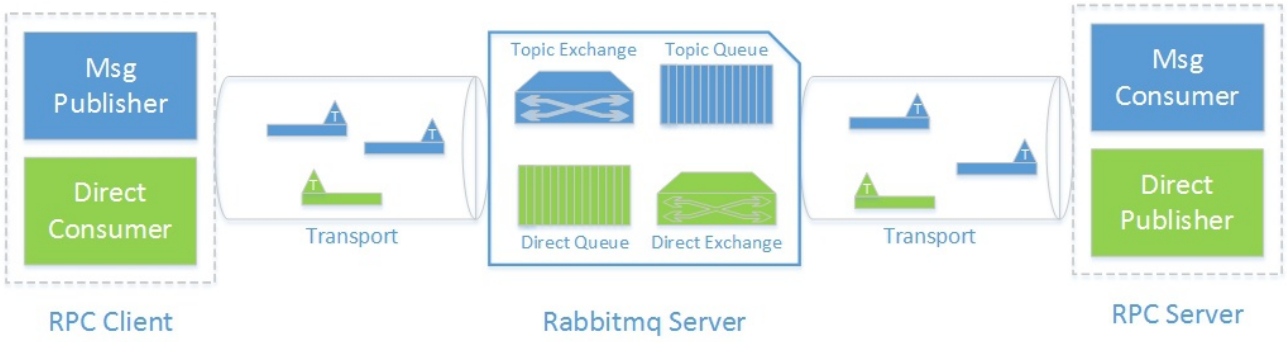
# RPC in oslo.messaging

This post introduces the logic of rpc client in oslo.messaging, which is widely used in Openstack ecological system.

## 1. Overview of RPC in oslo.messaging

The following picture is quoted from openstack official website, and you could read this linkto understand the components in this picture.




### The basic idea of RPC in oslo.messaging

The full name of RPC is "remote procedure call", and it is used widely in distributed systems. The "invoker" is the client side, and a client usually call a method and the method is run on the "worker", i.e., the server side.
Two kind of RPC semantics are supported in oslo:

- Call: a blocking RPC request, and the client will wait for the response from the server side.
- Cast: a non-blocking RPC request, and the client won't wait for any response.

In oslo.messaing, the RPC is implemented with messaging middle-ware.
For "call" semantic, two queues will be involved. The client and server are appointed to some queue, and the client side will put its request into this queue. The server receives the request and run corresponding logic and generates the response. The response is put into the reply queue. The client side pull the response and finish this round RPC.
For "cast" semantic, only one queue will be involved. Only the appointed queue, no reply

queue.

With the help of messaging middle-ware, the client request and the server response is asynchronous and decoupling.

### More details for call semantic

We take rabbitmq as an example of messaging middle-ware and introduce more details in a call semantic.

1. initialization: the rpc client and rpc server construct the transport object and listen on the same topic.
2. the 'msg publisher' in rpc client sends the request to the appointed topic by specify the target for this request. The request also specifies the name of the reply queue and the message id.
3. the 'msg consumer' in rpc server pulls the request and handle it.
4. the rpc server starts the direct publisher and send the response to the reply queue. The response is bound to the request message id.
5. the 'direct consumer' in rpc client pulls the response from the reply queue and give the response to proper call thread based on the message id.

## 2. Analyze the implementation on source code level

The following sections, I will analyze part of the source code. The code you can find here. It is very similar between RPC client and RPC server, so I take the client side as an example.

### RPCClient

```
# constructor
def __init__(self, transport, target,
             timeout=None, version_cap=None, serializer=None, retry=None):


# how to use it
```

```
class TestClient(object):
    def __init__(self, transport):
        target = messaging.Target(topic='test', version='2.0')
        self._client = messaging.RPCClient(transport, target)
    def test(self, ctxt, arg):
        return self._client.call(ctxt, 'test', arg=arg)
```

The comments in the source code are very clear. RPCClient has two important methods. "call", which invokes a method and wait for a reply. "cast", which invokes a method and return immediately. The RPCClient is an abstraction, and the implementation is in "_CallContext". Let's take a look at the implementations of the "call" method and "cast" method.

```
def call(self, ctxt, method, **kwargs):
    ...
    result = self.transport._send(self.target, msg_ctxt, msg,
                                  wait_for_reply=True, timeout=timeout,
                                  retry=self.retry)
    ...
    return self.serializer.deserialize_entity(ctxt, result)

def cast(self, ctxt, method, **kwargs):
    ...
    self.transport._send(self.target, ctxt, msg, retry=self.retry)
    ...
```

Note that the "call" method and the "cast" method both call the "_send" method of transport, which is one of the constructor parameter of RPCClient. The message to send and the "target" are changed to be parameters of "_send" method. There are two main differences between these two methods:

1. The "call" method returns the results while the "cast" method returns nothing, even something like a "future" in JAVA.
2. The "call" method has two more parameters: "wait_for_reply=True" and "timeout=timeout". It means the send method should wait until the server replies with in "timeout".

In a word, the "RPCClient" is like a wrapper of "transport" and exposes "call" and "cast" beyond the "_send" method of "transport".

## Transport

```
# constructor
def __init__(self, driver):
        self.conf = driver.conf
        self._driver =driver

# _send method
def _send(self, target, ctxt, message, wait_for_reply=None, timeout=None,
              retry=None):
        if not target.topic:
            raise exceptions.InvalidTarget('A topic is required to send',
target)
        return self._driver.send(target, ctxt, message,
                                 wait_for_reply=wait_for_reply,
                                 timeout=timeout, retry=retry)

# cleanup method
def cleanup(self):
        """Release all resources associated with this transport."""
        self._driver.cleanup()
```

Oh no.. It seems that "transport" is also an abstraction. It wraps a "_dirver" and the "_send" method and the "cleanup" method are all implemented by this driver. The driver could be "Rabbitmq" or "Zeromq" and so on and it handles the essential connection, sending and receiving.

Actually, we usually don't construct a "transport" directly. We just use the factory method, as follows:

```
# the factory method
def get_transport(conf, url=None, allowed_remote_exmods=None, aliases=None):
    allowed_remote_exmods = allowed_remote_exmods or[]
```

```
    conf.register_opts(_transport_opts)

    if not isinstance(url, TransportURL):
        url = url or conf.transport_url
        parsed = TransportURL.parse(conf, url, aliases)
        if not parsed.transport:
            raise InvalidTransportURL(url, 'No scheme specified in "%s"' %url)
        url =parsed

    kwargs = dict(default_exchange=conf.control_exchange,
                  allowed_remote_exmods=allowed_remote_exmods)

    try:
        mgr = driver.DriverManager('oslo.messaging.drivers',
                                   url.transport.split('+')[0],
                                   invoke_on_load=True,
                                   invoke_args=[conf, url],
                                   invoke_kwds=kwargs)
    except RuntimeError asex:
        raise DriverLoadFailure(url.transport, ex)

    return Transport(mgr.driver)

# how we get a transport
self.transport = oslo_messaging.get_transport(CONF)
```

The "CONF" is usually a global object which contains most of the configurations. The configurations related to "transport" contain the driver name, the server url of the driver, the username and password to use the driver server and so on. The factory method constructs the "TransportURL" first, which marks a "transport" and then gets the driver manager with "DriverManager".

The "DriverManager" implements a mechanism like reflection in JAVA.

```
# constructor
def __init__(self, namespace, name,
```

```
                invoke_on_load=False, invoke_args=(), invoke_kwds={},
                on_load_failure_callback=None,
                verify_requirements=False):
```

Drivers should be registered at somewhere and the "DriverManager" can then initialize it based on "namespace" and "name" with "invoke_args".
The driver condidates are defined in the "setup.cfg" config file, as follows:

```
oslo.messaging.drivers =
    rabbit = oslo_messaging._drivers.impl_rabbit:RabbitDriver
    zmq = oslo_messaging._drivers.impl_zmq:ZmqDriver
    amqp = oslo_messaging._drivers.impl_amqp1:ProtonDriver
```

More details for this part is beyond this wiki.

## Target

Before deep into the driver implementation, let's take a look at another parameter of RPCClient.

```
# constructor
def __init__(self, exchange=None, topic=None, namespace=None,
             version=None, server=None, fanout=None,
             legacy_namespaces=None):

# how we get target
target = oslo_messaging.Target(topic=self.topic,
                               server=self.server)
```

As the comment said, a "Target" encapsulates all the information to identify where a message should be sent or what messages a server is listening for. A RPC client should appoint the "topic" and optionally the "server" it wants to call. A target is like a bond between the RPC client and the RPC server. A target has nothing with the "connection".

## RabbitDriver

The Openstack implements many drivers. I take the default one, i.e., "RabbitDriver" as an example.

```
def __init__(self, conf, url,
             default_exchange=None,
             allowed_remote_exmods=None):
    ...
    connection_pool = pool.ConnectionPool(
        conf, conf.oslo_messaging_rabbit.rpc_conn_pool_size,
url, Connection)
    super(RabbitDriver, self).__init__(
conf, url,
connection_pool,
default_exchange,
allowed_remote_exmods
)
```

"RabbitDriver" almost has no implementation. Most importantly, it initializes a connection pool.

```
# constructor of ConnectionPool
def __init__(self, conf, rpc_conn_pool_size, url, connection_cls)

# create a instance to add to the pool
def create(self, purpose=None):
    if purpose is None:
        purpose = common.PURPOSE_SEND
    LOG.debug('Pool creating new connection')
    return self.connection_cls(self.conf, self.url, purpose)
```

The pool size is defined by the configuration "oslo_messaging_rabbit.rpc_conn_pool_size". The type in the pool is "Connection".

All the implementation methods are in the base class of "RabbitDriver".

## AMQPDriverBase

Aha, the big boss comes on the stage。

```
# the implementation of send method
def _send(self, target, ctxt, message,
          wait_for_reply=None, timeout=None,
          envelope=True, notify=False, retry=None):


    ...
    msg =message

    ifwait_for_reply:
        msg_id = uuid.uuid4().hex
        msg.update({'_msg_id': msg_id})
        msg.update({'_reply_q': self._get_reply_q()})
    ...
    ifwait_for_reply:
        self._waiter.listen(msg_id)
        log_msg = "CALL msg_id: %s " %msg_id
    else:
        log_msg = "CAST unique_id: %s " %unique_id

    try:
        with self._get_connection(rpc_common.PURPOSE_SEND) asconn:
            ifnotify:
                ...
                conn.notify_send(exchange, target.topic, msg, retry=retry)
            elif target.fanout:
                ...
                conn.fanout_send(target.topic, msg, retry=retry)
            else:
                topic = target.topic
                exchange = self._get_exchange(target)
                if target.server:
                    topic = '%s.%s' % (target.topic, target.server)
                ...
                conn.topic_send(exchange_name=exchange, topic=topic,
                                msg=msg, timeout=timeout, retry=retry)
```

```
            ifwait_for_reply:
                result = self._waiter.wait(msg_id, timeout)
                if isinstance(result, Exception):
                    raiseresult
                returnresult
        finally:
            ifwait_for_reply:
                self._waiter.unlisten(msg_id)
```

The code is a bit long so I removed unimportant lines. Do you still remember we set a parameter called "wait_for_reply" for "call" method in "transport". All the "wait_for_reply" branches are used for "call" method.

If this is a "call" method, a random message id, which is the message id for the response, will be generated and the queue used for reply will be set into the request message. A " waiter" then adds the message id into his waiting list and wait for this message within " timeout". Finally, the "waiter" removes the message id from his list.

The "target" is used before sending the message. The info on topic and server is joint and used as a param of the send method of "conn". The "conn" is get from "_get_connection" method, to simplify, a "conn" just a object picked form the connection pool and its type is "Connection".

Finally, the "send" operation is implemented by the "Connection". The "Connection" is a little complicated and the "waiter" also uses this class, so let's take a look at "waiter" first.

## ReplyWaiter

```
# construct ReplyWaiter
def _get_reply_q(self):
    with self._reply_q_lock:
        if self._reply_q is not None:
```

```
            return self._reply_q

        reply_q = 'reply_' + uuid.uuid4().hex

        conn = self._get_connection(rpc_common.PURPOSE_LISTEN)

        self._waiter =ReplyWaiter(reply_q, conn,
                                  self._allowed_remote_exmods)

        self._reply_q =reply_q
        self._reply_q_conn =conn

    return self._reply_q

# constructor of ReplyWaiter
def __init__(self, reply_q, conn, allowed_remote_exmods):
    self.conn =conn
    self.allowed_remote_exmods =allowed_remote_exmods
    self.msg_id_cache = rpc_amqp._MsgIdCache()
    self.waiters =ReplyWaiters()

    self.conn.declare_direct_consumer(reply_q, self)

    self._thread_exit_event = threading.Event()
    self._thread = threading.Thread(target=self.poll)
    self._thread.daemon = True
    self._thread.start()
```

Note that reply queue can only be set once in one driver, and the construction of "ReplyWaiter" is in this lock, so the one driver only have one "ReplyWaiter". The connection the " ReplyWaiter" used is also get from the connection pool.

In the constructor of "ReplyWaiter", the "declare_direct_consumer" of "Connection" is called. The waiters works like a cache and it contains the replied message. "ReplyWaiter" also instantiates a daemon thread, and this thread runs the "poll" method.

```
def poll(self):
        while not self._thread_exit_event.is_set():
            try:
                self.conn.consume()
            except Exception:
                LOG.exception(_LE("Failed to process incoming message, "
                                  "retrying..."))
```

The logic is extremely simple. It continues to call the "consume" method of "Connection" until the "thread_exit_event" is set. When "Connection" gets the message, it will call back the caller. The "ReplyWaiter" is the caller and it implements the "call" method.

```
 def __call__(self, message):
        message.acknowledge()
        incoming_msg_id = message.pop('_msg_id', None)
        if message.get('ending'):
            LOG.debug("received reply msg_id: %s", incoming_msg_id)
        self.waiters.put(incoming_msg_id, message)
```

Briefly, it just puts the message into the cache "waiters". Then, when we call the "wait" method in "send" method of driver, we get the message from "waiters".

```
def wait(self, msg_id, timeout):
        timer = rpc_common.DecayingTimer(duration=timeout)
        timer.start()
        final_reply = None
        ending = False
        while notending:
            timeout = timer.check_return(self._raise_timeout_exception, msg_id)
            try:
                message = self.waiters.get(msg_id, timeout=timeout)
            except moves.queue.Empty:
                self._raise_timeout_exception(msg_id)

            reply, ending = self._process_reply(message)
            if reply is not None:
```

```
            final_reply =reply
        returnfinal_reply
```

In a word, the "ReplyWaiter" consumes the data in reply queue continuously with a daemon thread and returns it to the driver.

## Connection

OK, here comes the final part----"Connection".
First of all, you need to be aware of the "Rabbitmq" doesn't implement the communication by itself. It just encapsulate the implementation of "kombu". We don't dig deep into the details of "kombu". We focus on the calling logic.

The implementation of "direct consumer" called by "ReplyWaiter" is "kombu direct consumer". We ignore the details of this class and take a look at the "consume" method of "Connection".

```
def _consume():
        ...
        poll_timeout = (self._poll_timeout if timeout is None
                        else min(timeout, self._poll_timeout))
        while True:
            if self._consume_loop_stopped:
                return
            ...
            try:
                self.connection.drain_events(timeout=poll_timeout)
                return
            except socket.timeout asexc:
                poll_timeout = timer.check_return(
                    _raise_timeout, exc, maximum=self._poll_timeout)
```

Do you remember that the "consume" method is called by the daemon thread of "ReplyWaiter" in a loop. So the mechanism is the "consumer" always try to consume data from the "

reply queue" continuously until some condition flags is set.

As you wish, the implementation of "topic_send" method is from "kombu" as well.

## The calling procedure of "cleanup" method

When we try to stop our service and all the object and connections on RPC should be released. As mentioned before, the "RPCClient" is just a wrapped class. The only thing we need to cleanup is the driver.

The entry point of the "cleanup" method of the driver is in "transport".

```
def cleanup(self):
    """Release all resources associated with this transport."""
    self._driver.cleanup()
```

The implementation of the "cleanup" method in driver is as follows:

```
def cleanup(self):
    if self._connection_pool:
        self._connection_pool.empty()
    self._connection_pool = None

    with self._reply_q_lock:
        if self._reply_q is not None:
            self._waiter.stop()
            self._reply_q_conn.close()
            self._reply_q_conn = None
            self._reply_q = None
            self._waiter = None
```

First, we need to release all the connections in the "connection pool". Then, if the reply queue is not "None", we need to release the resources of it. The "stop" method of " ReplyWaiter" is as follows:

```
# stop method of ReplyWaiter
def stop(self):
        if self._thread:
            self._thread_exit_event.set()
            self.conn.stop_consuming()
            self._thread.join()
            self._thread = None

# stop_consuming method of Connection
def stop_consuming(self):
        self._consume_loop_stopped = True
```

Fine, the "thread_exit_event" flag is set and the loop in the "pool" method of "ReplyWaiter" should not continue. The "consume_loop_stopped" flag is set and the loop in the "consume" method of "Connection" should not continue.

"_thread.join()" is a little tricky. When the "join" method is not passed in a "timeout" parameter, this thread will not be quitted immediately or wait for timeout. It keeps waiting until the thread exit by itself. It means if there is a blocking method in this method, the "join" method needs to wait.

Do we have a blocking method in this thread? We do have!

```
self.connection.drain_events(timeout=poll_timeout)
```

This method doesn't return until it gets the message or the timeout is reached. So, the "join" method have to wait for this "timeout" when the connection don't receive any message.

## 3. Conclusion

In this section, we conclude the classes in oslo.messaging.

1. RPCClient: An encapsulation of "transport". Expose two methods. The "call" method invokes a method and waits for a reply. The "cast" method invokes a method and return immediately. These two methods both call the "_send" method of "transport".

2. Transport: An abstraction of the connection between RPC client and the server of the driver. All the configurations about the driver is sent through this class. With the help of " DriverManager" provided by "stevedore", the transport can load different drivers without changing its interface. Wrap a driver in it and all the implementation is handed over to the driver.
3. Target: A bound between RPC client and RPC server. It contains all the information about topic and exchange and how to deliver a request to the RPC server.
4. Driver: The implementation of the RPC. It wraps a connection pool, which is used to maintain the connections between the RPC client and the server of the driver. All the "send" and "wait for reply" methods are implemented here. Based on several flags, the behavior is different between "call" method and "cast" method.
5. ReplyWaiter: Used when we invoke "call" method. It starts a daemon thread to read the response from RPC server and puts the results into a cache. And it exposes a "wait" method to provide the results to the "Driver".
6. Connection: It wrap the logic of "kombu". The instances in the "connection pool" are all type "Connection". Both the "Driver" and the "ReplyWaiter" relay on the "Connection" to "send" and "consume" message.