RPyC Documentation

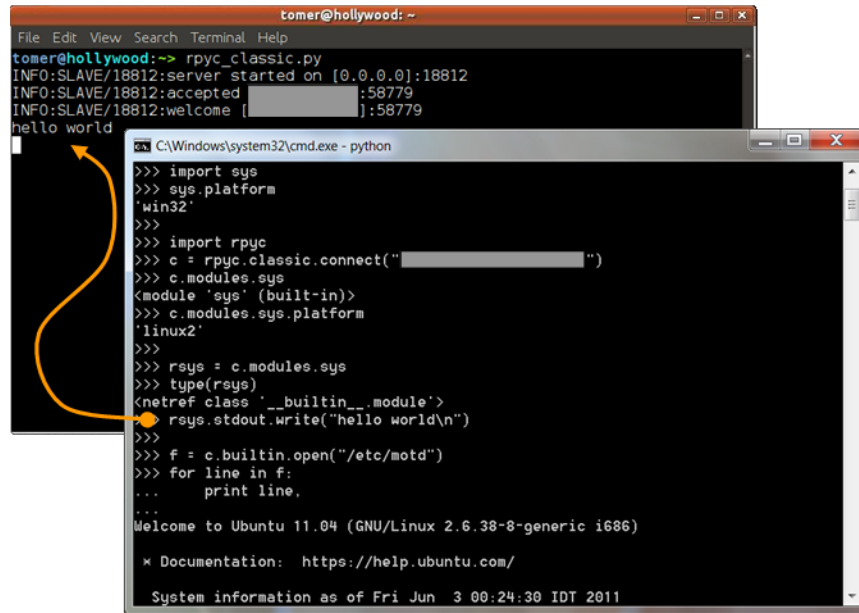*Release 6.0.0/2024-02-23*

Tomer Filiba

Feb 23, 2024

CONTENTS

Version 6.0.0 has been released!

Be sure to read the *changelog* before upgrading!

Please use the github issues to ask questions report problems. **Please do not email me directly**

RPyC (pronounced as *are-pie-see*), or *Remote Python Call*, is a **transparent** python library for **symmetrical** remote procedure calls, clustering and distributed-computing. RPyC makes use of object-proxying, a technique that employs python's dynamic nature, to overcome the physical boundaries between processes and computers, so that remote objects can be manipulated as if they were local.

Fig. 1: A screenshot of a Windows client connecting to a Linux server. Note that text written to the server's `stdout` is actually printed on the server's console.

GETTING STARTED

Installing RPyC is as easy as `pip install rpyc`.

If you're new to RPyC, be sure to check out the *Tutorial*. Next, refer to the *Documentation* and *API Reference*, as well as the *Mailing List*.

For an introductory reading (that may or may not be slightly outdated), David Mertz wrote a very thorough Charming Python installment about RPyC, explaining how it's different from existing alternatives (Pyro, XMLRPC, etc.), what roles it can play, and even show-cases some key features of RPyC (like the security model, remote monkey-patching, or remote resource utilization).

TWO

FEATURES

- **Transparent** - access to remote objects as if they were local; existing code works seamlessly with both local or remote objects.

- **Symmetric** - the protocol itself is completely symmetric, meaning both client and server can serve requests. This allows, among other things, for the server to invoke callbacks on the client side.

- **Synchronous** and *asynchronous* operation

- **Platform Agnostic** - 32/64 bit, little/big endian, Windows/Linux/Solaris/Mac… access objects across different architectures.

- **Low Overhead** - RpyC takes an *all-in-one* approach, using a compact binary protocol, and requiring no complex setup (name servers, HTTP, URL-mapping, etc.)

- **Secure** - employs a Capability based security model; integrates easily with SSH

- **Zero-Deploy Enabled** – Read more about *Zero-Deploy RPyC*

- **Integrates** with TLS/SSL, SSH and inetd.

USE CASES

- Excels in testing environments – run your tests from a central machine offering a convenient development environment, while the actual operations take place on remote ones.

- Control/administer multiple hardware or software platforms from a central point: any machine that runs Python is at your hand! No need to master multiple shell-script languages (BASH, Windows batch files, etc.) and use awkward tools (`awk`, `sed`, `grep`, . . .)

- Access remote hardware resources transparently. For instance, suppose you have some proprietary electronic testing equipment that comes with drivers only for HPUX, but no one wants to actually use HPUX. . . just connect to the machine and use the remote `ctypes` module (or open the `/dev` file directly).

- Monkey-patch local code or remote code. For instance, using monkey-patching you can cross network boundaries by replacing the `socket` module of one program with a remote one. Another example could be replacing the `os` module of your program with a remote module, causing `os.system` (for instance) to run remotely.

- Distributing workload among multiple machines with ease

- Implement remote services (like WSDL or RMI) quickly and concisely (without the overhead and limitations of these technologies)

CONTENTS

4.1 Download and Install

You can always download the latest releases of RPyC from the project's github page or its PyPI page. The easiest way to install RPyC, however, is using:

```
pip install rpyc
```

If you don't want to mess with virtualenvs or mess with system directories, install as user:

```
pip install rpyc --user
```

Be sure to read the *changelog* before upgrading versions! Also, always link your own applications against a fixed major version of rpyc!

4.1.1 Platforms and Interpreters

RPyC is a pure-python library, and as such can run on any architecture and platform that runs python (or one of its other implementations), both 32- and 64-bit. This is also true for a client and its server, which may run on different architectures. The latest release supports:

- **Python** (CPython) 2.7-3.7

- May work on py2.6

- May work with **Jython** and **IronPython**. However, these are not primary concerns for me. Breakage may occur at any time.

Cross-Interpreter Compatibility

Note that you **cannot** connect from a **Python 2.x** interpreter to a **3.x** one, or vice versa. Trying to do so will results in all kinds of strange exceptions, so beware. This is because Python 3 introduces major changes to the object model used by Python 2.x: some types were removed, added or unified into others. Byte- and Unicode- strings gave me a nightmare (and they still account for many bugs in the core interpreter). On top of that, many built-in modules and functions were renamed or removed, and many new language features were added. These changes render the two major versions of Python incompatible with one another, and sadly, this cannot be bridged automatically by RPyC at the serialization layer.

It's not that I didn't try – it's just too hard a feat. It's basically like writing a 100% working 2to3 tool, alongside with a matching `3to2` one; and that, I reckon, is comparable to the *halting problem* (of course I might be wrong here, but it still doesn't make it feasible).

Big words aside – you can connect from **Python 2.x to Python 2.y**, as long as you only use types/modules/features supported by both; and you can connect from **Python 3.x to Python 3.y**, under the same assumption.

Note: As a side note, do not try to mix different versions of RPyC (e.g., connecting a client machine running RPyC 3.1.0 to a server running RPyC 3.2.0). The wire-protocol has seen little changes since the release of RPyC 3.0, but the library itself has changed drastically. This might work, but don't count on it.

4.2 Development

4.2.1 Mailing List

There is an old mailing list that may contain useful information and that you should search before asking questions. Nowadays however, do not count on getting any answers for new questions there.

4.2.2 Repository

RPyC is developed on github, where you can always find the latest code or fork the project.

4.2.3 Bugs and Patches

We're using github's issue tracker for bug reports, feature requests, and overall status.

Patches are accepted through github pull requests.

4.2.4 Dependencies

The core of RPyC has no external dependencies, so you can use it out of the box for "simple" use. However, RPyC integrates with some other projects to provide more features, and if you wish to use any of those, you must install them:

- PyWin32 - Required for `PipeStream` on Windows
- zlib for IronPython - Required for IronPython prior to v2.7

4.3 Tutorial

Here's a little tutorial to get you started with RPyC in no time:

4.3.1 Part 1: Introduction to *Classic RPyC*

We'll kick-start the tutorial with what is known as *classic-style* RPyC, i.e., the methodology of RPyC 2.60. Since RPyC 3 is a complete redesign of the library, there are some minor changes, but if you were familiar with RPyC 2.60, you'll feel right at home. And even if you were not – we'll make sure you feel at home in a moment ;)

Running a Server

Let's start with the basics: running a server. In this tutorial we'll run both the server and the client on the same machine (the `localhost`). The classic server can be started using:

```
$ python bin/rpyc_classic.py
INFO:SLAVE/18812:server started on [127.0.0.1]:18812
```

This shows the parameters this server is running with:

- `SLAVE` indicates the `SlaveService` (you'll learn more about *services* later on), and

- `[127.0.0.1]:18812` is the address on which the server binds, in this case the server will only accept connections from localhost. If you run a server with `--host 0.0.0.0`, you are free for arbitrary code execution from anywhere.

Running a Client

The next step is running a client which connects to the server. The code needed to create a connection to the server is quite simple, you'd agree

```python
import rpyc
conn = rpyc.classic.connect("localhost")
```

If your server is not running on the default port (TCP `18812`), you'll have to pass the `port=` parameter to *classic. connect()*.

The `modules` Namespace

The `modules` property of connection objects exposes the server's module-space, i.e., it lets you access remote modules. Here's how:

```python
rsys = conn.modules.sys      # remote module on the server!
```

This *dot notation* only works for top level modules. Whenever you would require a nested import for modules contained within a package, you have to use the *bracket notation* to import the remote module, e.g.:

```python
minidom = conn.modules["xml.dom.minidom"]
```

With this alone you are already set to do almost anything. For example, here is how you see the server's command line:

```python
>>> rsys.argv
['bin/rpyc_classic.py']
```

. . . add module search paths for the server's import mechanism:

```python
>>> rsys.path.append('/tmp/totally-secure-package-location)
```

. . . change the current working directory of the server process:

```python
>>> conn.modules.os.chdir('..')
```

. . . or even print something on the server's stdout:

```
>>> print("Hello World!", file=conn.modules.sys.stdout)
```

The `builtins` Namespace

The `builtins` property of classic connection exposes all builtin functions available in the server's python environment.
You could use it for example to access a file on the server:

```
>>> f = conn.builtins.open('/home/oblivious/.ssh/id_rsa')
>>> f.read()
'-----BEGIN RSA PRIVATE KEY-----\nMIIJKQIBAAKCAgEA0...XuVmz/ywq+5m\n-----END RSA PRIVATE
→KEY-----\n'
```

Ooopsies, I just leaked my private key...;)

The `eval` and `execute` Methods

If you are not satisfied already, here is more: Classic connections also have properties `eval` and `execute` that allow
you to directly evaluate arbitrary expressions or even execute arbitrary statements on the server. For example:

```
>>> conn.execute('import math')
>>> conn.eval('2*math.pi')
6.283185307179586
```

But wait, this requires that rpyc classic connections have some notion of global variables, how can you see them? They
are accessible via the `namespace` property that will be initialized as empty dictionary for every new connection. So,
after our import, we now have:

```
>>> conn.namespace
{'__builtins__': <...>, 'math': <...>}
```

The aware reader will have noticed that neither of these shenanigans are strictly needed, as the same functionality
could be achieved by using the `conn.builtins.compile()` function, which is also accessible via `conn.modules.builtins.compile()`, and manually feeding it with a remotely created dict.

That's true, but we sometimes like a bit of sugar;)

The `teleport` method

There is another interesting method that allows you to transmit functions to the other sides and execute them over there:

```
>>> def square(x):
...     return x**2
>>> fn = conn.teleport(square)
>>> fn(2)
```

This calculates the square of two as expected, but the computation takes place on the remote!

Furthermore, teleported functions are automatically defined in the remote namespace:

```
>>> conn.eval('square(3)')
9
```

```
>>> conn.namespace['square'] is fn
True
```

And the teleported code can also access the namespace:

```
>>> conn.execute('import sys')
>>> version = conn.teleport(lambda: print(sys.version_info))
>>> version()
```

prints the version on the remote terminal.

Note that currently it is not possible to teleport arbitrary functions, in particular there can be issues with closures to non-trivial objects. In case of problems it may be worth taking a look at external libraries such as dill.

Continue to *Part 2: Netrefs and Exceptions*. . .

4.3.2 Part 2: Netrefs and Exceptions

In *Part 1: Introduction to Classic RPyC*, we have seen how to use rpyc classic connection to do almost anything remotely.

So far everything seemed normal. Now it's time to get our hands dirty and understand more what happens under the hood!

Setup

Start a classic server using:

```
python bin/rpyc_classic.py
```

And connect your client:

```
>>> import rpyc
>>> conn = rpyc.classic.connect("localhost")
```

Netrefs

We know that we can use `conn.modules.sys` to access the `sys` module the server. . . But what kind of magical object is that thing anyway?

```
>>> type(conn.modules.sys)
<netref class 'builtins.module'>
```

```
>>> type(conn.modules.sys.path)
<netref class 'builtins.list'>
```

```
>>> type(conn.modules.os.path.abspath)
<netref class 'builtins.function'>
```

Voila, **netrefs** (*network references*, also known as *transparent object proxies*) are special objects that delegate everything done on them locally to the corresponding remote objects. Netrefs may not be real lists of functions or modules, but

they "do their best" to look and feel like the objects they point to. . . in fact, they even fool python's introspection mechanisms!

```
>>> isinstance(conn.modules.sys.path, list)
True

>>> import inspect
>>> inspect.isbuiltin(conn.modules.os.listdir)
True
>>> inspect.isfunction(conn.modules.os.path.abspath)
True
>>> inspect.ismethod(conn.modules.os.path.abspath)
False
>>> inspect.ismethod(conn.modules.sys.stdout.write)
True
```

Cool, eh?

We all know that the best way to understand something is to smash it, slice it up and spill the contents into the world! So let's do that:

```
>>> dir(conn.modules.sys.path)
['____conn__', '____id_pack__', '__add__', '__class__', '__contains__', '__delattr__',
'__delitem__', '__delslice__', '__doc__', '__eq__', '__ge__', '__getattribute__',
'__getitem__', '__getslice__', '__gt__', '__hash__', '__iadd__', '__imul__',
'__init__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__',
'__setitem__', '__setslice__', '__str__', 'append', 'count', 'extend', 'index', 'insert',
'pop', 'remove', 'reverse', 'sort']
```

In addition to some expected methods and properties, you will have noticed ____conn__ and ____id_pack__. These properties store over which connection the object should be resolved and an identifier that allows the server to lookup the object from a dictionary.

Exceptions

Let's continue on this exhilarating path of destruction. After all, things are not always bright, and problems must be dealt with. When a client makes a request that fails (an exception is raised on the server side), the exception propagates transparently to the client. Have a look at this snippet:

```
>>> conn.modules.sys.path[300]              # there are only 12 elements in the list...
======= Remote traceback =======
Traceback (most recent call last):
  File "D:\projects\rpyc\core\protocol.py", line 164, in _dispatch_request
    res = self._handlers[handler](self, *args)
  File "D:\projects\rpyc\core\protocol.py", line 321, in _handle_callattr
    return attr(*args, **dict(kwargs))
IndexError: list index out of range

======= Local exception ========
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "D:\projects\rpyc\core\netref.py", line 86, in method
    return self.____sync_req__(consts.HANDLE_CALLATTR, name, args, kwargs)
```

(continues on next page)

```
  File "D:\projects\rpyc\core\netref.py", line 53, in ____sync_req__
    return self.____conn__.sync_request(handler, self.____id_pack__, *args)
  File "D:\projects\rpyc\core\protocol.py", line 224, in sync_request
    self.serve()
  File "D:\projects\rpyc\core\protocol.py", line 196, in serve
    self._serve(msg, seq, args)
  File "D:\projects\rpyc\core\protocol.py", line 189, in _serve
    self._dispatch_exception(seq, args)
  File "D:\projects\rpyc\core\protocol.py", line 182, in _dispatch_exception
    raise obj
IndexError: list index out of range
>>>
```

As you can see, we get two tracebacks: the remote one, showing what went wrong on the server, and a local one, showing what we did to cause it.

Custom Exception Handling Example

The server example:

```python
import rpyc
import urllib.error
from rpyc.utils.server import OneShotServer


class HelloService(rpyc.Service):
    def exposed_foobar(self, remote_str):
        raise urllib.error.URLError("test")


if __name__ == "__main__":
    rpyc.lib.setup_logger()
    server = OneShotServer(
        HelloService,
        port=12345,
        protocol_config={'import_custom_exceptions': True}
    )
    server.start()
```

The client example:

```python
import rpyc
import urllib.error
rpyc.core.vinegar._generic_exceptions_cache["urllib.error.URLError"] = urllib.error.
↪URLError


if __name__ == "__main__":
    conn = rpyc.connect("localhost", 12345)
    try:
        print(conn.root.foobar('hello'))
```

```
    except urllib.error.URLError:
        print('caught a URLError')
```

Continue to *Part 3: Services and New Style RPyC*...

4.3.3 Part 3: Services and *New Style* RPyC

So far we have covered the features of classic RPyC. However, the new model of RPyC programming (starting with RPyC 3.00), is based on *services*. As you might have noticed in the classic mode, the client basically gets full control over the server, which is why we (used to) call RPyC servers *slaves*. Luckily, this is no longer the case. The new model is *service oriented*: services provide a way to expose a well-defined set of capabilities to the other party, which makes RPyC a generic RPC platform. In fact, the *classic RPyC* that you've seen so far, is simply "yet another" service.

Services are quite simple really. To prove that, the SlaveService (the service that implements classic RPyC) is only 30 lines long, including comments ;). Basically, a service has the following boilerplate:

```python
import rpyc

class MyService(rpyc.Service):
    def on_connect(self, conn):
        # code that runs when a connection is created
        # (to init the service, if needed)
        pass

    def on_disconnect(self, conn):
        # code that runs after the connection has already closed
        # (to finalize the service, if needed)
        pass

    def exposed_get_answer(self): # this is an exposed method
        return 42

    exposed_the_real_answer_though = 43     # an exposed attribute

    def get_question(self):  # while this method is not exposed
        return "what is the airspeed velocity of an unladen swallow?"
```

Note: The conn argument for on_connect and on_disconnect are added in rpyc 4.0. This is backwards incompatible with previous versions where instead the service constructor is called with a connection parameter and stores it into self._conn.

As you can see, apart from the special initialization/finalization methods, you are free to define the class like any other class. Unlike regular classes, however, you can choose which attributes will be exposed to the other party: if the name starts with exposed_, the attribute will be remotely accessible, otherwise it is only locally accessible. In this example, clients will be able to call get_answer, but not get_question, as we'll see in a moment.

To expose your service to the world, however, you will need to start a server. There are many ways to do that, but the simplest is

```python
# ... continuing the code snippet from above ...
```

```python
if __name__ == "__main__":
    from rpyc.utils.server import ThreadedServer
    t = ThreadedServer(MyService, port=18861)
    t.start()
```

To the remote party, the service is exposed as the root object of the connection, e.g., `conn.root`. Now you know all you need to understand this short demo:

```python
>>> import rpyc
>>> c = rpyc.connect("localhost", 18861)
>>> c.root
<__main__.MyService object at 0x834e1ac>
```

This "root object" is a reference (netref) to the service instance living in the server process. It can be used access and invoke exposed attributes and methods:

```python
>>> c.root.get_answer()
42
>>> c.root.the_real_answer_though
43
```

Meanwhile, the question is not exposed:

```python
>>> c.root.get_question()
======= Remote traceback =======
...
  File "/home/tomer/workspace/rpyc/core/protocol.py", line 298, in sync_request
    raise obj
AttributeError: cannot access 'get_question'
```

Access policy

By default methods and attributes are only visible if they start with the `exposed_` prefix. This also means that attributes of builtin objects such as lists or dicts are not accessible by default. If needed, you can configure this by passing appropriate options when creating the server. For example:

```python
from rpyc.utils.server import ThreadedServer
server = ThreadedServer(MyService, port=18861, protocol_config={
    'allow_public_attrs': True,
})
server.start()
```

For a description of all available settings see the *DEFAULT_CONFIG*.

Shared service instance

Note that we have here passed the *class* `MyService` to the server with the effect that every incoming connection will use its own, independent `MyService` instance as root object.

If you pass in an *instance* instead, all incoming connections will use this instance as their shared root object, e.g.:

```
t = ThreadedServer(MyService(), port=18861)
```

Note the subtle difference (parentheses!) to the example above.

Note: Passing instances is supported starting with rpyc 4.0. In earlier versions, you can only pass a class of which every connection will receive a separate instance.

Passing arguments to the service

In the second case where you pass in a fully constructed service instance, it is trivial to pass additional arguments to the `__init__` function. However, the situation is slightly more tricky if you want to pass arguments while separating the root objects for each connection. In this case, use `classpartial()` like so:

```python
from rpyc.utils.helpers import classpartial

service = classpartial(MyService, 1, 2, pi=3)
t = ThreadedServer(service, port=18861)
```

Note: classpartial is added in version 4.0.

But Wait, There's More!

All services have a *name*, which is normally the name of the class, minus the `"Service"` suffix. In our case, the service name is `"MY"` (service names are case-insensitive). If you wish to define a custom name, or multiple names (aliases), you can do so by setting the `ALIASES` list. The first alias is considered to be the "formal name", while the rest are aliases:

```python
class MyService(rpyc.Service):
    ALIASES = ["floop", "bloop"]
    ...
```

In the original code snippet, this is what the client gets:

```python
>>> c.root.get_service_name()
'MY'
>>> c.root.get_service_aliases()
('MY',)
```

The reason services have names is for the **service registry**: normally, a server will broadcast its details to a nearby *registry server* for discovery. To use service discovery, a make sure you start the `bin/rpyc_registry.py`. This server listens on a broadcast UDP socket, and will answer to queries about which services are running where.

Once a registry server is running somewhere "broadcastable" on your network, and the servers are configured to auto-register with it (the default), clients can discover services *automagically*. To start a server:

```
>>> mysvc = rpyc.OneShotServer(service=MyService, port=18861, auto_register=True)
>>> mysvc.start()
```

To find servers running a given service name:

```
>>> rpyc.list_services()
>>> rpyc.discover("MY")
(('192.168.1.101', 18861),)
```

And if you don't care to which you server you connect, you use connect_by_service:

```
>>> c2 = rpyc.connect_by_service("MY")
>>> c2.root.get_answer()
42
```

Decoupled Services

So far we've discussed only about the service that the **server** exposes, but what about the client? Does the client expose a service too? After all, RPyC is a symmetric protocol – there's no difference between the client and the server. Well, as you might have guessed, the answer is yes: both client and server expose services. However, the services exposed by the two parties need not be the same – they are **decoupled**.

By default, clients (using one of the connect() functions to connect to a server) expose the VoidService. As the name suggests, this service exposes no functionality to the other party, meaning the server can't make requests to the client (except for explicitly passed capabilities, like function callbacks). You can set the service exposed by the client by passing the service = parameter to one of the *connect()* functions.

The fact that the services on both ends of the connection are decoupled, does not mean they can be arbitrary. For instance, "service A" might expect to be connected to "service B" – and runtime errors (mostly AttributeError) will ensue if this not the case. Many times the services on both ends can be different, but do keep it in mind that if you need interaction between the parties, both services must be "compatible".

Note: Classic mode: when using any of the *connect()* functions, the client-side service is set to SlaveService as well (being identical to the server).

Continue to *Part 4: Callbacks and Symmetry*...

4.3.4 Part 4: Callbacks and Symmetry

Before we dive into asynchronous invocation, we have to cover once last topic: callbacks. Passing a "callback function" means treating functions (or any callable objects in our case) as first-class objects, i.e., like any other value in the language. In C and C++ this is done with function pointers, but in python, there's no special machinery for it. Surely you've seen callbacks before:

```
>>> def f(x):
...     return x**2
...
>>> map(f, range(10))   # f is passed as an argument to map
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Since in python functions (as well as any other value) are objects, and since RPyC is symmetrical, local functions can be passed as arguments to remote objects, and vice versa. Here's an example

```
>>> import rpyc
>>> c = rpyc.classic.connect("localhost")
>>> rlist = c.modules.builtins.list((0,1,2,3,4,5,6,7,8,9))  # this is a remote list
>>> rlist
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
>>> def f(x):
...     return x**3
...
>>> list(c.modules.builtins.map(f, rlist))  # calling the remote map with the local␣
↪function f as an argument
[0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
>>>

# and to better understand the previous example
>>> def g(x):
...     print("hi, this is g, executing locally", x)
...     return x**3
...
>>> list(c.modules.builtins.map(g, rlist))
hi, this is g, executing locally 0
hi, this is g, executing locally 1
hi, this is g, executing locally 2
hi, this is g, executing locally 3
hi, this is g, executing locally 4
hi, this is g, executing locally 5
hi, this is g, executing locally 6
hi, this is g, executing locally 7
hi, this is g, executing locally 8
hi, this is g, executing locally 9
[0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
>>>
```

To explain what the symmetry of RPyC means, consider the following diagram:

As you can see, while the client is waiting for the result (a synchronous request), it will serve all incoming requests, meaning the server can invoke the callback it had received on the client. In other words, the symmetry of RPyC means that both the client and the server are ultimately "servers", and the "role" is more semantic than programmatic.

Continue to *Part 5: Asynchronous Operation and Events...*

4.3.5 Part 5: Asynchronous Operation and Events

Asynchronism

The last part of the tutorial deals with a more "advanced" issue of RPC programming, *asynchronous operation*, which is a key feature of RPyC. The code you have seen so far was *synchronous* – which is similar to the code you often write: when you invoke a function, you block until the result arrives. Asynchronous invocation allows you to start the request and continue rather than waiting. Instead of getting the result of the call, you get an object known as an `AsyncResult` (also known as a "future" or "promise") that will **eventually** hold the result.

Note that there is no guarantee on execution order for async requests!

To turn the invocation of a remote function (or any callable object) asynchronous, all you have to do is wrap it with
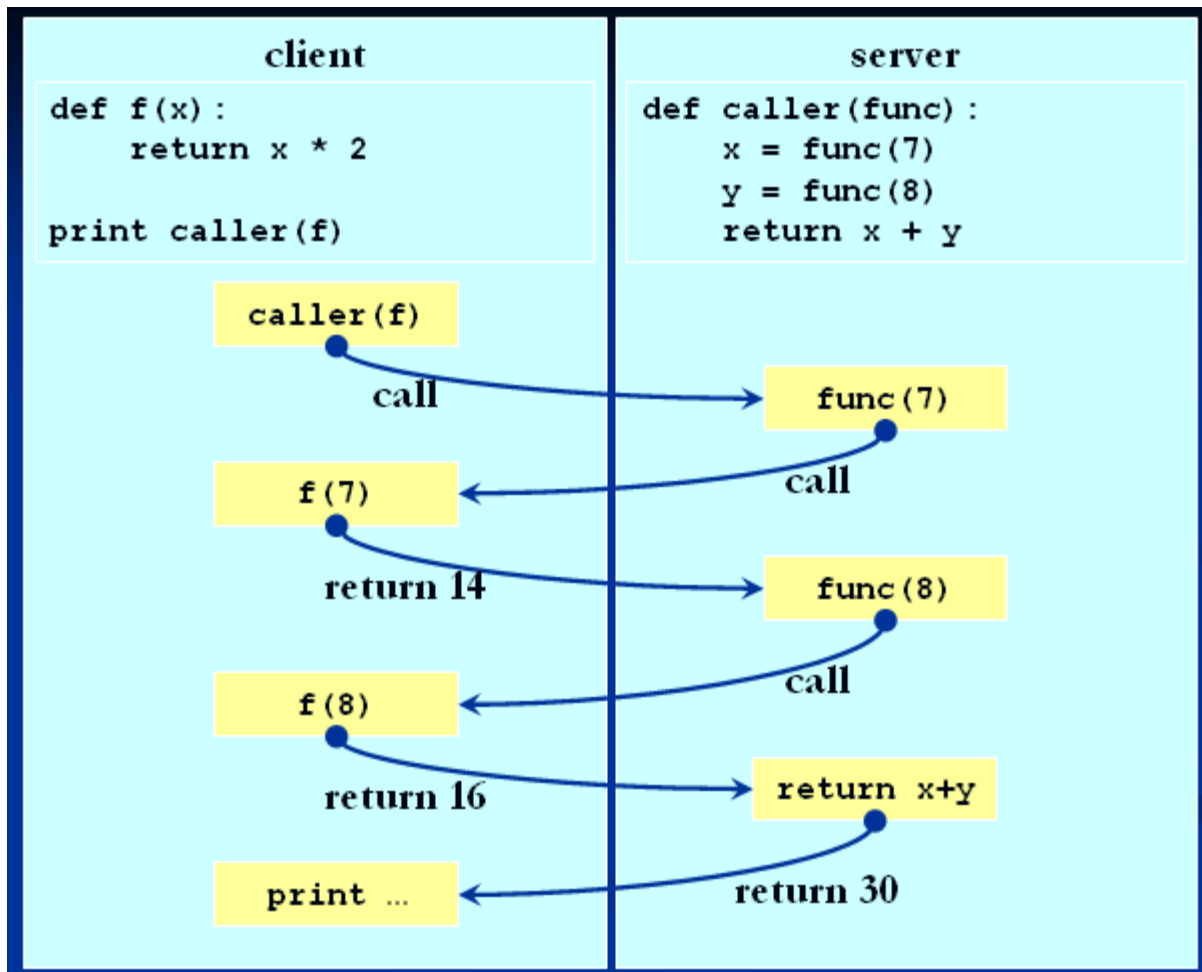
*async_*, which creates a wrapper function that will return an `AsyncResult` instead of blocking. `AsyncResult` objects have several properties and methods that

- `ready` - indicates whether or not the result arrived

- `error` - indicates whether the result is a value or an exception

- `expired` - indicates whether the AsyncResult object is expired (its time-to-wait has elapsed before the result has arrived). Unless set by `set_expiry`, the object will never expire

- `value` - the value contained in the AsyncResult. If the value has not yet arrived, accessing this property will block. If the result is an exception, accessing this property will raise it. If the object has expired, then an exception is raised. Otherwise, the value is returned

- `wait()` - wait for the result to arrive, or until the object is expired

- `add_callback(func)` - adds a callback to be invoked when the value arrives

- `set_expiry(seconds)` - sets the expiry time of the AsyncResult. By default, no expiry time is set

This may sound a bit complicated, so let us have a look at some real-life code, to convince you it is not that scary:

```
>>> import rpyc
>>> c=rpyc.classic.connect("localhost")
>>> c.modules.time.sleep
<built-in function sleep>
>>> c.modules.time.sleep(2) # i block for two seconds, until the call returns

 # wrap the remote function with async_(), which turns the invocation asynchronous
>>> asleep = rpyc.async_(c.modules.time.sleep)
>>> asleep
async_(<built-in function sleep>)

# invoking async functions yields an AsyncResult rather than a value
>>> res = asleep(15)
>>> res
<AsyncResult object (pending) at 0x0842c6bc>
>>> res.ready
False
>>> res.ready
False

# ... after 15 seconds...
>>> res.ready
True
>>> print(res.value)
None
>>> res
<AsyncResult object (ready) at 0x0842c6bc>
```

And here's a more interesting snippet:

```
>>> aint = rpyc.async_(c.builtins.int)  # async wrapper for the remote type int

# a valid call
>>> x = aint("8")
>>> x
<AsyncResult object (pending) at 0x0844992c>
```

```
>>> x.ready
True
>>> x.error
False
>>> x.value
8

# and now with an exception
>>> x = aint("this is not a valid number")
>>> x
<AsyncResult object (pending) at 0x0847cb0c>
>>> x.ready
True
>>> x.error
True
>>> x.value
Traceback (most recent call last):
...
  File "/opt/rpyc/rpyc/core/async_.py", line 102, in value
    raise self._obj
ValueError: invalid literal for int() with base 10: 'this is not a valid number'

========= Remote Traceback (1) =========
Traceback (most recent call last):
  File "/opt/rpyc/rpyc/core/protocol.py", line 324, in _dispatch_request
    res = self._HANDLERS[handler](self, *args)
...
ValueError: invalid literal for int() with base 10: 'this is not a valid number'
>>>
```

Events

Combining `async_` and callbacks yields a rather interesting result: *async callbacks*, also known as **events**. Generally speaking, events are sent by an "event producer" to notify an "event consumer" of relevant changes, and this flow is normally one-way (from producer to consumer). In other words, in RPC terms, events can be implemented as async callbacks, where the return value is ignored. To better illustrate the situation, consider the following `FileMonitor` example – it monitors a file (using `os.stat()`) for changes, and notifies the client when a change occurs (with the old and new `stat` results).

```
import rpyc
import os
import time
from threading import Thread


class FileMonitorService(rpyc.Service):
    class exposed_FileMonitor(object):   # exposing names is not limited to methods :)
        def __init__(self, filename, callback, interval = 1):
            self.filename = filename
            self.interval = interval
            self.last_stat = None
            self.callback = rpyc.async_(callback)   # create an async callback
```

```python
            self.active = True
            self.thread = Thread(target = self.work)
            self.thread.start()
        def exposed_stop(self):    # this method has to be exposed too
            self.active = False
            self.thread.join()
        def work(self):
            while self.active:
                stat = os.stat(self.filename)
                if self.last_stat is not None and self.last_stat != stat:
                    self.callback(self.last_stat, stat)    # notify the client of the
→change
                self.last_stat = stat
                time.sleep(self.interval)


if __name__ == "__main__":
    from rpyc.utils.server import ThreadedServer
    ThreadedServer(FileMonitorService, port = 18871).start()
```

And here's a live demonstration of events:

```python
>>> import rpyc
>>> f = open("/tmp/floop.bloop", "wb", buffering=0)
>>> conn = rpyc.connect("localhost", 18871)
>>> bgsrv = rpyc.BgServingThread(conn)  # creates a bg thread to process incoming events
>>>
>>> def on_file_changed(oldstat, newstat):
...     print("\nfile changed")
...     print(f"    old stat: {oldstat}")
...     print(f"    new stat: {newstat}")
...
>>> mon = conn.root.FileMonitor("/tmp/floop.bloop", on_file_changed)  # create a filemon


# wait a little for the filemon to have a look at the original file

>>> f.write(b"oloop")  # change the file size and wait for filemon to notice the change
file changed
    old stat: (33188, 1564681L, 2051L, 1, 1011, 1011, 0L, 1225204483, 1225204483,
→1225204483)
    new stat: (33188, 1564681L, 2051L, 1, 1011, 1011, 6L, 1225204483, 1225204556,
→1225204556)

>>> f.close()
>>> mon.stop()
>>> bgsrv.stop()
>>> conn.close()
```

Note that in this demo I used *BgServingThread*, which basically starts a background thread to serve all incoming requests, while the main thread is free to do as it wills. You don't have to open a second thread for that, if your application has a reactor (like gtk's gobject.io_add_watch): simply register the connection with the reactor for read, invoking conn.serve. If you don't have a reactor and don't wish to open threads, you should be aware that these notifications will not be processed until you make some interaction with the connection (which pulls all incoming requests). Here's an example of that:

```
>>> conn = rpyc.connect("localhost", 18871)
>>> mon = conn.root.FileMonitor("/tmp/floop.bloop", on_file_changed)
>>> f.write(b"zloop")  # change the file size

# Notice that nothing is printed. To print the file change messages,
# the RPyC connection must serve requests from filemon that contain stat data.
# Dispatching a request would implicitly make the connection serve existing requests.
# Executing conn.poll_all() would explicitly serve all requests, without an extra␣
→dispatch.
>>> conn.poll_all()
file changed
    old stat: (33188, 1564681L, 2051L, 1, 1011, 1011, 0L, 1225205197, 1225205197,␣
→1225205197)
    new stat: (33188, 1564681L, 2051L, 1, 1011, 1011, 6L, 1225205197, 1225205218,␣
→1225205218)
>>>
```

4.4 Documentation

4.4.1 Introduction

About RPyC

RPyC was inspired by the work of **Eyal Lotem** on pyinvoke, which pioneered in the field of "dynamic RPC" (where there's no predefined contract between the two sides). The two projects, however, are completely unrelated in any other way. RPyC is developed and maintained by Tomer Filiba (tomerfiliba@gmail.com).

Note: Please do not send questions directly to my email – use our the github issues instead

---

Contributors

Contributors for newer versions are visible from the git commit history.

v3.2.3

- Guy Rozendorn - backported lots of fixes from 3.3 branch
- Alon Horev - UNIX domain socket patch

v3.2.2

- Rotem Yaari - Add logging of exceptions to the protocol layer, investigate `EINTR` issue
- Anselm Kruis - Make RPyC more introspection-friendly
- Rüdiger Kessel - SSH on windows patch

v3.2.1

- Robert Hayward - adding missing import
- pyscripter - investigating python 3 incompatibilities
- xanep - handling `__cmp__` correctly

v3.2.0

- Alex - IPv6 support
- Sponce - added the `ThreadPoolServer`, several fixes to weak-references and `AsyncResult`
- Sagiv Malihi - Bug fix in classic server
- Miguel Alarcos - issue #8
- Pola Abram - Discovered several races when server threads trerminate
- Chris - Several bug fixes (#46, #49, #50)

v3.1.0
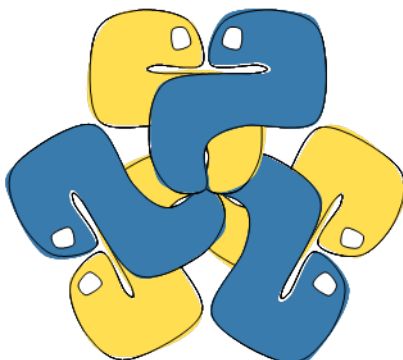
- Alex - better conventions, Jython support
- Fruch - testing, benchmarking
- Eyecue - porting to python3
- Jerome Delattre - IronPython support
- Akruis - bug fixes

v3.0.0-v3.0.7

- Noam Rapahel - provided the original Twisted-integration with RPyC.
- Gil Fidel - provided the original NamedPipeStream on Windows.
- Eyal Lotem - Consulting and spiritual support :)
- Serg Dobryak - backporting to python 2.3
- Jamie Kirkpatrick - patches for the registry server and client

Logo

The logo is derived from the Python logo, with explicit permission. I created it using *Power Point* (sorry, I'm no graphic designer :), and all the files are made available here:

-

-

-

- Also in `the original` Power Point master.

Theory of Operation

This is a short outline of the "Theory of Operation" of RPyC. It will introduce the main concepts and terminology that's required in order to understand the library's internals.

Theory

The most fundamental concept of computer programming, which almost all operating systems share, is the process. A process is a unit of code and data, contained within an address space – a region of (virtual) memory, owned solely by that process. This ensures that all processes are isolated from one another, so that they could run on the same hardware without interfering to each other. While this isolation is essential to operating systems and the programming model we normally use, it also has many downsides (most of which are out of the scope of this document). Most importantly, from RPyC's perspective, processes impose artificial boundaries between programs which forces programs to resort to monolithic structuring.

Several mechanism exist to overcome these boundaries, most notably remote procedure calls. Largely speaking, RPCs enable one process to execute code ("call procedures") that reside outside of its address space (in another process) and be aware of their results. Many such RPC frameworks exist, which all share some basic traits: they provide a way to describe what functions are exposed, define a serialization format, transport abstraction, and a client-side library/code-generator that allows clients utilize these remote functions.

RPyC is *yet another RPC*. However, unlike most RPCs, RPyC is **transparent**. This may sound like a rather weird virtue at first – but this is the key to RPyC's power: you can "plug" RPyC into existing code at (virtually) no cost. No need to write complicated definition files, configure name servers, set up transport (HTTP) servers, or even use special invocation syntax – RPyC fits the python programming model like a glove. For instance, a function that works on a local file object will work seamlessly on a remote file object – it's duck-typing to the extreme.

An interesting consequence of being transparent is **symmetry** – there's no longer a strict notion of what's a *server* as opposed to what's a *client* – both the parties may serve requests and dispatch replies; the server is simply the party that accepts incoming connections – but other than that, servers and clients are identical. Being symmetrical opens the doors to lots of previously unheard-of features, like callback functions.

The result of these two properties is that local and remote objects are "equal in front of the code": your program shouldn't even be aware of the "proximity" of object it is dealing with. In other words, two processes connected by RPyC can be thought of as a **single process**. I like to say that RPyC *unifies the address space* of both parties, although physically, this address space may be split between several computers.

Note: The notion of address-space unification is mostly true for "classic RPyC"; with new-style RPyC, where services dominate, the analogy is of "unifying selected parts of the address space".

In many situations, RPyC is employed in a master-slave relation, where the "client" takes full control over the "server". This mainly allows the client to access remote resources and perform operations on behalf of the server. However, RPyC can also be used as the basis for clustering and distributed computing: an array of RPyC servers on multiple machines can form a "huge computer" in terms of computation power.

Note: This would require some sort of framework to distribute workload and guarantee task completion. RPyC itself is just the mechanism.

Implementation

Boxing

A major concept in the implementation of RPyC is *boxing*, which is a form of *serialization* (encoding) that transfers objects between the two ends of the connection. Boxing relies on two methods of serialization:

- By Value - simple, immutable python objects (like strings, integers, tuples, etc.) are passed **by value**, meaning the value itself is passed to the other side. Since their value cannot change, there is no restriction on duplicating them on both sides.

- By Reference - all other objects are passed **by reference**, meaning a "reference" to the object is passed to the other side. This allows changes applied on the referencing (proxy) object to be reflected on the actual object. Passing objects by reference also allows passing of "location-aware" objects, like files or other operating system resources.

On the other side of the connection, the process of *unboxing* takes place: by-value data is converted ("deserialized") to local objects, while by-reference data is converted to *object proxies*.

Object Proxying

Object proxying is a technique of referencing a remote object transparently: since the remote object cannot be transferred by-value, a reference to it is passed. This reference is then wrapped by a special object, called a *proxy* that "looks and behaves" just like the actual object (the *target*). Any operation performed on the proxy is delivered transparently to the target, so that code need not be aware of whether the object is local or not.

Note: RPyC uses the term `netref` (network reference) for a proxy object

---

Most of the operations performed on object proxies are *synchronous*, meaning the party that issued the operation on the proxy waits for the operation to complete. However, sometimes you want *asynchronous* mode of operation, especially when invoking remote functions which might take a while to return their value. In this mode, you issue the operation and you will later be notified of its completion, without having to block until it arrives. RPyC supports both methods: proxy operations, are synchronous by default, but invocation of remote functions can be made asynchronous by wrapping the proxy with an asynchronous wrapper.

Services

In older versions of RPyC, up to version 2.60 (now referred to as *classic RPyC*), both parties had to "fully trust" each other and be "fully cooperative" – there was no way to limit the power of one party over the other. Either party could perform arbitrary operations on the other, and there was no way to restrict it.

RPyC 3.0 introduced the concept of *services*. RPyC itself is only a "sophisticated transport layer" – it is a mechanism, it does not set policies. RPyC allows each end of the connection to expose a (potentially different) *service* that is responsible for the "policy", i.e., the set of supported operations. For instance, *classic RPyC* is implemented by the `SlaveService`, which grants arbitrary access to all objects. Users of the library may define their own services, to meet their requirements.

How To's

This page contains a collection of useful concepts and examples for developing with RPyC

Redirecting Standard Input/Output

You can "rewire" `stdin`, `stdout` and `stderr` between RPyC hosts. For example, if you want to "forward" the `stdout` of a remote process to your local `tty`, you can use the following receipt:

```
>>> import rpyc
>>> c = rpyc.classic.connect("localhost")
>>> c.execute("print('hi there')")   # this will print on the host
>>> import sys
>>> c.modules.sys.stdout = sys.stdout
>>> c.execute("print('hi here')")   # now this will be redirected here
hi here
```

Also note that if you are using classic mode RPyC, you can use the context manager `rpyc.classic.redirected_stdio`:

```
>>> c.execute("print('hi there')")                    # printed on the server
>>>
>>> with rpyc.classic.redirected_stdio(c):
...     c.execute("print('hi here')")                 # printed on the client
...
hi here
>>> c.execute("print('hi there again')")              # printed on the server
>>>
```
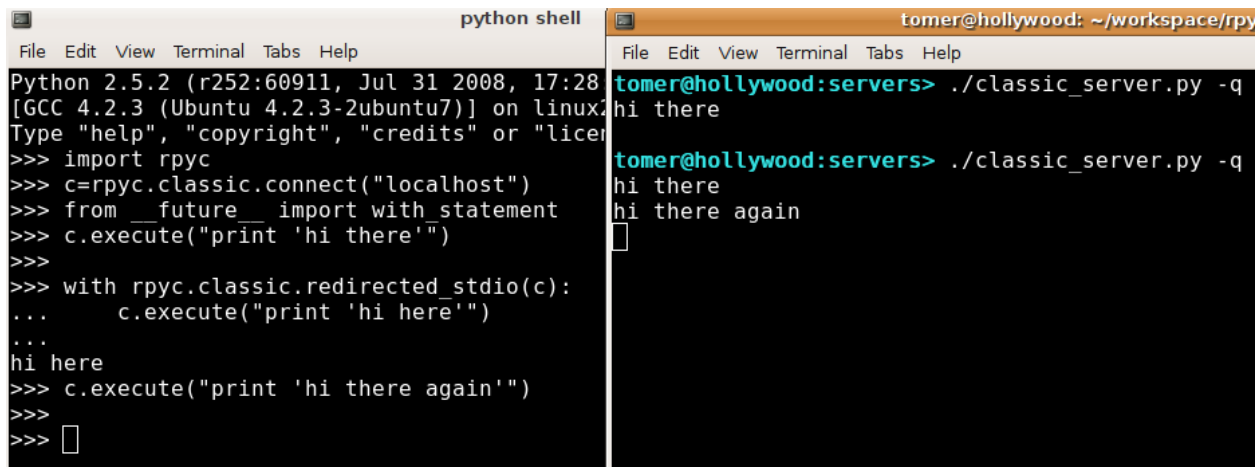
Fig. 1: A screenshot of an RPyC client redirecting standard output from the server to its own console.

Debugging

If you are using the classic mode, you will be glad to know that you can debug remote exceptions with pdb:

```
>>> c = rpyc.classic.connect("localhost")
>>> c.modules["xml.dom.minidom"].parseString("<<invalid xml>/>")
======= Remote traceback =======
Traceback (most recent call last):
...
  File "/usr/lib/python2.5/xml/dom/minidom.py", line 1925, in parseString
    return expatbuilder.parseString(string)
  File "/usr/lib/python2.5/xml/dom/expatbuilder.py", line 940, in parseString
    return builder.parseString(string)
  File "/usr/lib/python2.5/xml/dom/expatbuilder.py", line 223, in parseString
    parser.Parse(string, True)
ExpatError: not well-formed (invalid token): line 1, column 1
...
  File "/home/tomer/workspace/rpyc/core/protocol.py", line 298, in sync_request
    raise obj
xml.parsers.expat.ExpatError: not well-formed (invalid token): line 1, column 1
>>>
>>> rpyc.classic.pm(c) # start post-portem pdb
> /usr/lib/python2.5/xml/dom/expatbuilder.py(226)parseString()
-> pass
(Pdb) l
221             parser = self.getParser()
222             try:
223                 parser.Parse(string, True)
224                 self._setup_subset(string)
225             except ParseEscape:
226  ->             pass
227             doc = self.document
228             self.reset()
229             self._parser = None
230             return doc
```

(continues on next page)

```
231
(Pdb) w
...
  /home/tomer/workspace/rpyc/core/protocol.py(381)_handle_call()
-> return self._local_objects[oid](*args, **dict(kwargs))
  /usr/lib/python2.5/xml/dom/minidom.py(1925)parseString()
-> return expatbuilder.parseString(string)
  /usr/lib/python2.5/xml/dom/expatbuilder.py(940)parseString()
-> return builder.parseString(string)
> /usr/lib/python2.5/xml/dom/expatbuilder.py(226)parseString()
-> pass
(Pdb)
```

Tunneling

Many times, especially in testing environments, you have subnets, VLANs, VPNs, firewalls etc., which may prevent you from establishing a direct TCP connection between two machines, crossing network in two different networks. This may be done for security reasons or to simulate the environment where your product will be running, but it also hinders your ability to conduct tests. However, with RPyC you can overcome this limitation very easily: simply use the remote machine's `socket` module!

Consider the following diagram:
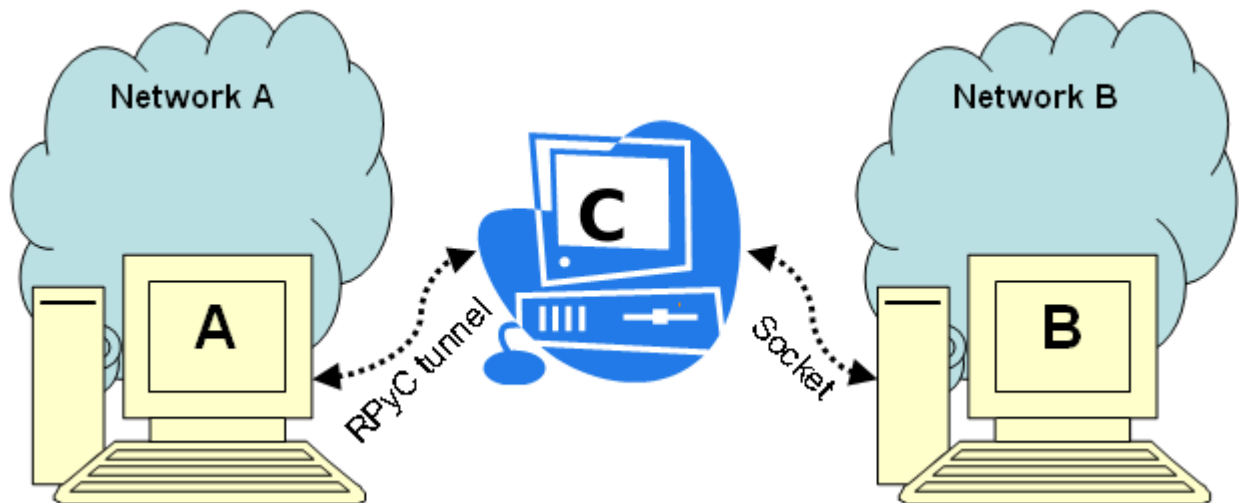
Machine `A` belongs to network `A`, and it wants to connect to machine B, which belongs to network B. Assuming there's a third machine, `C` that has access to both networks (for instance, it has multiple network cards or it belongs to multiple VLANs), you can use it as a transparent bridge between machines `A` and B very easily: simply run an RPyC server on machine C, to which machine `A` would connect, and use its `socket` module to connect to machine B. It's really simple:

```python
# this runs on machine `A`
import rpyc

machine_c = rpyc.classic.connect("machine-c")
sock = machine_c.modules.socket.socket()
sock.connect(("machine-b", 12345))
```

```
sock.send(...)
sock.recv(...)
```

Monkey-Patching

If you have python modules that make use of the `socket` module (say, `telnetlib` or `asyncore`), and you want them to be able to cross networks over such a bridge, you can use the recipe above to "inject" C's socket module into your third-party module, like so:

```python
import rpyc
import telnetlib

machine_c = rpyc.classic.connect("machine-c")
telnetlib.socket = rpyc.modules.socket
```

This is called monkey-patching, it's a very handy technique which you can use in other places as well, to override functions, classes and entire modules. For instance

```python
import mymodule
import rpyc
# ...
mymodule.os = conn.modules.os
mymodule.open = conn.builtins.open
mymodule.Telnet = conn.modules.telnetlib.Telnet
```

That way, when `mymodule` makes use of supposedly local modules, these modules actually perform operations on the remote machine, transparently.

Use Cases

This page lists some examples for tasks that RPyC excels in solving.

Remote ("Web") Services

Starting with RPyC 3.00, the library is *service-oriented*. This makes implementing **secure** remote services trivial: a service is basically a class that exposes a well-defined set of remote functions and objects. These exposed functions can be invoked by the clients of the service to obtain results. For example, a UPS-like company may expose a `TrackYourPackage` service with

```
get_current_location(pkgid)
get_location_history(pkgid)
get_delivery_status(pkgid)
report_package_as_lost(pkgid, info)
```

RPyC is configured (by default) to prevent the use of `getattr` on remote objects to all but "allowed attributes", and the rest of the security model is based on passing *capabilities*. Passing capabilities is explicit and fine grained – for instance, instead of allowing the other party call `open()` and attempting to block disallowed calls at the file-name level (which is weak), you can pass an open file object to the other party. The other party could manipulate the file (calling read/write/seek on it), but it would have no access to the rest of the file system.

Administration and Central Control

Efficient system administration is quite difficult: you have a variety of platforms that you need to control, of different endianities (big/little) or bit-widths (32/64), different administration tools, and different shell languages (`sh`, `tcsh`, batch files, WMI, etc.). Moreover, you have to work across numerous transport protocols (`telnet`, `ftp`, `ssh`, etc.), and most system tools are domain-specific (`awk`, `grep`) and quite limited (operating on lines of text), and are difficult to extend or compose together. System administration today is a mishmash of technologies.

Why not use python for that? It's a cross-platform, powerful and succinct programming language with loads of libraries and great support. All you have to do is `pip install rpyc` on all of your machines, set them up to start an RPyC server on boot (over SSH or SSL), and there you go! You can control every machine from a single place, using a unified set of tools and libraries.

Hardware Resources

Many times you find yourself in need of utilizing hardware ("physical") resources of one machine from another. For instance, some testgear or device can only connect to Solaris SPARC machines, but you're comfortable with developing on your Windows workstation. Assuming your device comes with C bindings, some command-line tool, or accepts commands via `ioctl` to some device node – you can just run an RPyC server on that machine, connect to it from your workstation, and access the device programmatically with ease (using `ctypes` or `popen` remotely).

Parallel Execution

In CPython, the GIL prevents multiple threads from executing python bytecode at once. This simplifies the design of the python interpreter, but the consequence of which is that CPython cannot utilize multiple/multicore CPUs. The only way to achieve scalable, CPU-bound python programs is to use multiple processes, instead of threads. The bright side of using processes over threads is reducing synchronization problems that are inherent to multithreading – but without a easy way to communicate between your processes, threads are more appealing.

Using RPyC, multiprocessing becomes very easy, since we can think of RPyC-connected processes as "one big process". Another modus operandi is having the "master" process spawn multiple worker processes and distribute workload between them.

Distributed Computation Platform

RPyC forms a powerful foundation for distributed computations and clustering: it is architecture and platform agnostic, supports synchronous and asynchronous invocation, and clients and servers are symmetric. On top of these features, it is easy to develop distributed-computing frameworks; for instance, such a framework will need to:

- Take care of nodes joining or leaving the cluster
- Handle workload balancing and node failures
- Collect results from workers
- Migrate objects and code based on runtime profiling

Note: RPyC itself is only a mechanism for distributed computing; it is not a distributed computing framework

---

Distributed algorithms could then be built on top of this framework to make computations faster.

Testing

The first and foremost use case of RPyC is in **testing environments**, where the concept of the library was conceived (initially as *pyinvoke*).

Classic-mode RPyC is the ideal tool for centralized testing across multiple machines and platforms: control your heterogeneous testing environment (simulators, devices and other test equipment) and test procedure from the comfort of your workstation. Since RPyC integrates so well with python, it is very easy to have your test logic run on machine A, while the side-effects happen on machine B.

There is no need to copy and keep your files synchronized across several machines, or work on remote file systems mounts. Also, since RPyC requires a lot of network "ping-pongs", and because of the inherent *security risks* of the *classic mode*, this mode works best on secure, fast local networks (which is usually the case in testing environments).

- *A little about RPyC* - related projects, contributors, and logo issues
- *Theory of Operation* - background on the inner workings of RPyC and the terminology
- *Use cases* - some common use-cases, demonstrating the power and ease of RPyC
- *How to's* - solutions to specific problems

4.4.2 Reference

RPyC Servers

Since RPyC is a symmetric protocol (where both client and server can process requests), an *RPyC server* is a largely just a main-loop that accepts incoming connections and calls `serve_all()`. RPyC comes with three built-in servers:

- Forking - forks a child-process to handle each incoming connection (POSIX only)
- Threaded - spawns a thread to handle each incoming connection (POSIX and Windows)
- Thread Pool - assigns a worker-thread for each incoming connection from the thread pool; if the thread pool is exhausted, the connection is dropped.

If you wish to implement new servers (say, reactor-based, etc.), you can derive from `rpyc.utils.server.Server` and implement `_accept_method()` to your own liking.

Note: RPyC uses the notion of *authenticators* to authenticate incoming connections. An authenticator object can be passed to the server instance upon construction, and it is used to validate incoming connections. See *Authenticators* for more info.

Classic Server

RPyC comes "bundled" with a *Classic*-mode server – `rpyc_classic.py`. This executable script takes several command-line switches and starts an RPyC server exposing the `ClassicService`. It is installed to your python's `scripts/` directory, and should be executable from the command line. Example usage:

```
$ ./rpyc_classic.py -m threaded -p 12333
INFO:SLAVE/12333:server started on [0.0.0.0]:12333
INFO:SLAVE/12333:accepted 127.0.0.1:34044
INFO:SLAVE/12333:welcome [127.0.0.1]:34044
INFO:SLAVE/12333:goodbye [127.0.0.1]:34044
```

(continues on next page)

```
^C
WARNING:SLAVE/12333:keyboard interrupt!
INFO:SLAVE/12333:server has terminated
INFO:SLAVE/12333:listener closed
```

The classic server takes the following command-line switches (try running it with `-h` for more info):

General switches

- `-m`, `--mode=MODE` - the serving mode (`threaded`, `forking`, or `stdio`). The default is `threaded`; `stdio` is useful for integration with inetd.

- `-p`, `--port=PORT` - the TCP port (only useful for `threaded` or `forking` modes). The default is `18812`; for SSL the default is `18821`.

- `--host=HOSTNAME` - the host to bind to. The default is `0.0.0.0`.

- `--ipv6` - if given, binds an IPv6 socket. Otherwise, binds an IPv4 socket (the default).

- `--logfile=FILENAME` - the log file to use. The default is `stderr`

- `-q`, `--quiet` - if given, sets quiet mode (no logging).

Registry switches

- `--register` - if given, the server will attempt to register with a registry server. By default, the server will **not** attempt to register.

The following switches are only relevant in conjunction with `--register`:

- `--registry-type=REGTYPE` - The registry type (UDP or TCP). The default is UDP, where the server sends timely UDP broadcasts, aimed at the registry server.

- `--registry-port=REGPORT` - The TCP/UDP port of the registry server. The default is `18811`.

- `--registry-host=REGHOST` - The host running the registry server. For UDP the default is broadcast (`255.255.255.255`); for TCP, this parameter is **required**.

SSL switches

If any of the following switches is given, the server uses the SSL authenticator. These cannot be used with conjunction with `--vdb`.

- `--ssl-keyfile=FILENAME` - the server's SSL key-file. Required for SSL

- `--ssl-certfile=FILENAME` - the server's SSL certificate file. Required for SSL

- `--ssl-cafile=FILENAME` - the certificate authority chain file. This switch is optional; if it's given, it enables client-side authentication.

Custom RPyC Servers

Starting an RPyC server that exposes your service is quite easy – when you construct the `rpyc.utils.server.Server` instance, pass it your `rpyc.core.service.Service` factory. You can use the following snippet:

```python
import rpyc
from rpyc.utils.server import ThreadedServer # or ForkingServer

class MyService(rpyc.Service):
    #
    # ... you service's implementation
    #
    pass

if __name__ == "__main__":
    server = ThreadedServer(MyService, port = 12345)
    server.start()
```

Refer to `rpyc.utils.server.Server` for the list all possible arguments.

Registry Server

RPyC comes with a simple command-line registry server, which can be configured quite extensively by command-line switches. The registry server is a bonjour-like agent, with which services may register and clients may perform queries. For instance, if you start an RPyC server that provides service `Foo` on `myhost:17777`, you can register that server with the registry server, which would allow clients to later query for the servers that expose that service (and get back a list of TCP endpoints). Example usage:

```
   $ ./bin/rpyc_registry.py --listing
   DEBUG:REGSRV/UDP/18811:registering 172.18.0.6:18861 as MY

For more info, see :ref:`api-registry`.
```

Switches

- `-m`, `--mode=MODE` - The registry mode; either UDP or TCP. The default is UDP.

- `-p`, `--port=PORT` - The UDP/TCP port to bind to. The default is `18811`.

- `-f`, `--file=FILE` - The log file to use. The default is `stderr`.

- `-q`, `--quiet` - If given, sets quiet mode (only errors are logged)

- `-t`, `--timeout=PRUNING_TIMEOUT` - Sets a custom pruning timeout, in seconds. The pruning time is the amount of time the registry server will keep a previously-registered service, when it no longer sends timely keepalives. The default is 4 minutes (240 seconds).

- `-l`, `--listing` - Give a boolean indicating if registry should allow sending the list of its known services. The default is False.

Classic

Prior to version 3, RPyC employed a modus-operandi that's now referred to as "classic mode". In this mode, the server was completely under the control of its client – there was no way to restrict what the client could do, and there was no notion of *services*. A client simply connected to a server and began to manipulate it.

Starting with version 3, RPyC became *service-oriented*, and now servers expose well-defined *services*, which define what a client can access. However, since the classic mode proved very useful and powerful, especially in testing environments, and in order to retain backwards compatibility, the classic mode is still exists in current versions – this time implemented as a `service`.

See also the *API reference*

Usage

RPyC installs `rpyc_classic.py` to your Python scripts directory (e.g., `C:\PythonXX\Scripts`, `/usr/local/bin`, etc.), which is a ready-to-run classic-mode server. It can be configured with *command-line parameters*. Once you have it running, you can connect to it like so

```python
conn = rpyc.classic.connect("hostname")    # use default TCP port (18812)

proc = conn.modules.subprocess.Popen("ls", stdout = -1, stderr = -1)
stdout, stderr = proc.communicate()
print(stdout.split())

remote_list = conn.builtin.range(7)

conn.execute("print('foo')")
```

Services

RPyC is oriented around the notion of *services*. Services are classes that derive from `rpyc.core.service.Service` and define "exposed methods" – normally, methods whose name explicitly begins with `exposed_`. Services also have a name, or a list of aliases. Normally, the name of the service is the name of its class (excluding a possible `Service` suffix), but you can override this behavior by specifying the `ALIASES` attribute in the class.

Let's have a look at a rather basic service – a calculator (see *Custom RPyC Servers* for more info)

```python
import rpyc

class CalculatorService(rpyc.Service):
    def exposed_add(self, a, b):
        return a + b
    def exposed_sub(self, a, b):
        return a - b
    def exposed_mul(self, a, b):
        return a * b
    def exposed_div(self, a, b):
        return a / b
    def foo(self):
        print("foo")
```

When a client connects, it can access any of the exposed members of the service

```
import rpyc

conn = rpyc.connect("hostname", 12345)
x = conn.root.add(4,7)
assert x == 11

try:
    conn.root.div(4,0)
except ZeroDivisionError:
    pass
```

As you can see, the `root` attribute of the connection gives you access to the service that's exposed by the other party. For security concerns, access is only granted to `exposed_` members. For instance, the `foo` method above is inaccessible (attempting to call it will result in an `AttributeError`).

Rather than having each method name start with `exposed_`, you may prefer to use a decorator. Let's revisit the calculator service, but this time we'll use decorators.

```
import rpyc

@rpyc.service
class CalculatorService(rpyc.Service):
    @rpyc.exposed
    def add(self, a, b):
        return a + b
    @rpyc.exposed
    def sub(self, a, b):
        return a - b
    @rpyc.exposed
    def mul(self, a, b):
        return a * b
    @rpyc.exposed
    def div(self, a, b):
        return a / b
    def foo(self):
        print("foo")
```

When implementing services, `@rpyc.service` and `@rpyc.exposed` can replace the `exposed_` naming convention.

Implementing Services

As previously explained, all `exposed_` members of your service class will be available to the other party. This applies to methods, but in fact, it applies to any attribute. For instance, you may expose a class:

```
class MyService(rpyc.Service):
    class exposed_MyClass(object):
        def __init__(self, a, b):
            self.a = a
            self.b = b
        def exposed_foo(self):
            return self.a + self.b
```

If you wish to change the name of your service, or specify a list of aliases, set the `ALIASES` (class-level) attribute to a

---

list of names. For instance:

```
class MyService(rpyc.Service):
    ALIASES = ["foo", "bar", "spam"]
```

The first name in this list is considered the "proper name" of the service, while the rest are considered aliases. This distinction is meaningless to the protocol and the registry server.

Your service class may also define two special methods: on_connect(self, conn) and on_disconnect(self, conn). The on_connect method is invoked when a connection has been established. From the client-side perspective, on_connect is invoked each time a client successfully invokes rpyc.connect or any other function provided by the connection factory module: rpyc.utils.factory. After the connection is dead, on_disconnect is invoked (you will not be able to access remote objects inside of on_disconnect).

Note: Try to avoid overriding the __init__ method of the service. Place all initialization-related code in on_connect.

Built-in Services

RPyC comes bundled with two built-in services:

- *VoidService*, which is an empty "do-nothing" service. It's useful when you want only one side of the connection to provide a service, while the other side a "consumer".

- *SlaveService*, which implements *Classic Mode* RPyC.

Decoupled Services

RPyC is a symmetric protocol, which means both ends of the connection can act as clients or servers – in other words – both ends may expose (possibly different) services. Normally, only the server exposes a service, while the client exposes the VoidService, but this is not constrained in any way. For instance, in the classic mode, both ends expose the SlaveService; this allows each party to execute arbitrary code on its peer. Although it's not the most common use case, two-sides services are quite useful. Consider this client:

```
class ClientService(rpyc.Service):
    def exposed_foo(self):
        return "foo"

conn = rpyc.connect("hostname", 12345, service = ClientService)
```

And this server:

```
class ServerService(rpyc.Service):
    def on_connect(self, conn):
        self._conn = conn

    def exposed_bar(self):
        return self._conn.root.foo() + "bar"
```

The client can invoke conn.root.bar() on the server, which will, in turn, invoke foo back on the client. The final result would be "foobar".

Another approach is to pass **callback functions**. Consider this server:

```
class ServerService(rpyc.Service):
    def exposed_bar(self, func):
        return func() + "bar"
```

And this client:

```
def foofunc():
    return "foo"

conn = rpyc.connect("hostname", 12345)
conn.root.bar(foofunc)
```

See also *Configuration Parameters*

Asynchronous Operation

Many times, especially when working in a client-server model, you may want to perform operations "in the background", i.e., send a batch of work to the server and continue with your local operation. At some later point, you may want to poll for the completion of the work, or perhaps be notified of its completion using a callback function.

RPyC is very well-suited for asynchronous work. In fact, the protocol itself is asynchronous, and synchronicity is layered on top of that – by issuing an asynchronous request and waiting for its completion. However, since the synchronous modus-operandi is the most common one, the library exposes a synchronous interface, and you'll need to explicitly enable asynchronous behavior.

async_()

The wrapper *async_()* takes any *callable netref* and returns an asynchronous-wrapper around that netref. When invoked, this wrapper object dispatches the request and immediately returns an *AsyncResult*, instead of waiting for the response.

Usage

Create an async wrapper around the server's `time.sleep` function

```
async_sleep = rpyc.async_(conn.modules.time.sleep)
```

And invoke it like any other function, but instead of blocking, it will immediately return an `AsyncResult`

```
res = async_sleep(5)
```

Which means your client can continue working normally, while the server performs the request. There are several pitfalls using `async_`, be sure to read the *Notes* section!

You can test for completion using `res.ready`, wait for completion using `res.wait()`, and get the result using `res.value`. You may set a timeout for the result using `res.set_expiry()`, or even register a callback function to be invoked when the result arrives, using `res.add_callback()`.

Notes

The returns async proxies are cached by a weak-reference. Therefore, you must hold a strong reference to the returned proxy. Particularly, this means that instead of doing

```
res = async_(conn.root.myfunc)(1,2,3)
```

Use

```
myfunc_async = async_(conn.root.myfunc)
res = myfunc_async(1,2,3)
```

Furthermore, async requests provide **no guarantee on execution order**. In particular, multiple subsequent async requests may be executed in reverse order.

timed()

timed allows you to set a timeout for a synchronous invocation. When a `timed` function is invoked, you'll synchronously wait for the result, but no longer than the specified timeout. Should the invocation take longer, a `AsyncResultTimeout` will be raised.

Under the hood, `timed` is actually implemented with `async_`: it begins dispatches the operation, sets a timeout on the `AsyncResult`, and waits for the response.

Example

```
# allow this function to take up to 6 seconds
timed_sleep = rpyc.timed(conn.modules.time.sleep, 6)

# wait for 3 seconds -- works
async_res = timed_sleep(3)  # returns immediately
async_res.value             # returns after 3 seconds

# wait for 10 seconds -- fails
async_res = timed_sleep(10) # returns immediately
async_res.value             # raises AsyncResultTimeout
```

Background Serving Thread

BgServingThread is a helper class that simply starts a background thread to serve incoming requests. Using it is quite simple:

```
bgsrv = rpyc.BgServingThread(conn)
# ...
# now you can do blocking stuff, while incoming requests are handled in the background
# ...
bgsrv.stop()
```

Using the `BgServingThread` allows your code (normally the client-side) to perform blocking calls, while still being able to process incoming request (normally from the server). This allows the server to send "events" (i.e., invoke callbacks on the client side) while the client is busy doing other things.

For a detailed example show-casing the `BgServingThread`, see *Events* in the tutorial.

Security

Operating over a network always involve a certain security risk, and requires some awareness. Version 3 of RPyC was a rewrite of the library, specifically targeting security and service-orientation. Unlike version 2.6, RPyC no longer makes use of insecure protocols like `pickle`, supports `security-related configuration parameters`, comes with strict defaults, and encourages the use of a capability-based security model. Even so, it behooves you to take a layered to secure programming and not let RPyC be a single point of failure.

CVE-2019-16328 is the first vulnerability since 2008, which made it possible for a remote attacker to bypass standard protocol security checks and modify the behavior of a service. The latent flaw was committed to master from September 2018 to October 2019 and affected versions *4.1.0* and *4.1.1*. As of version *4.1.2*, the vulnerability has been fixed.

RPyC is intuitive and secure when used properly. However, if not used properly, RPyC is also the perfect back-door... The general recommendation is not to use RPyC openly exposed over the Internet. It's wiser to use it only over secure local networks, where you trust your peers. This does not imply that there's anything wrong with the mechanism–but the implementation details are sometimes too subtle to be sure of. Of course, you can use RPyC over a *secure connection*, to mitigate these risks.

RPyC works by exposing a root object, which in turn may expose other objects (and so on). For instance, if you expose a module or an object that has a reference to the `sys` module, a user may be able to reach it. After reaching `sys`, the user can traverse `sys.modules` and gain access to all of the modules that the server imports. More complex methodologies, similar to those used in CVE-2019-16328, could leverage access to `builtins.str`, `builtins.type`, `builtins.object`, and `builtins.dict` and gain access to `sys` modules. The default configurations for RPyC are intended to mitigate access to dangerous objects. But if you enable `allow_public_attrs`, return uninitialized classes or override `_rpyc_getattr` such things are likely to slip under the radar (it's possible to prevent this – see below).

Wrapping

The recommended way to avoid over-exposing of objects is *wrapping*. For example, if your object has the attributes `foo`, `bar`, and `spam`, and you wish to restrict access to `foo` and `bar` alone – you can do

```python
class MyWrapper(object):
    def __init__(self, obj):
        self.foo = obj.foo
        self.bar = obj.bar
```

Since this is a common idiom, RPyC provides `restricted()`. This function returns a "restricted view" of the given object, limiting access only to the explicitly given set of attributes.

```python
class MyService(rpyc.Service):
    def exposed_open(self, filename):
        f = open(filename, "r")
        return restricted(f, ["read", "close"], [])  # allow access only to 'read' and
'close'
```

Assuming RPyC is configured to allow access only to safe attributes (the default), this would be secure.

When exposing modules, you can use the `__all__` list as your set of accessible attributes – but do keep in mind that this list may be unsafe.

Classic Mode

The classic mode (`SlaveService`) is **intentionally insecure** – in this mode, the server "gives up" on security and exposes everything to the client. This is especially useful for testing environments where you basically want your client to have full control over the server. Only ever use a classic mode server over secure local networks.

Configuration Parameters

By default, RPyC is configured to allow very little attribute access. This is useful when your clients are untrusted, but this may be a little too restrictive. If you get "strange" `AttributeError` exceptions, stating that access to certain attributes is denied – you may need to tweak the configuration parameters. Normally, users tend to enable `allow_public_attrs`, but, as stated above, this may have undesired implications.

Attribute Access

RPyC has a rather elaborate attribute access scheme, which is controlled by configuration parameters. However, in case you need more fine-grained control, or wish to completely override the configuration for some type of objects – you can implement the **RPyC attribute protocol**. This protocol consists of `_rpyc_getattr`, `_rpyc_setattr`, and `_rpyc_delattr`, which are parallel to `__getattr__` / `__setattr__` / `__delattr__`. Their signatures are

```
_rpyc_getattr(self, name)
_rpyc_delattr(self, name)
_rpyc_setattr(self, name, value)
```

Any object that implements this protocol (or part of it) will override the default attribute access policy. For example, if you generally wish to disallow access to protected attributes, but have to expose a certain protected attribute of some object, just define `_rpyc_getattr` for that object which allows it:

```python
class MyObjectThatExposesProtectedAttrs(object):
    def __init__(self):
        self._secret = 18
    def _rpyc_getattr(self, name):
        if name.startswith("__"):
            # disallow special and private attributes
            raise AttributeError("cannot accept private/special names")
        # allow all other attributes
        return getattr(self, name)
```

SSL

Using external tools, you can generate client and server certificates, and a certificate authority. After going through this setup stage, you can easily establish an SSL-enabled connection.

Server side:

```python
from rpyc.utils.authenticators import SSLAuthenticator
from rpyc.utils.server import ThreadedServer

# ...

authenticator = SSLAuthenticator("myserver.key", "myserver.cert")
```

(continues on next page)

```
server = ThreadedServer(SlaveService, port = 12345, authenticator = authenticator)
server.start()
```

Client side:

```
import rpyc

conn = rpyc.ssl_connect("hostname", port = 12345, keyfile="client.key",
                        certfile="client.cert")
```

For more info, see the documentation of ssl module.

Zero-Deploy RPyC

Setting up and managing servers is a headache. You need to start the server process, monitor it throughout its life span, make sure it doesn't hog up memory over time (or restart it if it does), make sure it comes up automatically after reboots, manage user permissions and make sure everything remains secure. Enter zero-deploy.

Zero-deploy RPyC does all of the above, but doesn't stop there: it allows you to dispatch an RPyC server on a machine that doesn't have RPyC installed, and even allows multiple instances of the server (each of a different port), while keeping it all 100% secure. In fact, because of the numerous benefits of zero-deploy, it is now considered the preferred way to deploy RPyC.

How It Works

Zero-deploy only requires that you have Plumbum (1.2 and later) installed on your client machine and that you can connect to the remote machine over SSH. It takes care of the rest:

1. Create a temporary directory on the remote machine

2. Copy the RPyC distribution (from the local machine) to that temp directory

3. Create a server file in the temp directory and run it (over SSH)

4. The server binds to an arbitrary port (call it *port A*) on the `localhost` interfaces of the remote machine, so it will only accept in-bound connections

5. The client machine sets up an SSH tunnel from a local port, *port B*, on the `localhost` to *port A* on the remote machine.

6. The client machine can now establish secure RPyC connections to the deployed server by connecting to `localhost`:*port B* (forwarded by SSH)

7. When the deployment is finalized (or when the SSH connection drops for any reason), the deployed server will remove the temporary directory and shut down, leaving no trace on the remote machine

Usage

There's a lot of detail here, of course, but the good thing is you don't have to bend your head around it – it requires only two lines of code:

```python
from rpyc.utils.zerodeploy import DeployedServer
from plumbum import SshMachine

# create the deployment
mach = SshMachine("somehost", user="someuser", keyfile="/path/to/keyfile")
server = DeployedServer(mach)

# and now you can connect to it the usual way
conn1 = server.classic_connect()
print(conn1.modules.sys.platform)

# you're not limited to a single connection, of course
conn2 = server.classic_connect()
print(conn2.modules.os.getpid())

# when you're done - close the server and everything will disappear
server.close()
```

The `DeployedServer` class can be used as a context-manager, so you can also write:

```python
with DeployedServer(mach) as server:
    conn = server.classic_connect()
    # ...
```

Here's a capture of the interactive prompt:

```python
>>> sys.platform
'win32'
>>>
>>> mach = SshMachine("192.168.1.100")
>>> server = DeployedServer(mach)
>>> conn = server.classic_connect()
>>> conn.modules.sys.platform
'linux2'
>>> conn2 = server.classic_connect()
>>> conn2.modules.os.getpid()
8148
>>> server.close()
>>> conn2.modules.os.getpid()
Traceback (most recent call last):
    ...
EOFError
```

You can deploy multiple instances of the server (each will live in a separate temporary directory), and create multiple RPyC connections to each. They are completely isolated from each other (up to the fact you can use them to run commands like `ps` to learn about their neighbors).

MultiServerDeployment

If you need to deploy on a group of machines a cluster of machines, you can also use `MultiServerDeployment`:

```python
from rpyc.utils.zerodeploy import MultiServerDeployment

m1 = SshMachine("host1")
m2 = SshMachine("host2")
m3 = SshMachine("host3")

dep = MultiServerDeployment([m1, m2, m3])
conn1, conn2, conn3 = dep.classic_connect_all()

# ...

dep.close()
```

On-Demand Servers

Zero-deploy is ideal for use-once, on-demand servers. For instance, suppose you need to connect to one of your machines periodically or only when a certain event takes place. Keeping an RPyC server up and running at all times is a waste of memory and a potential security hole. Using zero-deploy on demand is the best approach for such scenarios.

Security

Zero-deploy relies on SSH for security, in two ways. First, SSH authenticates the user and runs the RPyC server under the user's permissions. You can connect as an unprivileged user to make sure strayed RPyC processes can't `rm -rf /`. Second, it creates an SSH tunnel for the transport, so everything is kept encrypted on the wire. And you get these features for free – just configuring SSH accounts will do.

Timeouts

You can pass a `timeout` argument, in seconds, to the `close()` method. A `TimeoutExpired` is raised if any subprocess communication takes longer than the timeout, after the subprocess has been told to terminate. By default, the timeout is `None` i.e. infinite. A timeout value prevents a `close()` call blocking indefinitely.

Advanced Debugging

A guide to using Wireshark when debugging complex use such as chained-connections or version specific issues. To test more complex issues, we may wish to use *pyenv* or Docker in our development environment.

Testing Supported Python Versions via pyenv

Let's use *pyenv* to install Python versions under active development. Since development versions are pulled from a VCS, we wish to force install to get the latest commit before testing. The dependency *plumbum* needs to be installed as well (add *[dev]* for *plumbum* development dependencies). All together now!

```bash
versions=( 3.7 3.8 3.9 3.10 3.11 3.12)
for ver in ${versions[@]}; do
    pyenv install --force ${ver}
    pyenv global ${ver}
    pyenv exec pip install --upgrade pip setuptools wheel plumbum[dev]
    site="$(pyenv exec python -c 'import site; print(site.getsitepackages()[0])')"
    printf "${PWD}\n" > "${site}/rpyc.pth"
done
```

Each *venv* contains a *.pth* file that appends *rpyc* to *sys.path*. We can run *rpyc_classic.py* using *pyenv* like so.

```bash
PYENV_VERSION=3.10-dev pyenv exec python ./bin/rpyc_classic.py --host 127.0.0.1
PYENV_VERSION=3.9-dev pyenv exec python -c "import rpyc; conn = rpyc.utils.classic.
↪connect('127.0.0.1'); conn.modules.sys.stderr.write('hello world\n')"
```

Unit tests can be ran using your desired Python version as well.

```bash
PYENV_VERSION=3.9-dev pyenv exec python -m unittest discover -v -k test_affinity
PYENV_VERSION=3.8-dev pyenv exec python -m unittest discover
```

Testing Supported Python Versions via Docker

Testing RPyC often requires that you use specific Python versions. Docker will make your life easier when testing RPyC locally, especially when performing packet captures of RPyC communication across Python versions. The current settings will use bind mounts to simplify synchronization of RPyC source code within the containers. So, remember to checkout the commit you desire the containers to use on your host!

If desired, individual containers can be specified started

```bash
docker-compose -f ./docker/docker-compose.yml create
docker-compose -f ./docker/docker-compose.yml start rpyc-python-3.7
docker-compose -f ./docker/docker-compose.yml start rpyc-python-3.10
```

The registry server can be started like so

```bash
docker exec rpyc-3.8 /opt/rpyc/bin/rpyc_registry.py
```

The containers can then be used to test to your hearts desire

```bash
docker exec rpyc-3.7 /opt/rpyc/bin/rpyc_classic.py --host 0.0.0.0 &
docker exec -it rpyc-3.10 python -c "import rpyc;conn = rpyc.utils.classic.connect('rpyc-
↪3.7'); conn.modules.sys.stderr.write('hello world\n')"
```

Tips and Tricks

Display filtering for Wireshark

```
tcp.port == 18878 || tcp.port == 18879
(tcp.port == 18878 || tcp.port == 18879) && tcp.segment_data contains "rpyc.core.service.
→SlaveService"
```

Running the chained-connection unit test

```
python -m unittest discover -s ./tests -k test_get_id_pack.Test_get_id_pack.test_chained_
→connect
```

After stopping Wireshark, export specified packets, and open the PCAP. If not already configured, add a custom display column:

```
Title,        Type,    Fields,      Field Occurrence
Stream Index, Custom,  tcp.stream,  0
```

The stream index column makes it easier to decide which TCP stream to follow. Following a TCP provides a more human readable overview of requests and replies that can be printed as a PDF.
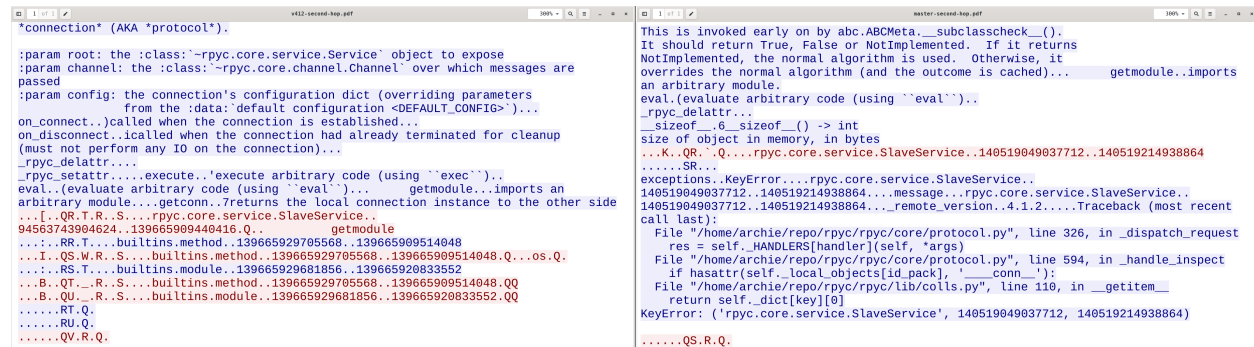
RPyC Release Process

A walkthrough of doing a RPyC Release.

1. Ensure a clean and current build environment (i.e., `git pull; git status`)

2. Describe commit history within `CHANGELOG.rst` (see *Generate Entry*)

3. Update `release_date` in `rpyc/version.py` and bump version (Semantic Versioning and Versioning using Hatch)

4. Verify changes and run `export ver=$(python -c 'import rpyc; print(rpyc.__version__)')`, `git add .`, and `git push`.

5. Create an Annotated tag: `git tag -a ${ver} -m "Updated CHANGELOG.rst and version for release ${ver}"`

6. Publish release tag: `git push origin ${ver}`

7. Install hatch: `pyenv exec pip install hatch`

8. Clean up any old build artifacts: `git clean -Xf -- dist/`

9. Create a wheel package: `pyenv exec hatch -v build`

10. Upload the wheel package: `pyenv exec hatch -v publish --user=__token__ --auth=${pypi_token} ; history -c && history -w`

11. Create new release such that the notes are from *CHANGELOG.rst* entry (%s/`#/#/g and %s/`_//g)

12. Make sure to add the wheel as an attachment to the release and you are done!

Generate CHANGELOG.rst Entry

To create an initial entry draft, run some shell commands.

```
owner="tomerfiliba-org"
repo="rpyc"
#url="https://github.com/${owner}/${repo}"
revisions="$(git rev-list $(hatch version)..HEAD | sed -z 's/\(.*\)\n/\1/;s/\n/|/g')"
numbers=( $(git log $(hatch version)..HEAD --no-merges --oneline | sed -nE 's/^.*#([0-
→9]+).*/\1/p' | sort -nu) )
issue_numbers="$(echo "${numbers[@]}" | sed 's/ /|/g')"
#
api_filter() {
    jq -rc ".[] | select( .${1} | . != null) | select(.${1} | tostring | test(\"${2}\"))
→" "${3}"
}
url="https://api.github.com/repos/${owner}/${repo}"
params="state=closed&accept=application/vnd.github+json"
tmp_issues="/tmp/issues.json"
tmp_pulls="/tmp/pulls.json"
curl "${url}/issues?${params}" > "${tmp_issues}"
curl "${url}/pulls?${params}" > "${tmp_pulls}"
# Pulls
gh_numbers=( )
bullets=( )
url_refs=( )
while IFS= read -r pull; do
    title="$(echo "${pull}" | jq -r .title)"
    number="$(echo "${pull}" | jq -r .number)"
    pull_url="$(echo "${pull}" | jq -r .html_url)"
    # Add GH number
    gh_numbers+=( "${number}" )
    # Add bullet
    bullets+=( "- \`#${number}\`_ ${title}" )
    # Add url ref
    url_ref=".. _#${number}: ${pull_url}"
    url_refs+=( "${url_ref}" )
done <<< "$(api_filter "merge_commit_sha" "${revisions}" "${tmp_pulls}")"
# Issues
while IFS= read -r issue; do
    title="$(echo "${issue}" | jq -r .title)"
    number="$(echo "${issue}" | jq -r .number)"
    issue_url="$(echo "${issue}" | jq -r .html_url)"
    # Add bullet
    bullets+=( "- \`#${number}\`_ ${title}" )
    # Add url ref
```

(continues on next page)

```
    url_ref=".. _#${number}: ${issue_url}"
    url_refs+=( "${url_ref}" )
done <<< "$(api_filter "number" "${issue_numbers}" "${tmp_issues}")"

# Header
printf '5.X.Y\n====\n'
printf 'Date: %s\n\n' "$(date --rfc-3339=date)"
for bullet in "${bullets[@]}"; do
    printf '%s\n' "${bullet}"
done
printf '\n'
for ref in "${url_refs[@]}"; do
    printf '%s\n' "${ref}"
done
```

Once insert this entry at the top of *CHANGELOG.rst*, review what it looks like with *instant-rst*.

```
instantRst -b chromium -p 8612 -f "CHANGELOG.rst"
```

Misc. References

- Wheel file name convention
- *Servers* - using the built-in servers and writing custom ones
- *Classic RPyC* - using RPyC in *slave mode* (AKA *classic mode*), where the client has unrestricted control over the server.
- *RPyC Services* - writing well-defined services which restrict the operations a client (or server) can carry out.
- *Asynchronous Operation* - invoking operations in the background, without having to wait for them to finish.
- *Security Concerns* - keeping security in mind when using RPyC
- *Secure Connections* - create an encrypted and authenticated connection over SSL or SSH
- *Zero-Deploy* - spawn temporary, short-lived RPyC server on remote machine with nothing more than SSH and a Python interpreter
- *Advanced Debugging* - debugging at the packet level

4.5 API Reference

4.5.1 Serialization

Brine

Brine is a simple, fast and secure object serializer for **immutable** objects.

The following types are supported: `int`, `bool`, `str`, `float`, `unicode`, `bytes`, `slice`, `complex`, `tuple` (of simple types), `frozenset` (of simple types) as well as the following singletons: `None`, `NotImplemented`, and `Ellipsis`.

Example::

```
>>> x = ("he", 7, u"llo", 8, (), 900, None, True, Ellipsis, 18.2, 18.2j + 13,
... slice(1,2,3), frozenset([5,6,7]), NotImplemented)
>>> dumpable(x)
True
>>> y = dump(x)
>>> y.encode("hex")

↪'140e0b686557080c6c6c6f5802160339303030003061840323333333333331b402a000000000000040323333333333319
↪'
>>> z = load(y)
>>> x == z
True
```

rpyc.core.brine.**dump**(*obj*)

> Converts (dumps) the given object to a byte-string representation
>
> > **Parameters**
> >
> > > **obj** – any *dumpable()* object
> >
> > **Returns**
> >
> > > a byte-string representation of the object

rpyc.core.brine.**load**(*data*)

> Recreates (loads) an object from its byte-string representation
>
> > **Parameters**
> >
> > > **data** – the byte-string representation of an object
> >
> > **Returns**
> >
> > > the dumped object

rpyc.core.brine.**dumpable**(*obj*)

> Indicates whether the given object is *dumpable* by brine
>
> > **Returns**
> >
> > > `True` if the object is dumpable (e.g., *dump()* would succeed), `False` otherwise

Vinegar

Vinegar ("when things go sour") is a safe serializer for exceptions. The *configuration parameters* control its mode of operation, for instance, whether to allow *old-style* exceptions (that do not derive from `Exception`), whether to allow the *load()* to import custom modules (imposes a security risk), etc.

Note that by changing the configuration parameters, this module can be made non-secure. Keep this in mind.

rpyc.core.vinegar.**dump**(*typ*, *val*, *tb*, *include_local_traceback*, *include_local_version*)

> Dumps the given exceptions info, as returned by `sys.exc_info()`
>
> > **Parameters**
> >
> > > - **typ** – the exception's type (class)
> > >
> > > - **val** – the exceptions' value (instance)
> > >
> > > - **tb** – the exception's traceback (a `traceback` object)
> > >
> > > - **include_local_traceback** – whether or not to include the local traceback in the dumped info. This may expose the other side to implementation details (code) and package structure, and may theoretically impose a security risk.

> **Returns**
>> A tuple of ((module name, exception name), arguments, attributes, traceback text). This tuple can be safely passed to `brine.dump`

rpyc.core.vinegar.**load**(*val*, *import_custom_exceptions*, *instantiate_custom_exceptions*, *instantiate_oldstyle_exceptions*)

Loads a dumped exception (the tuple returned by `dump()`) info a throwable exception object. If the exception cannot be instantiated for any reason (i.e., the security parameters do not allow it, or the exception class simply doesn't exist on the local machine), a `GenericException` instance will be returned instead, containing all of the original exception's details.

> **Parameters**
>> - **val** – the dumped exception
>>
>> - **import_custom_exceptions** – whether to allow this function to import custom modules (imposes a security risk)
>>
>> - **instantiate_custom_exceptions** – whether to allow this function to instantiate "custom exceptions" (i.e., not one of the built-in exceptions, such as `ValueError`, `OSError`, etc.)
>>
>> - **instantiate_oldstyle_exceptions** – whether to allow this function to instantiate exception classes that do not derive from `BaseException`. This is required to support old-style exceptions. Not applicable for Python 3 and above.
>
> **Returns**
>> A throwable exception object

exception rpyc.core.vinegar.**GenericException**

> A 'generic exception' that is raised when the exception the gotten from the other party cannot be instantiated locally

- *Brine* - A simple and fast serialization format for immutable data (numbers, string, tuples, etc.). Brine is the "over-the-wire" encoding format of RPyC.

- *Vinegar* - A configurable serializer for exceptions. Vinegar extracts the exception's details and stores them in a brine-friendly format.

4.5.2 IO Layer

Streams

An abstraction layer over OS-dependent file-like objects, that provides a consistent view of a *duplex byte stream*.

class rpyc.core.stream.**Stream**

> Base Stream
>
> **close()**
>> closes the stream, releasing any system resources associated with it
>
> **property closed**
>> tests whether the stream is closed or not
>
> **fileno()**
>> returns the stream's file descriptor
>
> **poll**(*timeout*)
>> indicates whether the stream has data to read (within *timeout* seconds)

read(*count*)

>   reads **exactly** *count* bytes, or raise EOFError

>   > **Parameters**
>   >
>   > **count** – the number of bytes to read
>   >
>   > **Returns**
>   >
>   > read data

write(*data*)

>   writes the entire *data*, or raise EOFError

>   > **Parameters**
>   >
>   > **data** – a string of binary data

class rpyc.core.stream.**SocketStream**(*sock*)

>   A stream over a socket

>   **classmethod connect**(*host*, *port*, ***kwargs*)

>   > factory method that creates a SocketStream over a socket connected to *host* and *port*

>   > > **Parameters**
>   > >
>   > > - **host** – the host name
>   > >
>   > > - **port** – the TCP port
>   > >
>   > > - **family** – specify a custom socket family
>   > >
>   > > - **socktype** – specify a custom socket type
>   > >
>   > > - **proto** – specify a custom socket protocol
>   > >
>   > > - **timeout** – connection timeout (default is 3 seconds)
>   > >
>   > > - **nodelay** – set the TCP_NODELAY socket option
>   > >
>   > > - **keepalive** – enable TCP keepalives. The value should be a boolean, but on Linux, it can also be an integer specifying the keepalive interval (in seconds)
>   > >
>   > > - **ipv6** – if True, creates an IPv6 socket (AF_INET6); otherwise an IPv4 (AF_INET) socket is created

>   > > **Returns**
>   > >
>   > > a *SocketStream*

>   **classmethod unix_connect**(*path*, *timeout=3*)

>   > factory method that creates a SocketStream over a unix domain socket located in *path*

>   > > **Parameters**
>   > >
>   > > - **path** – the path to the unix domain socket
>   > >
>   > > - **timeout** – socket timeout

>   **classmethod ssl_connect**(*host*, *port*, *ssl_kwargs*, ***kwargs*)

>   > factory method that creates a SocketStream over an SSL-wrapped socket, connected to *host* and *port* with the given credentials.

>   > > **Parameters**
>   > >
>   > > - **host** – the host name
>   > >
>   > > - **port** – the TCP port

- **ssl_kwargs** – a dictionary of keyword arguments for `ssl.SSLContext` and `ssl.SSLContext.wrap_socket`

- **kwargs** – additional keyword arguments: `family`, `socktype`, `proto`, `timeout`, `nodelay`, passed directly to the `socket` constructor, or `ipv6`.

- **ipv6** – if True, creates an IPv6 socket (`AF_INET6`); otherwise an IPv4 (`AF_INET`) socket is created

> **Returns**
> a *SocketStream*

property closed

> tests whether the stream is closed or not

close()

> closes the stream, releasing any system resources associated with it

fileno()

> returns the stream's file descriptor

read(*count*)

> reads **exactly** *count* bytes, or raise EOFError
>
> > **Parameters**
> > **count** – the number of bytes to read
> >
> > **Returns**
> > read data

write(*data*)

> writes the entire *data*, or raise EOFError
>
> > **Parameters**
> > **data** – a string of binary data

class rpyc.core.stream.**TunneledSocketStream**(*sock*)

> A socket stream over an SSH tunnel (terminates the tunnel when the connection closes)
>
> **close()**
>
> > closes the stream, releasing any system resources associated with it

class rpyc.core.stream.**Win32PipeStream**(*incoming*, *outgoing*)

> A stream over two simplex pipes (one used to input, another for output). This is an implementation for Windows pipes (which suck)
>
> **fileno()**
>
> > returns the stream's file descriptor
>
> **property closed**
>
> > tests whether the stream is closed or not
>
> **close()**
>
> > closes the stream, releasing any system resources associated with it
>
> **read**(*count*)
>
> > reads **exactly** *count* bytes, or raise EOFError
> >
> > > **Parameters**
> > > **count** – the number of bytes to read

Returns
read data

write(*data*)

writes the entire *data*, or raise EOFError

Parameters
data – a string of binary data

poll(*timeout*, *interval=0.001*)

a Windows version of select()

class rpyc.core.stream.**NamedPipeStream**(*handle*, *is_server_side*)

A stream over two named pipes (one used to input, another for output). Windows implementation.

classmethod create_server(*pipename*, *connect=True*)

factory method that creates a server-side `NamedPipeStream`, over a newly-created *named pipe* of the given name.

Parameters

- **pipename** – the name of the pipe. It will be considered absolute if it starts with \\.; otherwise \\.\pipe\rpyc will be prepended.

- **connect** – whether to connect on creation or not

Returns
a *NamedPipeStream* instance

connect_server()

connects the server side of an unconnected named pipe (blocks until a connection arrives)

classmethod create_client(*pipename*)

factory method that creates a client-side `NamedPipeStream`, over a newly-created *named pipe* of the given name.

Parameters
pipename – the name of the pipe. It will be considered absolute if it starts with \\.; otherwise \\.\pipe\rpyc will be prepended.

Returns
a *NamedPipeStream* instance

close()

closes the stream, releasing any system resources associated with it

read(*count*)

reads **exactly** *count* bytes, or raise EOFError

Parameters
count – the number of bytes to read

Returns
read data

write(*data*)

writes the entire *data*, or raise EOFError

Parameters
data – a string of binary data

> **poll**(*timeout*, *interval=0.001*)
>> Windows version of select()

class rpyc.core.stream.**PipeStream**(*incoming*, *outgoing*)

> A stream over two simplex pipes (one used to input, another for output)

> ### classmethod from_std()
>> factory method that creates a PipeStream over the standard pipes (`stdin` and `stdout`)
>>
>>> **Returns**
>>>> a *PipeStream* instance

> ### classmethod create_pair()
>> factory method that creates two pairs of anonymous pipes, and creates two PipeStreams over them. Useful for `fork()`.
>>
>>> **Returns**
>>>> a tuple of two *PipeStream* instances

> ### property closed
>> tests whether the stream is closed or not

> ### close()
>> closes the stream, releasing any system resources associated with it

> ### fileno()
>> returns the stream's file descriptor

> ### read(*count*)
>> reads **exactly** *count* bytes, or raise EOFError
>>
>>> **Parameters**
>>>> **count** – the number of bytes to read
>>>
>>> **Returns**
>>>> read data

> ### write(*data*)
>> writes the entire *data*, or raise EOFError
>>
>>> **Parameters**
>>>> **data** – a string of binary data

Channel

Channel is an abstraction layer over streams that works with *packets of data*, rather than an endless stream of bytes, and adds support for compression.

class rpyc.core.channel.**Channel**(*stream*, *compress=True*)

> Channel implementation.

> Note: In order to avoid problems with all sorts of line-buffered transports, we deliberately add \n at the end of each frame.

> ### close()
>> closes the channel and underlying stream

> ### property closed
>> indicates whether the underlying stream has been closed

fileno()

> returns the file descriptor of the underlying stream

poll(*timeout*)

> polls the underlying steam for data, waiting up to *timeout* seconds

recv()

> Receives the next packet (or *frame*) from the underlying stream. This method will block until the packet has been read completely
>
> > **Returns**
> > string of data

send(*data*)

> Sends the given string of data as a packet over the underlying stream. Blocks until the packet has been sent.
>
> > **Parameters**
> > **data** – the byte string to send as a packet

- *Streams* - The stream layer (byte-oriented, platform-agnostic streams)
- *Channel* - The channel layer (framing and compression)

4.5.3 Protocol

Netref

NetRef: a transparent *network reference*. This module contains quite a lot of *magic*, so beware.

rpyc.core.netref.**DELETED_ATTRS** = frozenset({'__array_interface__', '__array_struct__'})

> the set of attributes that are local to the netref object

rpyc.core.netref.**LOCAL_ATTRS** = frozenset({'____conn__', '____id_pack__',
'____refcount__', '__array_interface__', '__array_struct__', '__class__', '__cmp__',
'__del__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__exit__',
'__ge__', '__getattr__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__instancecheck__', '__le__', '__lt__', '__metaclass__', '__methods__', '__module__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__slots__', '__str__', '__weakref__'})

> a list of types considered built-in (shared between connections) this is needed because iterating the members of the builtins module is not enough, some types (e.g NoneType) are not members of the builtins module. TODO: this list is not complete.

rpyc.core.netref.**syncreq**(*proxy*, *handler*, **args*)

> Performs a synchronous request on the given proxy object. Not intended to be invoked directly.
>
> > **Parameters**
> > - **proxy** – the proxy on which to issue the request
> > - **handler** – the request handler (one of the HANDLE_XXX members of rpyc.protocol. consts)
> > - **args** – arguments to the handler
> >
> > **Raises**
> > any exception raised by the operation will be raised

       Returns

          the result of the operation

rpyc.core.netref.**asyncreq**(*proxy*, *handler*, **args*)

    Performs an asynchronous request on the given proxy object. Not intended to be invoked directly.

       Parameters

- **proxy** – the proxy on which to issue the request

- **handler** – the request handler (one of the HANDLE_XXX members of rpyc.protocol. consts)

- **args** – arguments to the handler

       Returns

          an *AsyncResult* representing the operation

class rpyc.core.netref.**NetrefMetaclass**

    A *metaclass* used to customize the __repr__ of netref classes. It is quite useless, but it makes debugging and interactive programming easier

class rpyc.core.netref.**BaseNetref**(*conn*, *id_pack*)

    The base netref class, from which all netref classes derive. Some netref classes are "pre-generated" and cached upon importing this module (those defined in the _builtin_types), and they are shared between all connections.

    The rest of the netref classes are created by rpyc.core.protocol.Connection._unbox(), and are private to the connection.

    Do not use this class directly; use *class_factory()* instead.

       Parameters

- **conn** – the *rpyc.core.protocol.Connection* instance

- **id_pack** – id tuple for an object ~ (name_pack, remote-class-id, remote-instance-id) (cont.) name_pack := __module__.__name__ (hits or misses on builtin cache and sys.module)

      remote-class-id := id of object class (hits or misses on netref classes cache and instance checks) remote-instance-id := id object instance (hits or misses on proxy cache)

      id_pack is usually created by rpyc.lib.get_id_pack

class rpyc.core.netref.**NetrefClass**(*class_obj*)

    a descriptor of the class being proxied

    Future considerations:

- there may be a cleaner alternative but lib.compat.with_metaclass prevented using __new__

- consider using __slot__ for this class

- revisit the design choice to use properties here

property **instance**

    accessor to class object for the instance being proxied

property **owner**

    accessor to the class object for the instance owner being proxied

rpyc.core.netref.**class_factory**(*id_pack*, *methods*)

> Creates a netref class proxying the given class
>
> > **Parameters**
> >
> > - **id_pack** – the id pack used for proxy communication
> >
> > - **methods** – a list of (`method name`, `docstring`) tuples, of the methods that the class defines
> >
> > **Returns**
> > a netref class

Async

class rpyc.core.async_.**AsyncResult**(*conn*)

> *AsyncResult* represents a computation that occurs in the background and will eventually have a result. Use the `value` property to access the result (which will block if the result has not yet arrived).
>
> **wait**()
>
> > Waits for the result to arrive. If the AsyncResult object has an expiry set, and the result did not arrive within that timeout, an `AsyncResultTimeout` exception is raised
>
> **add_callback**(*func*)
>
> > Adds a callback to be invoked when the result arrives. The callback function takes a single argument, which is the current AsyncResult (`self`). If the result has already arrived, the function is invoked immediately.
> >
> > > **Parameters**
> > > **func** – the callback function to add
>
> **set_expiry**(*timeout*)
>
> > Sets the expiry time (in seconds, relative to now) or `None` for unlimited time
> >
> > > **Parameters**
> > > **timeout** – the expiry time in seconds or `None`
>
> property **ready**
>
> > Indicates whether the result has arrived
>
> property **error**
>
> > Indicates whether the returned result is an exception
>
> property **expired**
>
> > Indicates whether the AsyncResult has expired
>
> property **value**
>
> > Returns the result of the operation. If the result has not yet arrived, accessing this property will wait for it. If the result does not arrive before the expiry time elapses, `AsyncResultTimeout` is raised. If the returned result is an exception, it will be raised here. Otherwise, the result is returned directly.

Protocol

The RPyC protocol

exception rpyc.core.protocol.**PingError**

> The exception raised should *Connection.ping()* fail

rpyc.core.protocol.**DEFAULT_CONFIG** = {'allow_all_attrs': False, 'allow_delattr': False,
'allow_exposed_attrs': True, 'allow_getattr': True, 'allow_pickle': False,
'allow_public_attrs': False, 'allow_safe_attrs': True, 'allow_setattr': False,
'before_closed': None, 'bind_threads': False, 'close_catchall': False, 'connid':
None, 'credentials': None, 'endpoints': None, 'exposed_prefix': 'exposed_',
'import_custom_exceptions': False, 'include_local_traceback': True,
'include_local_version': True, 'instantiate_custom_exceptions': False,
'instantiate_oldstyle_exceptions': False, 'log_exceptions': True, 'logger': None,
'propagate_KeyboardInterrupt_locally': True, 'propagate_SystemExit_locally': False,
'safe_attrs': {'__abs__', '__add__', '__and__', '__bool__', '__cmp__', '__contains__',
'__delitem__', '__delslice__', '__div__', '__divmod__', '__doc__', '__enter__', '__eq__',
'__exit__', '__float__', '__floordiv__', '__format__', '__ge__', '__getitem__',
'__getslice__', '__gt__', '__hash__', '__hex__', '__iadd__', '__iand__', '__idiv__',
'__ifloordiv__', '__ilshift__', '__imod__', '__imul__', '__index__', '__int__',
'__invert__', '__ior__', '__ipow__', '__irshift__', '__isub__', '__iter__',
'__itruediv__', '__ixor__', '__le__', '__len__', '__length_hint__', '__long__',
'__lshift__', '__lt__', '__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__next__',
'__nonzero__', '__oct__', '__or__', '__pos__', '__pow__', '__radd__', '__rand__',
'__rdiv__', '__rdivmod__', '__repr__', '__rfloordiv__', '__rlshift__', '__rmod__',
'__rmul__', '__ror__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__',
'__rtruediv__', '__rxor__', '__setitem__', '__setslice__', '__str__', '__sub__',
'__truediv__', '__xor__', 'next'}, 'sync_request_timeout': 30}

> The default configuration dictionary of the protocol. You can override these parameters by passing a different configuration dict to the *Connection* class.

Note: You only need to override the parameters you want to change. There's no need to repeat parameters whose values remain unchanged.

Parameter	Default value	Description
allow_safe_a	True	Whether to allow the use of *safe* attributes (only those listed as `safe_attrs`)
allow_expose	True	Whether to allow exposed attributes (attributes that start with the `exposed_prefix`)
allow_public_	False	Whether to allow public attributes (attributes that don't start with _)
allow_all_at	False	Whether to allow all attributes (including private)
safe_attrs	set([..])	The set of attributes considered safe
exposed_pref	"expo	The prefix of exposed attributes
allow_getatt	True	Whether to allow getting of attributes (`getattr`)
allow_setatt	False	Whether to allow setting of attributes (`setattr`)
allow_delatt	False	Whether to allow deletion of attributes (`delattr`)
allow_pickle	False	Whether to allow the use of `pickle`
include_loca	True	Whether to include the local traceback in the remote exception
instantiate_	False	Whether to allow instantiation of custom exceptions (not the built in ones)
import_custo	False	Whether to allow importing of exceptions from not-yet-imported modules
instantiate_	False	Whether to allow instantiation of exceptions which don't derive from `Exception`. This is not applicable for Python 3 and later.
propagate_Sy	False	Whether to propagate `SystemExit` locally (kill the server) or to the other party (kill the client)
propagate_Ke	False	Whether to propagate `KeyboardInterrupt` locally (kill the server) or to the other party (kill the client)
logger	None	The logger instance to use to log exceptions (before they are sent to the other party) and other events. If `None`, no logging takes place.
connid	None	**Runtime**: the RPyC connection ID (used mainly for debugging purposes)
credentials	None	**Runtime**: the credentials object that was returned by the server's *authenticator* or `None`
endpoints	None	**Runtime**: The connection's endpoints. This is a tuple made of the local socket endpoint (`getsockname`) and the remote one (`getpeername`). This is set by the server upon accepting a connection; client side connections do no have this configuration option set.
sync_request	30	Default timeout for waiting results
bind_threads	False	Whether to restrict request/reply by thread (experimental). The default value is False. Setting the environment variable *RPYC_BIND_THREADS* to *"true"* will enable this feature.

class rpyc.core.protocol.**Connection**(*root*, *channel*, *config={}*)

    The RPyC *connection* (AKA *protocol*).

Objects referenced over the connection are either local or remote. This class retains a strong reference to local objects that is deleted when the reference count is zero. Remote/proxied objects have a life-cycle controlled by a different address space. Since garbage collection is handled on the remote end, a weak reference is used for netrefs.

    Parameters

- **root** – the *Service* object to expose
- **channel** – the *Channel* over which messages are passed
- **config** – the connection's configuration dict (overriding parameters from the *default configuration*)

close()
>   closes the connection, releasing all held resources

property closed
>   Indicates whether the connection has been closed or not

fileno()
>   Returns the connectin's underlying file descriptor

ping(*data=None*, *timeout=3*)
>   Asserts that the other party is functioning properly, by making sure the *data* is echoed back before the *timeout* expires

>   **Parameters**

>   - **data** – the data to send (leave None for the default buffer)

>   - **timeout** – the maximal time to wait for echo

>   **Raises**
>   [`PingError`](#) if the echoed data does not match

>   **Raises**
>   EOFError if the remote host closes the connection

serve(*timeout=1*, *wait_for_lock=True*, *waiting=<function Connection.<lambda>>*)
>   Serves a single request or reply that arrives within the given time frame (default is 1 sec). Note that the dispatching of a request might trigger multiple (nested) requests, thus this function may be reentrant.

>   **Returns**
>   True if a request or reply were received, False otherwise.

poll(*timeout=0*)
>   Serves a single transaction, should one arrives in the given interval. Note that handling a request/reply may trigger nested requests, which are all part of a single transaction.

>   **Returns**
>   True if a transaction was served, False otherwise

serve_all()
>   Serves all requests and replies for as long as the connection is alive.

serve_threaded(*thread_count=10*)
>   Serves all requests and replies for as long as the connection is alive.

>   CAVEAT: using non-immutable types that require a netref to be constructed to serve a request, or invoking anything else that performs a sync_request, may timeout due to the sync_request reply being received by another thread serving the connection. A more conventional approach where each client thread opens a new connection would allow *ThreadedServer* to naturally avoid such multiplexing issues and is the preferred approach for threading procedures that invoke sync_request. See issue #345

poll_all(*timeout=0*)
>   Serves all requests and replies that arrive within the given interval.

>   **Returns**
>   True if at least a single transaction was served, False otherwise

sync_request(*handler*, **args*)
>   requests, sends a synchronous request (waits for the reply to arrive)

> **Raises**
>> any exception that the requests may be generated
>
> **Returns**
>> the result of the request

async_request(*handler*, **args*, ***kwargs*)

> Send an asynchronous request (does not wait for it to finish)
>
> **Returns**
>> an `rpyc.core.async_.AsyncResult` object, which will eventually hold the result (or exception)

property root

> Fetches the root object (service) of the other party

Service

Services are the heart of RPyC: each side of the connection exposes a *service*, which define the capabilities available to the other side.

Note that the services by both parties need not be symmetric, e.g., one side may exposed *service A*, while the other may expose *service B*. As long as the two can interoperate, you're good to go.

class rpyc.core.service.**Service**

> The service base-class. Derive from this class to implement custom RPyC services:
>
> - The name of the class implementing the `Foo` service should match the pattern `FooService` (suffixed by the word 'Service')

```python
class FooService(Service):
    pass

FooService.get_service_name() # 'FOO'
FooService.get_service_aliases() # ['FOO']
```

> - To supply a different name or aliases, use the `ALIASES` class attribute

```python
class Foobar(Service):
    ALIASES = ["foo", "bar", "lalaland"]

Foobar.get_service_name() # 'FOO'
Foobar.get_service_aliases() # ['FOO', 'BAR', 'LALALAND']
```

> - Override `on_connect()` to perform custom initialization
>
> - Override `on_disconnect()` to perform custom finalization
>
> - To add exposed methods or attributes, simply define them normally, but prefix their name by `exposed_`, e.g.

```python
class FooService(Service):
    def exposed_add(self, x, y):
        return x + y
```

> - All other names (not prefixed by `exposed_`) are local (not accessible to the other party)

> **Note:** You can override _rpyc_getattr, _rpyc_setattr and _rpyc_delattr to change attribute lookup –
> but beware of possible **security implications!**

on_connect(*conn*)

> called when the connection is established

on_disconnect(*conn*)

> called when the connection had already terminated for cleanup (must not perform any IO on the connection)

classmethod get_service_aliases()

> returns a list of the aliases of this service

classmethod get_service_name()

> returns the canonical name of the service (which is its first alias)

classmethod exposed_get_service_aliases()

> returns a list of the aliases of this service

classmethod exposed_get_service_name()

> returns the canonical name of the service (which is its first alias)

class rpyc.core.service.**VoidService**

> void service - an do-nothing service

class rpyc.core.service.**ModuleNamespace**(*getmodule*)

> used by the *SlaveService* to implement the magical 'module namespace'

class rpyc.core.service.**SlaveService**

> The SlaveService allows the other side to perform arbitrary imports and execution arbitrary code on the server.
> This is provided for compatibility with the classic RPyC (2.6) modus operandi.
>
> This service is very useful in local, secure networks, but it exposes a **major security risk** otherwise.

on_connect(*conn*)

> called when the connection is established

class rpyc.core.service.**FakeSlaveService**

> VoidService that can be used for connecting to peers that operate a *MasterService*, *ClassicService*, or the
> old SlaveService (pre v3.5) without exposing any functionality to them.

class rpyc.core.service.**MasterService**

> Peer for a new-style (>=v3.5) *SlaveService*. Use this service if you want to connect to a SlaveService
> without exposing any functionality to them.

on_connect(*conn*)

> called when the connection is established

class rpyc.core.service.**ClassicService**

> Full duplex master/slave service, i.e. both parties have full control over the other. Must be used by both parties.

class rpyc.core.service.**ClassicClient**

> MasterService that can be used for connecting to peers that operate a *MasterService*, *ClassicService* with-
> out exposing any functionality to them.

- *Protocol* - The RPyC protocol (*Connection* class)

- *Service* - The RPyC service model

- *Netref* - Implementation of transparent object proxies (netrefs)
- *Async* - Asynchronous object proxies (netrefs)

4.5.4 Server-Side

Server

rpyc plug-in server (threaded or forking)

class `rpyc.utils.server.Server`(*service*, *hostname=None*, *ipv6=False*, *port=0*, *backlog=4096*,
reuse_addr=True, *authenticator=None*, *registrar=None*,
auto_register=None, *protocol_config=None*, *logger=None*,
listener_timeout=0.5, *socket_path=None*)

Base server implementation

Parameters

- **service** – the *Service* to expose
- **hostname** – the host to bind to. By default, the 'wildcard address' is used to listen on all interfaces. if not properly secured, the server can receive traffic from unintended or even malicious sources.
- **ipv6** – whether to create an IPv6 or IPv4 socket. The default is IPv4
- **port** – the TCP port to bind to
- **backlog** – the socket's backlog (passed to `listen()`)
- **reuse_addr** – whether or not to create the socket with the SO_REUSEADDR option set.
- **authenticator** – the *Authenticators* to use. If `None`, no authentication is performed.
- **registrar** – the *RegistryClient* to use. If `None`, a default *UDPRegistryClient* will be used
- **auto_register** – whether or not to register using the *registrar*. By default, the server will attempt to register only if a registrar was explicitly given.
- **protocol_config** – the *configuration dictionary* that is passed to the RPyC connection
- **logger** – the `logger` to use (of the built-in `logging` module). If `None`, a default logger will be created.
- **listener_timeout** – the timeout of the listener socket; set to `None` to disable (e.g. on embedded platforms with limited battery)

close()

Closes (terminates) the server and all of its clients. If applicable, also unregisters from the registry server

fileno()

returns the listener socket's file descriptor

accept()

accepts an incoming socket connection (blocking)

start()

Starts the server (blocking). Use *close()* to stop

class `rpyc.utils.server.`**OneShotServer**(*service*, *hostname=None*, *ipv6=False*, *port=0*, *backlog=4096*, *reuse_addr=True*, *authenticator=None*, *registrar=None*, *auto_register=None*, *protocol_config=None*, *logger=None*, *listener_timeout=0.5*, *socket_path=None*)

> A server that handles a single connection (blockingly), and terminates after that

> Parameters: see *Server*

class `rpyc.utils.server.`**ThreadedServer**(*service*, *hostname=None*, *ipv6=False*, *port=0*, *backlog=4096*, *reuse_addr=True*, *authenticator=None*, *registrar=None*, *auto_register=None*, *protocol_config=None*, *logger=None*, *listener_timeout=0.5*, *socket_path=None*)

> A server that spawns a thread for each connection. Works on any platform that supports threads.

> Parameters: see *Server*

class `rpyc.utils.server.`**ThreadPoolServer**(**args*, ***kwargs*)

> This server is threaded like the ThreadedServer but reuses threads so that recreation is not necessary for each request. The pool of threads has a fixed size that can be set with the 'nbThreads' argument. The default size is 20. The server dispatches request to threads by batch, that is a given thread may process up to request_batch_size requests from the same connection in one go, before it goes to the next connection with pending requests. By default, self.request_batch_size is set to 10 and it can be overwritten in the constructor arguments.

> Contributed by @*sponce*

> Parameters: see *Server*

> **close**()
>
> > closes a ThreadPoolServer. In particular, joins the thread pool.

class `rpyc.utils.server.`**ForkingServer**(**args*, ***kwargs*)

> A server that forks a child process for each connection. Available on POSIX compatible systems only.

> Parameters: see *Server*

> **close**()
>
> > Closes (terminates) the server and all of its clients. If applicable, also unregisters from the registry server

class `rpyc.utils.server.`**GeventServer**(*service*, *hostname=None*, *ipv6=False*, *port=0*, *backlog=4096*, *reuse_addr=True*, *authenticator=None*, *registrar=None*, *auto_register=None*, *protocol_config=None*, *logger=None*, *listener_timeout=0.5*, *socket_path=None*)

> gevent based Server. Requires using `gevent.monkey.patch_all()`.

Authenticators

An *authenticator* is basically a callable object that takes a socket and "authenticates" it in some way. Upon success, it must return a tuple containing a **socket-like** object and its **credentials** (any object), or raise an *AuthenticationError* upon failure. The credentials are any object you wish to associate with the authentication, and it's stored in the connection's *configuration dict* under the key "credentials".

There are no constraints on what the authenticators, for instance:

```
def magic_word_authenticator(sock):
    if sock.recv(5) != "Ma6ik":
        raise AuthenticationError("wrong magic word")
    return sock, None
```

RPyC comes bundled with an authenticator for SSL (using certificates). This authenticator, for instance, both verifies the peer's identity and wraps the socket with an encrypted transport (which replaces the original socket).

Authenticators are used by *Server* to validate an incoming connection. Using them is pretty trivial

```
s = ThreadedServer(...., authenticator = magic_word_authenticator)
s.start()
```

exception rpyc.utils.authenticators.**AuthenticationError**

> raised to signal a failed authentication attempt

class rpyc.utils.authenticators.**SSLAuthenticator**(*keyfile*, *certfile*, *ca_certs=None*, *cert_reqs=None*, *ssl_version=None*, *ciphers=None*)

> An implementation of the authenticator protocol for SSL. The given socket is wrapped by `ssl.SSLContext.wrap_socket` and is validated based on certificates
>
> > **Parameters**
> >
> > - **keyfile** – the server's key file
> >
> > - **certfile** – the server's certificate file
> >
> > - **ca_certs** – the server's certificate authority file
> >
> > - **cert_reqs** – the certificate requirements. By default, if `ca_cert` is specified, the requirement is set to `CERT_REQUIRED`; otherwise it is set to `CERT_NONE`
> >
> > - **ciphers** – the list of ciphers to use, or `None`, if you do not wish to restrict the available ciphers. New in Python 2.7/3.2
> >
> > - **ssl_version** – the SSL version to use
>
> Refer to ssl.SSLContext for more info.
>
> Clients can connect to this authenticator using *rpyc.utils.factory.ssl_connect()*. Classic clients can use directly *rpyc.utils.classic.ssl_connect()* which sets the correct service parameters.

Registry

RPyC Registry Server maintains service information on RPyC services for *Service Registry and Discovery patterns*. Service Registry and Discovery patterns solve the connectivity problem for communication between services and external consumers. RPyC services will register with the server when `auto_register` is `True`.

Service registries such as Avahi and Bonjour are alternatives to the RPyC Registry Server. These alternatives do no support Windows and have more restrictive licensing.

Refer to `rpyc/scripts/rpyc_registry.py` for more info.

class rpyc.utils.registry.**RegistryServer**(*listenersock*, *pruning_timeout=None*, *logger=None*, *allow_listing=False*)

> Base registry server
>
> **on_service_added**(*name*, *addrinfo*)
>
> > called when a new service joins the registry (but not on keepalives). override this to add custom logic
>
> **on_service_removed**(*name*, *addrinfo*)
>
> > called when a service unregisters or is pruned. override this to add custom logic
>
> **cmd_query**(*host*, *name*)
>
> > implementation of the `query` command

cmd_list(*host*, *filter_host*)

> implementation for the `list` command

cmd_register(*host*, *names*, *port*)

> implementation of the `register` command

cmd_unregister(*host*, *port*)

> implementation of the `unregister` command

start()

> Starts the registry server (blocks)

close()

> Closes (terminates) the registry server

class rpyc.utils.registry.**UDPRegistryServer**(*host='0.0.0.0'*, *port=18811*, *pruning_timeout=None*, *logger=None*, *allow_listing=False*)

> UDP-based registry server. The server listens to UDP broadcasts and answers them. Useful in local networks, were broadcasts are allowed

class rpyc.utils.registry.**TCPRegistryServer**(*host='0.0.0.0'*, *port=18811*, *pruning_timeout=None*, *logger=None*, *reuse_addr=True*, *allow_listing=False*)

> TCP-based registry server. The server listens to a certain TCP port and answers requests. Useful when you need to cross routers in the network, since they block UDP broadcasts

class rpyc.utils.registry.**RegistryClient**(*ip*, *port*, *timeout*, *logger=None*)

> Base registry client. Also known as **registrar**

> **discover**(*name*)
>
> > Sends a query for the specified service name.
> >
> > > **Parameters**
> > > **name** – the service name (or one of its aliases)
> > >
> > > **Returns**
> > > a list of (`host, port`) tuples

> **list**(*filter_host=None*)
>
> > Send a query for the full lists of exposed servers :returns: a list of `` service_name ``

> **register**(*aliases*, *port*)
>
> > Registers the given service aliases with the given TCP port. This API is intended to be called only by an RPyC server.
> >
> > > **Parameters**
> > >
> > > • **aliases** – the [service's](#) aliases
> > >
> > > • **port** – the listening TCP port of the server

> **unregister**(*port*)
>
> > Unregisters the given RPyC server. This API is intended to be called only by an RPyC server.
> >
> > > **Parameters**
> > > **port** – the listening TCP port of the RPyC server to unregister

class rpyc.utils.registry.**UDPRegistryClient**(*ip='255.255.255.255'*, *port=18811*, *timeout=2*, *bcast=None*, *logger=None*, *ipv6=False*)

UDP-based registry clients. By default, it sends UDP broadcasts (requires special user privileges on certain OS's) and collects the replies. You can also specify the IP address to send to.

Example:

```
registrar = UDPRegistryClient()
list_of_services = registrar.list()
list_of_servers = registrar.discover("foo")
```

Note: Consider using *rpyc.utils.factory.discover()* instead

---

discover(*name*)

> Sends a query for the specified service name.
>
> > **Parameters**
> >   **name** – the service name (or one of its aliases)
> >
> > **Returns**
> >   a list of (`host, port`) tuples

list(*filter_host=None*)

> Send a query for the full lists of exposed servers :returns: a list of `` service_name ``

register(*aliases*, *port*, *interface=''*)

> Registers the given service aliases with the given TCP port. This API is intended to be called only by an RPyC server.
>
> > **Parameters**
> >
> > - **aliases** – the *service's* aliases
> >
> > - **port** – the listening TCP port of the server

unregister(*port*)

> Unregisters the given RPyC server. This API is intended to be called only by an RPyC server.
>
> > **Parameters**
> >   **port** – the listening TCP port of the RPyC server to unregister

class rpyc.utils.registry.**TCPRegistryClient**(*ip*, *port=18811*, *timeout=2*, *logger=None*)

TCP-based registry client. You must specify the host (registry server) to connect to.

Example:

```
registrar = TCPRegistryClient("localhost")
list_of_services = registrar.list()
list_of_servers = registrar.discover("foo")
```

Note: Consider using *rpyc.utils.factory.discover()* instead

---

discover(*name*)

> Sends a query for the specified service name.
>
> > **Parameters**
> >   **name** – the service name (or one of its aliases)

4.5. API Reference 69

> **Returns**
> a list of (`host, port`) tuples

list(*filter_host=None*)

> Send a query for the full lists of exposed servers :returns: a list of `` service_name ``

register(*aliases*, *port*, *interface=''*)

> Registers the given service aliases with the given TCP port. This API is intended to be called only by an RPyC server.
>
> > **Parameters**
> >
> > - **aliases** – the `service's` aliases
> >
> > - **port** – the listening TCP port of the server

unregister(*port*)

> Unregisters the given RPyC server. This API is intended to be called only by an RPyC server.
>
> > **Parameters**
> > **port** – the listening TCP port of the RPyC server to unregister

- *Server* - The core implementation of RPyC servers; includes the implementation of the forking and threaded servers.

- *Registry* - Implementation of the Service Registry; the registry is a bonjour-like discovery agent, with which RPyC servers register themselves, and allows clients to locate different servers by name.

- *Authenticators* - Implementation of two common authenticators, for SSL and TLSlite.

4.5.5 Client-Side

Factories

RPyC connection factories: ease the creation of a connection for the common cases)

exception rpyc.utils.factory.**DiscoveryError**

exception rpyc.utils.factory.**ForbiddenError**

rpyc.utils.factory.**connect_channel**(*channel*, *service=<class 'rpyc.core.service.VoidService'>*, *config={}*)

> creates a connection over a given channel
>
> > **Parameters**
> >
> > - **channel** – the channel to use
> >
> > - **service** – the local service to expose (defaults to Void)
> >
> > - **config** – configuration dict
> >
> > **Returns**
> > an RPyC connection

rpyc.utils.factory.**connect_stream**(*stream*, *service=<class 'rpyc.core.service.VoidService'>*, *config={}*)

> creates a connection over a given stream
>
> > **Parameters**
> >
> > - **stream** – the stream to use
> >
> > - **service** – the local service to expose (defaults to Void)

- **config** – configuration dict

> **Returns**
>> an RPyC connection

rpyc.utils.factory.**connect_pipes**(*input*, *output*, *service=<class 'rpyc.core.service.VoidService'>*, *config={}*)

> creates a connection over the given input/output pipes

>> **Parameters**

>>> - **input** – the input pipe

>>> - **output** – the output pipe

>>> - **service** – the local service to expose (defaults to Void)

>>> - **config** – configuration dict

>> **Returns**
>>> an RPyC connection

rpyc.utils.factory.**connect_stdpipes**(*service=<class 'rpyc.core.service.VoidService'>*, *config={}*)

> creates a connection over the standard input/output pipes

>> **Parameters**

>>> - **service** – the local service to expose (defaults to Void)

>>> - **config** – configuration dict

>> **Returns**
>>> an RPyC connection

rpyc.utils.factory.**connect**(*host*, *port*, *service=<class 'rpyc.core.service.VoidService'>*, *config={}*, *ipv6=False*, *keepalive=False*)

> creates a socket-connection to the given host and port

>> **Parameters**

>>> - **host** – the hostname to connect to

>>> - **port** – the TCP port to use

>>> - **service** – the local service to expose (defaults to Void)

>>> - **config** – configuration dict

>>> - **ipv6** – whether to create an IPv6 socket (defaults to `False`)

>>> - **keepalive** – whether to set TCP keepalive on the socket (defaults to `False`)

>> **Returns**
>>> an RPyC connection

rpyc.utils.factory.**unix_connect**(*path*, *service=<class 'rpyc.core.service.VoidService'>*, *config={}*)

> creates a socket-connection to the given unix domain socket

>> **Parameters**

>>> - **path** – the path to the unix domain socket

>>> - **service** – the local service to expose (defaults to Void)

>>> - **config** – configuration dict

>   **Returns**
>   an RPyC connection

rpyc.utils.factory.**ssl_connect**(*host*, *port*, *keyfile=None*, *certfile=None*, *ca_certs=None*, *cert_reqs=None*, *ssl_version=None*, *ciphers=None*, *service=<class 'rpyc.core.service.VoidService'>*, *config={}*, *ipv6=False*, *keepalive=False*, *verify_mode=None*)

creates an SSL-wrapped connection to the given host (encrypted and authenticated).

>   **Parameters**
>
>   - **host** – the hostname to connect to
>
>   - **port** – the TCP port to use
>
>   - **service** – the local service to expose (defaults to Void)
>
>   - **config** – configuration dict
>
>   - **ipv6** – whether to create an IPv6 socket or an IPv4 one(defaults to `False`)
>
>   - **keepalive** – whether to set TCP keepalive on the socket (defaults to `False`)
>
>   - **keyfile** – see `ssl.SSLContext.load_cert_chain`. May be `None`
>
>   - **certfile** – see `ssl.SSLContext.load_cert_chain`. May be `None`
>
>   - **ca_certs** – see `ssl.SSLContext.load_verify_locations`. May be `None`
>
>   - **cert_reqs** – see `ssl.SSLContext.verify_mode`. By default, if `ca_cert` is specified, the requirement is set to `CERT_REQUIRED`; otherwise it is set to `CERT_NONE`
>
>   - **ssl_version** – see `ssl.SSLContext`. The default is defined by `ssl.create_default_context`
>
>   - **ciphers** – see `ssl.SSLContext.set_ciphers`. May be `None`. New in Python 2.7/3.2
>
>   - **verify_mode** – see `ssl.SSLContext.verify_mode`
>
>   **Returns**
>   an RPyC connection

rpyc.utils.factory.**ssh_connect**(*remote_machine*, *remote_port*, *service=<class 'rpyc.core.service.VoidService'>*, *config={}*)

Connects to an RPyC server over an SSH tunnel (created by plumbum). See Plumbum tunneling for further details.

Note: This function attempts to allocate a free TCP port for the underlying tunnel, but doing so is inherently prone to a race condition with other processes who might bind the same port before sshd does. Albeit unlikely, there is no sure way around it.

>   **Parameters**
>
>   - **remote_machine** – an `plumbum.remote.RemoteMachine` instance
>
>   - **remote_port** – the port of the remote server
>
>   - **service** – the local service to expose (defaults to Void)
>
>   - **config** – configuration dict
>
>   **Returns**
>   an RPyC connection

rpyc.utils.factory.**discover**(*service_name*, *host=None*, *registrar=None*, *timeout=2*)

discovers hosts running the given service

> **Parameters**
>
> - **service_name** – the service to look for
>
> - **host** – limit the discovery to the given host only (None means any host)
>
> - **registrar** – use this registry client to discover services. if None, use the default UDPReg-istryClient with the default settings.
>
> - **timeout** – the number of seconds to wait for a reply from the registry if no hosts are found, raises DiscoveryError
>
> **Raises**
>
> `DiscoveryError` if no server is found
>
> **Returns**
>
> a list of (ip, port) pairs

rpyc.utils.factory.**connect_by_service**(*service_name*, *host=None*, *registrar=None*, *timeout=2*, *service=<class 'rpyc.core.service.VoidService'>*, *config={}*)

create a connection to an arbitrary server that exposes the requested service

> **Parameters**
>
> - **service_name** – the service to discover
>
> - **host** – limit discovery to the given host only (None means any host)
>
> - **service** – the local service to expose (defaults to Void)
>
> - **config** – configuration dict
>
> **Raises**
>
> `DiscoveryError` if no server is found
>
> **Returns**
>
> an RPyC connection

rpyc.utils.factory.**connect_subproc**(*args*, *service=<class 'rpyc.core.service.VoidService'>*, *config={}*)

runs an rpyc server on a child process that and connects to it over the stdio pipes. uses the subprocess module.

> **Parameters**
>
> - **args** – the args to Popen, e.g., [“python”, “-u”, “myfile.py”]
>
> - **service** – the local service to expose (defaults to Void)
>
> - **config** – configuration dict

rpyc.utils.factory.**connect_thread**(*service=<class 'rpyc.core.service.VoidService'>*, *config={}*, *remote_service=<class 'rpyc.core.service.VoidService'>*, *remote_config={}*)

starts an rpyc server on a new thread, bound to an arbitrary port, and connects to it over a socket.

> **Parameters**
>
> - **service** – the local service to expose (defaults to Void)
>
> - **config** – configuration dict
>
> - **remote_service** – the remote service to expose (of the server; defaults to Void)
>
> - **remote_config** – remote configuration dict (of the server)

rpyc.utils.factory.**connect_multiprocess**(*service=<class 'rpyc.core.service.VoidService'>, config={},*
remote_service=<class 'rpyc.core.service.VoidService'>,
remote_config={}, args={})

starts an rpyc server on a new process, bound to an arbitrary port, and connects to it over a socket. Basically a copy of connect_thread(). However if args is used and if these are shared memory then changes will be bi-directional. That is we now have access to shared memory.

> **Parameters**
>
> - **service** – the local service to expose (defaults to Void)
>
> - **config** – configuration dict
>
> - **remote_service** – the remote service to expose (of the server; defaults to Void)
>
> - **remote_config** – remote configuration dict (of the server)
>
> - **args** – dict of local vars to pass to new connection, form {'name':var}

Contributed by *@tvanzyl*

Classic

rpyc.utils.classic.**connect_channel**(*channel*)

Creates an RPyC connection over the given `channel`

> **Parameters**
> **channel** – the `rpyc.core.channel.Channel` instance
>
> **Returns**
> an RPyC connection exposing `SlaveService`

rpyc.utils.classic.**connect_stream**(*stream*)

Creates an RPyC connection over the given stream

> **Parameters**
> **channel** – the `rpyc.core.stream.Stream` instance
>
> **Returns**
> an RPyC connection exposing `SlaveService`

rpyc.utils.classic.**connect_stdpipes**()

Creates an RPyC connection over the standard pipes (`stdin` and `stdout`)

> **Returns**
> an RPyC connection exposing `SlaveService`

rpyc.utils.classic.**connect_pipes**(*input, output*)

Creates an RPyC connection over two pipes

> **Parameters**
>
> - **input** – the input pipe
>
> - **output** – the output pipe
>
> **Returns**
> an RPyC connection exposing `SlaveService`

rpyc.utils.classic.**connect**(*host*, *port=18812*, *ipv6=False*, *keepalive=False*)

> Creates a socket connection to the given host and port.
>
> > **Parameters**
> >
> > - **host** – the host to connect to
> >
> > - **port** – the TCP port
> >
> > - **ipv6** – whether to create an IPv6 socket or IPv4
> >
> > **Returns**
> >
> > an RPyC connection exposing `SlaveService`

rpyc.utils.classic.**unix_connect**(*path*)

> Creates a socket connection to the given host and port.
>
> > **Parameters**
> >
> > **path** – the path to the unix domain socket
> >
> > **Returns**
> >
> > an RPyC connection exposing `SlaveService`

rpyc.utils.classic.**ssl_connect**(*host*, *port=18821*, *keyfile=None*, *certfile=None*, *ca_certs=None*, *cert_reqs=None*, *ssl_version=None*, *ciphers=None*, *ipv6=False*)

> Creates a secure (SSL) socket connection to the given host and port, authenticating with the given certfile and CA file.
>
> > **Parameters**
> >
> > - **host** – the host to connect to
> >
> > - **port** – the TCP port to use
> >
> > - **ipv6** – whether to create an IPv6 socket or an IPv4 one
>
> The following arguments are passed to [ssl.SSLContext](#) and its corresponding methods:
>
> > **Parameters**
> >
> > - **keyfile** – see `ssl.SSLContext.load_cert_chain`. May be `None`
> >
> > - **certfile** – see `ssl.SSLContext.load_cert_chain`. May be `None`
> >
> > - **ca_certs** – see `ssl.SSLContext.load_verify_locations`. May be `None`
> >
> > - **cert_reqs** – see `ssl.SSLContext.verify_mode`. By default, if `ca_cert` is specified, the requirement is set to `CERT_REQUIRED`; otherwise it is set to `CERT_NONE`
> >
> > - **ssl_version** – see `ssl.SSLContext`. The default is defined by `ssl.create_default_context`
> >
> > - **ciphers** – see `ssl.SSLContext.set_ciphers`. May be `None`. New in Python 2.7/3.2
> >
> > **Returns**
> >
> > an RPyC connection exposing `SlaveService`

rpyc.utils.classic.**ssh_connect**(*remote_machine*, *remote_port*)

> Connects to the remote server over an SSH tunnel. See `rpyc.utils.factory.ssh_connect()` for more info.
>
> > **Parameters**
> >
> > - **remote_machine** – the `plumbum.remote.RemoteMachine` instance
> >
> > - **remote_port** – the remote TCP port

> **Returns**
>> an RPyC connection exposing `SlaveService`

rpyc.utils.classic.**connect_subproc**(*server_file=None*)

> Runs an RPyC classic server as a subprocess and returns an RPyC connection to it over stdio
>
>> **Parameters**
>>> **server_file** – The full path to the server script (`rpyc_classic.py`). If not given, `which` `rpyc_classic.py` will be attempted.
>>
>> **Returns**
>>> an RPyC connection exposing `SlaveService`

rpyc.utils.classic.**connect_thread**()

> Starts a SlaveService on a thread and connects to it. Useful for testing purposes. See *rpyc.utils.factory.* *connect_thread()*
>
>> **Returns**
>>> an RPyC connection exposing `SlaveService`

rpyc.utils.classic.**connect_multiprocess**(*args={}*)

> Starts a SlaveService on a multiprocess process and connects to it. Useful for testing purposes and running multicore code that's uses shared memory. See *rpyc.utils.factory.connect_multiprocess()*
>
>> **Returns**
>>> an RPyC connection exposing `SlaveService`

rpyc.utils.classic.**upload**(*conn*, *localpath*, *remotepath*, *filter=None*, *ignore_invalid=False*, *chunk_size=64000*)

> uploads a file or a directory to the given remote path
>
>> **Parameters**
>>> - **localpath** – the local file or directory
>>>
>>> - **remotepath** – the remote path
>>>
>>> - **filter** – a predicate that accepts the filename and determines whether it should be uploaded; None means any file
>>>
>>> - **chunk_size** – the IO chunk size

rpyc.utils.classic.**download**(*conn*, *remotepath*, *localpath*, *filter=None*, *ignore_invalid=False*, *chunk_size=64000*)

> download a file or a directory to the given remote path
>
>> **Parameters**
>>> - **localpath** – the local file or directory
>>>
>>> - **remotepath** – the remote path
>>>
>>> - **filter** – a predicate that accepts the filename and determines whether it should be downloaded; None means any file
>>>
>>> - **chunk_size** – the IO chunk size

rpyc.utils.classic.**upload_package**(*conn*, *module*, *remotepath=None*, *chunk_size=64000*)

> uploads a module or a package to the remote party
>
>> **Parameters**
>>> - **conn** – the RPyC connection to use

- **module** – the local module/package object to upload
- **remotepath** – the remote path (if `None`, will default to the remote system's python library (as reported by `distutils`)
- **chunk_size** – the IO chunk size

Note: `upload_module` is just an alias to `upload_package`

---

example:

```
import foo.bar
...
rpyc.classic.upload_package(conn, foo.bar)
```

rpyc.utils.classic.**upload_module**(*conn*, *module*, *remotepath=None*, *chunk_size=64000*)

uploads a module or a package to the remote party

Parameters

- **conn** – the RPyC connection to use
- **module** – the local module/package object to upload
- **remotepath** – the remote path (if `None`, will default to the remote system's python library (as reported by `distutils`)
- **chunk_size** – the IO chunk size

Note: `upload_module` is just an alias to `upload_package`

---

example:

```
import foo.bar
...
rpyc.classic.upload_package(conn, foo.bar)
```

rpyc.utils.classic.**obtain**(*proxy*)

obtains (copies) a remote object from a proxy object. the object is `pickled` on the remote side and `unpickled` locally, thus moved **by value**. changes made to the local object will not reflect remotely.

Parameters

proxy – an RPyC proxy object

Note: the remote object to must be `pickle`-able

Returns

a copy of the remote object

rpyc.utils.classic.**deliver**(*conn*, *localobj*)

delivers (recreates) a local object on the other party. the object is `pickled` locally and `unpickled` on the remote side, thus moved **by value**. changes made to the remote object will not reflect locally.

Parameters

- **conn** – the RPyC connection

- **localobj** – the local object to deliver

Note: the object must be `picklable`

> **Returns**
>   a proxy to the remote object

rpyc.utils.classic.**redirected_stdio**(*conn*)

> Redirects the other party's `stdin`, `stdout` and `stderr` to those of the local party, so remote IO will occur locally.
>
> Example usage:

```
with redirected_stdio(conn):
    conn.modules.sys.stdout.write("hello\n")   # will be printed locally
```

rpyc.utils.classic.**pm**(*conn*)

> same as `pdb.pm()` but on a remote exception
>
> > **Parameters**
> >   **conn** – the RPyC connection

rpyc.utils.classic.**interact**(*conn*, *namespace=None*)

> remote interactive interpreter
>
> > **Parameters**
>
> > - **conn** – the RPyC connection
>
> > - **namespace** – the namespace to use (a `dict`)

class rpyc.utils.classic.**MockClassicConnection**

> Mock classic RPyC connection object. Useful when you want the same code to run remotely or locally.

rpyc.utils.classic.**teleport_function**(*conn*, *func*, *globals=None*, *def_=True*)

> "Teleports" a function (including nested functions/closures) over the RPyC connection. The function is passed in bytecode form and reconstructed on the other side.
>
> The function cannot have non-brinable defaults (e.g., `def f(x, y=[8]):`, since a `list` isn't brinable), or make use of non-builtin globals (like modules). You can overcome the second restriction by moving the necessary imports into the function body, e.g.

```
def f(x, y):
    import os
    return (os.getpid() + y) * x
```

Note: While it is not forbidden to "teleport" functions across different Python versions, it *may* result in errors due to Python bytecode differences. It is recommended to ensure both the client and the server are of the same Python version when using this function.

> > **Parameters**
>
> > - **conn** – the RPyC connection
>
> > - **func** – the function object to be delivered to the other party

---

Helpers

Helpers and wrappers for common RPyC tasks

rpyc.utils.helpers.**buffiter**(*obj*, *chunk=10*, *max_chunk=1000*, *factor=2*)

> Buffered iterator - reads the remote iterator in chunks starting with *chunk*, multiplying the chunk size by *factor* every time, as an exponential-backoff, up to a chunk of *max_chunk* size.
>
> `buffiter` is very useful for tight loops, where you fetch an element from the other side with every iterator. Instead of being limited by the network's latency after every iteration, `buffiter` fetches a "chunk" of elements every time, reducing the amount of network I/Os.
>
> > **Parameters**
> >
> > - **obj** – An iterable object (supports `iter()`)
> >
> > - **chunk** – the initial chunk size
> >
> > - **max_chunk** – the maximal chunk size
> >
> > - **factor** – the factor by which to multiply the chunk size after every iterator (up to *max_chunk*). Must be >= 1.
> >
> > **Returns**
> >
> > an iterator
>
> Example:

```
cursor = db.get_cursor()
for id, name, dob in buffiter(cursor.select("Id", "Name", "DoB")):
    print id, name, dob
```

rpyc.utils.helpers.**restricted**(*obj*, *attrs*, *wattrs=None*)

> Returns a 'restricted' version of an object, i.e., allowing access only to a subset of its attributes. This is useful when returning a "broad" or "dangerous" object, where you don't want the other party to have access to all of its attributes.
>
> New in version 3.2.
>
> > **Parameters**
> >
> > - **obj** – any object
> >
> > - **attrs** – the set of attributes exposed for reading (`getattr`) or writing (`setattr`). The same set will serve both for reading and writing, unless wattrs is explicitly given.
> >
> > - **wattrs** – the set of attributes exposed for writing (`setattr`). If `None`, `wattrs` will default to `attrs`. To disable setting attributes completely, set to an empty tuple `()`.
> >
> > **Returns**
> >
> > a restricted view of the object
>
> Example:

```
class MyService(rpyc.Service):
    def exposed_open(self, filename):
        f = open(filename, "r")
        return rpyc.restricted(f, {"read", "close"})   # disallow access to `seek`
→or `write`
```

rpyc.utils.helpers.**async_**(*proxy*)

>   Creates an async proxy wrapper over an existing proxy. Async proxies are cached. Invoking an async proxy will return an AsyncResult instead of blocking

class rpyc.utils.helpers.**timed**(*proxy*, *timeout*)

>   Creates a timed asynchronous proxy. Invoking the timed proxy will run in the background and will raise an `rpyc.core.async_.AsyncResultTimeout` exception if the computation does not terminate within the given time frame
>
>>   **Parameters**
>>
>>>   • **proxy** – any **callable** RPyC proxy
>>>
>>>   • **timeout** – the maximal number of seconds to allow the operation to run
>>
>>   **Returns**
>>>   a `timed` wrapped proxy
>
>   Example:

```
t_sleep = rpyc.timed(conn.modules.time.sleep, 6) # allow up to 6 seconds
t_sleep(4) # okay
t_sleep(8) # will time out and raise AsyncResultTimeout
```

class rpyc.utils.helpers.**BgServingThread**(*conn*, *callback=None*, *serve_interval=0.0*, *sleep_interval=0.1*)

>   Runs an RPyC server in the background to serve all requests and replies that arrive on the given RPyC connection. The thread is started upon the the instantiation of the `BgServingThread` object; you can use the *stop()* method to stop the server thread.
>
>   CAVEAT: RPyC defaults to bind_threads as False. So, there is no guarantee that the background thread will serve the request. See issue #522 for an example of this behavior. As the bind_threads feature matures, we may change the default to to True in the future.
>
>   Example:

```
conn = rpyc.connect(...)
bg_server = BgServingThread(conn)
...
bg_server.stop()
```

>   **Note:** For a more detailed explanation of asynchronous operation and the role of the `BgServingThread`, see *Part 5: Asynchronous Operation and Events*

>   **stop**()
>
>>   stop the server thread. once stopped, it cannot be resumed. you will have to create a new BgServingThread object later.

rpyc.utils.helpers.**classpartial**(**args*, ***kwargs*)

>   Bind arguments to a class's __init__.

rpyc.utils.helpers.**async**(*proxy*)

>   Creates an async proxy wrapper over an existing proxy. Async proxies are cached. Invoking an async proxy will return an AsyncResult instead of blocking

• *Factories* - general-purpose connection factories (over pipes, sockets, SSL, SSH, TLSlite, etc.)

• *Classic* - *Classic-mode* factories and utilities

- *Helpers* - Various helpers (`timed`, `async_`, `buffiter`, `BgServingThread`, etc.)

4.5.6 Misc

Zero-Deploy RPyC

New in version 3.3.

Requires [plumbum](http://plumbum.readthedocs.org/)

class `rpyc.utils.zerodeploy.`**`DeployedServer`**(*remote_machine,*
server_class='rpyc.utils.server.ThreadedServer',
extra_setup='', python_executable=None)

Sets up a temporary, short-lived RPyC deployment on the given remote machine. It will:

1. Create a temporary directory on the remote machine and copy RPyC's code from the local machine to the remote temporary directory.

2. Start an RPyC server on the remote machine, binding to an arbitrary TCP port, allowing only in-bound connections (`localhost` connections). The server reports the chosen port over `stdout`.

3. An SSH tunnel is created from an arbitrary local port (on the local host), to the remote machine's chosen port. This tunnel is authenticated and encrypted.

4. You get a `DeployedServer` object that can be used to connect to the newly-spawned server.

5. When the deployment is closed, the SSH tunnel is torn down, the remote server terminates and the temporary directory is deleted.

   Parameters

   - **`remote_machine`** – a plumbum `SshMachine` or `ParamikoMachine` instance, representing an SSH connection to the desired remote machine

   - **`server_class`** – the server to create (e.g., `"ThreadedServer"`, `"ForkingServer"`)

   - **`extra_setup`** – any extra code to add to the script

    connect(*service=<class 'rpyc.core.service.VoidService'>, config={}*)

        Same as *connect()*, but with the `host` and `port` parameters fixed

    classic_connect()

        Same as *classic.connect*, but with the `host` and `port` parameters fixed

class `rpyc.utils.zerodeploy.`**`MultiServerDeployment`**(*remote_machines,*
server_class='rpyc.utils.server.ThreadedServer')

An 'aggregate' server deployment to multiple SSH machine. It deploys RPyC to each machine separately, but lets you manage them as a single deployment.

    connect_all(*service=<class 'rpyc.core.service.VoidService'>, config={}*)

        connects to all deployed servers; returns a list of connections (order guaranteed)

    classic_connect_all()

        connects to all deployed servers using classic_connect; returns a list of connections (order guaranteed)

- *Zero-Deploy RPyC* - Deploy short-living RPyC servers on remote machines with ease - all you'll need is SSH access and a Python interpreter installed on the host

4.6 License

RPyC is released under the *MIT license*:

```
Copyright (c) 2005-2013
  Tomer Filiba (tomerfiliba@gmail.com)
  Copyrights of patches are held by their respective submitters

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
THE SOFTWARE.
```

4.7 Release Change Log

4.8 6.0.0

Date: 2024-02-23

- #551 Resolves security issue that results in RCE. The fix breaks backwards compatibility for those that rely on the *__array__* attribute used by *numpy*. This RCE is only exploitable when the server-side gets the attribute *__array__* and calls it (e.g., *np.array(x)*). This issues effects all versions since major release 4.

4.9 5.3.1

Date: 2023-02-21

- #527 Resolved timeout issue that was introduced in 5.2.1

- #525 and #524 Fixed experimental thread binding struct for platforms where unsigned long is 8-bits

  – While the fix for thread binding is not backwards compatible, it only impacts people using an experimental feature. Hence, I did a patch version bump.

4.10 5.3.0

Date: 2022-11-25

- #515 Support for Python 3.11 is available after teleportation bug fix

- #507 Experimental support for threading is added (default is disabled for now)

- #516 Resolved server-side exceptions due to the logic for checking if a name is in *ModuleNamespace*

- #511 Improved documentation on the life-cycle of a netref/proxy-object

4.11 5.2.3

Date: 2022-08-03

- #503 *rpyc_classic.py* and *rpyc_registry.py* can now be resolved without the suffix as well.

4.12 5.2.1

Date: 2022-07-30

- #494 Added support for using decorators to expose methods (see #292)

- #499 Allow *BgServingThread* serve and sleep intervals to be customized

- #498 Avoid redefining *hasattr_static* on every *_check_attr* call

- #489 Updated SSL context usage to avoid deprecated aspects and changes

- #485 Add a configurable timeout on the zero deploy close method

- #484 Fixed *–mode* CLI argument for *rpyc_registry*

- #479 Fixed propagation of *AttributeErrors* raised by exposed descriptors

- #476 Allow filtering by host on list_services

- #493 and #502 Improved documentation and fixed typos

- #492 Some work around race conditions but proper fix is rather involved (see #491)

- 5.2.0 was skipped due to PyPi not allowing file name reuse

4.13 5.1.0

Date: 2022-02-26

- Add *types.MappingProxyType* to *_builtin_types* #470

- Updated documentation #469

- Fixed spradic dealock issues from wait within AsyncResult #463 and #455

- Fixed chained Classic RPyC connections #460

- Added ability to list Registry services #452

- Fixed bug that prevented RPyC from running on systems without SSL #451

- Fixed unexpected behavior with respect to *auto_register* #445
- Fixed propagation of *chunk_size* parameter for *download_dir* #433

4.13.1 5.0.1

Date: 1.11.2021

- Fixed unexpected behavior when using inspect.isfunction #426
- Now prevents installation of RPyC on unsupported versions of python #429

4.13.2 5.0.0

Date: 12.26.2020

Backwards Incompatible

- RPyC 5.0.0 cannot teleport functions to earlier versions
- Deprecated Python 2 support to coincide with it's EOL

Improvements

- Server hostname default supports IPv4 and IPv6 by using the wildcard address #425
- Added `docker/docker-compose.yml` for Python 3.6, 3.7, 3.8, 3.9, and 3.10 containers to improve local work-flow
- Fixed pickle failure on windows for `connect_multiprocess` and `connect_thread` #412
- Fixed teleport function behavior for keyword-only arguments with default #422
- Improved documentation on custom exception handling
- Fixed IPv6 support for server #407
- Added a simple asynchronous service example #400

4.13.3 4.1.5

Date: 4.25.2020

- Fixed mutable object used as kwarg for Server ctor #376
- Corrections to teleport example #374
- Lowered GIL-lock acquires for <64kb within channel sends to address slowness #280

4.13.4 4.1.4

Date: 1.30.2020

- Merged 3.7 and 3.8 teleportatio compat enhancement #371
- Fixed connection hanging due to namepack cursor #369
- Fixed test dependencies and is_py_* for 3.9

4.13.5 4.1.3

Date: 1.25.2020

- Performance improvements: #366 and #351
- Merged fix for propagate_KeyboardInterrupt_locally #364
- Fixed handling of exceptions for request callbacks #365
- Partially fixed return value for netref.__class__ #355

4.13.6 4.1.2

Date: 10.03.2019

- Fixed CVE-2019-16328 which was caused by a missing protocol security check
- Fixed RPyC over RPyC for mutable parameters and extended unit testing for #346

4.13.7 4.1.1

Date: 07.27.2019

- Fixed netref.class_factory id_pack usage per #339 and added test cases
- Name pack casted in _unbox to fix IronPython bug. Fixed #337
- Increased chunk size to improve multi-client response time and throughput of large data #329
- Added warning to _remote_tb when the major version of local and remote mismatch (#332)
- OneShotServer termination was fixed by WilliamBruneau (#343)
- Known issue with 3.8 for CodeType parameters (may drop Python2 support first)

4.13.8 4.1.0

Date: 05.25.2019

- Added connection back-off and attempts for congested workloads
- Fixed minor resource leak for ForkingServer (#304)
- Cross-connection instance check for cached netref classes (#316)
- Hashing fixed (#324)
- New ID Pack convention breaks compatibility between a client/server >= 4.10 with a client/server < 4.10

4.13.9 4.0.2

Date: 04.08.2018

- fix default hostname for ipv6 in rpyc_classic.py (#277)
- fix ThreadPoolServer not working (#283)

4.13.10 4.0.1

Date: 12.06.2018

- fix ValueError during install due to absolute PATH in SOURCES.txt (#276)

4.13.11 4.0.0

Date: 11.06.2018

This release brings a few minor backward incompatibilities, so be sure to read on before upgrading. However, fear not: the ones that are most likely relevant to you have a relatively simple migration path.

Backward Incompatibilities

- `classic.teleport_function` now executes the function in the connection's namespace by default. To get the old behaviour, use `teleport_function(conn, func, conn.modules[func.__module__].__dict__)` instead.
- Changed signature of `Service.on_connect` and `on_disconnect`, adding the connection as argument.
- Changed signature of `Service.__init__`, removing the connection argument
- no longer store connection as `self._conn`. (allows services that serve multiple clients using the same service object, see #198).
- `SlaveService` is now split into two asymmetric classes: `SlaveService` and `MasterService`. The slave exposes functionality to the master but can not anymore access remote objects on the master (#232, #248). If you were previously using `SlaveService`, you may experience problems when feeding the slave with netrefs to objects on the master. In this case, do any of the following:
    - use `ClassicService` (acts exactly like the old `SlaveService`)
    - use `SlaveService` with a `config` that allows attribute access etc
    - use `rpyc.utils.deliver` to feed copies rather than netrefs to the slave
- `RegistryServer.on_service_removed` is once again called whenever a service instance is removed, making it symmetric to `on_service_added` (#238) This reverts PR #173 on issue #172.
- Removed module `rpyc.experimental.splitbrain`. It's too confusing and undocumented for me and I won't be developing it, so better remove it altogether. (It's still available in the `splitbrain` branch)
- Removed module `rpyc.experimental.retunnel`. Seemingly unused anywhere, no documentation, no clue what this is about.
- `bin/rpyc_classic.py` will bind to `127.0.0.1` instead of `0.0.0.0` by default
- `SlaveService` no longer serves exposed attributes (i.e., it now uses `allow_exposed_attrs=False`)
- Exposed attributes no longer hide plain attributes if one otherwise has the required permissions to access the plain attribute. (#165)

What else is new

- teleported functions will now be defined by default in the globals dict

- Can now explicitly specify globals for teleported functions

- Can now use streams as context manager

- keep a hard reference to connection in netrefs, may fix some `EOFError` issues, in particular on Jython related (#237)

- handle synchronous and asynchronous requests uniformly

- fix deadlock with connections talking to each other multithreadedly (#270)

- handle timeouts cumulatively

- fix possible performance bug in `Win32PipeStream.poll` (oversleeping)

- use readthedocs theme for documentation (#269)

- actually time out sync requests (#264)

- clarify documentation concerning exceptions in `Connection.ping` (#265)

- fix `__hash__` for netrefs (#267, #268)

- rename `async` module to `async_` for py37 compatibility (#253)

- fix `deliver()` from IronPython to CPython2 (#251)

- fix brine string handling in py2 IronPython (#251)

- add gevent Server. For now, this requires using `gevent.monkey.patch_all()` before importing for rpyc. Client connections can already be made without further changes to rpyc, just using gevent's monkey patching. (#146)

- add function `rpyc.lib.spawn` to spawn daemon threads

- fix several bugs in `bin/rpycd.py` that crashed this script on startup (#231)

- fix problem with MongoDB, or more generally any remote objects that have a *catch-all* `__getattr__` (#165)

- fix bug when copying remote numpy arrays (#236)

- added `rpyc.utils.helpers.classpartial` to bind arguments to services (#244)

- can now pass services optionally as instance or class (could only pass as class, #244)

- The service is now charged with setting up the connection, doing so in `Service._connect`. This allows using custom protocols by e.g. subclassing `Connection`. More discussions and related features in #239-#247.

- service can now easily override protocol handlers, by updating `conn._HANDLERS` in `_connect` or `on_connect`. For example: `conn._HANDLERS[HANDLE_GETATTR] = self._handle_getattr`.

- most protocol handlers (`Connection._handle_XXX`) now directly get the object rather than its ID as first argument. This makes overriding individual handlers feel much more high-level. And by the way it turns out that this fixes two long-standing issues (#137, #153)

- fix bug with proxying context managers (#228)

- expose server classes from `rpyc` top level module

- fix logger issue on jython

4.13.12 3.4.4

Date: 07.08.2017

- Fix refcount leakage when unboxing from cache ([#196](#))
- Fix TypeError when dispatching exceptions on py2 (unicode)
- Respect `rpyc_protocol_config` for default Service getattr ([#202](#))
- Support unix domain sockets ([#100](#), [#208](#))
- Use first accessible server in `connect_by_service` ([#220](#))
- Fix deadlock problem with logging ([#207](#), [#212](#))
- Fix timeout problem for long commands ([#169](#))

4.13.13 3.4.3

Date: 26.07.2017

- Add missing endpoints config in ThreadPoolServer ([#222](#))
- Fix jython support ([#156](#), [#171](#))
- Improve documentation ([#158](#), [#185](#), [#189](#), [#198](#) and more)

4.13.14 3.4.2

Date: 14.06.2017

- Fix `export_function` on python 3.6

4.13.15 3.4.1

Date: 09.06.2017

- Fix issue high-cpu polling ([#191](#), [#218](#))
- Fix filename argument in logging ([#197](#))
- Improved log messages ([#191](#), [#204](#))
- Drop support for python 3.2 and py 2.5

4.13.16 3.4.0

Date: 29.05.2017

Please excuse the briefity for this versions changelist.

- Add keepalive interface [[#151](#)]
- Various fixes: [#136](#), [#140](#), [#143](#), [#147](#), [#149](#), [#151](#), [#159](#), [#160](#), [#166](#), [#173](#), [#176](#), [#179](#), [#174](#), [#182](#), [#183](#) and others.

4.13.17 3.3.0

- RPyC integrates with plumbum; plumbum is required for some features, like `rpyc_classic.py` and *zero deploy*, but the core of the library doesn't require it. It is, of course, advised to have it installed.

- `SshContext`, `SshTunnel` classes killed in favor of plumbum's SSH tunneling. The interface doesn't change much, except that `ssh_connect` now accept a `plumbum.SshMachine` instance instead of `SshContext`.

- Zero deploy: deploy RPyC to a remote machine over an SSH connection and form an SSH tunnel connected to it, in just one line of code. All you need is SSH access and a Python interpreter installed on the remote machine.

- Dropping Python 2.4 support. RPyC now requires Python 2.5 - 3.3.

- rpycd - a well-behaved daemon for `rpyc_classic.py`, based on python-daemon

- The `OneShotServer` is now exposed by `rpyc_classic -m oneshot`

- `scripts` directory renamed `bin`

- Introducing `Splitbrain Python` - running code on remote machines transparently. Although tested, it is still considered experimental.

- Removing the `BgServerThread` and all polling/timeout hacks in favor of a "global background reactor thread" that handles all incoming transport from all connections. This should solve all threading issues once and for all.

- Added `MockClassicConnection` - a mock RPyC "connection" that allows you to write code that runs either locally or remotely without modification

- Added `teleport_function`

4.13.18 3.2.3

- Fix (issue #76) for real this time

- Fix issue with `BgServingThread` (#89)

- Fix issue with `ThreadPoolServer` (#91)

- Remove RPyC's `excepthook` in favor of chaining the exception's remote tracebacks in the exception class' `__str__` method. This solves numerous issues with logging and debugging.

- Add `OneShotServer`

- Add UNIX domain sockets (#100)

4.13.19 3.2.2

- Windows: make SSH tunnels windowless (#68)

- Fixes a compatibility issue with IronPython on Mono (#72)

- Fixes an issue with introspection when an `AttributeError` is expected (#71)

- The server now logs all exceptions (#73)

- Forking server: call `siginterrupt(False)` in forked child (#76)

- Shutting down the old wikidot site

- Adding Travis CI integration

4.13.20 3.2.1

- Adding missing import (#52)

- Fixing site documentation issue (#54)

- Fixing Python 3 incompatibilities (#58, #59, #60, #61, #66)

- Fixing `slice` issue (#62)

- Added the `endpoints` parameter to the config dict of connection (only on the server side)

4.13.21 3.2.0

- Added support for IPv6 (#28)

- Added SSH tunneling support (`ssh_connect`)

- Added `restricted` object wrapping

- Several fixes to `AsyncResult` and weak references

- Added the `ThreadPoolServer`

- Fixed some minor (harmless) races that caused tracebacks occasionally when server-threads terminated

- Fixes issues #8, #41, #42, #43, #46, and #49.

- Converted all `CRLF` to `LF` (#40)

- Dropped TLSlite integration (#45). We've been dragging this corpse for too long.

- **New documentation** (both the website and docstrings) written in **Sphinx**

    - The site has moved to sourceforge. Wikidot had served us well over the past three years, but they began displaying way too many ads and didn't support uploading files over `rsync`, which made my life hard.

    - New docs are part of the git repository. Updating the site is as easy as `make upload`

- **Python 3.0-3.2** support

4.13.22 3.1.0

What's New

- Supports CPython 2.4-2.7, IronPython, and Jython

- tlslite has been ported to python 2.5-2.7 (the original library targeted 2.3 and 2.4)

- Initial python 3 support – not finished!

- Moves to a more conventional directory structure

- Moves to more standard facilities (`logging`, `nosetests`)

- Solves a major performance issue with the `BgServingThread` (#32), by removing the contention between the two threads that share the connection

- Fixes lots of issues concerning the ForkingServer (#3, #7, and #15)

- Many small bug fixes (#16, #13, #4, etc.)

- Integrates with the built-in `ssl` module for SSL support

- – `rpyc_classic.py` now takes several `--ssl-xxx` switches (see `--help` for more info)
- Fixes typos, running pylint, etc.

Breakage from 3.0.7

- Removing egg builds (we're pure python, and eggs just messed up the build)
- Package layout changed drastically, and some files were renamed
    - – The `servers/` directory was renamed `scripts/`
    - – `classic_server.py` was renamed `rpyc_classic.py`
    - – They scripts now install to your python scripts directory (no longer part of the package), e.g. `C:\python27\`
      `Scripts`
- `rpyc_classic.py` now takes `--register` in order to register, instead of `--dont-register`, which was a silly choice.
- `classic.tls_connect`, `factory.tls_connect` were renamed `tlslite_connect`, to distinguish it from the new `ssl_connect`.

4.13.23 3.0.7

- Moving to **git** as source control
- Build script: more egg formats; register in [pypi](#) ; remove svn; auto-generate `license.py` as well
- Cosmetic touches to `Connection`: separate `serve` into `_recv` and `dispatch`
- Shutdown socket before closing (`SHUT_RDWR`) to prevent `TIME_WAIT` and other problems with various Unixes
- `PipeStream`: use low-level file APIs (`os.read`, `os.write`) to prevent stdio-level buffering that messed up `select`
- `classic_server.py`: open logfile for writing (was opened for reading)
- `registry_server.py`: type of `timeout` is now `int` (was `str`)
- `utils/server.py`: better handling of sockets; fix python 2.4 syntax issue
- `ForkingServer`: re-register `SIGCHLD` handler after handling that signal, to support non-BSD-compliant platforms where after the invocation of the signal handler, the handler is reset

4.13.24 3.0.6

- Handle metaclasses better in `inspect_methods`
- `vinegar.py`: handle old-style-class exceptions better; python 2.4 issues
- `VdbAuthenticator`: when loading files, open for read only; API changes (`from_dict` instead of `from_users`), `from_file` accepts open-mode
- `ForkingServer`: better handling of SIGCHLD

4.13.25 3.0.5

- `setup.py` now also creates egg files

- Slightly improved `servers/vdbconf.py`

- Fixes to `utis/server.py`:

    - The authenticator is now invoked by `_accept_client`, which means it is invoked on the client's context (thread or child process). This solves a problem with the forking server having a TLS authenticator.

    - Changed the forking server to handle `SIGCHLD` instead of using double-fork.

4.13.26 3.0.4

- Fix: `inspect_methods` used `dir` and `getattr` to inspect the given object; this caused a problem with premature activation of properties (as they are activated by `getattr`). Now it inspects the object's type instead, following the MRO by itself, to avoid possible side effects.

4.13.27 3.0.3

- Changed versioning scheme: now 3.0.3 instead of 3.03, and the version tuple is (3, 0, 3)

- Added `servers/vdbconf.py` - a utility to manage verifier databases (used by `tlslite`)

- Added the `--vdb` switch to `classic_server.py`, which invokes a secure server (TLS) with the given VDB file.

4.13.28 3.02

- Authenticators: authenticated servers now store the credentials of the connection in conn._config.credentials

- `Registry`: added UDP and TCP registry servers and clients (`from rpyc.utils.registry import ...`)

- Minor bug fixes

- More tests

- The test-suite now runs under python 2.4 too

4.13.29 3.01

- Fixes some minor issues/bugs

- The registry server can now be instantiated (no longer a singleton) and customized, and RPyC server can be customized to use the different registry.

4.13.30 3.00

Known Issues

- **comparison** - comparing remote and local objects will usually not work, but there's nothing to do about it.

- **64bit platforms**: since channels use 32bit length field, you can't pass data/strings over 4gb. this is not a real limitation (unless you have a super-fast local network and tons of RAM), but as 64bit python becomes the defacto standard, I will upgrade channels to 64bit length field.

- **threads** - in face of no better solution, and after consulting many people, I resorted to setting a timeout on the underlying recv(). This is not an elegant way, but all other solution required rewriting all sorts of threading primitives and were not necessarily deadlock/race-free. as the zen says, "practicality beats purity".

- Windows - pipes supported, but Win32 pipes work like shit

4.13.31 3.00 RC2

Known Issues

- Windows - pipe server doesn't work

PYTHON MODULE INDEX

r

A

B

C