xRPC Documentation

Andrey Cizov

Aug 07, 2019

Contents:

CHAPTER 1

Indices and tables

- genindex
- modindex
- search

1.1 Core functionality

1.1.1 DSL

class xrpc.dsl.**RPCType**
The calling convention of the RPC point

Durable = 2
we only make sure the packet is received and do not wait for reply (UNDECIDED-RECEIVED)

Repliable = 1
Reply is expected from the receiver (OK-RECEIVED) Beware this does dead-lock services when they both try to send a repliable request at the same time to each other

Signalling = 3
we don't care if the packet is received (UNDECIDED-UNDECIDED)

class xrpc.dsl.**regular**(*initial*, *tick*)

initial
Initial wait time in seconds

tick
Run this function on every tick (this should affect the wait times)

class xrpc.dsl.**rpc**(*type*, *group*, *exc*)

exc
If an exception is raised while processing the packet from the transport, run this one

group
Alias for field number 1

type
Selected calling convention for the RPC call

class xrpc.dsl.**signal**(*codes*)

codes
Connect to this signal number

class xrpc.dsl.**startup**(*empty*)

empty
Alias for field number 0

1.1.2 The christmas-tree example

```
import logging
import random

from typing import Dict

from xrpc.dsl import rpc, RPCType, regular, signal
from xrpc.error import TerminationException
from xrpc.runtime import sender, service
# todo: the issue is actually that not only the request-reply pattern wouldn't work
# todo: but also the fact that an RPC might have circular dependencies
from xrpc.transport import Origin


# todo: please note that Request-Reply pattern does would not work with a service␣
↪that tries
# todo: to access itself.


class ExemplaryRPC:
    def __init__(self):
        # todo save the required local state here
        self.should_exit = False

    @rpc(RPCType.Signalling)
    def move_something(self, id: int, *xyzargs: int, pop: str, pip: int = 2,␣
↪**zzargs: int):
        #print('call made', 'ms', id, xyzargs, pop, pip, zzargs)
        # so we pack a call with args and kwargs and let the deserializer guess the␣
↪contents
        # how do we write a proper deserializer in such a scenario?
        pass

        # todo: we need an ability to save the sender
        # todo: we need an ability to automatically transform the sender to a␣
↪relevant object
```

(continues on next page)

(continued from previous page)

```
    @rpc(RPCType.Repliable)
    def reply(self, id: int, *xyzargs: int, pop: str, pip: int = 2, **zzargs: int) ->␣
↪float:
        # so we pack a call with args and kwargs and let the deserializer guess the␣
↪contents
        # how do we write a proper deserializer in such a scenario?

        return random.random()

    @regular()
    def regularly_executable(self, id: int = 1) -> int:
        return 1

    @regular()
    def regularly_executable_def(self, id: int = 1, b=6, a=5) -> int:
        return 2

    @regular()
    def regularly_executable_def2(self, id: int = 1, b=6, a=5) -> float:
        return 3

    @rpc(RPCType.Repliable)
    def exit(self):
        self.should_exit = True

    @regular()
    def exit_checket(self) -> float:
        if self.should_exit:
            raise TerminationException()
        return 1

    @signal()
    def on_exit(self) -> bool:
        # return True if we'd like to actually exit.
        # todo: save the relevant local state here.
        raise TerminationException()


class BroadcastClientRPC:
    def __init__(self, broadcast_addr: Origin):
        self.broadcast_addr = broadcast_addr
        self.pings_remaining = 5

    @rpc(type=RPCType.Signalling)
    def ping(self):
        self.pings_remaining -= 1
        logging.getLogger(__name__ + '.' + self.__class__.__name__).info(f'%d', self.
↪pings_remaining)

        if self.pings_remaining <= 0:
            raise TerminationException()

    @regular()
    def broadcast(self) -> float:
        while True:
            s = service(BroadcastRPC, self.broadcast_addr)
```

(continues on next page)

(continued from previous page)

```
            s.arrived()

            return 0.05


class BroadcastRPC:
    def __init__(self):
        self.origins: Dict[Origin, int] = {}
        self.origins_met = set()

    @rpc(type=RPCType.Signalling)
    def arrived(self):
        sdr = sender()
        self.origins[sdr] = 5

        if sdr not in self.origins_met:
            self.origins_met.add(sdr)

    @regular()
    def broadcast(self) -> float:
        for x in self.origins:
            c = service(BroadcastClientRPC, x)
            c.ping()

        for k in list(self.origins.keys()):
            self.origins[k] -= 1

            if self.origins[k] <= 0:
                del self.origins[k]

        logging.getLogger(__name__ + '.' + self.__class__.__name__).info(
            f'%d %s %s',
            len(self.origins_met), self.origins_met, self.origins)

        if len(self.origins_met) == 1 and len(self.origins) == 0:
            raise TerminationException()

        return 0.05
```

Python Module Index

x

Index