending outgoing method call.

class isc.client.**QueuedNotification**(*codec*, *event*, *data*)
Internal class. Represents pending outgoing notification.

class isc.client.**QueuedRequest**(*codec*, ***kwargs*)
Internal class. Represents pending outgoing message.

class isc.client.**ServiceProxy**(*client*, *service_name*)
Convenience wrapper for service.

It allows you to perform attribute chaining (e. g. `client.example.add(2, 3)`)

CHAPTER 8

Custom codecs

class isc.codecs.**AbstractCodec**
Abstract base class for implementing codecs.

"Codec" is a class that tells ISC how to encode & decode message payloads.

Server can support multiple codecs while client can use only one at a time.

Default codec is *PickleCodec*. You can implement your own codec by extending *AbstractCodec* and overriding its methods.

Important: Don't forget that both client and server should have the codec installed!

decode(*payload*)
Called when a message needs to be serialized for sending over AMQP channel.

encode(*message*)
Called when a message needs to be serialized for sending over AMQP channel.

class isc.codecs.**JSONCodec**
JSON codec implementation.

class isc.codecs.**PickleCodec**
Pickle codec implementation.

class isc.codecs.**TypedJSONCodec**
JSON codec implementation with support of timestamps & UUIDs.

Python Module Index

i

Index