compactor Documentation

Release 0.2.2

Brian Wickman

March 24, 2015

Contents

compactor is a pure python implementation of libprocess, the actor library underpinning mesos.

CHAPTER 1

compactor

1.1 Global methods

Some methods are proxied to the global singleton Context in order to make simple programs simpler to write. These methods do the Right Thing™ for most use-cases.

`compactor.`**`join`**`()`
Join against the global context – blocking until the context has been stopped.

`compactor.`**`spawn`**`(`**args*, ***kw*`)`
Spawn a process on the global context and return its pid.

Parameters **process** (`Process`) – The process to bind to the global context.

Returns pid The pid of the spawned process.

Return type `PID`

1.2 PIDs

```
from compactor.pid import PID
pid = PID.from_string('slave(1)@192.168.33.2:5051')
```

class `compactor.pid.`**`PID`**`(`*ip*, *port*, *id_*`)`

`__init__``(`*ip*, *port*, *id_*`)`
Construct a pid.

Parameters

- **ip** (`str`) – An IP address in string form.
- **port** (`int`) – The port of this pid.
- **id** (`str`) – The name of the process.

classmethod `from_string``(`*pid*`)`
Parse a PID from its string representation.

PIDs may be represented as name@ip:port, e.g.

```
pid = PID.from_string('master(1)@192.168.33.2:5051')
```

Parameters **pid** (`str`) – A string representation of a pid.

Returns The parsed pid.

Return type `PID`

Raises `ValueError` should the string not be of the correct syntax.

1.3 Processes

```
from compactor.process import Process

class PingProcess(Process):
  def initialize(self):
    super(PingProcess, self).initialize()
    self.pinged = threading.Event()

  @Process.install('ping')
  def ping(self, from_pid, body):
    self.pinged.set()
```

class `compactor.process.`**Process**(*name*)

__init__(*name*)
Create a process with a given name.

The process must still be bound to a context before it can send messages or link to other processes.

Parameters **name** (`str`) – The name of this process.

context
The context that this process is bound to.

Raises Will raise a `Process.UnboundProcess` exception if the process is not bound to a context.

exited(*pid*)
Called when a linked process terminates or its connection is severed.

Parameters **pid** (`PID`) – The pid of the linked process.

initialize()
Called when this process is spawned.

Once this is called, it means a process is now routable. Subclasses should implement this to initialize state or possibly initiate connections to remote processes.

classmethod install(*mbox*)
A decorator to indicate a remotely callable method on a process.

```
from compactor.process import Process

class PingProcess(Process):
  @Process.install('ping')
  def ping(self, from_pid, body):
    # do something
```

The installed method should take `from_pid` and `body` parameters. `from_pid` is the process calling the method. `body` is a `bytes` stream that was delivered with the message, possibly empty.

Parameters **mbox** (`str`) – Incoming messages to this "mailbox" will be dispatched to this method.

link(*to*)
Link to another process.

The `link` operation is not guaranteed to succeed. If it does, when the other process terminates, the `exited` method will be called with its pid.

Returns immediately.

Parameters **to** (`PID`) – The pid of the process to send a message.

Raises Will raise a `Process.UnboundProcess` exception if the process is not bound to a context.

Returns Nothing

pid
The pid of this process.

Raises Will raise a `Process.UnboundProcess` exception if the process is not bound to a context.

classmethod route(*path*)
A decorator to indicate that a method should be a routable HTTP endpoint.

```
from compactor.process import Process

class WebProcess(Process):
  @Process.route('/hello/world')
  def hello_world(self, handler):
    return handler.write('<html><title>hello world</title></html>')
```

The handler passed to the method is a tornado RequestHandler.

WARNING: This interface is alpha and may change in the future if or when we remove tornado as a compactor dependency.

Parameters **path** (`str`) – The endpoint to route to this method.

send(*to*, *method*, *body=None*)
Send a message to another process.

Sending messages is done asynchronously and is not guaranteed to succeed.

Returns immediately.

Parameters

- **to** (`PID`) – The pid of the process to send a message.
- **method** (`str`) – The method/mailbox name of the remote method.
- **body** (`bytes` or None) – The optional content to send with the message.

Raises Will raise a `Process.UnboundProcess` exception if the process is not bound to a context.

Returns Nothing

terminate()
Terminate this process.

This unbinds it from the context to which it is bound.

Raises Will raise a `Process.UnboundProcess` exception if the process is not bound to a context.

class `compactor.process.`**ProtobufProcess**(*name*)

Bases: `compactor.process.Process`

classmethod install(*message_type*)

A decorator to indicate a remotely callable method on a process using protocol buffers.

```
from compactor.process import ProtobufProcess
from messages_pb2 import RequestMessage, ResponseMessage

class PingProcess(ProtobufProcess):
  @ProtobufProcess.install(RequestMessage)
  def ping(self, from_pid, message):
    # do something with message, a RequestMessage
    response = ResponseMessage(...)
    # send a protocol buffer which will get serialized on the wire.
    self.send(from_pid, response)
```

The installed method should take `from_pid` and `message` parameters. `from_pid` is the process calling the method. `message` is a protocol buffer of the installed type.

Parameters **message_type** (*A generated protocol buffer stub*) – Incoming messages to this message_type will be dispatched to this method.

send(*to*, *message*)

Send a message to another process.

Same as `Process.send` except that `message` is a protocol buffer.

Returns immediately.

Parameters

- **to** (`PID`) – The pid of the process to send a message.
- **message** – The message to send

Raises Will raise a `Process.UnboundProcess` exception if the process is not bound to a context.

Returns Nothing

1.4 Contexts

```
from compactor.context import Context

context = Context(ip='127.0.0.1', port=8081)
context.start()

ping_process = PingProcess('ping')
ping_pid = context.spawn(ping_process)

context.join()
```

class `compactor.context.`**Context**(*delegate=''*, *loop=None*, *ip=None*, *port=None*)

A compactor context.

Compactor contexts control the routing and handling of messages between processes. At its most basic level, a context is a listening (ip, port) pair and an event loop.

__init__ (*delegate=''*, *loop=None*, *ip=None*, *port=None*)
Construct a compactor context.

Before any useful work can be done with a context, you must call `start` on the context.

Parameters

- **ip** (`str` or None) – The ip port of the interface on which the Context should listen. If none is specified, the context will attempt to bind to the ip specified by the `LIBPROCESS_IP` environment variable. If this variable is not set, it will bind on all interfaces.
- **port** (`int` or None) – The port on which the Context should listen. If none is specified, the context will attempt to bind to the port specified by the `LIBPROCESS_PORT` environment variable. If this variable is not set, it will bind to an ephemeral port.

delay (*amount*, *pid*, *method*, **args*)
Call a method on another process after a specified delay.

This is equivalent to `dispatch` except with an additional amount of time to wait prior to invoking the call.

This function returns immediately.

Parameters

- **amount** (`float` or `int`) – The amount of time to wait in seconds before making the call.
- **pid** (`PID`) – The pid of the process to be called.
- **method** (`str`) – The name of the method to be called.

Returns Nothing

dispatch (*pid*, *method*, **args*)
Call a method on another process by its pid.

The method on the other process does not need to be installed with `Process.install`. The call is serialized with all other calls on the context's event loop. The pid must be bound to this context.

This function returns immediately.

Parameters

- **pid** (`PID`) – The pid of the process to be called.
- **method** (`str`) – The name of the method to be called.

Returns Nothing

link (*pid*, *to*)
Link a local process to a possibly remote process.

Note: It is more idiomatic to call `link` directly on the bound Process object instead.

When `pid` is linked to `to`, the termination of the `to` process (or the severing of its connection from the Process `pid`) will result in the local process' `exited` method to be called with `to`.

This method returns immediately.

Parameters

- **pid** (`PID`) – The pid of the linking process.
- **to** (`PID`) – The pid of the linked process.

Returns Nothing

send(*from_pid*, *to_pid*, *method*, *body=None*)
Send a message method from one pid to another with an optional body.

Note: It is more idiomatic to send directly from a bound process rather than calling send on the context.

If the destination pid is on the same context, the Context may skip the wire and route directly to process itself. `from_pid` must be bound to this context.

This method returns immediately.

Parameters

- **from_pid** (`PID`) – The pid of the sending process.
- **to_pid** (`PID`) – The pid of the destination process.
- **method** (`str`) – The method name of the destination process.
- **body** (`bytes` or None) – Optional content to send along with the message.

Returns Nothing

spawn(*process*)
Spawn a process.

Spawning a process binds it to this context and assigns the process a pid which is returned. The process' `initialize` method is called.

Note: A process cannot send messages until it is bound to a context.

Parameters **process** (`Process`) – The process to bind to this context.

Returns The pid of the process.

Return type `PID`

start()
Start the context. This method must be called before calls to `send` and `spawn`.

This method is non-blocking.

stop()
Stops the context. This terminates all PIDs and closes all connections.

terminate(*pid*)
Terminate a process bound to this context.

When a process is terminated, all the processes to which it is linked will be have their `exited` methods called. Messages to this process will no longer be delivered.

This method returns immediately.

Parameters **pid** (`PID`) – The pid of the process to terminate.

Returns Nothing

CHAPTER 2

getting started

implementing a process is a matter of subclassing `compactor.Process`. you can "install" methods on processes using the `install` decorator. this makes them remotely callable.

```
import threading

from compactor import install, spawn, Process

class PingProcess(Process):
  def initialize(self):
    self.pinged = threading.Event()

  @install('ping')
  def ping(self, from_pid, body):
    self.pinged.set()

# construct the process
ping_process = PingProcess('ping_process')

# spawn the process, binding it to the current global context
spawn(ping_process)

# send a message to the process
client = Process('client')
spawn(client)
client.send(ping_process.pid, 'ping')

# ensure the message was delivered
ping_process.pinged.wait()
```

each context is, in essence, a listening (ip, port) pair.

by default there is a global, singleton context. use `compactor.spawn` to spawn a process on it. by default it will bind to `0.0.0.0` on an arbitrary port. this can be overridden using the `LIBPROCESS_IP` and `LIBPROCESS_PORT` environment variables.

alternately, you can create an instance of a `compactor.Context`, explicitly passing it `port=` and `ip=` keywords. you can then call the `spawn` method on it to bind processes.

spawning a process does two things: it binds the process to the context, creating a pid, and initializes the process. the pid is a unique identifier used for routing purposes. in practice, it consists of an (ip, port, name) tuple, where the ip and port are those of the context, and the name is the name of the process.

when a process is spawned, its `initialize` method is called. this can be used to initialize state or initiate connections to other services, as illustrated in the following example.

CHAPTER 3

leader/follower pattern

```
import threading
import uuid
from compactor import install
from compactor.process import Process

class Leader(Process):
  def __init__(self):
    super(Leader, self).__init__('leader')
    self.followers = set()

  @install('register')
  def register(self, from_pid, uuid):
    self.send(from_pid, 'registered', uuid)

class Follower(Process):
  def __init__(self, name, leader_pid):
    super(Follower, self).__init__(name)
    self.leader_pid = leader_pid
    self.uuid = uuid.uuid4().bytes
    self.registered = threading.Event()

  def initialize(self):
    super(Follower, self).initialize()
    self.send(self.leader_pid, 'register', self.uuid)

  def exited(self, from_pid):
    self.registered.clear()

  @install('registered')
  def registered(self, from_pid, uuid):
    if uuid == self.uuid:
      self.link(from_pid)
      self.registered.set()
```

with this, you can create two separate contexts:

```
from compactor import Context

leader_context = Context(port=5051)
leader = Leader()
leader_context.spawn(leader)

# at this point, leader_context.pid is a unique identifier for this leader process
```

```
# and can be disseminated via service discovery or passed explicitly to other services,
# e.g. 'leader@192.168.33.2:5051'.  the follower can be spawned in the same process,
# in a separate process, or on a separate machine.

follower_context = Context()
follower = Follower('follower1', leader_context.pid)
follower_context.spawn(follower)

follower.registered.wait()
```

this effectively initiates a handshake between the leader and follower processes, a common pattern building distributed systems using the actor model.

the `link` method links the two processes together. should the connection be severed, the `exited` method on the process will be called.

CHAPTER 4

protocol buffer processes

mesos uses protocol buffers over the wire to support RPC. compactor supports this natively. simply subclass `ProtobufProcess` instead and use `ProtobufProcess.install`

```
from compactor.process import ProtobufProcess
from service_pb2 import ServiceRequestMessage, ServiceResponseMessage

class Service(ProtobufProcess):
  @ProtobufProcess.install(ServiceRequestMessage)
  def request(self, from_pid, message):
    # message is a deserialized protobuf ServiceRequestMessage
    response = ServiceResponseMessage(...)
    # self.send automatically serializes the response, a protocol buffer, over the wire.
    self.send(from_pid, response)
```

C

Symbols

C

D

E

F

I

J

L

P

R

S

T