Pyro Documentation

Release 4.73

Irmen de Jong

Jul 07, 2018

Contents

CHAPTER 1

What is Pyro?

It is a library that enables you to build applications in which objects can talk to each other over the network, with minimal programming effort. You can just use normal Python method calls to call objects on other machines. Pyro is a pure Python library and runs on many different platforms and Python versions.

Pyro can be found on Pypi as Pyro4. Source on Github: https://github.com/irmen/Pyro4

CHAPTER 2

Contents

2.1 Intro and Example

This chapter contains a little overview of Pyro's features and a simple example to show how it looks like.

2.1.1 About Pyro: feature overview

Pyro is a library that enables you to build applications in which objects can talk to each other over the network, with minimal programming effort. You can just use normal Python method calls, with almost every possible parameter and return value type, and Pyro takes care of locating the right object on the right computer to execute the method. It is designed to be very easy to use, and to generally stay out of your way. But it also provides a set of powerful features that enables you to build distributed applications rapidly and effortlessly. Pyro is a pure Python library and runs on many different platforms and Python versions.

Here's a quick overview of Pyro's features:

- written in 100% Python so extremely portable, runs on Python 2.7, Python 3.4 and newer, IronPython, Pypy 2 and 3.
- works between different system architectures and operating systems.
- able to communicate between different Python versions transparently.
- defaults to a safe serializer (serpent) that supports many Python data types.

- supports different serializers (serpent, json, marshal, msgpack, pickle, cloudpickle, dill).
- support for all Python data types that are serializable when using the 'pickle', 'cloudpickle' or 'dill' serializers[1].
- can use IPv4, IPv6 and Unix domain sockets.
- optional secure connections via SSL/TLS (encryption, authentication and integrity), including certificate validation on both ends (2-way ssl).
- lightweight client library available for .NET and Java native code ('Pyrolite', provided separately).
- designed to be very easy to use and get out of your way as much as possible, but still provide a lot of flexibility when you do need it.
- name server that keeps track of your object's actual locations so you can move them around transparently.
- yellow-pages type lookups possible, based on metadata tags on registrations in the name server.
- support for automatic reconnection to servers in case of interruptions.
- automatic proxy-ing of Pyro objects which means you can return references to remote objects just as if it were normal objects.
- one-way invocations for enhanced performance.
- batched invocations for greatly enhanced performance of many calls on the same object.
- remote iterator on-demand item streaming avoids having to create large collections upfront and transfer them as a whole.
- you can define timeouts on network communications to prevent a call blocking forever if there's something wrong.
- asynchronous invocations if you want to get the results 'at some later moment in time'. Pyro will take care of gathering the result values in the background.
- remote exceptions will be raised in the caller, as if they were local. You can extract detailed remote traceback information.
- http gateway available for clients wanting to use http+json (such as browser scripts).
- stable network communication code that works reliably on many platforms.
- can hook onto existing sockets created for instance with socketpair() to communicate efficiently between threads or sub-processes.
- possibility to use Pyro's own event loop, or integrate it into your own (or third party) event loop.
- three different possible instance modes for your remote objects (singleton, one per session, one per call).
- many simple examples included to show various features and techniques.
- large amount of unit tests and high test coverage.
- reliable and established: built upon more than 15 years of existing Pyro history, with ongoing support and development.

[1] When configured to use the `pickle`, `cloudpickle` or `dill` serializer, your system may be vulnerable because of the security risks of these serialization protocols (possibility of arbitrary code execution). Pyro does have some security measures in place to mitigate this risk somewhat. They are described in the *Security* chapter. It is strongly advised to read it. By default, Pyro is configured to use the safe *serpent* serializer, so you won't have to deal with these issues unless you configure it explicitly to use one of the other serializers.

Pyro's history

I started working on the first Pyro version in 1998, when remote method invocation technology such as Java's RMI and CORBA were quite popular. I wanted something like that in Python and there was nothing available, so I decided to write my own. Over the years it slowly gained features till it reached version 3.10 or so. At that point it was clear that the code base had become quite ancient and couldn't reliably support any new features, so Pyro4 was born in early 2010, written from scratch.

`Pyro` is the package name of the old and no longer supported 3.x version of Pyro. `Pyro4` is the package name of the current version. Its concepts are similar to Pyro 3.x but it is not backwards compatible. To avoid conflicts, this version has a different package name.

If you're somehow still interested in the old version, here is its git repo and it is also still available on PyPi – but don't use it unless you really have to.

2.1.2 What can you use Pyro for?

Essentially, Pyro can be used to distribute and integrate various kinds of resources or responsibilities: computational (hardware) resources (cpu, storage, printers), informational resources (data, privileged information) and business logic (departments, domains).

An example would be a high performance compute cluster with a large storage system attached to it. Usually this is not accessible directly, rather, smaller systems connect to it and feed it with jobs that need to run on the big cluster. Later, they collect the results. Pyro could be used to expose the available resources on the cluster to other computers. Their client software connects to the cluster and calls the Python program there to perform its heavy duty work, and collect the results (either directly from a method call return value, or perhaps via asynchronous callbacks).

Remote controlling resources or other programs is a nice application as well. For instance, you could write a simple remote controller for your media server that is running on a machine somewhere in a closet. A simple remote control client program could be used to instruct the media server to play music, switch playlists, etc.

Another example is the use of Pyro to implement a form of privilege separation. There is a small component running with higher privileges, but just able to execute the few tasks (and nothing else) that require those higher privileges. That component could expose one or more Pyro objects that represent the privileged information or logic. Other programs running with normal privileges can talk to those Pyro objects to perform those specific tasks with higher privileges in a controlled manner.

Finally, Pyro can be a communication glue library to easily integrate various pars of a heterogeneous system, consisting of many different parts and pieces. As long as you have a working (and supported) Python version running on it, you should be able to talk to it using Pyro from any other part of the system.

Have a look at the `examples` directory in the source archive, perhaps one of the many example programs in there gives even more inspiration of possibilities.

2.1.3 Simple Example

This example will show you in a nutshell what it's like to use Pyro in your programs. A much more extensive introduction is found in the *Tutorial*. Here, we're making a simple greeting service that will return a personalized greeting message to its callers. First let's see the server code:

```
# saved as greeting-server.py
import Pyro4

@Pyro4.expose
class GreetingMaker(object):
    def get_fortune(self, name):
```

(continues on next page)

(continued from previous page)

```
        return "Hello, {0}. Here is your fortune message:\n" \
               "Behold the warranty -- the bold print giveth and the fine print␣
↪taketh away.".format(name)

daemon = Pyro4.Daemon()                # make a Pyro daemon
uri = daemon.register(GreetingMaker)   # register the greeting maker as a Pyro object

print("Ready. Object uri =", uri)      # print the uri so we can use it in the client␣
↪later
daemon.requestLoop()                   # start the event loop of the server to wait␣
↪for calls
```

Open a console window and start the greeting server:

```
$ python greeting-server.py
Ready. Object uri = PYRO:obj_edb9e53007ce4713b371d0dc6a177955@localhost:51681
```

Great, our server is running. Let's see the client code that invokes the server:

```
# saved as greeting-client.py
import Pyro4

uri = input("What is the Pyro uri of the greeting object? ").strip()
name = input("What is your name? ").strip()

greeting_maker = Pyro4.Proxy(uri)          # get a Pyro proxy to the greeting object
print(greeting_maker.get_fortune(name))    # call method normally
```

Start this client program (from a different console window):

```
$ python greeting-client.py
What is the Pyro uri of the greeting object?  <<paste the uri that the server printed␣
↪earlier>>
What is your name?  <<type your name; in my case: Irmen>>
Hello, Irmen. Here is your fortune message:
Behold the warranty -- the bold print giveth and the fine print taketh away.
```

As you can see the client code called the greeting maker that was running in the server elsewhere, and printed the resulting greeting string.

With a name server

While the example above works, it could become tiresome to work with object uris like that. There's already a big issue, *how is the client supposed to get the uri, if we're not copy-pasting it?* Thankfully Pyro provides a *name server* that works like an automatic phone book. You can name your objects using logical names and use the name server to search for the corresponding uri.

We'll have to modify a few lines in `greeting-server.py` to make it register the object in the name server:

```
# saved as greeting-server.py
import Pyro4

@Pyro4.expose
class GreetingMaker(object):
    def get_fortune(self, name):
```

(continues on next page)

(continued from previous page)

```
        return "Hello, {0}. Here is your fortune message:\n" \
               "Tomorrow's lucky number is 12345678.".format(name)

daemon = Pyro4.Daemon()                # make a Pyro daemon
ns = Pyro4.locateNS()                  # find the name server
uri = daemon.register(GreetingMaker)   # register the greeting maker as a Pyro object
ns.register("example.greeting", uri)   # register the object with a name in the name␣
↪server

print("Ready.")
daemon.requestLoop()                   # start the event loop of the server to wait␣
↪for calls
```

The `greeting-client.py` is actually simpler now because we can use the name server to find the object:

```
# saved as greeting-client.py
import Pyro4

name = input("What is your name? ").strip()

greeting_maker = Pyro4.Proxy("PYRONAME:example.greeting")     # use name server object␣
↪lookup uri shortcut
print(greeting_maker.get_fortune(name))
```

The program now needs a Pyro name server that is running. You can start one by typing the following command: **`python -m Pyro4.naming`** (or simply: **`pyro4-ns`**) in a separate console window (usually there is just *one* name server running in your network). After that, start the server and client as before. There's no need to copy-paste the object uri in the client any longer, it will 'discover' the server automatically, based on the object name (`example.greeting`). If you want you can check that this name is indeed known in the name server, by typing the command **`python -m Pyro4.nsc list`** (or simply: **`pyro4-nsc list`**), which will produce:

```
$ pyro4-nsc list
--------START LIST
Pyro.NameServer --> PYRO:Pyro.NameServer@localhost:9090
example.greeting --> PYRO:obj_663a31d2dde54b00bfe52ec2557d4f4f@localhost:51707
--------END LIST
```

(Once again the uri for our object will be random) This concludes this simple Pyro example.

Note: In the source archive there is a directory `examples` that contains a truckload of example programs that show the various features of Pyro. If you're interested in them (it is highly recommended to be so!) you will have to download the Pyro distribution archive. Installing Pyro only provides the library modules. For more information, see *Configuring Pyro*.

Other means of creating connections

The example above showed two of the basic ways to set up connections between your client and server code. There are various other options, have a look at the client code details: *Object discovery* and the server code details: *Pyro Daemon: publishing Pyro objects*. The use of the name server is optional, see *Name Server* for details.

2.1.4 Performance

Pyro4 is pretty fast. On a typical networked system you can expect:

- a few hundred new proxy connections per second to one server
- similarly, a few hundred initial remote calls per second to one server
- a few thousand remote method calls per second on a single proxy
- tens of thousands batched or oneway remote calls per second
- 10-100 Mb/sec data transfer

Results do vary depending on many factors such as:

- network speed
- machine and operating system
- I/O or CPU bound workload
- contents and size of the pyro call request and response messages
- the serializer being used
- python version being used

Experiment with the `benchmark`, `batchedcalls` and `hugetransfer` examples to see what results you get on your own setup.

2.2 Installing Pyro

This chapter will show how to obtain and install Pyro.

2.2.1 Compatibility

Pyro is written in 100% Python. It works on any recent operating system where a suitable supported Python implementation is available (2.7, or 3.4 and newer). It also works with Pypy (2 and 3) and IronPython. It will probably not work with Jython 2.7 at this time of writing. If you need this, try Pyro version 4.34 or older instead. (if you only need to write *client* code in Jython/Java, consider using *Pyrolite - client library for Java and .NET* instead!)

Note: When Pyro is configured to use pickle, cloudpickle, dill or marshal as its serialization format, it is required to have the same *major* Python versions on your clients and your servers. Otherwise the different parties cannot decipher each others serialized data. This means you cannot let Python 2.x talk to Python 3.x with Pyro, when using those serializers. However it should be fine to have Python 3.5 talk to Python 3.6 for instance. The other protocols (serpent, json) don't have this limitation!

2.2.2 Obtaining and installing Pyro

Debian Linux (or Debian derived distributions) You can install via the package manager: `apt install python3-pyro4` (for Python 3.x) or `apt install python2-pyro4` (for Python 2.x). Please pay attention to the packaged Pyro4 version, it can be quite old if you're not getting the package from the testing or unstable repositories.

Anaconda Anaconda users can install the Pyro4 package from conda-forge using `conda install -c conda-forge pyro4`

Pip `pip install Pyro4` should do the trick. Pyro is available here on pypi.

Manual installation Download the source distribution archive (Pyro4-X.YZ.tar.gz) from Pypi or Github, extract and `python setup.py install`. The serpent serialization library must also be installed. If you're using a version of Python older than 3.4, the selectors2 or selectors34 backported module must also be installed to be able to use the multiplex server type.

Attention: When using Python 3.4 or older it is better to install `selectors2` instead of `selectors34`. Pyro4's package requirements only refer to `selectors34` but Pyro will use `selectors2` first if it detects it. (reason: selectors2 deals with interrupted system calls better. Python 3.5 and newer already have this built-in.)

Github Source is on Github: https://github.com/irmen/Pyro4 The required serpent serializer library is there as well: https://github.com/irmen/Serpent

2.2.3 Third party libraries that Pyro4 uses

serpent - required, 1.24 or newer Should be installed automatically when you install Pyro4.

selectors34 - required on Python 3.3 or older Should be installed automatically when you install Pyro4.

selectors2 - optional on Python 3.4 or older Install this if you want better behavior for interrupted system calls on Python 3.4 or older.

dill - optional, 0.2.6 or newer Install this if you want to use the dill serializer.

cloudpickle - optional, 0.4.0 or newer Install this if you want to use the cloudpickle serializer.

msgpack - optional, 0.5.2 or newer Install this if you want to use the msgpack serializer.

2.2.4 Stuff you get extra in the source distribution archive and not with packaged versions

If you decide to download the distribution (.tar.gz) you have a bunch of extras over simply installing the Pyro library directly. It contains:

docs/ the Sphinx/RST sources for this manual

examples/ dozens of examples that demonstrate various Pyro features (highly recommended to examine these, many paragraphs in this manual refer to relevant examples here)

tests/ the unittest suite that checks for correctness and regressions

src/ The actual Pyro4 library's source code (only this part is installed if you install the `Pyro4` package)

and a couple of other files: a setup script and other miscellaneous files such as the license (see *Software License and Disclaimer*).

If you don't want to download anything, you can view all of this online on Github.

2.3 Tutorial

This tutorial will explain a couple of basic Pyro concepts, a little bit about the name server, and you'll learn to write a simple Pyro application. You'll do this by writing a warehouse system and a stock market simulator, that demonstrate

some key Pyro techniques.

2.3.1 Warm-up

Before proceeding, you should install Pyro if you haven't done so. For instructions about that, see *Installing Pyro*.

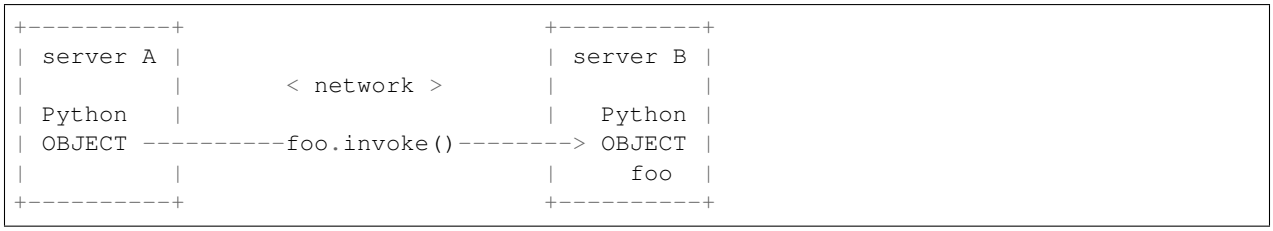
In this tutorial, you will use Pyro's default configuration settings, so once Pyro is installed, you're all set! All you need is a text editor and a couple of console windows. During the tutorial, you are supposed to run everything on a single machine. This avoids initial networking complexity.

Note: For security reasons, Pyro runs stuff on localhost by default. If you want to access things from different machines, you'll have to tell Pyro to do that explicitly. At the end is a small section *phase 3: running it on different machines* that tells you how you can run the various components on different machines.

Note: The code of the two tutorial 'projects' is included in the Pyro source archive. Just installing Pyro won't provide this. If you don't want to type all the code, you should extract the Pyro source archive (`Pyro4-X.Y.tar.gz`) somewhere. You will then have an `examples` directory that contains a truckload of examples, including the two tutorial projects we will be creating later in this tutorial, `warehouse` and `stockquotes`. (There is more in there as well: the `tests` directory contains the test suite with all the unittests for Pyro's code base.)

2.3.2 Pyro concepts and tools

Pyro enables code to call methods on objects even if that object is running on a remote machine:



Pyro is mainly used as a library in your code but it also has several supporting command line tools[1]. We won't explain every one of them here as you will only need the "name server" for this tutorial.

Key concepts

Here are a couple of key concepts you encounter when using Pyro:

Proxy A proxy is a substitute object for "the real thing". It intercepts the method calls you would normally do on an object as if it was the actual object. Pyro then performs some magic to transfer the call to the computer that contains the *real* object, where the actual method call is done, and the results are returned to the caller. This means the calling code doesn't have to know if it's dealing with a normal or a remote object, because the code is identical. The class implementing Pyro proxies is `Pyro4.Proxy` (shortcut for *`Pyro4.core.Proxy`*)

URI (Unique resource identifier) This is what Pyro uses to identify every object. (similar to what a web page URL is to point to the different documents on the web). Its string form is like this: "PYRO:" + object name + "@" + server name + port number. There are a few other forms it can take as well. You can write the protocol in

[1] Actually there are no scripts or command files included with Pyro right now. The *Command line tools* are invoked by starting their package directly using the `-m` argument of the Python interpreter.

lowercase too if you want ("pyro:") but it will automatically be converted to uppercase internally. The class implementing Pyro uris is `Pyro4.URI` (shortcut for *Pyro4.core.URI*)

Pyro object This is a normal Python object but it is registered with Pyro so that you can access it remotely. Pyro objects are written just as any other object but the fact that Pyro knows something about them makes them special, in the way that you can call methods on them from other programs. A class can also be a Pyro object, but then you will also have to tell Pyro about how it should create actual objects from that class when handling remote calls.

Pyro daemon (server) This is the part of Pyro that listens for remote method calls, dispatches them to the appropriate actual objects, and returns the results to the caller. All Pyro objects are registered in one or more daemons.

Pyro name server The name server is a utility that provides a phone book for Pyro applications: you use it to look up a "number" by a "name". The name in Pyro's case is the logical name of a remote object. The number is the exact location where Pyro can contact the object. Usually there is just *one* name server running in your network.

Serialization This is the process of transforming objects into streams of bytes that can be transported over the network. The receiver deserializes them back into actual objects. Pyro needs to do this with all the data that is passed as arguments to remote method calls, and their response data. Not all objects can be serialized, so it is possible that passing a certain object to Pyro won't work even though a normal method call would accept it just fine.

Configuration Pyro can be configured in a lot of ways. Using environment variables (they're prefixed with `PYRO_`) or by setting config items in your code. See the configuration chapter for more details. The default configuration should be ok for most situations though, so you many never have to touch any of these options at all!

Starting a name server

While the use of the Pyro name server is optional, we will use it in this tutorial. It also shows a few basic Pyro concepts, so let us begin by explaining a little about it. Open a console window and execute the following command to start a name server:

`python -m Pyro4.naming` (or simply: **`pyro4-ns`**)

The name server will start and it prints something like:

```
Not starting broadcast server for localhost.
NS running on localhost:9090 (127.0.0.1)
URI = PYRO:Pyro.NameServer@localhost:9090
```

Localhost

By default, Pyro uses *localhost* to run stuff on, so you can't by mistake expose your system to the outside world. You'll need to tell Pyro explicitly to use something else than *localhost*. But it is fine for the tutorial, so we leave it as it is.

The name server has started and is listening on *localhost port 9090*.

It also printed an URI. Remember that this is what Pyro uses to identify every object.

The name server can be stopped with a `control-c`, or on Windows, with `ctrl-break`. But let it run in the background for the rest of this tutorial.

Interacting with the name server

There's another command line tool that let you interact with the name server: "nsc" (name server control tool). You can use it, amongst other things, to see what all known registered objects in the naming server are. Let's do that right now. Type:

`python -m Pyro4.nsc list` (or simply: **`pyro4-nsc list`**)

and it will print something like this:

```
--------START LIST
Pyro.NameServer --> PYRO:Pyro.NameServer@localhost:9090
--------END LIST
```

The only object that is currently registered, is the name server itself! (Yes, the name server is a Pyro object itself. Pyro and the "nsc" tool are using Pyro to talk to it).

Note: As you can see, the name `Pyro.NameServer` is registered to point to the URI that we saw earlier. This is mainly for completeness sake, and is not often used, because there are different ways to get to talk to the name server (see below).

The NameServer object

The name server itself is a normal Pyro object which means the 'nsc' tool, and any other code that talks to it, is just using normal Pyro methods. The only "trickery" that makes it a bit different from other Pyro servers is perhaps the broadcast responder, and the two command line tools to interact with it (`Pyro4.naming` and `Pyro4.nsc`)

This is cool, but there's a little detail left unexplained: *How did the nsc tool know where the name server was?* Pyro has a couple of tactics to locate a name server. The nsc tool uses them too: Pyro uses a network broadcast to see if there's a name server available somewhere (the name server contains a broadcast responder that will respond "Yeah hi I'm here"). So in many cases you won't have to configure anything to be able to discover the name server. If nobody answers though, Pyro tries the configured default or custom location. If still nobody answers it prints a sad message and exits. However if it found the name server, it is then possible to talk to it and get the location of any other registered object. This means that you won't have to hard code any object locations in your code, and that the code is capable of dynamically discovering everything at runtime.

But enough of that. We need to start looking at how to actually write some code ourselves that uses Pyro!

2.3.3 Building a Warehouse

Hint: All code of this part of the tutorial can be found in the `examples/warehouse` directory.

You'll build a simple warehouse that stores items, and that everyone can visit. Visitors can store items and retrieve other items from the warehouse (if they've been stored there).

In this tutorial you'll first write a normal Python program that more or less implements the complete warehouse system, but in vanilla Python code. After that you'll add Pyro support to it, to make it a distributed warehouse system, where you can visit the central warehouse from many different computers.

phase 1: a simple prototype

To start with, write the vanilla Python code for the warehouse and its visitors. This prototype is fully working but everything is running in a single process. It contains no Pyro code at all, but shows what the system is going to look like later on.

The `Warehouse` object simply stores an array of items which we can query, and allows for a person to take an item or to store an item. Here is the code (`warehouse.py`):

```
from __future__ import print_function

class Warehouse(object):
    def __init__(self):
        self.contents = ["chair", "bike", "flashlight", "laptop", "couch"]

    def list_contents(self):
        return self.contents

    def take(self, name, item):
        self.contents.remove(item)
        print("{0} took the {1}.".format(name, item))

    def store(self, name, item):
        self.contents.append(item)
        print("{0} stored the {1}.".format(name, item))
```

Then there is a `Person` that can visit the warehouse. The person has a name and deposit and retrieve actions on a particular warehouse. Here is the code (`person.py`):

```
from __future__ import print_function
import sys

if sys.version_info < (3, 0):
    input = raw_input


class Person(object):
    def __init__(self, name):
        self.name = name

    def visit(self, warehouse):
        print("This is {0}.".format(self.name))
        self.deposit(warehouse)
        self.retrieve(warehouse)
        print("Thank you, come again!")

    def deposit(self, warehouse):
        print("The warehouse contains:", warehouse.list_contents())
        item = input("Type a thing you want to store (or empty): ").strip()
        if item:
            warehouse.store(self.name, item)

    def retrieve(self, warehouse):
        print("The warehouse contains:", warehouse.list_contents())
        item = input("Type something you want to take (or empty): ").strip()
        if item:
            warehouse.take(self.name, item)
```

Finally you need a small script that actually runs the code. It creates the warehouse and two visitors, and makes the

visitors perform their actions in the warehouse. Here is the code (`visit.py`):

```
# This is the code that runs this example.
from warehouse import Warehouse
from person import Person

warehouse = Warehouse()
janet = Person("Janet")
henry = Person("Henry")
janet.visit(warehouse)
henry.visit(warehouse)
```

Run this simple program. It will output something like this:

```
$ python visit.py
This is Janet.
The warehouse contains: ['chair', 'bike', 'flashlight', 'laptop', 'couch']
Type a thing you want to store (or empty): television   # typed in
Janet stored the television.
The warehouse contains: ['chair', 'bike', 'flashlight', 'laptop', 'couch', 'television
↪']
Type something you want to take (or empty): couch    # <-- typed in
Janet took the couch.
Thank you, come again!
This is Henry.
The warehouse contains: ['chair', 'bike', 'flashlight', 'laptop', 'television']
Type a thing you want to store (or empty): bricks   # <-- typed in
Henry stored the bricks.
The warehouse contains: ['chair', 'bike', 'flashlight', 'laptop', 'television',
↪'bricks']
Type something you want to take (or empty): bike   # <-- typed in
Henry took the bike.
Thank you, come again!
```

phase 2: first Pyro version

That wasn't very exciting but you now have working code for the basics of the warehouse system. Now you'll use Pyro to turn the warehouse into a standalone component, that people from other computers can visit. You'll need to add a couple of lines to the `warehouse.py` file so that it will start a Pyro server for the warehouse object. You can do this by registering your Pyro class with a 'Pyro daemon', the server that listens for and processes incoming remote method calls. One way to do that is like this (you can ignore the details about this for now):

```
Pyro4.Daemon.serveSimple(
        {
            Warehouse: "example.warehouse"
        },
        ns = False)
```

Next, we have to tell Pyro what parts of the class should be remotely accessible, and what parts aren't supposed to be accessible. This has to do with security. We'll be adding a `@Pyro4.expose` decorator on the Warehouse class definition to tell Pyro it is allowed to access the class remotely. You can ignore the `@Pyro4.behavior` line we also added for now (but it is required to properly have a persistent warehouse inventory). Finally we add a little `main` function so it will be started correctly, which should make the code now look like this (`warehouse.py`):

```
from __future__ import print_function
import Pyro4
```

(continues on next page)

(continued from previous page)

```
@Pyro4.expose
@Pyro4.behavior(instance_mode="single")
class Warehouse(object):
    def __init__(self):
        self.contents = ["chair", "bike", "flashlight", "laptop", "couch"]

    def list_contents(self):
        return self.contents

    def take(self, name, item):
        self.contents.remove(item)
        print("{0} took the {1}.".format(name, item))

    def store(self, name, item):
        self.contents.append(item)
        print("{0} stored the {1}.".format(name, item))


def main():
    Pyro4.Daemon.serveSimple(
            {
                Warehouse: "example.warehouse"
            },
            ns = False)

if __name__=="__main__":
    main()
```

Start the warehouse in a new console window, it will print something like this:

```
$ python warehouse.py
Object <__main__.Warehouse object at 0x025F4FF0>:
    uri = PYRO:example.warehouse@localhost:51279
Pyro daemon running.
```

It will become clear what you need to do with this output in a second. You now need to slightly change the `visit.py` script that runs the thing. Instead of creating a warehouse directly and letting the persons visit that, it is going to use Pyro to connect to the stand alone warehouse object that you started above. It needs to know the location of the warehouse object before it can connect to it. This is the **uri** that is printed by the warehouse program above (`PYRO:example.warehouse@localhost:51279`). You'll need to ask the user to enter that uri string into the program, and use Pyro to create a *proxy* to the remote object:

```
uri = input("Enter the uri of the warehouse: ").strip()
warehouse = Pyro4.Proxy(uri)
```

That is all you need to change. Pyro will transparently forward the calls you make on the warehouse object to the remote object, and return the results to your code. So the code will now look like this (`visit.py`):

```
# This is the code that visits the warehouse.
import sys
import Pyro4
from person import Person

if sys.version_info<(3,0):
```

(continues on next page)

(continued from previous page)

```
    input = raw_input

uri = input("Enter the uri of the warehouse: ").strip()
warehouse = Pyro4.Proxy(uri)
janet = Person("Janet")
henry = Person("Henry")
janet.visit(warehouse)
henry.visit(warehouse)
```

Notice that the code of `Warehouse` and `Person` classes didn't change *at all*.

Run the program. It will output something like this:

```
$ python visit.py
Enter the uri of the warehouse: PYRO:example.warehouse@localhost:51279  # copied from␣
↪warehouse output
This is Janet.
The warehouse contains: ['chair', 'bike', 'flashlight', 'laptop', 'couch']
Type a thing you want to store (or empty): television   # typed in
The warehouse contains: ['chair', 'bike', 'flashlight', 'laptop', 'couch', 'television
↪']
Type something you want to take (or empty): couch   # <-- typed in
Thank you, come again!
This is Henry.
The warehouse contains: ['chair', 'bike', 'flashlight', 'laptop', 'television']
Type a thing you want to store (or empty): bricks   # <-- typed in
The warehouse contains: ['chair', 'bike', 'flashlight', 'laptop', 'television',
↪'bricks']
Type something you want to take (or empty): bike    # <-- typed in
Thank you, come again!
```

And notice that in the other console window, where the warehouse server is running, the following is printed:

```
Janet stored the television.
Janet took the couch.
Henry stored the bricks.
Henry took the bike.
```

phase 3: final Pyro version

The code from the previous phase works fine and could be considered to be the final program, but is a bit cumbersome because you need to copy-paste the warehouse URI all the time to be able to use it. You will simplify it a bit in this phase by using the Pyro name server. Also, you will use the Pyro excepthook to print a nicer exception message if anything goes wrong (by taking something from the warehouse that is not present! Try that now with the code from phase 2. You will get a `ValueError: list.remove(x): x not in list` but with a not so useful stack trace).

Note: Once again you can leave code of the `Warehouse` and `Person` classes **unchanged**. As you can see, Pyro is not getting in your way at all here. You can often use it with only adding a couple of lines to your existing code.

Okay, stop the warehouse program from phase 2 if it is still running, and check if the name server that you started in *Starting a name server* is still running in its own console window.

In `warehouse.py` locate the statement `Pyro4.Daemon.serveSimple(...` and change the `ns = False`

argument to `ns = True`. This tells Pyro to use a name server to register the objects in. (The `Pyro4.Daemon.serveSimple` is a very easy way to start a Pyro server but it provides very little control. Look here *Oneliner Pyro object publishing: serveSimple()* for some more details, and you will learn about another way of starting a server in *Building a Stock market simulator*).

In `visit.py` remove the input statement that asks for the warehouse uri, and change the way the warehouse proxy is created. Because you are now using a name server you can ask Pyro to locate the warehouse object automatically:

```
warehouse = Pyro4.Proxy("PYRONAME:example.warehouse")
```

Finally, install the `Pyro4.util.excepthook` as excepthook. You'll soon see what this does to the exceptions and stack traces your program produces when something goes wrong with a Pyro object. So the code should look something like this (`visit.py`):

```
# This is the code that visits the warehouse.
import sys
import Pyro4
import Pyro4.util
from person import Person

sys.excepthook = Pyro4.util.excepthook

warehouse = Pyro4.Proxy("PYRONAME:example.warehouse")
janet = Person("Janet")
henry = Person("Henry")
janet.visit(warehouse)
henry.visit(warehouse)
```

Start the warehouse program again in a separate console window. It will print something like this:

```
$ python warehouse.py
Object <__main__.Warehouse object at 0x02496050>:
    uri = PYRO:obj_426e82eea7534fb5bc78df0b5c0b6a04@localhost:51294
    name = example.warehouse
Pyro daemon running.
```

As you can see the uri is different this time, it now contains some random id code instead of a name. However it also printed an object name. This is the name that is now used in the name server for your warehouse object. Check this with the 'nsc' tool: **`python -m Pyro4.nsc list`** (or simply: **`pyro4-nsc list`**), which will print something like:

```
--------START LIST
Pyro.NameServer --> PYRO:Pyro.NameServer@localhost:9090
example.warehouse --> PYRO:obj_426e82eea7534fb5bc78df0b5c0b6a04@localhost:51294
--------END LIST
```

This means you can now refer to that warehouse object using the name `example.warehouse` and Pyro will locate the correct object for you automatically. This is what you changed in the `visit.py` code so run that now to see that it indeed works!

Remote exception: You also installed Pyro's custom excepthook so try that out. Run the `visit.py` script and try to take something from the warehouse that is not present (for instance, batteries):

```
Type something you want to take (or empty): batteries
Traceback (most recent call last):
  File "visit.py", line 12, in <module>
    janet.visit(warehouse)
```

(continues on next page)

(continued from previous page)

```
  File "d:\PROJECTS\Pyro4\examples\warehouse\phase3\person.py", line 14, in visit
    self.retrieve(warehouse)
  File "d:\PROJECTS\Pyro4\examples\warehouse\phase3\person.py", line 25, in retrieve
    warehouse.take(self.name, item)
  File "d:\PROJECTS\Pyro4\src\Pyro4\core.py", line 161, in __call__
    return self.__send(self.__name, args, kwargs)
  File "d:\PROJECTS\Pyro4\src\Pyro4\core.py", line 314, in _pyroInvoke
    raise data
ValueError: list.remove(x): x not in list
 +--- This exception occured remotely (Pyro) - Remote traceback:
 | Traceback (most recent call last):
 |   File "d:\PROJECTS\Pyro4\src\Pyro4\core.py", line 824, in handleRequest
 |     data=method(*vargs, **kwargs)   # this is the actual method call to the Pyro
↪object
 |   File "warehouse.py", line 14, in take
 |     self.contents.remove(item)
 | ValueError: list.remove(x): x not in list
 +--- End of remote traceback
```

What you can see now is that you not only get the usual exception traceback, *but also the exception that occurred in the remote warehouse object on the server* (the "remote traceback"). This can greatly help locating problems! As you can see it contains the source code lines from the warehouse code that is running in the server, as opposed to the normal local traceback that only shows the remote method call taking place inside Pyro.

2.3.4 Building a Stock market simulator

Hint: All of the code of this part of the tutorial can be found in the `examples/stockquotes` directory.

simplified example

The tutorial here is a simplified version of a stock quote simulation example. There's a more elaborate example available called `examples/stockquotes-old` (it used to be this example in older versions of the documentation)

We'll build a simple stock quote system. The idea is that we have multiple stock markets producing stock symbol quotes. There are viewers that aggregate and filter all stock quotes from the markets and display those from the companies we are interested in.

Stockmarket 1	→		Viewer
Stockmarket 2	→	→	Viewer
Stockmarket 3	→		Viewer
...			...

phase 1: simple prototype

Again, like the previous application (the warehouse), you first create a working version of the system by only using normal Python code. This simple prototype will be functional but everything will be running in a single process. It contains no Pyro code at all, but shows what the system is going to look like later on.

First create a file `stockmarket.py` that will simulate a stock market that is producing stock quotes for registered companies. For simplicity we will use a generator function that produces individual random stock quotes. The code is as follows:

```
# stockmarket.py
import random
import time


class StockMarket(object):
    def __init__(self, marketname, symbols):
        self.name = marketname
        self.symbols = symbols

    def quotes(self):
        while True:
            symbol = random.choice(self.symbols)
            yield symbol, round(random.uniform(5, 150), 2)
            time.sleep(random.random()/2.0)
```

For the actual viewer application we create a new file `viewer.py` that iterates over the symbols produced by various stock markets. It prints the symbols from the companies we're interested in:

```
# viewer.py
from __future__ import print_function
from stockmarket import StockMarket


class Viewer(object):
    def __init__(self):
        self.markets = set()
        self.symbols = set()

    def start(self):
        print("Shown quotes:", self.symbols)
        quote_sources = {
            market.name: market.quotes() for market in self.markets
        }
        while True:
            for market, quote_source in quote_sources.items():
                quote = next(quote_source)  # get a new stock quote from the source
                symbol, value = quote
                if symbol in self.symbols:
                    print("{0}.{1}: {2}".format(market, symbol, value))


def main():
    nasdaq = StockMarket("NASDAQ", ["AAPL", "CSCO", "MSFT", "GOOG"])
    newyork = StockMarket("NYSE", ["IBM", "HPQ", "BP"])
    viewer = Viewer()
    viewer.markets = {nasdaq, newyork}
    viewer.symbols = {"IBM", "AAPL", "MSFT"}
    viewer.start()


if __name__ == "__main__":
    main()
```

If you run this file `viewer.py` it will print a stream of stock symbol quote updates that are being generated by the two stock markets (but only the few symbols that the viewer wants to see):

```
$ python viewer.py
Shown quotes: {'MSFT', 'IBM', 'AAPL'}
NYSE.IBM: 19.59
NASDAQ.MSFT: 25.06
NYSE.IBM: 89.54
NYSE.IBM: 44.08
NASDAQ.MSFT: 9.73
NYSE.IBM: 80.57
....
```

phase 2: Pyro version

Now you use Pyro to make the various components fully distributed. Pyro is used to make them talk to each other. The actual code for each component class hasn't really changed since phase 1, it is just the plumbing that you need to write to glue them together. Pyro is making this a matter of just a few lines of code that is Pyro-specific, the rest of the code is needed anyway to start up and configure the system. To be able to see the final result, the code is listed once more with comments on what changed with respect to the version in phase 1.

stockmarket

The `stockmarket.py` is changed slightly. You have to add the `@Pyro4.expose` decorator on the methods (or class) that must be accessible remotely. Also, to access the `name` and `symbols` attributes of the class you have to turn them into real Python properties. Finally there is now a bit of startup logic to create some stock markets and make them available as Pyro objects. Notice that we gave each market their own defined name, this will be used in the viewer application later.

For sake of example we are not using the `serveSimple` method here to publish our objects via Pyro. Rather, the daemon and name server are accessed by our own code. Notice that to ensure tidy cleanup of connectoin resources, they are both used as context managers in a `with` statement.

Also notice that we can leave the generator function in the stockmarket class as-is; since version 4.49 Pyro is able to turn it into a remote generator without your client program ever noticing.

The complete code for the Pyro version of `stockmarket.py` is as follows:

```
# stockmarket.py
from __future__ import print_function
import random
import time
import Pyro4


@Pyro4.expose
class StockMarket(object):
    def __init__(self, marketname, symbols):
        self._name = marketname
        self._symbols = symbols

    def quotes(self):
        while True:
            symbol = random.choice(self.symbols)
            yield symbol, round(random.uniform(5, 150), 2)
```

(continues on next page)

(continued from previous page)

```
            time.sleep(random.random()/2.0)

    @property
    def name(self):
        return self._name

    @property
    def symbols(self):
        return self._symbols


if __name__ == "__main__":
    nasdaq = StockMarket("NASDAQ", ["AAPL", "CSCO", "MSFT", "GOOG"])
    newyork = StockMarket("NYSE", ["IBM", "HPQ", "BP"])
    # for example purposes we will access the daemon and name server ourselves and␣
↪not use serveSimple
    with Pyro4.Daemon() as daemon:
        nasdaq_uri = daemon.register(nasdaq)
        newyork_uri = daemon.register(newyork)
        with Pyro4.locateNS() as ns:
            ns.register("example.stockmarket.nasdaq", nasdaq_uri)
            ns.register("example.stockmarket.newyork", newyork_uri)
        print("Stockmarkets available.")
        daemon.requestLoop()
```

viewer

You don't need to change the actual code in the `Viewer`, other than how to tell it what stock market objects it should use. Rather than hard coding the fixed set of stockmarket names, it is more flexible to utilize Pyro's name server and ask that to return all stock markets it knows about. The `Viewer` class itself remains unchanged:

```
# viewer.py
from __future__ import print_function
import Pyro4


class Viewer(object):
    def __init__(self):
        self.markets = set()
        self.symbols = set()

    def start(self):
        print("Shown quotes:", self.symbols)
        quote_sources = {
            market.name: market.quotes() for market in self.markets
        }
        while True:
            for market, quote_source in quote_sources.items():
                quote = next(quote_source)  # get a new stock quote from the source
                symbol, value = quote
                if symbol in self.symbols:
                    print("{0}.{1}: {2}".format(market, symbol, value))


def find_stockmarkets():
```

(continues on next page)

(continued from previous page)

```
    # You can hardcode the stockmarket names for nasdaq and newyork, but it
    # is more flexible if we just look for every available stockmarket.
    markets = []
    with Pyro4.locateNS() as ns:
        for market, market_uri in ns.list(prefix="example.stockmarket.").items():
            print("found market", market)
            markets.append(Pyro4.Proxy(market_uri))
    if not markets:
        raise ValueError("no markets found! (have you started the stock markets first?
↪)")
    return markets


def main():
    viewer = Viewer()
    viewer.markets = find_stockmarkets()
    viewer.symbols = {"IBM", "AAPL", "MSFT"}
    viewer.start()


if __name__ == "__main__":
    main()
```

running the program

To run the final stock quote system you need to do the following:

- open a new console window and start the Pyro name server (**`python -m Pyro4.naming`**, or simply: **`pyro4-ns`**).
- open another console window and start the stock market server
- open another console window and start the viewer

The stock market program doesn't print much by itself but it sends stock quotes to the viewer, which prints them:

```
$ python viewer.py
found market example.stockmarket.newyork
found market example.stockmarket.nasdaq
Shown quotes: {'AAPL', 'IBM', 'MSFT'}
NASDAQ.AAPL: 82.58
NYSE.IBM: 85.22
NYSE.IBM: 124.68
NASDAQ.AAPL: 88.55
NYSE.IBM: 40.97
NASDAQ.MSFT: 38.83
...
```

If you're interested to see what the name server now contains, type **`python -m Pyro4.nsc list`** (or simply: **`pyro4-nsc list`**):

```
$ pyro4-nsc list
--------START LIST
Pyro.NameServer --> PYRO:Pyro.NameServer@localhost:9090
    metadata: ['class:Pyro4.naming.NameServer']
example.stockmarket.nasdaq --> PYRO:obj_
↪3896de2eb38b4bed9d12ba91703539a4@localhost:51479
```

(continues on next page)

(continued from previous page)

```
example.stockmarket.newyork --> PYRO:obj_
↪1ab1a322e5c14f9e984a0065cd080f56@localhost:51479
--------END LIST
```

phase 3: running it on different machines

Before presenting the changes in phase 3, let's introduce some additional notions when working with Pyro.

It's important for you to understand that, for security reasons, Pyro runs stuff on localhost by default. If you want to access things from different machines, you'll have to tell Pyro to do that explicitly. Here we show you how you can do this:

Let's assume that you want to start the *name server* in such a way that it is accessible from other machines. To do that, type in the console one of two options (with an appropriate -n argument):

> $ python -m Pyro4.naming -n your_hostname # i.e. your_hostname = "192.168.1.99"

or simply:

> $ pyro4-ns -n your_hostname

If you want to implement this concept on the *warehouse server*, you'll have to modify `warehouse.py`. Then, right before the `serveSimple` call, you have to tell it to bind the daemon on your hostname instead of localhost. One way to do this is by setting the `HOST` config item:

```
Pyro4.config.HOST = "your_hostname_here"
Pyro4.Daemon.serveSimple(...)
```

Optionally, you can choose to leave the code alone, and instead set the `PYRO_HOST` environment variable before starting the warehouse server. Another choice is to pass the required host (and perhaps even port) arguments to `serveSimple`:

```
Pyro4.Daemon.serveSimple(
        {
            Warehouse: "example.warehouse"
        },
        host = 'your_hostname_here',
        ns = True)
```

Remember that if you want more details, refer to the chapters in this manual about the relevant Pyro components.

Now, back on the new version of the *stock market server*, notice that this example already creates a daemon object instead of using the `serveSimple()` call. You'll have to modify `stockmarket.py` because that is the one creating a daemon. But you'll only have to add the proper `host``and ``port` arguments to the construction of the Daemon, to set it to your machine name instead of the default of localhost. Let's see the few minor changes that are required in the code:

> ... HOST_IP = "192.168.1.99" HOST_PORT = 9092 ... with Pyro4.Daemon(host=HOST_IP, port=HOST_PORT) as daemon: ...

Of course, you could also change the `HOST` config item (either in the code itself, or by setting the `PYRO_HOST` environment variable before launching).

2.3.5 Other means of creating connections

In both tutorials above we used the Name Server for easy object lookup. The use of the name server is optional, see *Name Server* for details. There are various other options for connecting your client code to your Pyro objects, have a

look at the client code details: *Object discovery* and the server code details: *Pyro Daemon: publishing Pyro objects*.

2.3.6 Ok, what's next?

Congratulations! You completed the Pyro tutorials in which you built a simple warehouse storage system, and a stock market simulation system consisting of various independent components that talk to each other using Pyro. The Pyro distribution archive contains a truckload of example programs with short descriptions that you could study to see how to use the various features that Pyro has to offer. Or just browse the manual for more detailed information. Happy remote object programming!

2.4 Command line tools

Pyro has several command line tools that you will be using sooner or later. They are generated and installed when you install Pyro.

- **pyro4-ns** (name server)
- **pyro4-nsc** (name server client tool)
- **pyro4-test-echoserver** (test echo server)
- **pyro4-check-config** (prints configuration)
- **pyro4-flameserver** (flame server)
- **pyro4-httpgateway** (http gateway server)

If you prefer, you can also invoke the various "executable modules" inside Pyro directly, by using Python's "-m" command line argument.

Some of these tools are described in detail in their respective sections of the manual:

Name server tools: See *Starting the Name Server* and *Name server control tool* for detailed information.

HTTP gateway server: See *Pyro via HTTP and JSON* for detailed information.

2.4.1 Test echo server

python -m Pyro4.test.echoserver [options] (or simply: **pyro4-test-echoserver [options]**)

This is a simple built-in server that can be used for testing purposes. It launches a Pyro object that has several methods suitable for various tests (see below). Optionally it can also directly launch a name server. This way you can get a simple Pyro server plus name server up with just a few keystrokes.

A short explanation of the available options can be printed with the help option:

-h, --help
Print a short help message and exit.

The echo server object is available by the name `test.echoserver`. It exposes the following methods:

echo(*argument*)
Simply returns the given argument object again.

error()
Generates a run time exception.

shutdown()
Terminates the echo server.

2.4.2 Configuration check

python -m Pyro4.configuration (or simply: **pyro4-check-config**) This is the equivalent of:

```
>>> import Pyro4
>>> print(Pyro4.config.dump())
```

It prints the Pyro version, the location it is imported from, and a dump of the active configuration items.

2.5 Clients: Calling remote objects

This chapter explains how you write code that calls remote objects. Often, a program that calls methods on a Pyro object is called a *client* program. (The program that provides the object and actually runs the methods, is the *server*. Both roles can be mixed in a single program.)

Make sure you are familiar with Pyro's *Key concepts* before reading on.

2.5.1 Object discovery

To be able to call methods on a Pyro object, you have to tell Pyro where it can find the actual object. This is done by creating an appropriate URI, which contains amongst others the object name and the location where it can be found. You can create it in a number of ways.

- **directly use the object name and location.** This is the easiest way and you write an URI directly like this: `PYRO:someobjectid@servername:9999` It requires that you already know the object id, servername, and port number. You could choose to use fixed object names and fixed port numbers to connect Pyro daemons on. For instance, you could decide that your music server object is always called "musicserver", and is accessible on port 9999 on your server musicbox.my.lan. You could then simply use:

  ```
  uri_string = "PYRO:musicserver@musicbox.my.lan:9999"
  # or use Pyro4.URI("...") for an URI object instead of a string
  ```

  Most examples that come with Pyro simply ask the user to type this in on the command line, based on what the server printed. This is not very useful for real programs, but it is a simple way to make it work. You could write the information to a file and read that from a file share (only slightly more useful, but it's just an idea).

- **use a logical name and look it up in the name server.** A more flexible way of locating your objects is using logical names for them and storing those in the Pyro name server. Remember that the name server is like a phone book, you look up a name and it gives you the exact location. To continue on the previous bullet, this means your clients would only have to know the logical name "musicserver". They can then use the name server to obtain the proper URI:

  ```
  import Pyro4
  nameserver = Pyro4.locateNS()
  uri = nameserver.lookup("musicserver")
  # ... uri now contains the URI with actual location of the musicserver object
  ```

  You might wonder how Pyro finds the Name server. This is explained in the separate chapter *Name Server*.

- **use a logical name and let Pyro look it up in the name server for you.** Very similar to the option above, but even more convenient, is using the *meta*-protocol identifier `PYRONAME` in your URI string. It lets Pyro know that it should lookup the name following it, in the name server. Pyro should then use the resulting URI from the name server to contact the actual object. See *The PYRONAME protocol type*. This means you can write:

```
uri_string = "PYRONAME:musicserver"
# or Pyro4.URI("PYRONAME:musicserver") for an URI object
```

  You can use this URI everywhere you would normally use a normal uri (using `PYRO`). Everytime Pyro encounters the `PYRONAME` uri it will use the name server automatically to look up the object for you.[1]

- **use object metadata tagging to look it up (yellow-pages style lookup).** You can do this directly via the name server for maximum control, or use the `PYROMETA` protocol type. See *The PYROMETA protocol type*. This means you can write:

```
uri_string = "PYROMETA:metatag1,metatag2"
# or Pyro4.URI("PYROMETA:metatag1,metatag2") for an URI object
```

  You can use this URI everywhere you would normally use a normal uri. Everytime Pyro encounters the `PYROMETA` uri it will use the name server automatically to find a random object for you with the given metadata tags.[1]

2.5.2 Calling methods

Once you have the location of the Pyro object you want to talk to, you create a Proxy for it. Normally you would perhaps create an instance of a class, and invoke methods on that object. But with Pyro, your remote method calls on Pyro objects go trough a proxy. The proxy can be treated as if it was the actual object, so you write normal python code to call the remote methods and deal with the return values, or even exceptions:

```
# Continuing our imaginary music server example.
# Assume that uri contains the uri for the music server object.

musicserver = Pyro4.Proxy(uri)
try:
    musicserver.load_playlist("90s rock")
    musicserver.play()
    print("Currently playing:", musicserver.current_song())
except MediaServerException:
    print("Couldn't select playlist or start playing")
```

For normal usage, there's not a single line of Pyro specific code once you have a proxy!

2.5.3 Accessing remote attributes

You can access exposed attributes of your remote objects directly via the proxy. If you try to access an undefined or unexposed attribute, the proxy will raise an AttributeError stating the problem. Note that direct remote attribute access only works if the metadata feature is enabled (`METADATA` config item, enabled by default).

```
import Pyro4

p = Pyro4.Proxy("...")
```

(continues on next page)

[1] this is not very efficient if it occurs often. Have a look at the *Tips & Tricks* chapter for some hints about this.

(continued from previous page)

```
velo = p.velocity     # attribute access, no method call
print("velocity = ", velo)
```

See the `attributes` example for more information.

2.5.4 Serialization

Pyro will serialize the objects that you pass to the remote methods, so they can be sent across a network connection. Depending on the serializer that is being used, there will be some limitations on what objects you can use.

- **serpent**: serializes into Python literal expressions. Accepts quite a lot of different types. Many will be serialized as dicts. You might need to explicitly translate literals back to specific types on the receiving end if so desired, because most custom classes aren't dealt with automatically. Requires third party library module, but it will be installed automatically as a dependency of Pyro. This serializer is the default choice.
- **json**: more restricted as serpent, less types supported. Part of the standard library. Not particularly fast, so you might want to look for a faster 3rd party implementation (such as simplejson). Be sure to benchmark before switching! Use the *JSON_MODULE* config item to tell Pyro to use the other module instead. Note that it has to support the advanced parameters such as *default*, not all 3rd party implementations do that.
- **marshal**: a very limited but fast serializer. Can deal with a small range of builtin types only, no custom classes can be serialized. Part of the standard library.
- **msgpack**: See https://pypi.python.org/pypi/msgpack Reasonably fast serializer (and a lot faster if you're using the C module extension). Can deal with many builtin types, but not all. Not enabled by default because it's optional, but it's safe to add to the accepted serializers config item if you have it installed.
- **pickle**: the legacy serializer. Fast and supports almost all types. Part of the standard library. Has security problems, so it's better to avoid using it.
- **cloudpickle**: See https://pypi.python.org/pypi/cloudpickle It is similar to pickle serializer, but more capable. Extends python's 'pickle' module for serializing and de-serializing python objects to the majority of the built-in python types. Has security problems though, just as pickle.
- **dill**: See https://pypi.python.org/pypi/dill It is similar to pickle serializer, but more capable. Extends python's 'pickle' module for serializing and de-serializing python objects to the majority of the built-in python types. Has security problems though, just as pickle.

You select the serializer to be used by setting the `SERIALIZER` config item. (See the *Configuring Pyro* chapter). The valid choices are the names of the serializer from the list mentioned above. If you're using pickle or dill, and need to control the protocol version that is used, you can do so with the `PICKLE_PROTOCOL_VERSION` or `DILL_PROTOCOL_VERSION` config items. If you're using cloudpickle, you can control the protocol version with `PICKLE_PROTOCOL_VERSION` as well. By default Pyro will use the highest one available.

It is possible to override the serializer on a particular proxy. This allows you to connect to one server using the default serpent serializer and use another proxy to connect to a different server using the json serializer, for instance. Set the desired serializer name in `proxy._pyroSerializer` to override.

Note: Since Pyro 4.20 the default serializer is "`serpent`". Serpent is secure but cannot serialize all types (by design). Some types are serialized into a different form such as a string or a dict. Strings are serialized/deserialized into unicode at all times – be aware of this if you're using Python 2.x (strings in Python 3.x are always unicode already).

Note: The serializer(s) that a Pyro server/daemon accepts, is controlled by a different config item

(SERIALIZERS_ACCEPTED). This can be a set of one or more serializers. By default it accepts the set of 'safe' serializers, so "pickle", "cloudpickle" and "dill" are excluded. If the server doesn't accept the serializer that you configured for your client, it will refuse the requests and respond with an exception that tells you about the unsupported serializer choice. If it *does* accept your requests, the server response will use the same serializer that was used for the request.

Note: Because the name server is just a regular Pyro server as well, you will have to tell it to allow the pickle, cloudpickle or dill serializers if your client code uses them. See *Using the name server with pickle, cloudpickle or dill serializers*.

Changing the way your custom classes are (de)serialized

Note: The information in this paragraph is not relevant when using the pickle, cloudpickle or dill serialization protocols, they have their own ways of serializing custom classes.

By default, custom classes are serialized into a dict. They are not deserialized back into instances of your custom class. This avoids possible security issues. An exception to this however are certain classes in the Pyro4 package itself (such as the URI and Proxy classes). They *are* deserialized back into objects of that certain class, because they are critical for Pyro to function correctly.

There are a few hooks however that allow you to extend this default behaviour and register certain custom converter functions. These allow you to change the way your custom classes are treated, and allow you to actually get instances of your custom class back from the deserialization if you so desire.

The hooks are provided via several classmethods: *Pyro4.util.SerializerBase.register_class_to_dict()* and *Pyro4.util.SerializerBase.register_dict_to_class()*

and their unregister-counterparts: *Pyro4.util.SerializerBase.unregister_class_to_dict()* and *Pyro4.util.SerializerBase.unregister_dict_to_class()*

Click on the method link to see its apidoc, or have a look at the ser_custom example and the test_serialize unit tests for more information. It is recommended to avoid using these hooks if possible, there's a security risk to create arbitrary objects from serialized data that is received from untrusted sources.

Upgrading older code that relies on pickle

What do you have to do with code that relies on pickle, and worked fine in older Pyro versions, but now crashes?

You have three options:

1. Redesign remote interfaces
2. Configure Pyro to eable the use of pickle again
3. Stick to Pyro 4.18 (less preferable)

You can redesign the remote interface to only include types that can be serialized (python's built-in types and exception classes, and a few Pyro specific classes such as URIs). That way you benefit from the new security that the alternative serializers provide. If you can't do this, you have to tell Pyro to enable pickle again. This has been made an explicit step because of the security implications of using pickle. Here's how to do this:

Client code configuration Tell Pyro to use pickle as serializer for outgoing communication, by setting the `SERIALIZER` config item to `pickle`. For instance, in your code: `Pyro4.config.SERIALIZER = 'pickle'` or set the appropriate environment variable.

Server code configuration Tell Pyro to accept pickle as incoming serialization format, by including `pickle` in the `SERIALIZERS_ACCEPTED` config item list. For instance, in your code: `Pyro4.config.SERIALIZERS_ACCEPTED.add('pickle')`. Or set the appropriate environment variable, for instance: `export PYRO_SERIALIZERS_ACCEPTED=serpent,json,marshal,pickle`. If your server also uses Pyro to call other servers, you may also need to configure it as mentioned above at 'client code'. This is because the incoming and outgoing serializer formats are configured independently.

2.5.5 Proxies, connections, threads and cleaning up

Here are some rules:

- Every single Proxy object will have its own socket connection to the daemon.
- You can share Proxy objects among threads, it will re-use the same socket connection.
- Usually every connection in the daemon has its own processing thread there, but for more details see the *Servers: hosting Pyro objects* chapter.
- The connection will remain active for the lifetime of the proxy object. Hence, consider cleaning up a proxy object explicitly if you know you won't be using it again in a while. That will free up resources and socket connections. You can do this in two ways:
  1. calling `_pyroRelease()` on the proxy.
  2. using the proxy as a context manager in a `with` statement. *This is the preffered way of creating and using Pyro proxies.* This ensures that when you're done with it, or an error occurs (inside the with-block), the connection is released:

```
with Pyro4.Proxy(".....") as obj:
    obj.method()
```

  Note: you can still use the proxy object when it is disconnected: Pyro will reconnect it as soon as it's needed again.
- At proxy creation, no actual connection is made. The proxy is only actually connected at first use, or when you manually connect it using the `_pyroReconnect()` or `_pyroBind()` methods.

2.5.6 Oneway calls

Normal method calls always block until the response is returned. This can be any normal return value, `None`, or an error in the form of a raised exception. The client code execution is suspended until the method call has finished and produced its result.

Some methods never return any response or you are simply not interested in it (including errors and exceptions!), or you don't want to wait until the result is available but rather continue immediately. You can tell Pyro that calls to these methods should be done as *one-way calls*. For calls to such methods, Pyro will not wait for a response from the remote object. The return value of these calls is always `None`, which is returned *immediately* after submitting the method invocation to the server. The server will process the call while your client continues execution. The client can't tell if the method call was successful, because no return value, no errors and no exceptions will be returned! If you want to find out later what - if anything - happened, you have to call another (non-oneway) method that does return a value.

Note that this is different from *Asynchronous ('future') remote calls & call chains*: they are also executed while your client code continues with its work, but they *do* return a value (but at a later moment in time). Oneway calls are more efficient because they immediately produce `None` as result and that's it.

How to make methods one-way: You mark the methods of your class *in the server* as one-way by using a special *decorator*. See *Creating a Pyro class and exposing its methods and properties* for details on how to do this. See the `oneway` example for some code that demonstrates the use of oneway methods.

2.5.7 Batched calls

Doing many small remote method calls in sequence has a fair amount of latency and overhead. Pyro provides a means to gather all these small calls and submit it as a single 'batched call'. When the server processed them all, you get back all results at once. Depending on the size of the arguments, the network speed, and the amount of calls, doing a batched call can be *much* faster than invoking every call by itself. Note that this feature is only available for calls on the same proxy object.

How it works:

1. You create a batch proxy object for the proxy object.
2. Call all the methods you would normally call on the regular proxy, but use the batch proxy object instead.
3. Call the batch proxy object itself to obtain the generator with the results.

You create a batch proxy using this: `batch = Pyro4.batch(proxy)` or this (equivalent): `batch = proxy._pyroBatch()`. The signature of the batch proxy call is as follows:

`batchproxy.__call__`([*oneway=False*, *asynchronous=False*])

Invoke the batch and when done, returns a generator that produces the results of every call, in order. If `oneway==True`, perform the whole batch as one-way calls, and return `None` immediately. If `asynchronous==True`, perform the batch asynchronously, and return an asynchronous call result object immediately.

Simple example:

```
batch = Pyro4.batch(proxy)
batch.method1()
batch.method2()
# more calls ...
batch.methodN()
results = batch()    # execute the batch
for result in results:
    print(result)    # process result in order of calls...
```

Oneway batch:

```
results = batch(oneway=True)
# results==None
```

Asynchronous batch

The result value of an asynchronous batch call is a special object. See *Asynchronous ('future') remote calls & call chains* for more details about it. This is some simple code doing an asynchronous batch:

```
results = batch(asynchronous=True)
# do some stuff... until you're ready and require the results of the asynchronous␣
↪batch:
for result in results.value:
    print(result)     # process the results
```

See the `batchedcalls` example for more details.

2.5.8 Remote iterators/generators

Since Pyro 4.49 it is possible to simply iterate over a remote iterator or generator function as if it was a perfectly normal Python iterable. Pyro will fetch the items one by one from the server that is running the remote iterator until all elements have been consumed or the client disconnects.

Filter on the server

If you plan to filter the items that are returned from the iterator, it is strongly suggested to do that on the server and not in your client. Because otherwise it is possible that you first have to serialize and transfer all possible items from the server only to select a few out of them, which is very inefficient.

Beware of many small items

Pyro has to do a remote call to get every next item from the iterable. If your iterator produces lots of small individual items, this can be quite inefficient (many small network calls). Either chunk them up a bit or use larger individual items.

So you can write in your client:

```
proxy = Pyro4.Proxy("...")
for item in proxy.things():
    print(item)
```

The implementation of the `things` method can return a normal list but can also return an iterator or even be a generator function itself. This has the usual benefits of "lazy" generators: no need to create the full collection upfront which can take a lot of memory, possibility of infinite sequences, and spreading computation load more evenly.

By default the remote item streaming is enabled in the server and there is no time limit set for how long iterators and generators can be 'alive' in the server. You can configure this however if you want to restrict resource usage or disable this feature altogether, via the `ITER_STREAMING` and `ITER_STREAM_LIFETIME` config items.

Lingering when disconnected: the `ITER_STREAM_LINGER` config item controls the number of seconds a remote generator is kept alive when a disconnect happens. It defaults to 30 seconds. This allows you to reconnect the proxy and continue using the remote generator as if nothing happened (see *`Pyro4.core.Proxy._pyroReconnect()`* or even *Automatic reconnecting*). If you reconnect the proxy and continue iterating again *after* the lingering timeout period expired, an exception is thrown because the remote generator has been discarded in the meantime. Lingering can be disabled completely by setting the value to 0, then all remote generators from a proxy will immediately be discarded in the server if the proxy gets disconnected or closed.

Notice that you can also use this in your Java or .NET/C# programs that connect to Python via Pyrolite! Version 4.14 or newer of that library supports Pyro item streaming. It returns normal Java and .NET iterables to your code that you can loop over normally with foreach or other things.

There are several examples that use the remote iterator feature. Have a look at the `stockquotes` tutorial example, or the `filetransfer` example.

2.5.9 Asynchronous ('future') remote calls & call chains

You can execute a remote method call and tell Pyro: "hey, I don't need the results right now. Go ahead and compute them, I'll come back later once I need them". The call will be processed in the background and you can collect the results at a later time. If the results are not yet available (because the call is *still* being processed) your code blocks but only at the line you are actually retrieving the results. If they have become available in the meantime, the code doesn't block at all and can process the results immediately. It is possible to define one or more callables (the "call chain") that should be invoked automatically by Pyro as soon as the result value becomes available.

You set a proxy in asynchronous mode using this: `Pyro4.asyncproxy(proxy)` or (equivalent): `proxy._pyroAsync()`. Every remote method call you make on the asynchronous proxy, returns a *Pyro4.futures.FutureResult* object immediately. This object means 'the result of this will be available at some moment in the future' and has the following interface:

value
: This property contains the result value from the call. If you read this and the value is not yet available, execution is halted until the value becomes available. If it is already available you can read it as usual.

ready
: This property contains the readiness of the result value (`True` meaning that the value is available).

wait(*[timeout=None]*)
: Waits for the result value to become available, with optional wait timeout (in seconds). Default is None, meaning infinite timeout. If the timeout expires before the result value is available, the call will return `False`. If the value has become available, it will return `True`.

then(*callable[, *args, **kwargs]*)
: Add a callable to the call chain, to be invoked when the results become available. The result of the current call will be used as the first argument for the next call. Optional extra arguments can be provided via `args` and `kwargs`.

iferror(*errorhandler*)
: Specify the exception handler to be invoked (with the exception object as only argument) when asking for the result raises an exception. If no exception handler is set, any exception result will be silently ignored (unless you explicitly ask for the value). Returns self so you can easily chain other calls.

A simple piece of code showing an asynchronous method call:

```
proxy._pyroAsync()
asyncresult = proxy.remotemethod()
print("value available?", asyncresult.ready)
# ...do some other stuff...
print("resultvalue=", asyncresult.value)
```

Note: *Batched calls* can also be executed asynchronously. Asynchronous calls are implemented using a background thread that waits for the results. Callables from the call chain are invoked sequentially in this background thread.

See the `async` example for more details and example code for call chains.

Async calls for normal callables (not only for Pyro proxies)

The asynchrnous proxy discussed above is only available when you are dealing with Pyro proxies. It provides a convenient syntax to call the methods on the proxy asynchronously. For normal Python code it is sometimes useful to have a similar mechanism as well. Pyro provides this too, see *Asynchronous ('future') normal function calls* for more information.

2.5.10 Pyro Callbacks

Usually there is a nice separation between a server and a client. But with some Pyro programs it is not that simple. It isn't weird for a Pyro object in a server somewhere to invoke a method call on another Pyro object, that could even be running in the client program doing the initial call. In this case the client program is a server itself as well.

These kinds of 'reverse' calls are labeled *callbacks*. You have to do a bit of work to make them possible, because normally, a client program is not running the required code to also act as a Pyro server to accept incoming callback calls.

In fact, you have to start a Pyro daemon and register the callback Pyro objects in it, just as if you were writing a server program. Keep in mind though that you probably have to run the daemon's request loop in its own background thread. Or make heavy use of oneway method calls. If you don't, your client program won't be able to process the callback requests because it is by itself still waiting for results from the server.

Exceptions in callback objects: If your callback object raises an exception, Pyro will return that to the server doing the callback. Depending on what the server does with it, you might never see the actual exception, let alone the stack trace. This is why Pyro provides a decorator that you can use on the methods in your callback object in the client program: `@Pyro4.callback`. This way, an exception in that method is not only returned to the caller, but also logged locally in your client program, so you can see it happen including the stack trace (if you have logging enabled):

```
import Pyro4

class Callback(object):

    @Pyro4.expose
    @Pyro4.callback
    def call(self):
        print("callback received from server!")
        return 1//0    # crash!
```

Also notice that the callback method (or the whole class) has to be decorated with `@Pyro4.expose` as well to allow it to be called remotely at all. See the `callback` example for more details and code.

2.5.11 Miscellaneous features

Pyro provides a few miscellaneous features when dealing with remote method calls. They are described in this section.

Error handling

You can just do exception handling as you would do when writing normal Python code. However, Pyro provides a few extra features when dealing with errors that occurred in remote objects. This subject is explained in detail its own chapter: *Exceptions and remote tracebacks*.

See the `exceptions` example for more details.

Timeouts

Because calls on Pyro objects go over the network, you might encounter network related problems that you don't have when using normal objects. One possible problems is some sort of network hiccup that makes your call unresponsive because the data never arrived at the server or the response never arrived back to the caller.

By default, Pyro waits an indefinite amount of time for the call to return. You can choose to configure a *timeout* however. This can be done globally (for all Pyro network related operations) by setting the timeout config item:

```
Pyro4.config.COMMTIMEOUT = 1.5      # 1.5 seconds
```

You can also do this on a per-proxy basis by setting the timeout property on the proxy:

```
proxy._pyroTimeout = 1.5    # 1.5 seconds
```

There is also a server setting related to oneway calls, that says if oneway method calls should be executed in a separate thread or not. If this is set to `False`, they will execute in the same thread as the other method calls. This means that if the oneway call is taking a long time to complete, the other method calls from the client may actually stall, because they're waiting on the server to complete the oneway call that came before them. To avoid this problem you can set this config item to True (which is the default). This runs the oneway call in its own thread (regardless of the server type that is used) and other calls can be processed immediately:

```
Pyro4.config.ONEWAY_THREADED = True      # this is the default
```

See the `timeout` example for more details.

Also, there is a automatic retry mechanism for timeout or connection closed (by server side), in order to use this automatically retry:

```
Pyro4.config.MAX_RETRIES = 3      # attempt to retry 3 times before raise the␣
↪exception
```

You can also do this on a pre-proxy basis by setting the max retries property on the proxy:

```
proxy._pyroMaxRetries = 3      # attempt to retry 3 times before raise the exception
```

Be careful to use when remote functions have a side effect (e.g.: calling twice results in error)! See the `autoretry` example for more details.

Automatic reconnecting

If your client program becomes disconnected to the server (because the server crashed for instance), Pyro will raise a *Pyro4.errors.ConnectionClosedError*. You can use the automatic retry mechanism to handle this exception, see the `autoretry` example for more details. Alternatively, it is also possible to catch this and tell Pyro to attempt to reconnect to the server by calling `_pyroReconnect()` on the proxy (it takes an optional argument: the number of attempts to reconnect to the daemon. By default this is almost infinite). Once successful, you can resume operations on the proxy:

```
try:
    proxy.method()
except Pyro4.errors.ConnectionClosedError:
    # connection lost, try reconnecting
    obj._pyroReconnect()
```

This will only work if you take a few precautions in the server. Most importantly, if it crashed and comes up again, it needs to publish its Pyro objects with the exact same URI as before (object id, hostname, daemon port number).

See the `autoreconnect` example for more details and some suggestions on how to do this.

The `_pyroReconnect()` method can also be used to force a newly created proxy to connect immediately, rather than on first use.

Proxy sharing

Due to internal locking you can freely share proxies among threads. The lock makes sure that only a single thread is actually using the proxy's communication channel at all times. This can be convenient *but* it may not be the best way to approach things. The lock essentially prevents parallelism. If you want calls to go in parallel, give each thread its own proxy.

Here are a couple of suggestions on how to make copies of a proxy:

1. use the `copy` module, `proxy2 = copy.copy(proxy)`

2. create a new proxy from the uri of the old one: `proxy2 = Pyro4.Proxy(proxy._pyroUri)`
3. simply create a proxy in the thread itself (pass the uri to the thread instead of a proxy)

See the `proxysharing` example for more details.

Metadata from the daemon

A proxy contains some meta-data about the object it connects to. It obtains the data via the (public) *Pyro4.core.DaemonObject.get_metadata()* method on the daemon that it connects to. This method returns the following information about the object (or rather, its class): what methods and attributes are defined, and which of the methods are to be called as one-way. This information is used to properly execute one-way calls, and to do client-side validation of calls on the proxy (for instance to see if a method or attribute is actually available, without having to do a round-trip to the server). Also this enables a properly working `hasattr` on the proxy, and efficient and specific error messages if you try to access a method or attribute that is not defined or not exposed on the Pyro object. Lastly the direct access to attributes on the remote object is also made possible, because the proxy knows about what attributes are available.

For backward compatibility with old Pyro4 versions (4.26 and older) you can disable this mechanism by setting the `METADATA` config item to `False` (it's `True` by default). You can tell if you need to do this if you're getting errors in your proxy saying that 'DaemonObject' has no attribute 'get_metadata'. Either upgrade the Pyro version of the server, or set the `METADATA` config item to False in your client code.

2.6 Servers: hosting Pyro objects

This chapter explains how you write code that publishes objects to be remotely accessible. These objects are then called *Pyro objects* and the program that provides them, is often called a *server* program.

(The program that calls the objects is usually called the *client*. Both roles can be mixed in a single program.)

Make sure you are familiar with Pyro's *Key concepts* before reading on.

See also:

Configuring Pyro for several config items that you can use to tweak various server side aspects.

2.6.1 Creating a Pyro class and exposing its methods and properties

Exposing classes, methods and properties is done using the `@Pyro4.expose` decorator. It lets you mark the following items to be available for remote access:

- methods (including classmethod and staticmethod). You cannot expose a 'private' method, i.e. name starting with underscore. You *can* expose a 'dunder' method with double underscore for example `__len__`. There is a short list of dunder methods that will never be remoted though (because they are essential to let the Pyro proxy function correctly). Make sure you put the `@expose` decorator after other decorators on the method, if any.
- properties (these will be available as remote attributes on the proxy) It's not possible to expose a 'private' property (name starting with underscore). You can't expose attributes directly. It is required to provide a @property for them and decorate that with `@expose`, if you want to provide a remotely accessible attribute.
- classes as a whole (exposing a class has the effect of exposing every nonprivate method and property of the class automatically)

private members

In the spirit of being secure by default, Pyro doesn't allow remote access to anything of your class unless explicitly told to do so. It will never allow remote access to 'private' members (where private means that the name starts with a single or double underscore, with a special exception for the regular 'dunder' names with double underscores such as `__len__`)

Anything that isn't decorated with `@expose` is not remotely accessible.

Here's a piece of example code that shows how a partially exposed Pyro class may look like:

```
import Pyro4

class PyroService(object):

    value = 42                  # not exposed

    def __dunder__(self):       # exposed
        pass

    def _private(self):         # not exposed
        pass

    def __private(self):        # not exposed
        pass

    @Pyro4.expose
    def get_value(self):        # exposed
        return self.value

    @Pyro4.expose
    @property
    def attr(self):             # exposed as 'proxy.attr' remote attribute
        return self.value

    @Pyro4.expose
    @attr.setter
    def attr(self, value):      # exposed as 'proxy.attr' writable
        self.value = value
```

Note: Prior to Pyro version 4.46, the default behavior was different: Pyro exposed everything, no special action was needed in your server side code to make it available to remote calls. Probably the easiest way to make old code that was written for this model to fit the new default behavior is to add a single `@Pyro4.expose` decorator on all of your Pyro classes. Better (safer) is to only add it to the methods and properties of the classes that are accessed remotely. If you cannot (or don't want to) change your code to be compatible with the new behavior, you can set the `REQUIRE_EXPOSE` config item back to `False` (it is now `True` by default). This will restore the old behavior.

Notice that it has been possible for a long time already for older code to utilize the `@expose` decorator and the current, safer, behavior by having `REQUIRE_EXPOSE` set to `True`. That choice has now simply become the default. Before upgrading to Pyro 4.46 or newer you can try setting it to `True` yourself and then adding `@expose` decorators to your Pyro classes or methods as required. Once everything works as it should you can then effortlessly upgrade Pyro itself.

Specifying one-way methods using the @Pyro4.oneway decorator:

You decide on the class of your Pyro object on the server, what methods are to be called as one-way. You use the `@Pyro4.oneway` decorator on these methods to mark them for Pyro. When the client proxy connects to the server

it gets told automatically what methods are one-way, you don't have to do anything on the client yourself. Any calls your client code makes on the proxy object to methods that are marked with `@Pyro4.oneway` on the server, will happen as one-way calls:

```
import Pyro4

@Pyro4.expose
class PyroService(object):

    def normal_method(self, args):
        result = do_long_calculation(args)
        return result

    @Pyro4.oneway
    def oneway_method(self, args):
        result = do_long_calculation(args)
        # no return value, cannot return anything to the client
```

See *Oneway calls* for the documentation about how client code handles this. See the `oneway` example for some code that demonstrates the use of oneway methods.

2.6.2 Exposing classes and methods without changing existing source code

In the case where you cannot or don't want to change existing source code, it's not possible to use the `@expose` decorator to tell Pyro what methods should be exposed. This can happen if you're dealing with third-party library classes or perhaps a generic module that you don't want to 'taint' with a Pyro dependency because it's used elsewhere too.

There are a few possibilities to deal with this:

Don't use @expose at all (not recommended)

You can disable the requirement for adding `@expose` to classes/methods by setting `REQUIRE_EXPOSE` back to False. This is a global setting however and will affect all your Pyro classes in the server, so be careful.

Use adapter classes

The preferred solution is to not use the classes from the third party library directly, but create an adapter class yourself with the appropriate `@expose` set on it or on its methods. Register this adapter class instead. Then use the class from the library from within your own adapter class. This way you have full control over what exactly is exposed, and what parameter and return value types travel over the wire.

Create exposed classes by using ''@expose'' as a function

Creating adapter classes is good but if you're looking for the most convenient solution we can do better. You can still use `@expose` to make a class a proper Pyro class with exposed methods, *without having to change the source code* due to adding @expose decorators, and without having to create extra classes yourself. Remember that Python decorators are just functions that return another function (or class)? This means you can also call them as a regular function yourself, which allows you to use classes from third party libraries like this:

```
from awesome_thirdparty_library import SomeClassFromLibrary
import Pyro4

# expose the class from the library using @expose as wrapper function:
ExposedClass = Pyro4.expose(SomeClassFromLibrary)

daemon.register(ExposedClass)     # register the exposed class rather than the library␣
↪class itself
```

There are a few caveats when using this:

1. You can only expose the class and all its methods as a whole, you can't cherrypick methods that should be exposed
2. You have no control over what data is returned from the methods. It may still be required to deal with serialization issues for instance when a method of the class returns an object whose type is again a class from the library.

See the `thirdpartylib` example for a little server that deals with such a third party library.

2.6.3 Pyro Daemon: publishing Pyro objects

To publish a regular Python object and turn it into a Pyro object, you have to tell Pyro about it. After that, your code has to tell Pyro to start listening for incoming requests and to process them. Both are handled by the *Pyro daemon*.

In its most basic form, you create one or more classes that you want to publish as Pyro objects, you create a daemon, register the class(es) with the daemon, and then enter the daemon's request loop:

```
import Pyro4

@Pyro4.expose
class MyPyroThing(object):
    # ... methods that can be called go here...
    pass

daemon = Pyro4.Daemon()
uri = daemon.register(MyPyroThing)
print(uri)
daemon.requestLoop()
```

Once a client connects, Pyro will create an instance of the class and use that single object to handle the remote method calls during one client proxy session. The object is removed once the client disconnects. Another client will cause another instance to be created for its session. You can control more precisely when, how, and for how long Pyro will create an instance of your Pyro class. See *Controlling Instance modes and Instance creation* below for more details.

Anyway, when you run the code printed above, the uri will be printed and the server sits waiting for requests. The uri that is being printed looks a bit like this: `PYRO:obj_dcf713ac20ce4fb2a6e72acaeba57dfd@localhost:51850` Client programs use these uris to access the specific Pyro objects.

Note: From the address in the uri that was printed you can see that Pyro by default binds its daemons on localhost. This means you cannot reach them from another machine on the network (a security measure). If you want to be able to talk to the daemon from other machines, you have to explicitly provide a hostname to bind on. This is done by giving a `host` argument to the daemon, see the paragraphs below for more details on this.

Note: Private methods: Pyro considers any method or attribute whose name starts with at least one underscore ('_'), private. These cannot be accessed remotely. An exception is made for the 'dunder' methods with double underscores, such as `__len__`. Pyro follows Python itself here and allows you to access these as normal methods, rather than treating them as private.

Note: You can publish any regular Python object as a Pyro object. However since Pyro adds a few Pyro-specific attributes to the object, you can't use:

- types that don't allow custom attributes, such as the builtin types (`str` and `int` for instance)
- types with `__slots__` (a possible way around this is to add Pyro's custom attributes to your `__slots__`, but that isn't very nice)

Note: Most of the the time a Daemon will keep running. However it's still possible to nicely free its resources when the request loop terminates by simply using it as a context manager in a `with` statement, like so:

```
with Pyro4.Daemon() as daemon:
    daemon.register(...)
    daemon.requestLoop()
```

Oneliner Pyro object publishing: serveSimple()

Ok not really a one-liner, but one statement: use `serveSimple()` to publish a dict of objects/classes and start Pyro's request loop. The code above could also be written as:

```
import Pyro4

@Pyro4.expose
class MyPyroThing(object):
    pass

obj = MyPyroThing()
Pyro4.Daemon.serveSimple(
    {
        MyPyroThing: None,     # register the class
        obj: None              # register one specific instance
    },
    ns=False)
```

You can perform some limited customization:

static Daemon.**serveSimple**(*objects [host=None, port=0, daemon=None, ns=True, verbose=True]*)

Very basic method to fire up a daemon that hosts a bunch of objects. The objects will be registered automatically in the name server if you specify this. API reference: *Pyro4.core.Daemon.serveSimple()*

Parameters

- **objects** (*dict*) – mapping of objects/classes to names, these are the Pyro objects that will be hosted by the daemon, using the names you provide as values in the mapping. Normally you'll provide a name yourself but in certain situations it may be useful to set it to `None`. Read below for the exact behavior there.
- **host** (*str or None*) – optional hostname where the daemon should be accessible on. Necessary if you want to access the daemon from other machines.
- **port** (*int*) – optional port number where the daemon should be accessible on
- **daemon** (`Pyro4.core.Daemon`) – optional existing daemon to use, that you created yourself. If you don't specify this, the method will create a new daemon object by itself.
- **ns** (*bool*) – optional, if True (the default), the objects will also be registered in the name server (located using *Pyro4.locateNS()*) for you. If this parameters is False, your objects will only be hosted in the daemon and are not published in a name server. Read below about the exact behavior of the object names you provide in the `objects` dictionary.

- **verbose** (*bool*) – optional, if True (the default), print out a bit of info on the objects that are registered

Returns nothing, it starts the daemon request loop and doesn't return until that stops.

If you set `ns=True` your objects will appear in the name server as well (this is the default setting). Usually this means you provide a logical name for every object in the `objects` dictionary. If you don't (= set it to `None`), the object will still be available in the daemon (by a generated name) but will *not* be registered in the name server (this is a bit strange, but hey, maybe you don't want all the objects to be visible in the name server).

When not using a name server at all (`ns=False`), the names you provide are used as the object names in the daemon itself. If you set the name to `None` in this case, your object will get an automatically generated internal name, otherwise your own name will be used.

Important:

- The names you provide for each object have to be unique (or `None`). For obvious reasons you can't register multiple objects with the same names.
- if you use `None` for the name, you have to use the `verbose` setting as well, otherwise you won't know the name that Pyro generated for you. That would make your object more or less unreachable.

The uri that is used to register your objects in the name server with, is of course generated by the daemon. So if you need to influence that, for instance because of NAT/firewall issues, it is the daemon's configuration you should be looking at.

If you don't provide a daemon yourself, `serveSimple()` will create a new one for you using the default configuration or with a few custom parameters you can provide in the call, as described above. If you don't specify the `host` and `port` parameters, it will simple create a Daemon using the default settings. If you *do* specify `host` and/or `port`, it will use these as parameters for creating the Daemon (see next paragraph). If you need to further tweak the behavior of the daemon, you have to create one yourself first, with the desired configuration. Then provide it to this function using the `daemon` parameter. Your daemon will then be used instead of a new one:

```
custom_daemon = Pyro4.Daemon(host="example", nathost="example")    # some additional␣
↪custom configuration
Pyro4.Daemon.serveSimple(
    {
        MyPyroThing: None
    },
    daemon = custom_daemon)
```

Creating a Daemon

Pyro's daemon is `Pyro4.Daemon` (shortcut to *Pyro4.core.Daemon*). It has a few optional arguments when you create it:

Daemon([*host=None*, *port=0*, *unixsocket=None*, *nathost=None*, *natport=None*, *interface=DaemonObject*])
Create a new Pyro daemon.

Parameters

- **host** (*str or None*) – the hostname or IP address to bind the server on. Default is `None` which means it uses the configured default (which is localhost). It is necessary to set this argument to a visible hostname or ip address, if you want to access the daemon from other machines.
- **port** (*int*) – port to bind the server on. Defaults to 0, which means to pick a random port.

- **unixsocket** (*str or None*) – the name of a Unix domain socket to use instead of a TCP/IP socket. Default is None (don't use).
- **nathost** – hostname to use in published addresses (useful when running behind a NAT firewall/router). Default is None which means to just use the normal host. For more details about NAT, see *Pyro behind a NAT router/firewall*.
- **natport** – port to use in published addresses (useful when running behind a NAT firewall/router). If you use 0 here, Pyro will replace the NAT-port by the internal port number to facilitate one-to-one NAT port mappings.
- **interface** (Pyro4.core.DaemonObject) – optional alternative daemon object implementation (that provides the Pyro API of the daemon itself)

Registering objects/classes

Every object you want to publish as a Pyro object needs to be registered with the daemon. You can let Pyro choose a unique object id for you, or provide a more readable one yourself.

Daemon.**register**(*obj_or_class*[, *objectId=None*, *force=False*])
Registers an object with the daemon to turn it into a Pyro object.

Parameters

- **obj_or_class** – the singleton instance or class to register (class is the preferred way)
- **objectId** (*str or None*) – optional custom object id (must be unique). Default is to let Pyro create one for you.
- **force** (*bool*) – optional flag to force registration, normally Pyro checks if an object had already been registered. If you set this to True, the previous registration (if present) will be silently overwritten.

Returns an uri for the object

Return type *Pyro4.core.URI*

It is important to do something with the uri that is returned: it is the key to access the Pyro object. You can save it somewhere, or perhaps print it to the screen. The point is, your client programs need it to be able to access your object (they need to create a proxy with it).

Maybe the easiest thing is to store it in the Pyro name server. That way it is almost trivial for clients to obtain the proper uri and connect to your object. See *Name Server* for more information (*Registering object names*), but it boils down to getting a name server proxy and using its register method:

```
uri = daemon.register(some_object)
ns = Pyro4.locateNS()
ns.register("example.objectname", uri)
```

Note: If you ever need to create a new uri for an object, you can use *Pyro4.core.Daemon.uriFor()*. The reason this method exists on the daemon is because an uri contains location information and the daemon is the one that knows about this.

Intermission: Example 1: server and client not using name server

A little code example that shows the very basics of creating a daemon and publishing a Pyro object with it. Server code:

```
import Pyro4

@Pyro4.expose
class Thing(object):
    def method(self, arg):
        return arg*2

# ------ normal code ------
daemon = Pyro4.Daemon()
uri = daemon.register(Thing)
print("uri=",uri)
daemon.requestLoop()

# ------ alternatively, using serveSimple -----
Pyro4.Daemon.serveSimple(
    {
        Thing: None
    },
    ns=False, verbose=True)
```

Client code example to connect to this object:

```
import Pyro4
# use the URI that the server printed:
uri = "PYRO:obj_b2459c80671b4d76ac78839ea2b0fb1f@localhost:49383"
thing = Pyro4.Proxy(uri)
print(thing.method(42))    # prints 84
```

With correct additional parameters –described elsewhere in this chapter– you can control on which port the daemon is listening, on what network interface (ip address/hostname), what the object id is, etc.

Intermission: Example 2: server and client, with name server

A little code example that shows the very basics of creating a daemon and publishing a Pyro object with it, this time using the name server for easier object lookup. Server code:

```
import Pyro4

@Pyro4.expose
class Thing(object):
    def method(self, arg):
        return arg*2

# ------ normal code ------
daemon = Pyro4.Daemon(host="yourhostname")
ns = Pyro4.locateNS()
uri = daemon.register(Thing)
ns.register("mythingy", uri)
daemon.requestLoop()

# ------ alternatively, using serveSimple -----
Pyro4.Daemon.serveSimple(
    {
        Thing: "mythingy"
    },
    ns=True, verbose=True, host="yourhostname")
```

Client code example to connect to this object:

```
import Pyro4
thing = Pyro4.Proxy("PYRONAME:mythingy")
print(thing.method(42))    # prints 84
```

Unregistering objects

When you no longer want to publish an object, you need to unregister it from the daemon:

`Daemon.`**`unregister`**`(`*objectOrId*`)`

> **Parameters** **`objectOrId`**`(`*object itself or its id string*`)` – the object to unregister

Running the request loop

Once you've registered your Pyro object you'll need to run the daemon's request loop to make Pyro wait for incoming requests.

`Daemon.`**`requestLoop`**`(`[*loopCondition*]`)`

> **Parameters** **`loopCondition`** – optional callable returning a boolean, if it returns False the request loop will be aborted and the call returns

This is Pyro's event loop and it will take over your program until it returns (it might never.) If this is not what you want, you can control it a tiny bit with the `loopCondition`, or read the next paragraph.

Integrating Pyro in your own event loop

If you want to use a Pyro daemon in your own program that already has an event loop (aka main loop), you can't simply call `requestLoop` because that will block your program. A daemon provides a few tools to let you integrate it into your own event loop:

- *`Pyro4.core.Daemon.sockets`* - list of all socket objects used by the daemon, to inject in your own event loop
- *`Pyro4.core.Daemon.events()`* - method to call from your own event loop when Pyro needs to process requests. Argument is a list of sockets that triggered.

For more details and example code, see the `eventloop` and `gui_eventloop` examples. They show how to use Pyro including a name server, in your own event loop, and also possible ways to use Pyro from within a GUI program with its own event loop.

Combining Daemon request loops

In certain situations you will be dealing with more than one daemon at the same time. For instance, when you want to run your own Daemon together with an 'embedded' Name Server Daemon, or perhaps just another daemon with different settings.

Usually you run the daemon's *`Pyro4.core.Daemon.requestLoop()`* method to handle incoming requests. But when you have more than one daemon to deal with, you have to run the loops of all of them in parallel somehow. There are a few ways to do this:

1. multithreading: run each daemon inside its own thread

2. multiplexing event loop: write a multiplexing event loop and call back into the appropriate daemon when one of its connections send a request. You can do this using `selectors` or `select` and you can even integrate other (non-Pyro) file-like selectables into such a loop. Also see the paragraph above.
3. use *Pyro4.core.Daemon.combine()* to combine several daemons into one, so that you only have to call the requestLoop of that "master daemon". Basically Pyro will run an integrated multiplexed event loop for you. You can combine normal Daemon objects, the NameServerDaemon and also the name server's BroadcastServer. Again, have a look at the `eventloop` example to see how this can be done. (Note: this will only work with the `multiplex` server type, not with the `thread` type)

Cleaning up

To clean up the daemon itself (release its resources) either use the daemon object as a context manager in a `with` statement, or manually call *Pyro4.core.Daemon.close()*.

Of course, once the daemon is running, you first need a clean way to stop the request loop before you can even begin to clean things up.

You can use force and hit ctrl-C or ctrl-or ctrl-Break to abort the request loop, but this usually doesn't allow your program to clean up neatly as well. It is therefore also possible to leave the loop cleanly from within your code (without using `sys.exit()` or similar). You'll have to provide a `loopCondition` that you set to `False` in your code when you want the daemon to stop the loop. You could use some form of semi-global variable for this. (But if you're using the threaded server type, you have to also set `COMMTIMEOUT` because otherwise the daemon simply keeps blocking inside one of the worker threads).

Another possibility is calling *Pyro4.core.Daemon.shutdown()* on the running daemon object. This will also break out of the request loop and allows your code to neatly clean up after itself, and will also work on the threaded server type without any other requirements.

If you are using your own event loop mechanism you have to use something else, depending on your own loop.

2.6.4 Controlling Instance modes and Instance creation

While it is possible to register a single singleton *object* with the daemon, it is actually preferred that you register a *class* instead. When doing that, it is Pyro itself that creates an instance (object) when it needs it. This allows for more control over when and for how long Pyro creates objects.

Controlling the instance mode and creation is done by decorating your class with `Pyro4.behavior` and setting its `instance_mode` or/and `instance_creator` parameters. It can only be used on a class definition, because these behavioral settings only make sense at that level.

By default, Pyro will create an instance of your class per *session* (=proxy connection) Here is an example of registering a class that will have one new instance for *every single method call* instead:

```
import Pyro4

@Pyro4.behavior(instance_mode="percall")
class MyPyroThing(object):
    @Pyro4.expose
    def method(self):
        return "something"

daemon = Pyro4.Daemon()
uri = daemon.register(MyPyroThing)
print(uri)
daemon.requestLoop()
```

There are three possible choices for the `instance_mode` parameter:

- `session`: (the default) a new instance is created for every new proxy connection, and is reused for all the calls during that particular proxy session. Other proxy sessions will deal with a different instance.
- `single`: a single instance will be created and used for all method calls, regardless what proxy connection we're dealing with. This is the same as creating and registering a single object yourself (the old style of registering code with the deaemon). Be aware that the methods on this object can be called from separate threads concurrently.
- `percall`: a new instance is created for every single method call, and discarded afterwards.

Instance creation

Instance creation is lazy

When you register a class in this way, be aware that Pyro only creates an actual instance of it when it is first needed. If nobody connects to the deamon requesting the services of this class, no instance is ever created.

Normally Pyro will simply use a default parameterless constructor call to create the instance. If you need special initialization or the class's init method requires parameters, you have to specify an `instance_creator` callable as well. Pyro will then use that to create an instance of your class. It will call it with the class to create an instance of as the single parameter.

See the `instancemode` example to learn about various ways to use this. See the `usersession` example to learn how you could use it to build user-bound resource access without concurrency problems.

2.6.5 Autoproxying

Pyro will automatically take care of any Pyro objects that you pass around through remote method calls. It will replace them by a proxy automatically, so the receiving side can call methods on it and be sure to talk to the remote object instead of a local copy. There is no need to create a proxy object manually. All you have to do is to register the new object with the appropriate daemon:

```
def some_pyro_method(self):
    thing=SomethingNew()
    self._pyroDaemon.register(thing)
    return thing    # just return it, no need to return a proxy
```

This feature can be enabled or disabled by a config item, see *Configuring Pyro*. (it is on by default). If it is off, a copy of the object itself is returned, and the client won't be able to interact with the actual new Pyro object in the server. There is a `autoproxy` example that shows the use of this feature, and several other examples also make use of it.

Note that when using the marshal serializer, this feature doesn't work. You have to use one of the other serializers to use autoproxying. Also, it doesn't work correctly when you are using old-style classes (but they are from Python 2.2 and earlier, you should not be using these anyway).

2.6.6 Server types and Concurrency model

Pyro supports multiple server types (the way the Daemon listens for requests). Select the desired type by setting the `SERVERTYPE` config item. It depends very much on what you are doing in your Pyro objects what server type is most suitable. For instance, if your Pyro object does a lot of I/O, it may benefit from the parallelism provided by the thread pool server. However if it is doing a lot of CPU intensive calculations, the multiplexed server may be more appropriate. If in doubt, go with the default setting.

1. **threaded server (servertype `"thread"`, this is the default)** This server uses a dynamically adjusted thread pool to handle incoming proxy connections. If the max size of the thread pool is too small for the number of proxy connections, new proxy connections will fail with an exception. The size of the pool is configurable via some config items:

   - `THREADPOOL_SIZE` this is the maximum number of threads that Pyro will use
   - `THREADPOOL_SIZE_MIN` this is the minimum number of threads that must remain standby

   Every proxy on a client that connects to the daemon will be assigned to a thread to handle the remote method calls. This way multiple calls can potentially be processed concurrently. *This means your Pyro object may have to be made thread-safe*! If you registered the pyro object's class with instance mode `single`, that single instance will be called concurrently from different threads. If you used instance mode `session` or `percall`, the instance will not be called from different threads because a new one is made per connection or even per call. But in every case, if you access a shared resource from your Pyro object, you may need to take thread locking measures such as using Queues.

2. **multiplexed server (servertype `"multiplex"`)** This server uses a connection multiplexer to process all remote method calls sequentially. No threads are used in this server. It uses the best supported selector available on your platform (kqueue, poll, select). It means only one method call is running at a time, so if it takes a while to complete, all other calls are waiting for their turn (even when they are from different proxies). The instance mode used for registering your class, won't change the way the concurrent access to the instance is done: in all cases, there is only one call active at all times. Your objects will never be called concurrently from different threads, because there are no threads. It does still affect when and how often Pyro creates an instance of your class.

Note: If the `ONEWAY_THREADED` config item is enabled (it is by default), *oneway* method calls will be executed in a separate worker thread, regardless of the server type you're using.

When to choose which server type? With the threadpool server at least you have a chance to achieve concurrency, and you don't have to worry much about blocking I/O in your remote calls. The usual trouble with using threads in Python still applies though: Python threads don't run concurrently unless they release the GIL (Global Interpreter Lock). If they don't, you will still hang your server process. For instance if a particular piece of your code doesn't release the GIL during a longer computation, the other threads will remain asleep waiting to acquire the GIL. One of these threads may be the Pyro server loop and then your whole Pyro server will become unresponsive. Doing I/O usually means the GIL is released. Some C extension modules also release it when doing their work. So, depending on your situation, not all hope is lost.

With the multiplexed server you don't have threading problems: everything runs in a single main thread. This means your requests are processed sequentially, but it's easier to make the Pyro server unresponsive. Any operation that uses blocking I/O or a long-running computation will block all remote calls until it has completed.

2.6.7 Serialization

Pyro will serialize the objects that you pass to the remote methods, so they can be sent across a network connection. Depending on the serializer that is being used for your Pyro server, there will be some limitations on what objects you can use, and what serialization format is required of the clients that connect to your server.

You specify one or more serializers that are accepted in the daemon/server by setting the `SERIALIZERS_ACCEPTED` config item. This is a set of serializer names that are allowed to be used with your server. It defaults to the set of 'safe' serializers. A client that successfully talks to your server will get responses using the same serializer as the one used to send requests to the server.

If your server also uses Pyro client code/proxies, you might also need to select the serializer for these by setting the `SERIALIZER` config item.

See the *Configuring Pyro* chapter for details about the config items. See *Serialization* for more details about serialization, the new config items, and how to deal with existing code that relies on pickle.

Note: Since Pyro 4.20 the default serializer is "`serpent`". It used to be "`pickle`" in older versions. The default set of accepted serializers in the server is the set of 'safe' serializers, so "`pickle`" and "`dill`" are not among the default.

2.6.8 Other features

Attributes added to Pyro objects

The following attributes will be added to your object if you register it as a Pyro object:

- `_pyroId` - the unique id of this object (a `str`)
- `_pyroDaemon` - a reference to the *Pyro4.core.Daemon* object that contains this object

Even though they start with an underscore (and are private, in a way), you can use them as you so desire. As long as you don't modify them! The daemon reference for instance is useful to register newly created objects with, to avoid the need of storing a global daemon object somewhere.

These attributes will be removed again once you unregister the object.

Network adapter binding and localhost

All Pyro daemons bind on localhost by default. This is because of security reasons. This means only processes on the same machine have access to your Pyro objects. If you want to make them available for remote machines, you'll have to tell Pyro on what network interface address it must bind the daemon. This also extends to the built in servers such as the name server.

Warning: Read chapter *Security* before exposing Pyro objects to remote machines!

There are a few ways to tell Pyro what network address it needs to use. You can set a global config item `HOST`, or pass a `host` parameter to the constructor of a Daemon, or use a command line argument if you're dealing with the name server. For more details, refer to the chapters in this manual about the relevant Pyro components.

Pyro provides a couple of utility functions to help you with finding the appropriate IP address to bind your servers on if you want to make them publicly accessible:

- *Pyro4.socketutil.getIpAddress()*
- *Pyro4.socketutil.getInterfaceAddress()*

Cleaning up / disconnecting stale client connections

A client proxy will keep a connection open even if it is rarely used. It's good practice for the clients to take this in consideration and release the proxy. But the server can't enforce this, some clients may keep a connection open for a long time. Unfortunately it's hard to tell when a client connection has become stale (unused). Pyro's default behavior is to accept this fact and not kill the connection. This does mean however that many stale client connections will eventually block the server's resources, for instance all workers threads in the threadpool server.

There's a simple possible solution to this, which is to specify a communication timeout on your server. For more information about this, read *'After X simultaneous proxy connections, Pyro seems to freeze!' Fix: Release your proxies when you can.*.

Daemon Pyro interface

A rather interesting aspect of Pyro's Daemon is that it (partly) is a Pyro object itself. This means it exposes a couple of remote methods that you can also invoke yourself if you want. The object exposed is *Pyro4.core.DaemonObject* (as you can see it is a bit limited still).

You access this object by creating a proxy for the `"Pyro.Daemon"` object. That is a reserved object name. You can use it directly but it is preferable to use the constant `Pyro4.constants.DAEMON_NAME`. An example follows that accesses the daemon object from a running name server:

```
>>> import Pyro4
>>> daemon=Pyro4.Proxy("PYRO:"+Pyro4.constants.DAEMON_NAME+"@localhost:9090")
>>> daemon.ping()
>>> daemon.registered()
['Pyro.NameServer', 'Pyro.Daemon']
```

2.7 Name Server

The Pyro Name Server is a tool to help keeping track of your objects in your network. It is also a means to give your Pyro objects logical names instead of the need to always know the exact object name (or id) and its location.

Pyro will name its objects like this:

```
PYRO:obj_dcf713ac20ce4fb2a6e72acaeba57dfd@localhost:51850
PYRO:custom_name@localhost:51851
```

It's either a generated unique object id on a certain host, or a name you chose yourself. But to connect to these objects you'll always need to know the exact object name or id and the exact hostname and port number of the Pyro daemon where the object is running. This can get tedious, and if you move servers around (or Pyro objects) your client programs can no longer connect to them until you update all URIs.

Enter the *name server*. This is a simple phone-book like registry that maps logical object names to their corresponding URIs. No need to remember the exact URI anymore. Instead, you can ask the name server to look it up for you. You only need to give it the logical object name.

Note: Usually you only need to run *one single instance* of the name server in your network. You can start multiple name servers but they are unconnected; you'll end up with a partitioned name space.

Example scenario: Assume you've got a document archive server that publishes a Pyro object with several archival related methods in it. This archive server can register this object with the name server, using a logical name such as "Department.ArchiveServer". Any client can now connect to it using only the name "Department.ArchiveServer". They don't need to know the exact Pyro id and don't even need to know the location. This means you can move the archive server to another machine and as long as it updates its record in the name server, all clients won't notice anything and can keep on running without modification.

2.7.1 Starting the Name Server

The easiest way to start a name server is by using the command line tool.

synopsys: **python -m Pyro4.naming [options]** (or simply: **pyro4-ns [options]**)

Starts the Pyro Name Server. It can run without any arguments but there are several that you can use, for instance to control the hostname and port that the server is listening on. A short explanation of the available options can be printed with the help option. When it starts, it prints a message similar to this ('neptune' is the hostname of the machine it is running on):

```
$ pyro4-ns -n neptune
Broadcast server running on 0.0.0.0:9091
NS running on neptune:9090 (192.168.1.100)
URI = PYRO:Pyro.NameServer@neptune:9090
```

As you can see it prints that it started a broadcast server (and its location), a name server (and its location), and it also printed the URI that clients can use to access it directly.

The nameserver uses a fast but volatile in-memory database by default. With a command line argument you can select a persistent storage mechanism (see below). If you're using that, your registrations will not be lost when the nameserver stops/restarts. The server will print the number of existing registrations at startup time if it discovers any.

Note: Pyro by default binds its servers on localhost which means you cannot reach them from another machine on the network. This behavior also applies to the name server. If you want to be able to talk to the name server from other machines, you have to explicitly provide a hostname to bind on.

There are several command line options for this tool:

-h, --help
Print a short help message and exit.

-n HOST, **--host**=HOST
Specify hostname or ip address to bind the server on. The default is localhost, note that your name server will then not be visible from the network If the server binds on localhost, *no broadcast responder* is started either. Make sure to provide a hostname or ip address to make the name server reachable from other machines, if you want that.

-p PORT, **--port**=PORT
Specify port to bind server on (0=random).

-u UNIXSOCKET, **--unixsocket**=UNIXSOCKET
Specify a Unix domain socket name to bind server on, rather than a normal TCP/IP socket.

--bchost=BCHOST
Specify the hostname or ip address to bind the broadcast responder on. Note: if the hostname where the name server binds on is localhost (or 127.0.x.x), no broadcast responder is started.

--bcport=BCPORT
Specify the port to bind the broadcast responder on (0=random).

--nathost=NATHOST
Specify the external host name to use in case of NAT

--natport=NATPORT
Specify the external port use in case of NAT

-s STORAGE, **--storage**=STORAGE
Specify the storage mechanism to use. You have several options:

- `memory` - fast, volatile in-memory database. This is the default.
- `dbm:dbfile` - dbm-style persistent database table. Provide the filename to use. This storage type does not support metadata.

- `sql:sqlfile` - sqlite persistent database. Provide the filename to use.

-x, --nobc
Don't start a broadcast responder. Clients will not be able to use the UDP-broadcast lookup to discover this name server. (The broadcast responder listens to UDP broadcast packets on the local network subnet, to signal its location to clients that want to talk to the name server)

-k, --key
Specify hmac key to use. Deprecated: use SSL instead, or if you must, set the key via the PYRO_HMAC_KEY environment variable before starting the name server.

2.7.2 Starting the Name Server from within your own code

Another way to start up a name server is by doing it from within your own code. This is more complex than simply launching it via the command line tool, because you have to integrate the name server into the rest of your program (perhaps you need to merge event loops?). For your convenience, two helper functions are available to create a name server yourself: *Pyro4.naming.startNS()* and *Pyro4.naming.startNSloop()*. Look at the `eventloop` example to see how you can use this.

Custom storage mechanism: The utility functions allow you to specify a custom storage mechanism (via the `storage` parameter). By default the in memory storage `Pyro4.naming.MemoryStorage` is used. In the *Pyro4.naming_storage* module you can find the two other implementations (for the dbm and for the sqlite storage). You could also build your own, as long as it has the same interface.

2.7.3 Configuration items

There are a couple of config items related to the nameserver. They are used both by the name server itself (to configure the values it will use to start the server with), and the client code that locates the name server (to give it optional hints where the name server is located). Often these can be overridden with a command line option or with a method parameter in your code.

Configuration item	description
HOST	hostname that the name server will bind on (being a regular Pyro daemon).
NS_HOST	the hostname or ip address of the name server. Used for locating in clients only.
NS_PORT	the port number of the name server. Used by the server and for locating in clients.
NS_BCHOST	the hostname or ip address of the name server's broadcast responder. Used only by the server.
NS_BCPORT	the port number of the name server's broadcast responder. Used by the server and for locating in clients.
NATHOST	the external hostname in case of NAT. Used only by the server.
NATPORT	the external port in case of NAT. Used only by the server.
NS_AUTOCLEAN	a recurring period in seconds where the Name server checks its registrations, and removes the ones that are no longer available. Defaults to 0.0 (off).

2.7.4 Name server control tool

The name server control tool (or 'nsc') is used to talk to a running name server and perform diagnostic or maintenance actions such as querying the registered objects, adding or removing a name registration manually, etc.

synopsis: **`python -m Pyro4.nsc [options] command [arguments]`** (or simply: **`pyro4-nsc [options] command [arguments]`**)

-h, --help
Print a short help message and exit.

-n HOST, **--host**=HOST
Provide the hostname or ip address of the name server. The default is to do a broadcast lookup to search for a name server.

-p PORT, **--port**=PORT
Provide the port of the name server, or its broadcast port if you're doing a broadcast lookup.

-u UNIXSOCKET, **--unixsocket**=UNIXSOCKET
Provide the Unix domain socket name of the name server, rather than a normal TCP/IP socket.

-k, --key
Specify hmac key to use. Deprecated: use SSL instead, or if you must, set the key via the PYRO_HMAC_KEY environment variable before starting the nsc tool.

-v, --verbose
Print more output that could be useful.

The available commands for this tool are:

list [list [prefix]] List all objects registered in the name server. If you supply a prefix, the list will be filtered to show only the objects whose name starts with the prefix.

listmatching [listmatching pattern] List only the objects with a name matching the given regular expression pattern.

lookup [lookup name] Looks up a single name registration and prints the uri.

listmeta_all [listmeta_all metadata [metadata...]] List the objects having *all* of the given metadata tags

listmeta_any [listmeta_any metadata [metadata...]] List the objects having *any one* (or multiple) of the given metadata tags

register [register name uri] Registers a name to the given Pyro object URI.

remove [remove name] Removes the entry with the exact given name from the name server.

removematching [removematching pattern] Removes all entries matching the given regular expression pattern.

setmeta [setmeta name [metadata...]] Sets the new list of metadata tags for the given Pyro object. If you don't specify any metadata tags, the metadata of the object is cleared.

ping Does nothing besides checking if the name server is running and reachable.

Example:

```
$ pyro4-nsc ping
Name server ping ok.

$ pyro4-nsc list Pyro
--------START LIST - prefix 'Pyro'
Pyro.NameServer --> PYRO:Pyro.NameServer@localhost:9090
--------END LIST - prefix 'Pyro'
```

2.7.5 Locating the Name Server and using it in your code

The name server is a Pyro object itself, and you access it through a normal Pyro proxy. The object exposed is *Pyro4.naming.NameServer*. Getting a proxy for the name server is done using the following function: *Pyro4.naming.locateNS()* (also available as *Pyro4.locateNS()*).

By far the easiest way to locate the Pyro name server is by using the broadcast lookup mechanism. This goes like this: you simply ask Pyro to look up the name server and return a proxy for it. It automatically figures out where in your subnet it is running by doing a broadcast and returning the first Pyro name server that responds. The broadcast is a simple UDP-network broadcast, so this means it usually won't travel outside your network subnet (or through routers) and your firewall needs to allow UDP network traffic.

There is a config item `BROADCAST_ADDRS` that contains a comma separated list of the broadcast addresses Pyro should use when doing a broadcast lookup. Depending on your network configuration, you may have to change this list to make the lookup work. It could be that you have to add the network broadcast address for the specific network that the name server is located on.

Note: You can only talk to a name server on a different machine if it didn't bind on localhost (that means you have to start it with an explicit host to bind on). The broadcast lookup mechanism only works in this case as well – it doesn't work with a name server that binds on localhost. For instance, the name server started as an example in *Starting the Name Server* was told to bind on the host name 'neptune' and it started a broadcast responder as well. If you use the default host (localhost) a broadcast responder will not be created.

Normally, all name server lookups are done this way. In code, it is simply calling the locator function without any arguments. If you want to circumvent the broadcast lookup (because you know the location of the server already, somehow) you can specify the hostname. As soon as you provide a specific hostname to the name server locator (by using a host argument to the `locateNS` call, or by setting the `NS_HOST` config item, etc) it will no longer use a broadcast too try to find the name server.

locateNS([*host=None*, *port=None*, *broadcast=True*, *hmac_key=key*])

Get a proxy for a name server somewhere in the network. If you're not providing host or port arguments, the configured defaults are used. Unless you specify a host, a broadcast lookup is done to search for a name server. (api reference: *Pyro4.naming.locateNS()*)

Parameters

- **host** – the hostname or ip address where the name server is running. Default is `None` which means it uses a network broadcast lookup. If you specify a host, no broadcast lookup is performed.
- **port** – the port number on which the name server is running. Default is `None` which means use the configured default. The exact meaning depends on whether the host parameter is given:
  - host parameter given: the port now means the actual name server port.
  - host parameter not given: the port now means the broadcast port.
- **broadcast** – should a broadcast be used to locate the name server, if no location is specified? Default is True.
- **hmac_key** – optional hmac key to use

2.7.6 The PYRONAME protocol type

To create a proxy and connect to a Pyro object, Pyro needs an URI so it can find the object. Because it is so convenient, the name server logic has been integrated into Pyro's URI mechanism by means of the special `PYRONAME` protocol type (rather than the normal `PYRO` protocol type). This protocol type tells Pyro to treat the URI as a logical object name instead, and Pyro will do a name server lookup automatically to get the actual object's URI. The form of a PYRONAME uri is very simple:

```
PYRONAME:some_logical_object_name
PYRONAME:some_logical_object_name@nshostname            # with optional host name
PYRONAME:some_logical_object_name@nshostname:nsport     # with optional host name +␣
↪port
```

where "some_logical_object_name" is the name of a registered Pyro object in the name server. When you also provide the `nshostname` and perhaps even `nsport` parts in the uri, you tell Pyro to look for the name server on that specific location (instead of relying on a broadcast lookup mechanism). (You can achieve more or less the same by setting the `NS_HOST` and `NS_PORT` config items)

All this means that instead of manually resolving objects like this:

```
nameserver=Pyro4.locateNS()
uri=nameserver.lookup("Department.BackupServer")
proxy=Pyro4.Proxy(uri)
proxy.backup()
```

you can write this instead:

```
proxy=Pyro4.Proxy("PYRONAME:Department.BackupServer")
proxy.backup()
```

An additional benefit of using a PYRONAME uri in a proxy is that the proxy isn't strictly tied to a specific object on a specific location. This is useful in scenarios where the server objects might move to another location, for instance when a disconnect/reconnect occurs. See the `autoreconnect` example for more details about this.

Note: Pyro has to do a lookup every time it needs to connect one of these PYRONAME uris. If you connect/disconnect many times or with many different objects, consider using PYRO uris (you can type them directly or create them by resolving as explained in the following paragraph) or call *`Pyro4.core.Proxy._pyroBind()`* on the proxy to bind it to a fixed PYRO uri instead.

2.7.7 The PYROMETA protocol type

Next to the `PYRONAME` protocol type there is another 'magic' protocol `PYROMETA`. This protocol type tells Pyro to treat the URI as metadata tags, and Pyro will ask the name server for any (randomly chosen) object that has the given metadata tags. The form of a PYROMETA uri is:

```
PYROMETA:metatag
PYROMETA:metatag1,metatag2,metatag3
PYROMETA:metatag@nshostname           # with optional host name
PYROMETA:metatag@nshostname:nsport    # with optional host name + port
```

So you can write this to connect to any random printer (given that all Pyro objects representing a printer have been registered in the name server with the `resource.printer` metadata tag):

```
proxy=Pyro4.Proxy("PYROMETA:resource.printer")
proxy.printstuff()
```

You have to explicitly add metadata tags when registering objects with the name server, see *Yellow-pages ability of the Name Server (metadata tags)*. Objects without metadata tags cannot be found via `PYROMETA` obviously. Note that the name server supports more advanced metadata features than what `PYROMETA` provides: in a PYROMETA uri you cannot use white spaces, and you cannot ask for an object that has one or more of the given tags – multiple tags means that the object must have all of them.

Metadata tags can be listed if you query the name server for registrations.

2.7.8 Resolving object names

'Resolving an object name' means to look it up in the name server's registry and getting the actual URI that belongs to it (with the actual object name or id and the location of the daemon in which it is running). This is not normally needed in user code (Pyro takes care of it automatically for you), but it can still be useful in certain situations.

So, resolving a logical name can be done in several ways:

1. The easiest way: let Pyro do it for you! Simply pass a `PYRONAME` URI to the proxy constructor, and forget all about the resolving happening under the hood:

```
obj = Pyro4.Proxy("PYRONAME:objectname")
obj.method()
```

2. obtain a name server proxy and use its `lookup` method (*Pyro4.naming.NameServer.lookup()*). You could then use this resolved uri to get an actual proxy, or do other things with it:

```
ns = Pyro4.locateNS()
uri = ns.lookup("objectname")
# uri now is the resolved 'objectname'
obj = Pyro4.Proxy(uri)
obj.method()
```

3. use a `PYRONAME` URI and resolve it using the `resolve` utility function *Pyro4.naming.resolve()* (also available as *Pyro4.resolve()*):

```
uri = Pyro4.resolve("PYRONAME:objectname")
# uri now is the resolved 'objectname'
obj = Pyro4.Proxy(uri)
obj.method()
```

4. use a `PYROMETA` URI and resolve it using the `resolve` utility function *Pyro4.naming.resolve()* (also available as *Pyro4.resolve()*):

```
uri = Pyro4.resolve("PYROMETA:metatag1,metatag2")
# uri is now randomly chosen from all objects having the given meta tags
obj = Pyro4.Proxy(uri)
```

2.7.9 Registering object names

'Registering an object' means that you associate the URI with a logical name, so that clients can refer to your Pyro object by using that name. Your server has to register its Pyro objects with the name server. It first registers an object with the Daemon, gets an URI back, and then registers that URI in the name server using the following method on the name server proxy:

register(*name*, *uri*, *safe=False*)
Registers an object (uri) under a logical name in the name server.

Parameters

- **name** (*string*) – logical name that the object will be known as
- **uri** (string or *Pyro4.core.URI*) – the URI of the object (you get it from the daemon)

- **safe** (*bool*) – normally registering the same name twice silently overwrites the old registration. If you set safe=True, the same name cannot be registered twice.

You can unregister objects as well using the `unregister()` method. The name server also supports automatically checking for registrations that are no longer available, for instance because the server process crashed or a network problem occurs. It will then automatically remove those registrations after a certain timeout period. This feature is disabled by default (it potentially requires the NS to periodically create a lot of network connections to check for each of the registrations if it is still available). You can enable it by setting the `NS_AUTOCLEAN` config item to a non zero value; it then specifies the recurring period in seconds for the nameserver to check all its registrations. Choose an appropriately large value, the minimum allowed is 3.

2.7.10 Free connections to the NS quickly

By default the Name server uses a Pyro socket server based on whatever configuration is the default. Usually that will be a threadpool based server with a limited pool size. If more clients connect to the name server than the pool size allows, they will get a connection error.

It is suggested you apply the following pattern when using the name server in your code:

1. obtain a proxy for the NS
2. look up the stuff you need, store it
3. free the NS proxy (See *Proxies, connections, threads and cleaning up*)
4. use the uri's/proxies you've just looked up

This makes sure your client code doesn't consume resources in the name server for an excessive amount of time, and more importantly, frees up the limited connection pool to let other clients get their turn. If you have a proxy to the name server and you let it live for too long, it may eventually deny other clients access to the name server because its connection pool is exhausted. So if you don't need the proxy anymore, make sure to free it up.

There are a number of things you can do to improve the matter on the side of the Name Server itself. You can control its behavior by setting certain Pyro config items before starting the server:

- You can set `SERVERTYPE=multiplex` to create a server that doesn't use a limited connection (thread) pool, but multiplexes as many connections as the system allows. However, the actual calls to the server must now wait on eachother to complete before the next call is processed. This may impact performance in other ways.
- You can set `THREADPOOL_SIZE` to an even larger number than the default.
- You can set `COMMTIMEOUT` to a certain value, which frees up unused connections after the given time. But the client code may now crash with a TimeoutError or ConnectionClosedError when it tries to use a proxy it obtained earlier. (You can use Pyro's autoreconnect feature to work around this but it makes the code more complex)

2.7.11 Using the name server with pickle, cloudpickle or dill serializers

If you find yourself in the unfortunate situation where you absolutely have to use the pickle, cloudpickle or dill serializers, you have to pay attention when also using the name server. Because these serializers are disabled by default, the name server will not reply to messages from clients that are using them, unless you enable them in the name server as well.

The symptoms are usually that your client code seems unable to contact the name server:

```
Pyro4.errors.NamingError: Failed to locate the nameserver
```

The name server will show a user warning message on the console:

```
Pyro protocol error occurred: message used serializer that is not accepted
```

And if you enable logging for the name server you will likely see in its logfile:

```
accepted serializers: {'json', 'marshal', 'serpent'}
...
...
Pyro4.errors.ProtocolError: message used serializer that is not accepted: [4,5]
```

The way to solve this is to stop using the these serializers, or if you must use them, tell the name server that it is okay to accept them. You do that by setting the `SERIALIZERS_ACCEPTED` config item to a set of serializers that includes them, and then restart the name server. For instance:

```
$ export PYRO_SERIALIZERS_ACCEPTED=serpent,json,marshal,pickle,cloudpickle,dill
$ pyro4-ns
```

If you enable logging you will then see that the name server says that pickle, cloudpickle and dill are among the accepted serializers.

2.7.12 Yellow-pages ability of the Name Server (metadata tags)

Since Pyro 4.40, it is possible to tag object registrations in the name server with one or more Metadata tags. These are simple strings but you're free to put anything you want in it. One way of using it, is to provide a form of Yellow-pages object lookup: instead of directly asking for the registered object by its unique name (telephone book), you're asking for any registration from a certain *category*. You get back a list of registered objects from the queried category, from which you can then choose the one you want.

Note: Metadata tags are case-sensitive.

As an example, imagine the following objects registered in the name server (with the metadata as shown):

Name	Uri	Metadata
printer.secondfloor	PYRO:printer1@host:1234	printer
printer.hallway	PYRO:printer2@host:1234	printer
storage.diskcluster	PYRO:disks1@host:1234	storage
storage.ssdcluster	PYRO:disks2@host:1234	storage

Instead of having to know the exact name of a required object you can query the name server for all objects having a certain set of metadata. So in the above case, your client code doesn't have to 'know' that it needs to lookup the `printer.hallway` object to get the uri of a printer (in this case the one down in the hallway). Instead it can just ask for a list of all objects having the `printer` metadata tag. It will get a list containing both `printer.secondfloor` and `printer.hallway` so you will still have to choose the object you want to use - or perhaps even use both. The objects tagged with `storage` won't be returned.

Arguably the most useful way to deal with the metadata is to use it for Yellow-pages style lookups. You can ask for all objects having some set of metadata tags, where you can choose if they should have *all* of the given tags or only *any one* (or more) of the given tags. Additional or other filtering must be done in the client code itself. So in the above example, querying with `metadata_any={'printer', 'storage'}` will return all four objects, while querying with `metadata_all={'printer', 'storage'}` will return an empty list (because there are no objects that are both a printer and storage).

Setting metadata in the name server

Object registrations in the name server by default have an empty set of metadata tags associated with them. However the `register` method (*Pyro4.naming.NameServer.register()*) has an optional `metadata` argument, you can set that to a set of strings that will be the metadata tags associated with the object registration. For instance:

```
ns.register("printer.secondfloor", "PYRO:printer1@host:1234", metadata={"printer"})
```

Getting metadata back from the name server

The `lookup` (*Pyro4.naming.NameServer.lookup()*) and `list` (*Pyro4.naming.NameServer.list()*) methods of the name server have an optional `return_metadata` argument. By default it is False, and you just get back the registered URI (lookup) or a dictionary with the registered names and their URI as values (list). If you set it to True however, you'll get back tuples instead: (uri, set-of-metadata-tags):

```
ns.lookup("printer.secondfloor", return_metadata=True)
# returns: (<Pyro4.core.URI at 0x6211e0, PYRO:printer1@host:1234>, {'printer'})

ns.list(return_metadata=True)
# returns something like:
#   {'printer.secondfloor': ('PYRO:printer1@host:1234', {'printer'}),
#    'Pyro.NameServer': ('PYRO:Pyro.NameServer@localhost:9090', {'class:Pyro4.naming.
↪NameServer'})}
# (as you can see the name server itself has also been registered with a metadata tag)
```

Querying on metadata (Yellow-page lookup)

You can ask the name server to list all objects having some set of metadata tags. The `list` (*Pyro4.naming.NameServer.list()*) method of the name server has two optional arguments to allow you do do this: `metadata_all` and `metadata_any`.

1. `metadata_all`: give all objects having *all* of the given metadata tags:

   ```
   ns.list(metadata_all={"printer"})
   # returns: {'printer.secondfloor': 'PYRO:printer1@host:1234'}
   ns.list(metadata_all={"printer", "communication"})
   # returns: {}   (there is no object that's both a printer and a communication␣
   ↪device)
   ```

2. `metadata_any`: give all objects having *one* (or more) of the given metadata tags:

   ```
   ns.list(metadata_any={"storage", "printer", "communication"})
   # returns: {'printer.secondfloor': 'PYRO:printer1@host:1234'}
   ```

Querying on metadata via ''PYROMETA'' uri (Yellow-page lookup in uri)

As a convenience, similar to the `PYRONAME` uri protocol, you can use the `PYROMETA` uri protocol to let Pyro do the lookup for you. It only supports `metadata_all` lookup, but it allows you to conveniently get a proxy like this:

```
Pyro4.Proxy("PYROMETA:resource.printer,performance.fast")
```

this will connect to a (randomly chosen) object with both the `resource.printer` and `performance.fast` metadata tags. Also see *The PYROMETA protocol type*.

You can find some code that uses the metadata API in the `ns-metadata` example. Note that the `nsc` tool (*Name server control tool*) also allows you to manipulate the metadata in the name server from the command line.

2.7.13 Other methods in the Name Server API

The name server has a few other methods that might be useful at times. For instance, you can ask it for a list of all registered objects. Because the name server itself is a regular Pyro object, you can access its methods through a regular Pyro proxy, and refer to the description of the exposed class to see what methods are available: *Pyro4.naming.NameServer*.

2.8 Security

Warning: Do not publish any Pyro objects to remote machines unless you've read and understood everything that is discussed in this chapter. This is also true when publishing Pyro objects with different credentials to other processes on the same machine. Why? In short: using Pyro has several security risks. Pyro has a few countermeasures to deal with them. Understanding the risks, the countermeasures, and their limits, is very important to avoid creating systems that are very easy to compromise by malicious entities.

2.8.1 Pickle, cloudpickle and dill as serialization formats (optional)

When configured to do so, Pyro is able to use the `pickle`, `cloudpickle` or `dill` modules to serialize objects and then sends them over the network. It is well known that using these serializers for this purpose is a security risk. The main problem is that allowing a program to deserialize this type of serialized data can cause arbitrary code execution and this may wreck or compromise your system. Because of this the default serializer is serpent, which doesn't have this security problem. Some other means to enhance security are discussed below.

2.8.2 Network interface binding

By default Pyro binds every server on localhost, to avoid exposing things on a public network or over the internet by mistake. If you want to expose your Pyro objects to anything other than localhost, you have to explicitly tell Pyro the network interface address it should use. This means it is a conscious effort to expose Pyro objects to other machines.

It is possible to tell Pyro the interface address via an environment variable or global config item (`HOST`). In some situations - or if you're paranoid - it is advisable to override this setting in your server program by setting the config item from within your own code, instead of depending on an externally configured setting.

2.8.3 Running Pyro servers with different credentials/user id

The following is not a Pyro specific problem, but is important nonetheless: If you want to run your Pyro server as a different user id or with different credentials as regular users, *be very careful* what kind of Pyro objects you expose like this!

Treat this situation as if you're exposing your server on the internet (even when it's only running on localhost). Keep in mind that it is still possible that a random user on the same machine connects to the local server. You may need additional security measures to prevent random users from calling your Pyro objects.

2.8.4 Secure communication via SSL/TLS

Pyro itself doesn't encrypt the data it sends over the network. This means if you use the default configuration, you must never transfer sensitive data on untrusted networks (especially user data, passwords, and such) because eavesdropping is possible.

You can run Pyro over a secure network (VPN, ssl/ssh tunnel) where the encryption is taken care of externally. It is also possible however to enable SSL/TLS in Pyro itself, so that all communication is secured via this industry standard that provides encryption, authentication, and anti-tampering (message integrity).

Using SSL/TLS

Enable it by setting the `SSL` config item to True, and configure the other SSL config items as required. You'll need to specify the cert files to use, private keys, and passwords if any. By default, the SSL mode only has a cert on the server (which is similar to visiting a https url in your browser). This means your *clients* can be sure that they are connecting to the expected server, but the *server* has no way to know what clients are connecting. You can solve this by using a HMAC key (see *... by using a HMAC signature via a shared private key*), but if you're already using SSL, a better way is to do custom certificate verification. You can do this in your client (checks the server's cert) but you can also tell your clients to use certs as well and check these in your server. This makes it 2-way-SSL or mutual authentication. For more details see here *... by using 2-way-SSL and certificate verificiation*. The SSL config items are in *Overview of Config Items*.

For example code on how to set up a 2-way-SSL Pyro client and server, with cert verification, see the `ssl` example.

Important: You must use at least Python 2.7.11 / 3.4.4 or newer for proper SSL support.

2.8.5 Dotted names (object traversal)

Using dotted names on Pyro proxies (such as `proxy.aaa.bbb.ccc()`) is not possible in Pyro, because it is a security vulnerability (for similar reasons as described here http://www.python.org/news/security/PSF-2005-001/).

2.8.6 Environment variables overriding config items

Almost all config items can be overwritten by an environment variable. If you can't trust the environment in which your script is running, it may be a good idea to reset the config items to their default builtin values, without using any environment variables. See *Configuring Pyro* for the proper way to do this.

2.8.7 Preventing arbitrary connections

... by using a HMAC signature via a shared private key

You can use a HMAC signature on every network transfer to prevent malicious requests. The idea is to only have legit clients connect to your Pyro server. Using the HMAC signature ensures that only clients with the correct secret key can create valid requests, and that it is impossible to modify valid requests (even though the network data is not encrypted). The hashing algorithm that is used in the HMAC is SHA-1.

consider alternatives

For industry standard encryption and connection verification, consider using SSL/TLS instead.

You need to create and configure a secure shared key yourself. The key is a byte string and must be cryptographically secure (there are various methods to create such a key). Your server needs to set this key and every client that wants to connect to it also needs to set it. You can set the shared key via the `_pyroHmacKey` property on a proxy or a daemon:

```
daemon._pyroHmacKey = b"secretkey"
proxy._pyroHmacKey = b"secretkey"
```

Warning: It is hard to keep a shared secret key actually secret! People might read the source code of your software and extract the key from it. Pyro itself provides no facilities to help you with this, sorry. The Diffie-Hellman Key Exchange algorithm is one example of a secure solution to this problem. There's the `diffie-hellman` example that shows the basics, but DO NOT use it directly as being "the secure way to do this" – it's only demo code.

. . . by using 2-way-SSL and certificate verificiation

When using SSL, you should also do some custom certificate verification, such as checking the serial number and commonName. This way your code is not only certain that the communication is encrypted, but also that it is talking to the intended party and nobody else (middleman). The server hostname and cert expiration dates *are* checked automatically, but other attributes you have to verify yourself.

This is fairly easy to do: you can use *Connection handshake* for this. You can then get the peer certificate using `Pyro4.socketutil.SocketConnection.getpeercert()`.

If you configure a client cert as well as a server cert, you can/should also do verification of client certificates in your server. This is a good way to be absolutely certain that you only allow clients that you know and trust, because you can check the required unique certificate attributes.

Having certs on both client and server is called 2-way-SSL or mutual authentication.

It's a bit too involved to fully describe here but it not much harder than the basic SSL configuration described earlier. You just have to make sure you supply a client certificate and that the server requires a client certificate (and verifies some properties of it). The `ssl` example shows how to do all this.

2.9 Exceptions and remote tracebacks

There is an example that shows various ways to deal with exceptions when writing Pyro code. Have a look at the `exceptions` example in the `examples` directory.

2.9.1 Pyro exceptions

Pyro's exception classes can be found in *`Pyro4.errors`*. They are used by Pyro if something went wrong inside Pyro itself or related to something Pyro was doing.

2.9.2 Remote exceptions

More interesting are the exceptions that occur in *your own* objects (the remote Pyro objects). Pyro is doing its best to make the remote objects appear as normal, local, Python objects. That also means that if they raise an error, Pyro will make it appear in the caller, as if the error occurred locally.

Say you have a remote object that can divide arbitrary numbers. It will probably raise a `ZeroDivisionError` when you supply `0` as the divisor. This can be dealt with as follows:

```
import Pyro4

divider=Pyro4.Proxy( ... )
try:
    result = divider.div(999,0)
except ZeroDivisionError:
    print("yup, it crashed")
```

Just catch the exception as if you were writing code that deals with normal objects.

But, since the error occurred in a *remote* object, and Pyro itself raises it again on the client side, you lose some information: the actual traceback of the error at the time it occurred in the server. Pyro fixes this because it stores the traceback information on a special attribute on the exception object (`_pyroTraceback`). The traceback is stored as a list of strings (each is a line from the traceback text, including newlines). You can use this data on the client to print or process the traceback text from the exception as it occurred in the Pyro object on the server.

There is a utility function in *Pyro4.util* to make it easy to deal with this: *Pyro4.util.getPyroTraceback()*

You use it like this:

```
import Pyro4.util
try:
    result = proxy.method()
except Exception:
    print("Pyro traceback:")
    print("".join(Pyro4.util.getPyroTraceback()))
```

Also, there is another function that you can install in `sys.excepthook`, if you want Python to automatically print the complete Pyro traceback including the remote traceback, if any: *Pyro4.util.excepthook()*

A full Pyro exception traceback, including the remote traceback on the server, looks something like this:

```
Traceback (most recent call last):
  File "client.py", line 50, in <module>
    print(test.complexerror())      # due to the excepthook, the exception will show␣
↪the pyro error
  File "E:\Projects\Pyro4\src\Pyro4\core.py", line 130, in __call__
    return self.__send(self.__name, args, kwargs)
  File "E:\Projects\Pyro4\src\Pyro4\core.py", line 242, in _pyroInvoke
    raise data
TypeError: unsupported operand type(s) for //: 'str' and 'int'
 +--- This exception occured remotely (Pyro) - Remote traceback:
 | Traceback (most recent call last):
 |   File "E:\Projects\Pyro4\src\Pyro4\core.py", line 760, in handleRequest
 |     data=method(*vargs, **kwargs)   # this is the actual method call to the Pyro␣
↪object
 |   File "E:\projects\Pyro4\examples\exceptions\excep.py", line 17, in complexerror
 |     x.crash()
 |   File "E:\projects\Pyro4\examples\exceptions\excep.py", line 22, in crash
 |     s.crash2('going down...')
 |   File "E:\projects\Pyro4\examples\exceptions\excep.py", line 25, in crash2
 |     x=arg//2
 | TypeError: unsupported operand type(s) for //: 'str' and 'int'
 +--- End of remote traceback
```

As you can see, the first part is only the exception as it occurs locally on the client (raised by Pyro). The indented part marked with 'Remote traceback' is the exception as it occurred in the remote Pyro object.

2.9.3 Detailed traceback information

There is another utility that Pyro has to make it easier to debug remote object exceptions. If you enable the DETAILED_TRACEBACK config item on the server (see *Overview of Config Items*), the remote traceback is extended with details of the values of the local variables in every frame:

```
+--- This exception occured remotely (Pyro) - Remote traceback:
| ----------------------------------------------------
|  EXCEPTION <type 'exceptions.TypeError'>: unsupported operand type(s) for //: 'str'␣
↪and 'int'
|  Extended stacktrace follows (most recent call last)
| ----------------------------------------------------
| File "E:\Projects\Pyro4\src\Pyro4\core.py", line 760, in Daemon.handleRequest
| Source code:
|     data=method(*vargs, **kwargs)   # this is the actual method call to the Pyro␣
↪object
| ----------------------------------------------------
| File "E:\projects\Pyro4\examples\exceptions\excep.py", line 17, in TestClass.
↪complexerror
| Source code:
|     x.crash()
| Local values:
|     self = <excep.TestClass object at 0x02392830>
|         self._pyroDaemon = <Pyro4.core.Daemon object at 0x02392330>
|         self._pyroId = 'obj_c63d47dd140f44dca8782151643e0c55'
|     x = <excep.Foo object at 0x023929D0>
| ----------------------------------------------------
| File "E:\projects\Pyro4\examples\exceptions\excep.py", line 22, in Foo.crash
| Source code:
|     self.crash2('going down...')
| Local values:
|     self = <excep.Foo object at 0x023929D0>
| ----------------------------------------------------
| File "E:\projects\Pyro4\examples\exceptions\excep.py", line 25, in Foo.crash2
| Source code:
|     x=arg//2
| Local values:
|     arg = 'going down...'
|     self = <excep.Foo object at 0x023929D0>
| ----------------------------------------------------
|  EXCEPTION <type 'exceptions.TypeError'>: unsupported operand type(s) for //: 'str'␣
↪and 'int'
| ----------------------------------------------------
+--- End of remote traceback
```

You can immediately see why the call produced a TypeError without the need to have a debugger running (the arg variable is a string and dividing that string by 2 is the cause of the error).

Of course it is also possible to enable DETAILED_TRACEBACK on the client, but it is not as useful there (normally it is no problem to run the client code inside a debugger).

2.10 Flame: Foreign Location Automatic Module Exposer

Pyro Flame is an easy way of exposing remote modules and builtins, and even a remote interactive Python console. It is available since Pyro 4.10. With Flame, you don't need to write any server side code anymore, and still be able to call objects, modules and other things on the remote machine. Flame does this by giving a client direct access to any module or builtin that is available on the remote machine.

Flame can be found in the *Pyro4.utils.flame* module.

> **Warning:** Be very sure about what you are doing before enabling Flame.
>
> Flame is disabled by default. You need to explicitly set a config item to true, and start a Flame server yourself, to make it available. This is because it allows client programs full access to *everything* on your system. Only use it if you fully trust your environment and the clients that can connect to your machines.
>
> (Flame is also symbolic for burning server machines that got totally owned by malicious clients.)

2.10.1 Enabling Flame

Flame is actually a special Pyro object that is exposed via a normal Pyro daemon. You need to start it explicitly in your daemon. This is done by calling a utility function with your daemon that you want to enable flame on:

```
import Pyro4.utils.flame
Pyro4.utils.flame.start(daemon)
```

Additionally, you have to make some configuration changes:

- flame server: set the `FLAME_ENABLED` config item to True
- flame server: set the `SERIALIZERS_ACCEPTED` config item to `{"pickle"}`
- flame client: set the `SERIALIZER` config item to `pickle`

You'll have to explicitly enable Flame. When you don't, you'll get an error when trying to start Flame. The config item is False by default to avoid unintentionally running Flame servers. Also, Flame requires the pickle serializer. It doesn't work when using one of the secure serializers, because it needs to be able to transfer custom python objects.

2.10.2 Command line server

There's a little command line server program that will launch a flame enabled Pyro daemon, to avoid the hassle of having to write a custom server program yourself everywhere you want to provide a Flame server:

`python -m Pyro4.utils.flameserver` (or simply: **`pyro4-flameserver`**)

The command line arguments are similar to the echo server (see *Test echo server*). Use `-h` to make it print a short help text. For the command line server you'll also have to set the `FLAME_ENABLED` config item to True, otherwise you'll get an error when trying to start it. Because we're talking about command line clients, the most convenient way to do so is probably by setting the environment variable in your shell: `PYRO_FLAME_ENABLED=true`.

2.10.3 Flame object and examples

A Flame server exposes a `"Pyro.Flame"` object (you can hardcode this name or use the constant *Pyro4.constants.FLAME_NAME*). Its interface is described in the API documentation, see *Pyro4.utils.flame.Flame*.

Connecting to the flame server can be done as usual (by creating a Pyro proxy yourself) or by using the convenience function *Pyro4.utils.flame.connect()*. It creates a Pyro proxy to the flame server for you using a slightly simpler interface. (If you started the flame server with a hmac key, you also must provide the `hmac_key` parameter here.)

A little example follows. You have to have running flame server, and then you can write a client like this:

```
import Pyro4.utils.flame

Pyro4.config.SERIALIZER = "pickle"    # flame requires pickle serializer

flame = Pyro4.utils.flame.connect("hostname:9999")    # or whatever the server runs at

socketmodule = flame.module("socket")
osmodule = flame.module("os")
print("remote host name=", socketmodule.gethostname())
print("remote server directory contents=", osmodule.listdir("."))

flame.execute("import math")
root = flame.evaluate("math.sqrt(500)")
print("calculated square root=", root)
print("remote exceptions also work", flame.evaluate("1//0"))

# print something on the remote std output
flame.builtin("print")("Hello there, remote server stdout!")
```

A remote interactive console can be started like this:

```
with flame.console() as console:
    console.interact()
    # ... you can repeat sessions if you want
```

... which will print something like:

```
Python 2.7.2 (default, Jun 12 2011, 20:46:48)
[GCC 4.2.1 (Apple Inc. build 5577)] on darwin
(Remote console on charon:9999)
>>> # type stuff here and it gets executed on the remote machine
>>> import socket
>>> socket.gethostname()
'charon.local'
>>> ^D
(Remote session ended)
```

Note: The `getfile` and `sendfile` functions can be used for *very* basic file transfer.

The `getmodule` and `sendmodule` functions can be used to send module source files to other machines so it is possible to execute code that wasn't available before. This is a *very* experimental replacement of the mobile code feature that Pyro 3.x had. It also is a very easy way of totally owning the server because you can make it execute anything you like. Be very careful.

Note: The remote module proxy that Flame provides does *not* support direct attribute access. For instance, you cannot do the following:

```
flame = Pyro4.utils.flame.connect("....")
ros = flame.module("os")
print(ros.name)      #  doesn't work as you might expect
```

The `flame.evaluate` method (*Pyro4.utils.flame.Flame.evaluate()*) provides an alternative though:

```
print(flame.evaluate("os.name"))      # properly prints the os.name of the remote␣
↪server
```

Note: *Pyrolite - client library for Java and .NET* also supports convenient access to a Pyro Flame server. This includes the remote interactive console.

See the `flame` example for example code including uploading module source code to the server.

2.11 Tips & Tricks

2.11.1 Best practices

Avoid using insecure features.

Avoid using the `pickle` (and `dill`, and `cloudpickle`) serializers, they will make your solution insecure. Avoid using Flame (it requires pickle, but has severe security implications by itself).

Make as little as possible remotely accessible.

Avoid sticking a `@expose` on the whole class, and instead mark only those methods exposed that you really want to be remotely accessible. Alternatively, make sure your exposed Pyro server class only consists of methods that are okay to be accessed remotely.

Avoid circular communication topologies.

When you can have a circular communication pattern in your system (A–>B–>C–>A) this can cause some problems:

- when reusing a proxy it causes a deadlock because the proxy is already being used for an active remote call. See the `deadlock` example.
- with the multiplex servertype, the server itself may also block for all other remote calls because the handling of the first is not yet completed.

Avoid circularity, or use *oneway* method calls on at least one of the links in the chain. Another possible way out of a lock situation is to set `COMMTIMEOUT` so that after a certain period in a locking situation the caller aborts with a TimeoutError, effectively breaking the deadlock.

'After X simultaneous proxy connections, Pyro seems to freeze!' Fix: Release your proxies when you can.

A connected proxy that is unused takes up resources on the server. In the case of the threadpool server type, it locks up a single thread. If you have too many connected proxies at the same time, the server may run out of threads and won't be able to accept new connections.

You can use the `THREADPOOL_SIZE` config item to increase the maximum number of threads that Pyro will use. Or use the multiplex server instead, which doesn't have this limitation. Another option is to set `COMMTIMEOUT` to a certain value *on your server*, which will free up unused connections after the given time. But your client code may now crash with a TimeoutError or ConnectionClosedError when it tries to use a proxy that worked earlier. You can use Pyro's autoreconnect feature to work around this but it makes the code more complex.

It is however advised to close (release) proxies that your program no longer needs, to free resources both in the client and in the server. Don't worry about reconnecting, Pyro does that automatically for you once the proxy is used again. You can use explicit `_pyroRelease` calls or use the proxy from within a context manager. It's not a good idea to release it after every single remote method call though, because then the cost of reconnecting the socket can be bad for performance.

Avoid large binary blobs over the wire.

Pyro is not designed to efficiently transfer large amounts of binary data over the network. Try to find another protocol that better suits this requirement. Read *Binary data transfer / file transfer* for some more details about this. How to deal with Numpy data (large *or* small) is explained here *Pyro and Numpy*.

Note that Pyro has a 2 gigabyte message size limitation at this time.

Minimize object graphs that travel over the wire.

Pyro will serialize the whole object graph you're passing, even when only a tiny fraction of it is used on the receiving end. Be aware of this: it may be necessary to define special lightweight objects for your Pyro interfaces that hold the data you need, rather than passing a huge object structure. It's good design practice as well to have an "external API" that is different from your internal code, and tuned for minimal communication overhead or complexity.

Consider using basic data types instead of custom classes.

Because Pyro serializes the objects you're passing, it needs to know how to serialize custom types. While you can teach Pyro about these (see *Changing the way your custom classes are (de)serialized*) it may sometimes be easier to just use a builtin datatype instead. For instance if you have a custom class whose state essentially is a set of numbers, consider then that it may be easier to just transfer a `set` or a `list` of those numbers rather than an instance of your custom class. It depends on your class and data of course, and whether the receiving code expects just the list of numbers or really needs an instance of your custom class.

2.11.2 Logging

If you configure it (see *Overview of Config Items*) Pyro will write a bit of debug information, errors, and notifications to a log file. It uses Python's standard `logging` module for this (See https://docs.python.org/2/library/logging.html

). Once enabled, your own program code could use Pyro's logging setup as well. But if you want to configure your own logging, make sure you do that before any Pyro imports. Then Pyro will skip its own autoconfig.

A little example to enable logging by setting the required environment variables from the shell:

```
$ export PYRO_LOGFILE=pyro.log
$ export PYRO_LOGLEVEL=DEBUG
$ python my_pyro_program.py
```

Another way is by modifiying `os.environ` from within your code itself, *before* any import of Pyro4 is done:

```
import os
os.environ["PYRO_LOGFILE"] = "pyro.log"
os.environ["PYRO_LOGLEVEL"] = "DEBUG"

import Pyro4
# do stuff...
```

Finally, it is possible to initialize the logging by means of the standard Python `logging` module only, but then you still have to tell Pyro4 what log level it should use (or it won't log anything):

```
import logging
logging.basicConfig()  # or your own sophisticated setup
logging.getLogger("Pyro4").setLevel(logging.DEBUG)
logging.getLogger("Pyro4.core").setLevel(logging.DEBUG)
# ... set level of other logger names as desired ...

import Pyro4
# do stuff...
```

The various logger names are similar to the module that uses the logger, so for instance logging done by code in `Pyro4.core` will use a logger category name of `Pyro4.core`. Look at the top of the source code of the various modules from Pyro to see what the exact names are.

2.11.3 Multiple network interfaces

This is a difficult subject but here are a few short notes about it. *At this time, Pyro doesn't support running on multiple network interfaces at the same time*. You can bind a deamon on INADDR_ANY (0.0.0.0) though, including the name server. But weird things happen with the URIs of objects published through these servers, because they will point to 0.0.0.0 and your clients won't be able to connect to the actual objects.

The name server however contains a little trick. The broadcast responder can also be bound on 0.0.0.0 and it will in fact try to determine the correct ip address of the interface that a client needs to use to contact the name server on. So while you cannot run Pyro daemons on 0.0.0.0 (to respond to requests from all possible interfaces), sometimes it is possible to run only the name server on 0.0.0.0. The success ratio of all this depends heavily on your network setup.

2.11.4 Same major Python version required when using pickle, cloudpickle, dill or marshal

When Pyro is configured to use pickle, cloudpickle, dill or marshal as its serialization format, it is required to have the same *major* Python versions on your clients and your servers. Otherwise the different parties cannot decipher each others serialized data. This means you cannot let Python 2.x talk to Python 3.x with Pyro when using these serializers. However it should be fine to have Python 3.5 talk to Python 3.6 for instance. It may still be required to specify the pickle or dill protocol version though, because that needs to be the same on both ends as well. For instance, Python 3.4 introduced version 4 of the pickle protocol and as such won't be able to talk to Python 3.3 which is stuck on version 3

pickle protocol. You'll have to tell the Python 3.4 side to step down to protocol 3. There is a config item for that. The same will apply for dill protocol versions. If you are using cloudpickle, you can just set the pickle protocol version (as pickle is used under the hood).

The implementation independent serialization protocols serpent and json don't have these limitations.

2.11.5 Wire protocol version

Here is a little tip to find out what wire protocol version a given Pyro server is using. This could be useful if you are getting `ProtocolError: invalid data or unsupported protocol version` or something like that. It also works with Pyro 3.x.

Server

This is a way to figure out the protocol version number a given Pyro server is using: by reading the first 6 bytes from the server socket connection. The Pyro daemon will respond with a 4-byte string "`PYRO`" followed by a 2-byte number that is the protocol version used:

```
$ nc <pyroservername> <pyroserverport> | od -N 6 -t x1c
0000000  50  59  52  4f  00  05
          P   Y   R   O  \0 005
```

This one is talking protocol version `00 05` (5). This low number means it is a Pyro 3.x server. When you try it on a Pyro 4 server:

```
$ nc <pyroservername> <pyroserverport> | od -N 6 -t x1c
0000000  50  59  52  4f  00  2c
          P   Y   R   O  \0   ,
```

This one is talking protocol version `00 2c` (44). For Pyro4 the protocol version started at 40 for the first release and is now at 46 for the current release at the time of writing.

Client

To find out the protocol version that your client code is using, you can use this:

```
$ python -c "import Pyro4.constants as c; print(c.PROTOCOL_VERSION)"
```

2.11.6 Asynchronous ('future') normal function calls

Pyro provides an asynchronous proxy to call remote methods asynchronously, see *Asynchronous ('future') remote calls & call chains*. For normal Python code, Python provides a similar mechanism in the form of the *`Pyro4.futures.Future`* class (also available as `Pyro4.Future`). With a syntax that is slightly different from normal method calls, it provides the same asynchronous function calls as the asynchronous proxy has. Note that Python itself has a similar thing in the standard library since version 3.2, see http://docs.python.org/3/library/concurrent.futures.html#future-objects . However Pyro's Future object is available on older Python versions too. It works slightly differently and perhaps a little bit easier as well.

You create a `Future` object for a callable that you want to execute in the background, and receive its results somewhere in the future:

```
def add(x,y):
    return x+y

futurecall = Pyro4.Future(add)
result = futurecall(4,5)
```

(continues on next page)

(continued from previous page)

```
# do some other stuff... then access the value
summation = result.value
```

Actually calling the *Future* object returns control immediately and results in a *Pyro4.futures.FutureResult* object. This is the exact same class as with the asynchrnous proxy. The most important attributes are `value`, `ready` and the `wait` method. See *Asynchronous ('future') remote calls & call chains* for more details.

You can also chain multiple calls, so that the whole call chain is executed sequentially in the background. You can do this directly on the `Future` object, with the *Pyro4.futures.Future.then()* method. It has the same signature as the `then` method from the `FutureResult` class:

```
futurecall = Pyro4.Future(something) \
    .then(somethingelse, 44) \
    .then(lastthing, optionalargument="something")
```

There's also a *Pyro4.futures.Future.iferror()* method that allows you to register a callback to be invoked when an exception occurs. This method also exists on the `FutureResult` class. See the `futures` example for more details and example code.

You can delay the execution of the future for a number of seconds via the *Pyro4.futures.Future.delay()* method, and you can cancel it altogether via the *Pyro4.futures.Future.cancel()* method (which only works if the future hasn't been evaluated yet).

2.11.7 DNS setup

Pyro depends on a working DNS configuration, at least for your local hostname (i.e. 'pinging' your local hostname should work). If your local hostname doesn't resolve to an IP address, you'll have to fix this. This can usually be done by adding an entry to the hosts file. For OpenSUSE, you can also use Yast to fix it (go to Network Settings, enable "Assign hostname to loopback IP").

If Pyro detects a problem with the dns setup it will log a WARNING in the logfile (if logging is enabled), something like: `weird DNS setup: your-computer-hostname resolves to localhost (127.x.x.x)`

2.11.8 Pyro behind a NAT router/firewall

You can run Pyro behind a NAT router/firewall. Assume the external hostname is 'pyro.server.com' and the external port is 5555. Also assume the internal host is 'server1.lan' and the internal port is 9999. You'll need to have a NAT rule that maps pyro.server.com:5555 to server1.lan:9999. You'll need to start your Pyro daemon, where you specify the `nathost` and `natport` arguments, so that Pyro knows it needs to 'publish' URIs containing that *external* location instead of just using the internal addresses:

```
# running on server1.lan
d = Pyro4.Daemon(port=9999, nathost="pyro.server.com", natport=5555)
uri = d.register(Something, "thing")
print(uri)      # "PYRO:thing@pyro.server.com:5555"
```

As you see, the URI now contains the external address.

Pyro4.core.Daemon.uriFor() by default returns URIs with a NAT address in it (if `nathost` and `natport` were used). You can override this by setting `nat=False`:

```
# d = Pyro4.Daemon(...)
print(d.uriFor("thing"))                  # "PYRO:thing@pyro.server.com:5555"
```

(continues on next page)

(continued from previous page)

```
print(d.uriFor("thing", nat=False))      # "PYRO:thing@localhost:36124"
uri2 = d.uriFor(uri.object, nat=False)   # get non-natted uri
```

The Name server can also be started behind a NAT: it has a couple of command line options that allow you to specify a nathost and natport for it. See *Starting the Name Server*.

Note: The broadcast responder always returns the internal address, never the external NAT address. Also, the name server itself won't translate any URIs that are registered with it. So if you want it to publish URIs with 'external' locations in them, you have to tell the Daemon that registers these URIs to use the correct nathost and natport as well.

Note: In some situations the NAT simply is configured to pass through any port one-to-one to another host behind the NAT router/firewall. Pyro facilitates this by allowing you to set the natport to 0, in which case Pyro will replace it by the internal port number.

2.11.9 'Failed to locate the nameserver' or 'Connection refused' error, what now?

Usually when you get an error like "failed to locate the name server" or "connection refused" it is because there is a configuration problem in your network setup, such as a firewall blocking certain network connections. Sometimes it can be because you configured Pyro wrong. A checklist to follow to diagnose your issue can be as follows:

- is the name server on a network interface that is visible on the network? If it's on localhost, then it's definitely not! (check the URI)
- is the Pyro object's daemon on a network interface that is visible on the network? If it's on localhost, then it's definitely not! (check the URI)
- with what URI is the Pyro object registered in the Name server? See previous item.
- can you ping the server from your client machine?
- can you telnet to the given host+port from your client machine?
- dealing with IPV4 versus IPV6: do both client and server use the same protocol?
- is the server's ip address as shown one of an externally reachable network interface?
- do you have your server behind a NAT router? See *Pyro behind a NAT router/firewall*.
- do you have a firewall or packetfilter running that prevents the connection?
- do you have the same Pyro versions on both server and client?
- what does the pyro logfiles tell you (enable it via the config items on both the server and the client, including the name server. See *Logging*.
- (if not using the default:) do you have a compatible serializer configuration?
- (if not using the default:) do you have a symmetric hmac key configuration?
- can you obtain a few bytes from the wire using netcat, see *Wire protocol version*.

2.11.10 Binary data transfer / file transfer

. . . if you do want to use Pyro for this. . .

At the end of this paragraph, a few alternative approaches of reasonably efficient binary data transfer are presented, where (almost) all of the code still uses just Pyro's high level abstractions.

Pyro is not meant to transfer large amounts of binary data (images, sound files, video clips): the protocol is not designed nor optimized for these kinds of data. The occasional transmission of such data is fine (*Flame: Foreign Location Automatic Module Exposer* even provides a convenience method for that, if you like: `Pyro4.utils.flame.Flame.sendfile()`) but if you're dealing with a lot of them or with big files, it is usually better to use something else to do the actual data transfer (file share+file copy, ftp, http, scp, rsync).

Also, Pyro has a 2 gigabyte message size limitation at this time (if your Python implementation and system memory even allow the process to reach this size). You can avoid this problem if you use the remote iterator feature (return chunks via an iterator or generator function and consume them on demand in your client).

Note: Serpent and binary data: If you do transfer binary data using the serpent serializer, you have to be aware of the following. The wire protocol is text based so serpent has to encode any binary data. It uses base-64 to do that. This means on the receiving side, instead of the raw bytes, you get a little dictionary like this instead: `{'data': 'aXJtZW4gZGUgam9uZw==', 'encoding': 'base64'}` Your client code needs to be aware of this and to get the original binary data back, it has to base-64 decode the data element by itself. This is perhaps done the easiest by using the `serpent.tobytes` helper function from the `serpent` library, which will convert the result to actual bytes if needed (and leave it untouched if it is already in bytes form)

The following table is an indication of the relative speeds when dealing with large amounts of binary data. It lists the results of the `hugetransfer` example, using python 3.5, over a 1000 Mbps LAN connection:

serializer	str mb/sec	bytes mb/sec	bytearray mb/sec	bytearray w/iterator
pickle	77.8	79.6	69.9	35.0
marshal	71.0	73.0	73.0	37.8
serpent	25.0	14.1	13.5	13.5
json	31.5	not supported	not supported	not supported

The json serializer only works with strings, it can't serialize binary data at all. The serpent serializer can, but read the note above about why it's quite inefficent there. Marshal and pickle are relatively efficient, speed-wise. But beware, when using `pickle`, there's quite a difference in dealing with various types:

pickle datatype differences

`str` *Python 2.x:* efficient; directly encoded as a byte sequence, because that's what it is. *Python 3.x:* inefficient; encoded in UTF-8 on the wire, because it is a unicode string.

`bytes` *Python 2.x:* same as `str` (Python 2.7) *Python 3.x:* efficient; directly encoded as a byte sequence.

`bytearray` Inefficient; encoded as UTF-8 on the wire (pickle does this in both Python 2.x and 3.x)

`array("B")` (array of unsigned ints of size 1) *Python 2.x:* very inefficient; every element is encoded as a separate token+value. *Python 3.x:* efficient; uses machine type encoding on the wire (a byte sequence).

`numpy arrays` usually cannot be transferred directly, see *Pyro and Numpy*.

Alternative: avoid most of the serialization overhead by (ab)using annotations

Pyro allows you to add custom annotation chunks to the request and response messages (see *Message annotations*). Because these are binary chunks they will not be passed through the serializer at all. There is a 64Kb total annotation size limit on messages though, so you have to split up larger files. The `filetransfer` example contains fully working example code to see this in action. It combines this with the remote iterator capability of Pyro to easily get all

chunks of the file. It has to split up the file in small chunks but is still quite a bit faster than transmitting bytes through regular response values. Also it is using only regular Pyro high level logic and no low level network or socket code.

Alternative: integrating raw socket transfer in a Pyro server

It is possible to get data transfer speeds that are close to the limit of your network adapter by doing the actual data transfer via low-level socket code and everything else via Pyro. This keeps the amount of low-level code to a minimum. Have a look at the `filetransfer` example again, to see a possible way of doing this. It creates a special Daemon subclass that uses Pyro for everything as usual, but for actual file transfer it sets up a dedicated temporary socket connection over which the file data is transmitted.

2.11.11 MSG_WAITALL socket option

Pyro will use the `MSG_WAITALL` socket option to receive large messages, if it decides that the feature is available and working correctly. This avoids having to use a slower function that needs a loop to get all data. On most systems that define the `socket.MSG_WAITALL` symbol, it works fine, except on Windows: even though the option is there, it doesn't work reliably. Pyro thus won't use it by default on Windows, and will use it by default on other systems. You should set the `USE_MSG_WAITALL` config item to False yourself, if you find that your system has an unreliable implementation of this socket option. Please let me know what system (os/python version) it is so we could teach Pyro to select the correct option automatically in a new version.

2.11.12 IPV6 support

Pyro4 supports IPv6 since version 4.18. You can use IPv6 addresses in the same places where you would normally have used IPv4 addresses. There's one exception: the address notation in a Pyro URI. For a numeric IPv6 address in a Pyro URI, you have to enclose it in brackets. For example:

```
PYRO:objectname@[::1]:3456
```

points at a Pyro object located on the IPv6 "::1" address (localhost). When Pyro displays a numeric IPv6 location from an URI it will also use the bracket notation. This bracket notation is only used in Pyro URIs, everywhere else you just type the IPv6 address without brackets.

To tell Pyro to prefer using IPv6 you can use the `PREFER_IP_VERSION` config item. It is set to 4 by default, for backward compatibility reasons. This means that unless you change it to 6 (or 0), Pyro will be using IPv4 addressing.

There is a new method to see what IP addressing is used: *`Pyro4.socketutil.getIpVersion()`*, and a few other methods in *`Pyro4.socketutil`* gained a new optional argument to tell it if it needs to deal with an ipv6 address rather than ipv4, but these are rarely used in client code.

2.11.13 Pyro and Numpy

Pyro doesn't support Numpy out of the box. You'll see certain errors occur when trying to use numpy objects (ndarrays, etcetera) with Pyro:

```
TypeError: array([1, 2, 3]) is not JSON serializable
  or
TypeError: don't know how to serialize class <type 'numpy.ndarray'>
  or
TypeError: don't know how to serialize class <class 'numpy.int64'>
```

These errors are caused by Numpy datatypes not being serializable by serpent or json serializers. There are several reasons these datatypes are not supported out of the box:

1. numpy is a third party library and there are many, many others. It is not Pyro's responsibility to understand all of them.

2. numpy is often used in scenarios with large amounts of data. Sending these large arrays over the wire through Pyro is often not the best solution. It is not useful to provide transparent support for numpy types when you'll be running into trouble often such as slow calls and large network overhead.
3. Pyrolite (*Pyrolite - client library for Java and .NET*) would have to get numpy support as well and that is a lot of work (because every numpy type would require a mapping to the appropriate Java or .NET type)

If you understand this but still want to use numpy with Pyro, and pass numpy objects over the wire, you can do it! Choose one of the following options:

1. Don't use Numpy datatypes as arguments or return values. Convert them to standard Python datatypes before using them in Pyro. So instead of just `na = numpy.array(...); return na;`, use this instead: `return na.tolist()`. Or perhaps even `return array.array('i', na)` (serpent understands `array.array` just fine). Note that the elements of a numpy array usually are of a special numpy datatype as well (such as `numpy.int32`). If you don't convert these individually as well, you will still get serialization errors. That is why something like `list(na)` doesn't work: it seems to return a regular python list but the elements are still numpy datatypes. You have to use the full conversions as mentioned earlier. Note that you'll have to do a bit more work to deal with multi-dimensional arrays: you have to convert the shape of the array separately.
2. If possible don't return the whole array. Redesign your API so that you might perhaps only return a single element from it, or a few, if that is all the client really needs.
3. Tell Pyro to use `pickle`, `cloudpickle` or `dill` as serializer. These serializers *can* deal with numpy datatypes out of the box. However they have security implications. See *Security*. (If you choose to use them anyway, also be aware that you must tell your name server about it as well, see *Using the name server with pickle, cloudpickle or dill serializers*)

2.11.14 Pyro via HTTP and JSON

advanced topic

This is an advanced/low-level Pyro topic.

Pyro provides a HTTP gateway server that translates HTTP requests into Pyro calls. It responds with JSON messages. This allows clients (including web browsers) to use a simple http interface to call Pyro objects. Pyro's JSON serialization format is used so the gateway simply passes the JSON response messages back to the caller. It also provides a simple web page that shows how stuff works.

Starting the gateway:

You can launch the HTTP gateway server via the command line tool. This will create a web server using Python's `wsgiref` server module. Because the gateway is written as a wsgi app, you can also stick it into a wsgi server of your own choice. Import `pyro_app` from `Pyro4.utils.httpgateway` to do that (that's the app you need to use).

synopsys: **`python -m Pyro4.utils.httpgateway [options]`** (or simply: **`pyro4-httpgateway [options]`**)

A short explanation of the available options can be printed with the help option:

`-h, --help`
Print a short help message and exit.

Most other options should be self explanatory; you can set the listening host and portname etc. An important option is the exposed names regex option: this controls what objects are accessible from the http gateway interface. It defaults to something that won't just expose every internal object in your system. If you want to toy a bit with the examples provided in the gateway's web page, you'll have to change the option to something like: `r'Pyro\.|test\.'` so

that those objects are exposed. This regex is the same as used when listing objects from the name server, so you can use the `nsc` tool to check it (with the listmatching command).

Setting Hmac keys for use by the gateway:

The `-k` and/or `-g` command line options to set the optional Hmac keys are deprecated since Pyro 4.72 because setting a hmac key like this is a security issue. You should set these keys with the PYRO_HMAC_KEY and PYRO_HTTPGATEWAY_KEY environment variables instead, before starting the gateway.

Using the gateway:

You request the url `http://localhost:8080/pyro/<<objectname>>/<<method>>` to invoke a method on the object with the given name (yes, every call goes through a naming server lookup). Parameters are passed via a regular query string parameter list (in case of a GET request) or via form post parameters (in case of a POST request). The response is a JSON document. In case of an exception, a JSON encoded exception object is returned. You can easily call this from your web page scripts using `XMLHttpRequest` or something like JQuery's `$.ajax()`. Have a look at the page source of the gateway's web page to see how this could be done. Note that you have to comply with the browser's same-origin policy: if you want to allow your own scripts to access the gateway, you'll have to make sure they are loaded from the same website.

The http gateway server is *stateless* at the moment. This means every call you do will end be processed by a new Pyro proxy in the gateway server. This is not impacting your client code though, because every call that it does is also just a stateless http call. It only impacts performance: doing large amounts of calls through the http gateway will perform much slower as the same calls processed by a native Pyro proxy (which you can instruct to operate in batch mode as well). However because Pyro is quite efficient, a call through the gateway is still processed in just a few milliseconds, naming lookup and json serialization all included.

Special http request headers:

- `X-Pyro-Options`: add this header to the request to set certain pyro options for the call. Possible values (comma-separated):
  - `oneway`: force the Pyro call to be a oneway call and return immediately. The gateway server still returns a 200 OK http response as usual, but the response data is empty. This option is to override the semantics for non-oneway method calls if you so desire.
- `X-Pyro-Gateway-Key`: add this header to the request to set the http gateway key. You can also set it on the request with a `$key=...` querystring parameter.

Special Http response headers:

- `X-Pyro-Correlation-Id`: contains the correlation id Guid that was used for this request/response.

Http response status codes:

- 200 OK: all went well, response is the Pyro response message in JSON serialized format
- 403 Forbidden: you're trying to access an object that is not exposed by configuration
- 404 Not Found: you're requesting a non existing object
- 500 Internal server error: something went wrong during request processing, response is serialized exception object (if available)

2.11.15 Client information on the current_context, correlation id

advanced topic

This is an advanced/low-level Pyro topic.

Pyro provides a *thread-local* object with some information about the current Pyro method call, such as the client that's performing the call. It is available as *Pyro4.current_context* (shortcut to *Pyro4.core.current_context*). When accessed in a Pyro server it contains various attributes:

`Pyro4.current_context.`**`client`**
(*Pyro4.socketutil.SocketConnection*) this is the socket connection with the client that's doing the request. You can check the source to see what this is all about, but perhaps the single most useful attribute exposed here is `sock`, which is the socket connection. So the client's IP address can for instance be obtained via `Pyro4.current_context.client.sock.getpeername()[0]` . However, since for oneway calls the socket connection will likely be closed already, this is not 100% reliable. Therefore Pyro stores the result of the `getpeername` call in a separate attribute on the context: `client_sock_addr` (see below)

`Pyro4.current_context.`**`client_sock_addr`**
(*tuple*) the socket address of the client doing the call. It is a tuple of the client host address and the port.

`Pyro4.current_context.`**`seq`**
(*int*) request sequence number

`Pyro4.current_context.`**`msg_flags`**
(*int*) message flags, see *Pyro4.message.Message*

`Pyro4.current_context.`**`serializer_id`**
(*int*) numerical id of the serializer used for this communication, see *Pyro4.message.Message* .

`Pyro4.current_context.`**`annotations`**
(*dict*) message annotations, key is a 4-letter string and the value is a byte sequence. Used to send and receive annotations with Pyro requests. See *Message annotations* for more information about that.

`Pyro4.current_context.`**`response_annotations`**
(*dict*) message annotations, key is a 4-letter string and the value is a byte sequence. Used in client code, the annotations returned by a Pyro server are available here. See *Message annotations* for more information about that.

`Pyro4.current_context.`**`correlation_id`**
(`uuid.UUID`, optional) correlation id of the current request / response. If you set this (in your client code) before calling a method on a Pyro proxy, Pyro will transfer the correlation id to the server context. If the server on their behalf invokes another Pyro method, the same correlation id will be passed along. This way it is possible to relate all remote method calls that originate from a single call. To make this work you'll have to set this to a new `uuid.UUID` in your client code right before you call a Pyro method. Note that it is required that the correlation id is of type `uuid.UUID`. Note that the HTTP gateway (see *Pyro via HTTP and JSON*) also creates a correlation id for every request, and will return it via the `X-Pyro-Correlation-Id` HTTP-header in the response. It will also accept this header optionally on a request in which case it will use the value from the header rather than generating a new id.

For an example of how this information can be retrieved, and how to set the `correlation_id`, see the `callcontext` example. See the `usersession` example to learn how you could use it to build user-bound resource access without concurrency problems.

2.11.16 Automatically freeing resources when client connection gets closed

advanced topic

This is an advanced/low-level Pyro topic.

A client can call remote methods that allocate stuff in the server. Normally the client is responsible to call other methods once the resources should be freed.

However if the client forgets this or the connection to the server is forcefully closed before the client can free the resources, the resources in the server will usually not be freed anymore.

You may be able to solve this in your server code yourself (perhaps using some form of keepalive/timeout mechanism) but Pyro 4.63 and newer provides a built-in mechanism that can help: resource tracking on the client connection. Your server will register the resources when they are allocated, thereby making them tracked resources on the client connection. These tracked resources will be automatically freed by Pyro if the client connection is closed.

For this to work, the resource object should have a `close` method (Pyro will call this). If needed, you can also override *`Pyro4.core.Daemon.clientDisconnect()`* and do the cleanup yourself with the `tracked_resources` on the connection object.

Resource tracking and untracking is done in your server class on the `Pyro4.current_context` object:

`Pyro4.current_context.`**`track_resource`**(*resource*)
Let Pyro track the resource on the current client connection.

`Pyro4.current_context.`**`untrack_resource`**(*resource*)
Untrack a previously tracked resource, useful if you have freed it normally.

See the `resourcetracking` example for working code utilizing this.

Note: The order in which the resources are freed is arbitrary. Also, if the resource can be garbage collected normally by Python, it is removed from the tracked resources. So the `close` method should not be the only way to properly free such resources (maybe you need a `__del__` as well).

2.11.17 Message annotations

advanced topic

This is an advanced/low-level Pyro topic.

Pyro's wire protocol allows for a very flexible messaging format by means of *annotations*. Annotations are extra information chunks that are added to the pyro messages traveling over the network. Pyro internally uses a couple of chunks to exchange extra data between a proxy and a daemon: correlation ids (annotation `CORR`) and hmac signatures (annotation `HMAC`). These chunk types are reserved and you should not touch them. All other annotation types are free to use in your own code (and will be ignored by Pyro itself). There's no limit on the number of annotations you can add to a message, but each individual annotation cannot be larger than 64 Kb.

reserved annotation chunks

The following annotation chunks are used by Pyro internally and should not be touched or used: `CORR`, `HMAC`, `STRM` and `BLBI`.

An annotation is a low level datastructure (to optimize the generation of network messages): a chunk identifier string of exactly 4 characters (such as "CODE"), and its value, a byte sequence. If you want to put specific data structures into an annotation chunk value, you have to encode them to a byte sequence yourself (of course, you could utilize a Pyro serializer for this). When processing a custom annotation, you have to decode it yourself as well. Communicating annotations with Pyro is done via a normal dictionary of chunk id -> data bytes. Pyro will take care of encoding this dictionary into the wire message and extracting it out of a response message.

Custom user annotations:

You can add your own annotations to messages. For server code, you do this by setting the `response_annotations` property of the *Pyro4.current_context* in your Pyro object, right before returning the regular response value. Pyro will add the annotations dict to the response message. In client code, you can set the `annotations` property of the *Pyro4.current_context* object right before the proxy method call. Pyro will then add that annotations dict to the request message.

The older method to to this (before Pyro 4.56) was to create a subclass of `Proxy` or `Daemon` and override the methods *Pyro4.core.Proxy._pyroAnnotations()* or *Pyro4.core.Daemon.annotations()* respectively. These methods should return the custom annotations dict that should be added to request/response messages. This is still possible to not break older code.

Reacting on annotations:

In your server code, in the Daemon, you can use the *Pyro4.current_context* to access the `annotations` of the last message that was received. In your client code, you can do that as well, but you should look at the `response_annotations` of this context object instead. If you're using large annotation chunks, it is advised to clear these fields after use. See *Client information on the current_context, correlation id*.

The older method to do this (before Pyro 4.56) for client code was to create a proxy subclass and override the method *Pyro4.core.Proxy._pyroResponseAnnotations()*. Pyro calls this method with the dictionary of any annotations received in a response message from the daemon, and the message type identifier of the response message. This still works to not break older code.

For an example of how you can work with custom message annotations, see the `callcontext` example.

2.11.18 Connection handshake

advanced topic

This is an advanced/low-level Pyro topic.

When a proxy is first connecting to a Pyro daemon, it exchanges a few messages to set up and validate the connection. This is called the connection *handshake*. Part of it is the daemon returning the object's metadata (see *Metadata from the daemon*). You can hook into this mechanism and influence the data that is initially exchanged during the connection setup, and you can act on this data. You can disallow the connection based on this, for example.

You can set your own data on the proxy attribute *Pyro4.core.Proxy._pyroHandshake*. You can set any serializable object. Pyro will send this as the handshake message to the daemon when the proxy tries to connect. In the daemon, override the method *Pyro4.core.Daemon.validateHandshake()* to customize/validate the connection setup. This method receives the data from the proxy and you can either raise an exception if you don't want to allow the connection, or return a result value if you are okay with the new connection. The result value again can be any serializable object. This result value will be received back in the Proxy where you can act on it if you subclass the proxy and override *Pyro4.core.Proxy._pyroValidateHandshake()*.

For an example of how you can work with connections handshake validation, see the `handshake` example. It implements a (bad!) security mechanism that requires the client to supply a "secret" password to be able to connect to the daemon.

2.11.19 Efficient dispatchers or gateways that don't de/reserialize messages

advanced topic

This is an advanced/low-level Pyro topic.

Imagine you're designing a setup where a Pyro call is essentially dispatched or forwarded to another server. The dispatcher (sometimes also called gateway) does nothing else than deciding who the message is for, and then forwarding the Pyro call to the actual object that performs the operation.

This can be built easily with Pyro by 'intercepting' the call in a dispatcher object, and performing the remote method call *again* on the actual server object. There's nothing wrong with this except for perhaps two things:

1. Pyro will deserialize and reserialize the remote method call parameters on every hop, this can be quite inefficient if you're dealing with many calls or large argument data structures.
2. The dispatcher object is now dependent on the method call argument data types, because Pyro has to be able to de/reserialize them. This often means the dispatcher also needs to have access to the same source code files that define the argument data types, that the client and server use.

As long as the dispatcher itself *doesn't have to know what is even in the actual message*, Pyro provides a way to avoid both issues mentioned above: use the *Pyro4.core.SerializedBlob*. If you use that as the (single) argument to a remote method call, Pyro will not deserialize the message payload *until you ask for it* by calling the `deserialized()` method on it. Which is something you only do in the actual server object, and not in the dispatcher. Because the message is then never de/reserialized in the dispatcher code, you avoid the serializer overhead, and also don't have to include the source code for the serialized types in the dispatcher. It just deals with a blob of serialized bytes.

An example that shows how this mechanism can be used, can be found as `blob-dispatch` in the examples folder.

2.11.20 Hooking onto existing connected sockets such as from socketpair()

For communication between threads or sub-processes, there is `socket.socketpair()`. It creates spair of connected sockets that you can share between the threads or processes. Since Pyro 4.70 it is possible to tell Pyro to use a user-created socket like that, instead of creating new sockets itself, which means you can use Pyro to talk between threads or sub-processes over an efficient and isolated channel. You do this by creating a socket (or a pair) and providing it as the `connected_socket` parameter to the `Daemon` and `Proxy` classes. For the Daemon, don't pass any other arguments because they won't be used anyway. For the Proxy, set only the first parameter (`uri`) to just the *name* of the object in the daemon you want to connect to. So don't use a PYRO or PYRONAME prefix for the uri in this case.

Closing the proxy or the daemon will *not* close the underlying user-supplied socket so you can use it again for another proxy (to access a different object). You created the socket(s) yourself, and you also have to close the socket(s) yourself. Also because the socketpair is internal to the process that created it, it's safe to use the pickle serializer on this connection. This can improve communication performance even further.

See the `socketpair` example for two example programs (one using threads, the other using fork to create a child process).

2.12 Configuring Pyro

Pyro can be configured using several *configuration items*. The current configuration is accessible from the `Pyro4.config` object, it contains all config items as attributes. You can read them and update them to change Pyro's configuration. (usually you need to do this at the start of your program). For instance, to enable message compression and change the server type, you add something like this to the start of your code:

```
Pyro4.config.COMPRESSION = True
Pyro4.config.SERVERTYPE = "multiplex"
```

You can also set them outside of your program, using environment variables from the shell. **To avoid conflicts, the environment variables have a ''PYRO_'' prefix.** This means that if you want to change the same two settings as above, but by using environment variables, you would do something like:

```
$ export PYRO_COMPRESSION=true
$ export PYRO_SERVERTYPE=multiplex

(or on windows:)
C:\> set PYRO_COMPRESSION=true
C:\> set PYRO_SERVERTYPE=multiplex
```

This environment defined configuration is simply used as initial values for Pyro's configuration object. Your code can still overwrite them by setting the items to other values, or by resetting the config as a whole.

2.12.1 Resetting the config to default values

`Pyro4.config.`**`reset`**`(`[*useenvironment=True*]`)`

Resets the configuration items to their builtin default values. If *useenvironment* is True, it will overwrite builtin config items with any values set by environment variables. If you don't trust your environment, it may be a good idea to reset the config items to just the builtin defaults (ignoring any environment variables) by calling this method with *useenvironment* set to False. Do this before using any other part of the Pyro library.

2.12.2 Inspecting current config

To inspect the current configuration you have several options:

1. Access individual config items: `print(Pyro4.config.COMPRESSION)`
2. Dump the config in a console window: **`python -m Pyro4.configuration`** (or simply **`pyro4-check-config`**) This will print something like:

```
Pyro version: 4.6
Loaded from: E:\Projects\Pyro4\src\Pyro4
Active configuration settings:
AUTOPROXY = True
COMMTIMEOUT = 0.0
COMPRESSION = False
...
```

3. Access the config as a dictionary: `Pyro4.config.asDict()`
4. Access the config string dump (used in #2): `Pyro4.config.dump()`

2.12.3 Overview of Config Items

config item	type	default	meaning
AUTOPROXY	bool	True	Enable to make Pyro automatically replace Pyro objects b
COMMTIMEOUT	float	0.0	network communication timeout in seconds. 0.0=no time
COMPRESSION	bool	False	Enable to make Pyro compress the data that travels over t
DETAILED_TRACEBACK	bool	False	Enable to get detailed exception tracebacks (including the
HOST	str	localhost	Hostname where Pyro daemons will bind on
MAX_MESSAGE_SIZE	int	0	Maximum size in bytes of the messages sent or received o

Tabl

config item	type	default	meaning
NS_HOST	str	*equal to HOST*	Hostname for the name server. Used for locating in client
NS_PORT	int	9090	TCP port of the name server. Used by the server and for l
NS_BCPORT	int	9091	UDP port of the broadcast responder from the name serve
NS_BCHOST	str	None	Hostname for the broadcast responder of the name server.
NS_AUTOCLEAN	float	0.0	Specify a recurring period in seconds where the Name ser
NATHOST	str	None	External hostname in case of NAT (used by the server)
NATPORT	int	None	External port in case of NAT (used by the server)
BROADCAST_ADDRS	str	<broadcast>, 0.0.0.0	List of comma separated addresses that Pyro should send
ONEWAY_THREADED	bool	True	Enable to make oneway calls be processed in their own se
POLLTIMEOUT	float	2.0	For the multiplexing server only: the timeout of the select
SERVERTYPE	str	thread	Select the Pyro server type. thread=thread pool based, mu
SOCK_REUSE	bool	True	Should SO_REUSEADDR be used on sockets that Pyro c
PREFER_IP_VERSION	int	4	The IP address type that is preferred (4=ipv4, 6=ipv6, 0=l
THREADPOOL_SIZE	int	40	For the thread pool server: maximum number of threads r
THREADPOOL_SIZE_MIN	int	4	For the thread pool server: minimum number of threads r
FLAME_ENABLED	bool	False	Should Pyro Flame be enabled on the server
SERIALIZER	str	serpent	The wire protocol serializer to use for clients/proxies (one
SERIALIZERS_ACCEPTED	set	json,marshal,serpent	The wire protocol serializers accepted in the server/daemo
PICKLE_PROTOCOL_VERSION	int	highest possible	The pickle protocol version to use, if pickle is selected as
DILL_PROTOCOL_VERSION	int	highest possible	The dill protocol version to use, if dill is selected as serial
JSON_MODULE	str	json	The json module to use for the json serializer. (json is inc
LOGWIRE	bool	False	If wire-level message data should be written to the logfile
METADATA	bool	True	Client: Get remote object metadata from server automatic
REQUIRE_EXPOSE	bool	True	Server: Is @expose required to make members remotely a
USE_MSG_WAITALL	bool	True (False if on Windows)	Some systems have broken socket MSG_WAITALL supp
MAX_RETRIES	int	0	Automatically retry network operations for some exceptio
ITER_STREAMING	bool	True	Should iterator item streaming support be enabled in the s
ITER_STREAM_LIFETIME	float	0.0	Maximum lifetime in seconds for item streams (default=0
ITER_STREAM_LINGER	float	30.0	Linger time in seconds to keep an item stream alive after
SSL	bool	False	Should SSL/TSL communication security be used? Enabl
SSL_SERVERCERT	str	*empty str*	Location of the server's certificate file
SSL_SERVERKEY	str	*empty str*	Location of the server's private key file
SSL_SERVERKEYPASSWD	str	*empty str*	Password for the server's private key
SSL_REQUIRECLIENTCERT	bool	False	Should the server require clients to connect with their ow
SSL_CLIENTCERT	str	*empty str*	Location of the client's certificate file
SSL_CLIENTKEY	str	*empty str*	Location of the client's private key file
SSL_CLIENTKEYPASSWD	str	*empty str*	Password for the client's private key

There are two special config items that control Pyro's logging, and that are only available as environment variable settings. This is because they are used at the moment the Pyro4 package is being imported (which means that modifying them as regular config items after importing Pyro4 is too late and won't work).

It is up to you to set the environment variable you want to the desired value. You can do this from your OS or shell, or perhaps by modifying `os.environ` in your Python code *before* importing Pyro4.

environment variable	type	default	meaning
PYRO_LOGLEVEL	string	*not set*	The log level to use for Pyro's logger (DEBUG, WARN, ...) See Python's standard `logging` module for the allowed values (https://docs.python.org/2/library/logging.html#levels). If it is not set, no logging is being configured.
PYRO_LOGFILE	string	pyro.log	The name of the log file. Use {stderr} to make the log go to the standard error output.

2.13 Pyro4 library API

This chapter describes Pyro's library API. All Pyro classes and functions are defined in sub packages such as *Pyro4.core*, but for ease of use, the most important ones are also placed in the *Pyro4* package scope.

2.13.1 `Pyro4` — Main API package

Pyro4 is the main package of Pyro4. It imports most of the other packages that it needs and provides shortcuts to the most frequently used objects and functions from those packages. This means you can mostly just `import Pyro4` in your code to start using Pyro.

The classes and functions provided are:

symbol in *Pyro4*	referenced location
Pyro4.**config**	the current configuration settings, Pyro4.configuration.config
Pyro4.**current_context**	the current client context, *Pyro4.core.current_context*
class Pyro4.**URI**	*Pyro4.core.URI*
class Pyro4.**Proxy**	*Pyro4.core.Proxy*
class Pyro4.**Daemon**	*Pyro4.core.Daemon*
class Pyro4.**Future**	*Pyro4.futures.Future*
Pyro4.**callback**()	*Pyro4.core.callback()*
Pyro4.**batch**()	*Pyro4.core.batch()*
Pyro4.**asyncproxy**()	*Pyro4.core.asyncproxy()*
Pyro4.**locateNS**()	*Pyro4.naming.locateNS()*
Pyro4.**resolve**()	*Pyro4.naming.resolve()*
Pyro4.**expose**()	*Pyro4.core.expose()* (decorator @expose)
Pyro4.**oneway**()	*Pyro4.core.oneway()* (decorator @oneway)
Pyro4.**behavior**()	*Pyro4.core.behavior()* (decorator @behavior)

See also:

Module *Pyro4.core* The core Pyro classes and functions.

Module *Pyro4.naming* The Pyro name server logic.

2.13.2 Pyro4.core — core Pyro logic

Core logic (uri, daemon, proxy stuff).

class Pyro4.core.**URI**(*uri*)

Pyro object URI (universal resource identifier). The uri format is like this: PYRO:objectid@location where location is one of:

- hostname:port (tcp/ip socket on given port)
- ./u:sockname (Unix domain socket on localhost)

There is also a 'Magic format' for simple name resolution using Name server:
`PYRONAME:objectname[@location]` (optional name server location, can also omit location port)

And one that looks up things in the name server by metadata: `PYROMETA:meta1,meta2,...[@location]` (optional name server location, can also omit location port)

You can write the protocol in lowercase if you like (`pyro:...`) but it will automatically be converted to uppercase internally.

asString()
the string representation of this object

static isUnixsockLocation(*location*)
determine if a location string is for a Unix domain socket

location
property containing the location string, for instance `"servername.you.com:5555"`

class Pyro4.core.**Proxy**(*uri*, *connected_socket=None*)
Pyro proxy for a remote object. Intercepts method calls and dispatches them to the remote object.

_pyroBind()
Bind this proxy to the exact object from the uri. That means that the proxy's uri will be updated with a direct PYRO uri, if it isn't one yet. If the proxy is already bound, it will not bind again.

_pyroRelease()
release the connection to the pyro daemon

_pyroReconnect(*tries=100000000*)
(Re)connect the proxy to the daemon containing the pyro object which the proxy is for. In contrast to the _pyroBind method, this one first releases the connection (if the proxy is still connected) and retries making a new connection until it succeeds or the given amount of tries ran out.

_pyroBatch()
returns a helper class that lets you create batched method calls on the proxy

_pyroAsync(*asynchronous=True*, ***kwargs*)

_pyroAnnotations()
Override to return a dict with custom user annotations to be sent with each request message. Code using Pyro 4.56 or newer can skip this and instead set the annotations directly on the context object.

_pyroResponseAnnotations(*annotations*, *msgtype*)
Process any response annotations (dictionary set by the daemon). Usually this contains the internal Pyro annotations such as hmac and correlation id, and if you override the annotations method in the daemon, can contain your own annotations as well. Code using Pyro 4.56 or newer can skip this and instead read the response_annotations directly from the context object.

_pyroValidateHandshake(*response*)
Process and validate the initial connection handshake response data received from the daemon. Simply return without error if everything is ok. Raise an exception if something is wrong and the connection should not be made.

_pyroTimeout
The timeout in seconds for calls on this proxy. Defaults to `None`. If the timeout expires before the remote method call returns, Pyro will raise a *Pyro4.errors.TimeoutError*

_pyroHmacKey
the HMAC key (bytes) that this proxy uses

_pyroMaxRetries
Number of retries to perform on communication calls by this proxy, allows you to override the default setting.

_pyroSerializer
Name of the serializer to use by this proxy, allows you to override the default setting.

_pyroHandshake
The data object that should be sent in the initial connection handshake message. Can be any serializable object.

class Pyro4.core.**Daemon**(*host=None*, *port=0*, *unixsocket=None*, *nathost=None*, *natport=None*, *interface=<class 'Pyro4.core.DaemonObject'>*, *connected_socket=None*)
Pyro daemon. Contains server side logic and dispatches incoming remote method calls to the appropriate objects.

annotations()
Override to return a dict with custom user annotations to be sent with each response message.

clientDisconnect(*conn*)
Override this to handle a client disconnect. Conn is the SocketConnection object that was disconnected.

close()
Close down the server and release resources

combine(*daemon*)
Combines the event loop of the other daemon in the current daemon's loop. You can then simply run the current daemon's requestLoop to serve both daemons. This works fine on the multiplex server type, but doesn't work with the threaded server type.

events(*eventsockets*)
for use in an external event loop: handle any requests that are pending for this daemon

handleRequest(*conn*)
Handle incoming Pyro request. Catches any exception that may occur and wraps it in a reply to the calling side, as to not make this server side loop terminate due to exceptions caused by remote invocations.

housekeeping()
Override this to add custom periodic housekeeping (cleanup) logic. This will be called every few seconds by the running daemon's request loop.

locationStr = None
The location (str of the form `host:portnumber`) on which the Daemon is listening

natLocationStr = None
The NAT-location (str of the form `nathost:natportnumber`) on which the Daemon is exposed for use with NAT-routing

objectsById = None
Dictionary from Pyro object id to the actual Pyro object registered by this id

proxyFor(*objectOrId*, *nat=True*)
Get a fully initialized Pyro Proxy for the given object (or object id) for this daemon. If nat is False, the configured NAT address (if any) is ignored. The object or id must be registered in this daemon, or you'll get an exception. (you can't get a proxy for an unknown object)

register(*obj_or_class*, *objectId=None*, *force=False*)
Register a Pyro object under the given id. Note that this object is now only known inside this daemon, it is not automatically available in a name server. This method returns a URI for the registered object. Pyro checks if an object is already registered, unless you set force=True. You can register a class or an object (instance) directly. For a class, Pyro will create instances of it to handle the remote calls according

to the instance_mode (set via @expose on the class). The default there is one object per session (=proxy connection). If you register an object directly, Pyro will use that single object for *all* remote calls.

requestLoop(*loopCondition=<function Daemon.<lambda>>*)
Goes in a loop to service incoming requests, until someone breaks this or calls shutdown from another thread.

resetMetadataCache(*objectOrId*, *nat=True*)
Reset cache of metadata when a Daemon has available methods/attributes dynamically updated. Clients will have to get a new proxy to see changes

selector
the multiplexing selector used, if using the multiplex server type

static serveSimple(*objects*, *host=None*, *port=0*, *daemon=None*, *ns=True*, *verbose=True*)
Basic method to fire up a daemon (or supply one yourself). objects is a dict containing objects to register as keys, and their names (or None) as values. If ns is true they will be registered in the naming server as well, otherwise they just stay local. If you need to publish on a unix domain socket you can't use this shortcut method. See the documentation on 'publishing objects' (in chapter: Servers) for more details.

shutdown()
Cleanly terminate a daemon that is running in the requestloop.

sock
the server socket used by the daemon

sockets
list of all sockets used by the daemon (server socket and all active client sockets)

unregister(*objectOrId*)
Remove a class or object from the known objects inside this daemon. You can unregister the class/object directly, or with its id.

uriFor(*objectOrId*, *nat=True*)
Get a URI for the given object (or object id) from this daemon. Only a daemon can hand out proper uris because the access location is contained in them. Note that unregistered objects cannot be given an uri, but unregistered object names can (it's just a string we're creating in that case). If nat is set to False, the configured NAT address (if any) is ignored and it will return an URI for the internal address.

validateHandshake(*conn*, *data*)
Override this to create a connection validator for new client connections. It should return a response data object normally if the connection is okay, or should raise an exception if the connection should be denied.

class Pyro4.core.**DaemonObject**(*daemon*)
The part of the daemon that is exposed as a Pyro object.

get_metadata(*objectId*, *as_lists=False*)
Get metadata for the given object (exposed methods, oneways, attributes). If you get an error in your proxy saying that 'DaemonObject' has no attribute 'get_metadata', you're probably connecting to an older Pyro version (4.26 or earlier). Either upgrade the Pyro version or set METADATA config item to False in your client code.

info()
return some descriptive information about the daemon

ping()
a simple do-nothing method for testing purposes

registered()
returns a list of all object names registered in this daemon

Pyro4.core.**callback**(*method*)
decorator to mark a method to be a 'callback'. This will make Pyro raise any errors also on the callback side, and not only on the side that does the callback call.

Pyro4.core.**batch**(*proxy*)
convenience method to get a batch proxy adapter

Pyro4.core.**asyncproxy**(*proxy*, *asynchronous=True*, ***kwargs*)
convenience method to set proxy to asynchronous or sync mode.

Pyro4.core.**expose**(*method_or_class*)
Decorator to mark a method or class to be exposed for remote calls (relevant when REQUIRE_EXPOSE=True) You can apply it to a method or a class as a whole. If you need to change the default instance mode or instance creator, also use a @behavior decorator.

Pyro4.core.**behavior**(*instance_mode='session'*, *instance_creator=None*)
Decorator to specify the server behavior of your Pyro class.

Pyro4.core.**oneway**(*method*)
decorator to mark a method to be oneway (client won't wait for a response)

Pyro4.core.**current_context = <Pyro4.core._CallContext object>**
the context object for the current call. (thread-local)

class Pyro4.core.**_StreamResultIterator**(*streamId*, *proxy*)
Pyro returns this as a result of a remote call which returns an iterator or generator. It is a normal iterable and produces elements on demand from the remote iterator. You can simply use it in for loops, list comprehensions etc.

class Pyro4.core.**SerializedBlob**(*info*, *data*, *is_blob=False*)
Used to wrap some data to make Pyro pass this object transparently (it keeps the serialized payload as-is) Only when you need to access the actual client data you can deserialize on demand. This makes efficient, transparent gateways or dispatchers and such possible: they don't have to de/reserialize the message and are independent from the serialized class definitions. You have to pass this as the only parameter to a remote method call for Pyro to understand it. Init arguments: `info` = some (small) descriptive data about the blob. Can be a simple id or name or guid. Must be marshallable. `data` = the actual client data payload that you want to transfer in the blob. Can be anything that you would otherwise have used as regular remote call arguments.

deserialized()
Retrieves the client data stored in this blob. Deserializes the data automatically if required.

2.13.3 Pyro4.naming — Pyro name server

Name Server and helper functions.

Pyro4.naming.**locateNS**(*host=None*, *port=None*, *broadcast=True*, *hmac_key=None*)
Get a proxy for a name server somewhere in the network.

Pyro4.naming.**resolve**(*uri*, *hmac_key=None*)
Resolve a 'magic' uri (PYRONAME, PYROMETA) into the direct PYRO uri. It finds a name server, and use that to resolve a PYRONAME uri into the direct PYRO uri pointing to the named object. If uri is already a PYRO uri, it is returned unmodified. You can consider this a shortcut function so that you don't have to locate and use a name server proxy yourself. Note: if you need to resolve more than a few names, consider using the name server directly instead of repeatedly calling this function, to avoid the name server lookup overhead from each call.

Pyro4.naming.**type_meta**(*class_or_object*, *prefix='class:'*)
extracts type metadata from the given class or object, can be used as Name server metadata.

Pyro4.naming.**startNSloop**(*host=None*, *port=None*, *enableBroadcast=True*, *bchost=None*, *bcport=None*, *unixsocket=None*, *nathost=None*, *natport=None*, *storage=None*, *hmac=None*)
utility function that starts a new Name server and enters its requestloop.

Pyro4.naming.**startNS**(*host=None*, *port=None*, *enableBroadcast=True*, *bchost=None*, *bcport=None*, *unixsocket=None*, *nathost=None*, *natport=None*, *storage=None*, *hmac=None*)
utility fuction to quickly get a Name server daemon to be used in your own event loops. Returns (nameserverUri, nameserverDaemon, broadcastServer).

class Pyro4.naming.**NameServer**(*storageProvider=None*)
Pyro name server. Provides a simple flat name space to map logical object names to Pyro URIs. Default storage is done in an in-memory dictionary. You can provide custom storage types.

list(*prefix=None*, *regex=None*, *metadata_all=None*, *metadata_any=None*, *return_metadata=False*)
Retrieve the registered items as a dictionary name-to-URI. The URIs in the resulting dict are strings, not URI objects. You can filter by prefix or by regex or by metadata subset (separately)

lookup(*name*, *return_metadata=False*)
Lookup the given name, returns an URI if found. Returns tuple (uri, metadata) if return_metadata is True.

ping()
A simple test method to check if the name server is running correctly.

register(*name*, *uri*, *safe=False*, *metadata=None*)
Register a name with an URI. If safe is true, name cannot be registered twice. The uri can be a string or an URI object. Metadata must be None, or a collection of strings.

remove(*name=None*, *prefix=None*, *regex=None*)
Remove a registration. returns the number of items removed.

set_metadata(*name*, *metadata*)
update the metadata for an existing registration

Name Server persistent storage implementations.

class Pyro4.naming_storage.**DbmStorage**(*dbmfile*)
Storage implementation that uses a persistent dbm file. Because dbm only supports strings as key/value, we encode/decode them in utf-8. Dbm files cannot be accessed concurrently, so a strict concurrency model is used where only one operation is processed at the same time (this is very slow when compared to the in-memory storage) DbmStorage does NOT support storing metadata! It only accepts empty metadata, and always returns empty metadata.

clear() → None. Remove all items from D.

class Pyro4.naming_storage.**SqlStorage**(*dbfile*)
Sqlite-based storage. It is just a single (name,uri) table for the names and another table for the metadata. Sqlite db connection objects aren't thread-safe, so a new connection is created in every method.

clear() → None. Remove all items from D.

2.13.4 Pyro4.util — Utilities and serializers

Miscellaneous utilities, and serializers.

class Pyro4.util.**CloudpickleSerializer**
A (de)serializer that wraps the Cloudpickle serialization protocol. It can optionally compress the serialized data, and is thread safe.

class Pyro4.util.**DillSerializer**
A (de)serializer that wraps the Dill serialization protocol. It can optionally compress the serialized data, and is thread safe.

class Pyro4.util.**JsonSerializer**
(de)serializer that wraps the json serialization protocol.

class Pyro4.util.**MarshalSerializer**
(de)serializer that wraps the marshal serialization protocol.

classmethod class_to_dict(*obj*)
Convert a non-serializable object to a dict. Partly borrowed from serpent. Not used for the pickle serializer.

class Pyro4.util.**MsgpackSerializer**
(de)serializer that wraps the msgpack serialization protocol.

class Pyro4.util.**PickleSerializer**
A (de)serializer that wraps the Pickle serialization protocol. It can optionally compress the serialized data, and is thread safe.

class Pyro4.util.**SerializerBase**
Base class for (de)serializer implementations (which must be thread safe)

classmethod class_to_dict(*obj*)
Convert a non-serializable object to a dict. Partly borrowed from serpent. Not used for the pickle serializer.

deserializeCall(*data*, *compressed=False*)
Deserializes the given call data back to (object, method, vargs, kwargs) tuple. Set compressed to True to decompress the data first.

deserializeData(*data*, *compressed=False*)
Deserializes the given data (bytes). Set compressed to True to decompress the data first.

classmethod dict_to_class(*data*)
Recreate an object out of a dict containing the class name and the attributes. Only a fixed set of classes are recognized. Not used for the pickle serializer.

classmethod register_class_to_dict(*clazz*, *converter*, *serpent_too=True*)
Registers a custom function that returns a dict representation of objects of the given class. The function is called with a single parameter; the object to be converted to a dict.

classmethod register_dict_to_class(*classname*, *converter*)
Registers a custom converter function that creates objects from a dict with the given classname tag in it. The function is called with two parameters: the classname and the dictionary to convert to an instance of the class.

This mechanism is not used for the pickle serializer.

serializeCall(*obj*, *method*, *vargs*, *kwargs*, *compress=False*)
Serialize the given method call parameters, try to compress if told so. Returns a tuple of the serialized data and a bool indicating if it is compressed or not.

serializeData(*data*, *compress=False*)
Serialize the given data object, try to compress if told so. Returns a tuple of the serialized data (bytes) and a bool indicating if it is compressed or not.

classmethod unregister_class_to_dict(*clazz*)
Removes the to-dict conversion function registered for the given class. Objects of the class will be serialized by the default mechanism again.

classmethod unregister_dict_to_class(*classname*)
Removes the converter registered for the given classname. Dicts with that classname tag will be deserialized by the default mechanism again.

This mechanism is not used for the pickle serializer.

class Pyro4.util.**SerpentSerializer**
(de)serializer that wraps the serpent serialization protocol.

classmethod dict_to_class(*data*)
Recreate an object out of a dict containing the class name and the attributes. Only a fixed set of classes are recognized. Not used for the pickle serializer.

Pyro4.util.**excepthook**(*ex_type*, *ex_value*, *ex_tb*)
An exception hook you can use for `sys.excepthook`, to automatically print remote Pyro tracebacks

Pyro4.util.**fixIronPythonExceptionForPickle**(*exceptionObject*, *addAttributes*)
Function to hack around a bug in IronPython where it doesn't pickle exception attributes. We piggyback them into the exception's args. Bug report is at https://github.com/IronLanguages/main/issues/943 Bug is still present in Ironpython 2.7.7

Pyro4.util.**formatTraceback**(*ex_type=None*, *ex_value=None*, *ex_tb=None*, *detailed=False*)
Formats an exception traceback. If you ask for detailed formatting, the result will contain info on the variables in each stack frame. You don't have to provide the exception info objects, if you omit them, this function will obtain them itself using `sys.exc_info()`.

Pyro4.util.**getAttribute**(*obj*, *attr*)
Resolves an attribute name to an object. Raises an AttributeError if any attribute in the chain starts with a '_'. Doesn't resolve a dotted name, because that is a security vulnerability. It treats it as a single attribute name (and the lookup will likely fail).

Pyro4.util.**getPyroTraceback**(*ex_type=None*, *ex_value=None*, *ex_tb=None*)
Returns a list of strings that form the traceback information of a Pyro exception. Any remote Pyro exception information is included. Traceback information is automatically obtained via `sys.exc_info()` if you do not supply the objects yourself.

Pyro4.util.**get_exposed_members**(*obj*, *only_exposed=True*, *as_lists=False*, *use_cache=True*)
Return public and exposed members of the given object's class. You can also provide a class directly. Private members are ignored no matter what (names starting with underscore). If only_exposed is True, only members tagged with the @expose decorator are returned. If it is False, all public members are returned. The return value consists of the exposed methods, exposed attributes, and methods tagged as @oneway. (All this is used as meta data that Pyro sends to the proxy if it asks for it) as_lists is meant for python 2 compatibility.

Pyro4.util.**get_exposed_property_value**(*obj*, *propname*, *only_exposed=True*)
Return the value of an @exposed @property. If the requested property is not a @property or not exposed, an AttributeError is raised instead.

Pyro4.util.**is_private_attribute**(*attr_name*)
returns if the attribute name is to be considered private or not.

Pyro4.util.**reset_exposed_members**(*obj*, *only_exposed=True*, *as_lists=False*)
Delete any cached exposed members forcing recalculation on next request

Pyro4.util.**set_exposed_property_value**(*obj*, *propname*, *value*, *only_exposed=True*)
Sets the value of an @exposed @property. If the requested property is not a @property or not exposed, an AttributeError is raised instead.

2.13.5 Pyro4.socketutil — Socket related utilities

Low level socket utilities.

class Pyro4.socketutil.**SocketConnection**(*sock*, *objectId=None*, *keep_open=False*)
A wrapper class for plain sockets, containing various methods such as `send()` and `recv()`

Pyro4.socketutil.**bindOnUnusedPort**(*sock*, *host='localhost'*)
Bind the socket to a free port and return the port number. This code is based on the code in the stdlib's test.test_support module.

Pyro4.socketutil.**createBroadcastSocket**(*bind=None*, *reuseaddr=False*, *timeout=<object object>*, *ipv6=False*)
Create a udp broadcast socket. Set ipv6=True to create an IPv6 socket rather than IPv4. Set ipv6=None to use the PREFER_IP_VERSION config setting.

Pyro4.socketutil.**createSocket**(*bind=None*, *connect=None*, *reuseaddr=False*, *keepalive=True*, *timeout=<object object>*, *noinherit=False*, *ipv6=False*, *nodelay=True*, *sslContext=None*)
Create a socket. Default socket options are keepalive and IPv4 family, and nodelay (nagle disabled). If 'bind' or 'connect' is a string, it is assumed a Unix domain socket is requested. Otherwise, a normal tcp/ip socket is used. Set ipv6=True to create an IPv6 socket rather than IPv4. Set ipv6=None to use the PREFER_IP_VERSION config setting.

Pyro4.socketutil.**findProbablyUnusedPort**(*family=<AddressFamily.AF_INET: 2>*, *socktype=<SocketKind.SOCK_STREAM: 1>*)
Returns an unused port that should be suitable for binding (likely, but not guaranteed). This code is copied from the stdlib's test.test_support module.

Pyro4.socketutil.**getInterfaceAddress**(*ip_address*)
tries to find the ip address of the interface that connects to the given host's address

Pyro4.socketutil.**getIpAddress**(*hostname*, *workaround127=False*, *ipVersion=None*)
Returns the IP address for the given host. If you enable the workaround, it will use a little hack if the ip address is found to be the loopback address. The hack tries to discover an externally visible ip address instead (this only works for ipv4 addresses). Set ipVersion=6 to return ipv6 addresses, 4 to return ipv4, 0 to let OS choose the best one or None to use config.PREFER_IP_VERSION.

Pyro4.socketutil.**getIpVersion**(*hostnameOrAddress*)
Determine what the IP version is of the given hostname or ip address (4 or 6). First, it resolves the hostname or address to get an IP address. Then, if the resolved IP contains a ':' it is considered to be an ipv6 address, and if it contains a '.', it is ipv4.

Pyro4.socketutil.**getSSLcontext**(*servercert=''*, *serverkey=''*, *clientcert=''*, *clientkey=''*, *cacerts=''*, *keypassword=''*)
creates an SSL context and caches it, so you have to set the parameters correctly before doing anything

Pyro4.socketutil.**interruptSocket**(*address*)
bit of a hack to trigger a blocking server to get out of the loop, useful at clean shutdowns

Pyro4.socketutil.**receiveData**(*sock*, *size*)
Retrieve a given number of bytes from a socket. It is expected the socket is able to supply that number of bytes. If it isn't, an exception is raised (you will not get a zero length result or a result that is smaller than what you asked for). The partial data that has been received however is stored in the 'partialData' attribute of the exception object.

Pyro4.socketutil.**sendData**(*sock*, *data*)
Send some data over a socket. Some systems have problems with `sendall()` when the socket is in non-blocking mode. For instance, Mac OS X seems to be happy to throw EAGAIN errors too often. This function falls back to using a regular send loop if needed.

Pyro4.socketutil.**setKeepalive**(*sock*)
sets the SO_KEEPALIVE option on the socket, if possible.

Pyro4.socketutil.**setNoDelay**(*sock*)
sets the TCP_NODELAY option on the socket (to disable Nagle's algorithm), if possible.

Pyro4.socketutil.**setNoInherit**(*sock*)
Mark the given socket fd as non-inheritable to child processes

Pyro4.socketutil.**setReuseAddr**(*sock*)
sets the SO_REUSEADDR option on the socket, if possible.

2.13.6 Pyro4.message — Pyro wire protocol message

The pyro wire protocol message.

class Pyro4.message.**Message**(*msgType*, *databytes*, *serializer_id*, *flags*, *seq*, *annotations=None*, *hmac_key=None*)
Pyro write protocol message.

Wire messages contains of a fixed size header, an optional set of annotation chunks, and then the payload data. This class doesn't deal with the payload data: (de)serialization and handling of that data is done elsewhere. Annotation chunks are only parsed, except the 'HMAC' chunk: that is created and validated because it is used as a message digest.

The header format is:

```
4   id ('PYRO')
2   protocol version
2   message type
2   message flags
2   sequence number
4   data length    (i.e. 2 Gb data size limitation)
2   data serialization format (serializer id)
2   annotations length (total of all chunks, 0 if no annotation chunks present)
2   (reserved)
2   checksum
```

After the header, zero or more annotation chunks may follow, of the format:

```
4   id (ASCII)
2   chunk length
x   annotation chunk databytes
```

After that, the actual payload data bytes follow.

The sequencenumber is used to check if response messages correspond to the actual request message. This prevents the situation where Pyro would perhaps return the response data from another remote call (which would not result in an error otherwise!) This could happen for instance if the socket data stream gets out of sync, perhaps due To some form of signal that interrupts I/O.

The header checksum is a simple sum of the header fields to make reasonably sure that we are dealing with an actual correct PYRO protocol header and not some random data that happens to start with the 'PYRO' protocol identifier.

Pyro now uses two annotation chunks that you should not touch yourself: 'HMAC' contains the hmac digest of the message data bytes and all of the annotation chunk data bytes (except those of the HMAC chunk itself). 'CORR' contains the correlation id (guid bytes) Other chunk names are free to use for custom purposes, but Pyro has the right to reserve more of them for internal use in the future.

decompress_if_needed()
Decompress the message data if it is compressed.

classmethod from_header(*headerData*)
Parses a message header. Does not yet process the annotations chunks and message data.

hmac()
returns the hmac of the data and the annotation chunk values (except HMAC chunk itself)

static ping(*pyroConnection*, *hmac_key=None*)
Convenience method to send a 'ping' message and wait for the 'pong' response

classmethod recv(*connection*, *requiredMsgTypes=None*, *hmac_key=None*)
Receives a pyro message from a given connection. Accepts the given message types (None=any, or pass a sequence). Also reads annotation chunks and the actual payload data. Validates a HMAC chunk if present.

send(*connection*)
send the message as bytes over the connection

to_bytes()
creates a byte stream containing the header followed by annotations (if any) followed by the data

Pyro4.message.**secure_compare**()
compare_digest(a, b) -> bool

Return 'a == b'. This function uses an approach designed to prevent timing analysis, making it appropriate for cryptography. a and b must both be of the same type: either str (ASCII only), or any bytes-like object.

Note: If a and b are of different lengths, or if an error occurs, a timing attack could theoretically reveal information about the types and lengths of a and b–but not their values.

MSG_*
(*int*) The various message type identifiers

FLAGS_*
(*int*) Various bitflags that specify the characteristics of the message, can be bitwise or-ed together

2.13.7 Pyro4.constants — Constant value definitions

Pyro4.constants.**VERSION**
The library version string (currently "4.73").

Pyro4.constants.**DAEMON_NAME**
Standard object name for the Daemon itself, preferred over hardcoding it as a string literal.

Pyro4.constants.**NAMESERVER_NAME**
Standard object name for the Name server itself, preferred over hardcoding it as a string literal.

Pyro4.constants.**FLAME_NAME**
Standard object name for the Flame server, preferred over hardcoding it as a string literal.

Pyro4.constants.**PROTOCOL_VERSION**
Pyro's network protocol version number.

2.13.8 Pyro4.config — Configuration items

Pyro's configuration is available in the `Pyro4.config` object. Detailed information about the API of this object is available in the *Configuring Pyro* chapter.

Note: creation of the `Pyro4.config` object

This object is constructed when you import Pyro4. It is an instance of the `Pyro4.configuration.Configuration` class. The package initializer code creates it and the initial configuration is determined (from defaults and environment variable settings). It is then assigned to `Pyro4.config`.

2.13.9 `Pyro4.errors` — Exception classes

The exception hierarchy is as follows:

```
Exception
  |
  +-- PyroError
        |
        +-- NamingError
        +-- DaemonError
        +-- SecurityError
        +-- CommunicationError
              |
              +-- ConnectionClosedError
              +-- TimeoutError
              +-- ProtocolError
                    |
                    +-- SerializeError
```

Definition of the various exceptions that are used in Pyro.

exception `Pyro4.errors.`**CommunicationError**
Base class for the errors related to network communication problems.

exception `Pyro4.errors.`**ConnectionClosedError**
The connection was unexpectedly closed.

exception `Pyro4.errors.`**DaemonError**
The Daemon encountered a problem.

exception `Pyro4.errors.`**MessageTooLargeError**
Pyro received a message or was trying to send a message that exceeds the maximum message size as configured.

exception `Pyro4.errors.`**NamingError**
There was a problem related to the name server or object names.

exception `Pyro4.errors.`**ProtocolError**
Pyro received a message that didn't match the active Pyro network protocol, or there was a protocol related error.

exception `Pyro4.errors.`**PyroError**
Generic base of all Pyro-specific errors.

exception `Pyro4.errors.`**SecurityError**
A security related error occurred.

exception `Pyro4.errors.`**SerializeError**
Something went wrong while (de)serializing data.

exception `Pyro4.errors.`**TimeoutError**
A call could not be completed within the set timeout period, or the network caused a timeout.

2.13.10 Pyro4.test.echoserver — Built-in echo server for testing purposes

Echo server for test purposes. This is usually invoked by starting this module as a script:

python -m Pyro4.test.echoserver or simply: **pyro4-test-echoserver**

It is also possible to use the *EchoServer* in user code but that is not terribly useful.

class Pyro4.test.echoserver.**EchoServer**

The echo server object that is provided as a Pyro object by this module. If its `verbose` attribute is set to `True`, it will print messages as it receives calls.

echo(*message*)

return the message

error()

generates a simple exception without text

error_with_text()

generates a simple exception with message

generator()

a generator function that returns some elements on demand

oneway_echo(*message*)

just like echo, but oneway; the client won't wait for response

oneway_slow()

prints a message after a certain delay, and returns; but the client won't wait for it

shutdown()

called to signal the echo server to shut down

slow()

returns (and prints) a message after a certain delay

2.13.11 Pyro4.utils.flame — Foreign Location Automatic Module Exposer

Pyro FLAME: Foreign Location Automatic Module Exposer. Easy but potentially very dangerous way of exposing remote modules and builtins. Flame requires the pickle serializer to be used.

Pyro4.utils.flame.**connect**(*location*, *hmac_key=None*)

Connect to a Flame server on the given location, for instance localhost:9999 or ./u:unixsock This is just a convenience function to creates an appropriate Pyro proxy.

Pyro4.utils.flame.**start**(*daemon*)

Create and register a Flame server in the given daemon. Be *very* cautious before starting this: it allows the clients full access to everything on your system.

Pyro4.utils.flame.**createModule**(*name*, *source*, *filename='<dynamic-module>'*, *namespace=None*)

Utility function to create a new module with the given name (dotted notation allowed), directly from the source string. Adds it to sys.modules, and returns the new module object. If you provide a namespace dict (such as `globals()`), it will import the module into that namespace too.

class Pyro4.utils.flame.**Flame**

The actual FLAME server logic. Usually created by using `core.Daemon.startFlame()`. Be *very* cautious before starting this: it allows the clients full access to everything on your system.

builtin(*name*)

returns a proxy to the given builtin on the server

console()
get a proxy for a remote interactive console session

evaluate(*expression*)
evaluate an expression and return its result

execute(*code*)
execute a piece of code

getfile(*filename*)
read any accessible file from the server

getmodule(*modulename*)
obtain the source code from a module on the server

module(*name*)
Import a module on the server given by the module name and returns a proxy to it. The returned proxy does not support direct attribute access, if you want that, you should use the `evaluate` method instead.

sendfile(*filename*, *filedata*)
store a new file on the server

sendmodule(*modulename*, *modulesource*)
Send the source of a module to the server and make the server load it. Note that you still have to actually `import` it on the server to access it. Sending a module again will replace the previous one with the new.

2.13.12 Pyro4.futures — asynchronous calls

Support for Futures (asynchronously executed callables). If you're using Python 3.2 or newer, also see http://docs.python.org/3/library/concurrent.futures.html#future-objects

class Pyro4.futures.**Future**(*somecallable*)
Holds a callable that will be executed asynchronously and provide its result value some time in the future. This is a more general implementation than the AsyncRemoteMethod, which only works with Pyro proxies (and provides a bit different syntax). This class has a few extra features as well (delay, canceling).

cancel()
Cancels the execution of the future altogether. If the execution hasn't been started yet, the cancellation is succesful and returns True. Otherwise, it failed and returns False.

delay(*seconds*)
Delay the evaluation of the future for the given number of seconds. Return True if succesful otherwise False if the future has already been evaluated.

iferror(*exceptionhandler*)
Specify the exception handler to be invoked (with the exception object as only argument) when calculating the result raises an exception. If no exception handler is set, any exception raised in the asynchronous call will be silently ignored. Returns self so you can easily chain other calls.

then(*call*, **args*, ***kwargs*)
Add a callable to the call chain, to be invoked when the results become available. The result of the current call will be used as the first argument for the next call. Optional extra arguments can be provided in args and kwargs. Returns self so you can easily chain then() calls.

class Pyro4.futures.**FutureResult**
The result object for asynchronous Pyro calls. Unfortunatley it should be similar to the more general Future class but it is still somewhat limited (no delay, no canceling).

iferror(*exceptionhandler*)
Specify the exception handler to be invoked (with the exception object as only argument) when asking for

the result raises an exception. If no exception handler is set, any exception result will be silently ignored (unless you explicitly ask for the value). Returns self so you can easily chain other calls.

ready
Boolean that contains the readiness of the asynchronous result

then(*call*, **args*, ***kwargs*)
Add a callable to the call chain, to be invoked when the results become available. The result of the current call will be used as the first argument for the next call. Optional extra arguments can be provided in args and kwargs. Returns self so you can easily chain then() calls.

value
The result value of the call. Reading it will block if not available yet.

wait(*timeout=None*)
Wait for the result to become available, with optional timeout (in seconds). Returns True if the result is ready, or False if it still isn't ready.

2.13.13 Socket server API contract

For now, this is an internal API, used by the Pyro Daemon. The various servers in Pyro4.socketserver implement this.

class SocketServer_API
Methods:

init(*daemon*, *host*, *port*, *unixsocket=None*)
Must bind the server on the given host and port (can be None). daemon is the object that will receive Pyro invocation calls (see below). When host or port is None, the server can select something appropriate itself. If possible, use `Pyro4.config.COMMTIMEOUT` on the sockets (see *Pyro4.config — Configuration items*). Set `self.sock` to the daemon server socket. If unixsocket is given the name of a Unix domain socket, that type of socket will be created instead of a regular tcp/ip socket.

loop(*loopCondition*)
Start an endless loop that serves Pyro requests. loopCondition is an optional function that is called every iteration, if it returns False, the loop is terminated and this method returns.

events(*eventsockets*)
Called from external event loops: let the server handle events that occur on one of the sockets of this server. eventsockets is a sequence of all the sockets for which an event occurred.

shutdown()
Initiate shutdown of a running socket server, and close it.

close()
Release resources and close a stopped server. It can no longer be used after calling this, until you call initServer again.

wakeup()
This is called to wake up the `requestLoop()` if it is in a blocking state.

Properties:

sockets
must be the list of all sockets used by this server (server socket + all connected client sockets)

sock
must be the server socket itself.

locationStr
must be a string of the form `"serverhostname:serverport"` can be different from the host:port

arguments passed to initServer. because either of those can be None and the server will choose something appropriate. If the socket is a Unix domain socket, it should be of the form `"./u:socketname"`.

2.14 Running on alternative Python implementations

Pyro is a pure Python library so you should be able to use it with any compatible Python implementation. There are a few gotchas however. If possible please use the most recent version available of your Python implementation.

Note: You may have to install the serpent serialization library manually (this is a dependency). Check that you can `import serpent` to make sure it is installed.

2.14.1 IronPython

IronPython is a Python implementation running on the .NET virtual machine.

- Pyro runs with IronPython 2.7.5. Older versions may or may not work, and can lack required modules such as zlib.
- IronPython cannot properly serialize exception objects, which could lead to problems when dealing with Pyro's enhanced tracebacks. For now, Pyro contains a workaround for this bug.
- You may have to use the `-X:Frames` command line option when starting Ironpython. (one of the libraries Pyro4 depends on when running in Ironpython, requires this)

2.14.2 Pypy

Pypy is a Python implementation written in Python itself, and it usually is quite a lot faster than the default implementation because it has a JIT (Just in time)-compiler.

Pyro runs happily on recent versions of Pypy.

2.15 Pyrolite - client library for Java and .NET

This library allows your Java or .NET program to interface very easily with the Python world. It uses the Pyro protocol to call methods on remote objects. It also supports convenient access to a Pyro Flame server including the remote interactive console.

Pyrolite is a tiny library that implements a part of the client side Pyro library, hence its name 'lite'. Pyrolite has no additional dependencies. So if you don't need Pyro's full feature set, and don't require your Java/.NET code to host Pyro objects itself, Pyrolite may be a good choice to connect java or .NET and python.

Pyrolite also contains a feature complete implementation of Python's `pickle` protocol (with fairly intelligent mapping of datatypes between Python and Java/.NET), and a small part of Pyro's client network protocol and proxy logic. It can use the Serpent serialization format as well.

Getting the .NET version: The .NET version is available using the nuget package manager, package name is `Razorvine.Pyrolite` (and `Razorvine.Serpent`, which is a dependency). Package info.

Getting the Java version: The Java library can be obtained from Maven, groupid `net.razorvine` artifactid `pyrolite`.

Source is on Github: https://github.com/irmen/Pyrolite

Readme: https://raw.githubusercontent.com/irmen/Pyrolite/master/README.txt

Small code example in Java:

```
import net.razorvine.pyro.*;

NameServerProxy ns = NameServerProxy.locateNS(null);
PyroProxy remoteobject = new PyroProxy(ns.lookup("Your.Pyro.Object"));
Object result = remoteobject.call("pythonmethod", 42, "hello", new int[]{1,2,3});
String message = (String)result;  // cast to the type that 'pythonmethod' returns
System.out.println("result message="+message);
remoteobject.close();
ns.close();
```

You can also read a more elaborate example. That writeup is an elaboration of the Pyro simple example greeting.py appearing in the introduction chapter, but with a Java (rather than Python) client.

The same example in C#:

```
using Razorvine.Pyro;

using( NameServerProxy ns = NameServerProxy.locateNS(null) )
{
    using( PyroProxy something = new PyroProxy(ns.lookup("Your.Pyro.Object")) )
    {
        object result = something.call("pythonmethod", 42, "hello", new int[]{1,2,3});
        string message = (string)result;  // cast to the type that 'pythonmethod'␣
↪returns
        Console.WriteLine("result message="+message);
    }
}
```

You can also use Pyro Flame rather conveniently because of some wrapper classes:

```
Config.SERIALIZER = Config.SerializerType.pickle;   // flame requires the pickle␣
↪serializer
PyroProxy flame = new PyroProxy(hostname, port, "Pyro.Flame");
FlameModule r_module = (FlameModule) flame.call("module", "socket");
System.out.println("hostname=" + r_module.call("gethostname"));

FlameRemoteConsole console = (FlameRemoteConsole) flame.call("console");
console.interact();
console.close();
```

2.16 Change Log

Pyro 4.73

- include LICENSE file in distribution
- avoid decode error when dealing with memoryview annotations

Pyro 4.72

- (source files: normalized line endings to LF)
- the -k command line option to supply a HMAC encryption key on the command line for the name server, nsc, echoserver, flameserver and httpgateway tools is now deprecated (and will print a warning if used). It is a

security issue because the key used is plainly visible. If you require proper security, use Pyro's 2-way SSL feature. Alternatively, set the HMAC key in the (new) environment variable PYRO_HMAC_KEY if you still have to use it before launching the aforementioned tools.

Pyro 4.71

- updated `msgpack` dependency (was `msgpack-python` but that name is now deprecated)
- fixed restart and force reload commands of the contrib/init.d/pyro4-nsd script, and changed its port binding from 9999 back to 9090 which is Pyro's default.
- serpent 1.24 library now required to fix some api deprecation warnings when using Python 3.7 or newer.
- updated sphinx documentation theme

Pyro 4.70

- bump to version 4.70 to emphasize the following change:
- **incompatible API change** for python 3.7 compatibility: renaming of `async` function and keyword arguments in the API: Renamed `Pyro4.core.async` to `Pyro4.core.asyncproxy` (and its occurrence in `Pyro4`) and the `async` keyword argument in some methods to `asynchronous`. This had to be done because `async` (and `await`) are now parsed as keywords in Python 3.7 and using them otherwise will result in a SyntaxError when loading the module. It is suggested you stop using the `asyncproxy` function and instead create asynchronous proxies using the `_pyroAsync` method on the regular proxy.
- For existing code running on Python *older than 3.7*, a backwards compatibility feature is present to still provide the `async` function and keyword arguments as they were supported on previous Pyro versions. But also for that older environments, it's advised to migrate away from them and start using the new names.
- Proxy and Daemon have a new 'connected_socket' parameter. You can set it to a user-supplied connected socket that must be used by them instead of creating a new socket for you. Connected sockets can be created using the socket.socketpair() function for instance, and allow for easy and efficient communication over an internal socket between parent-child processes or threads, using Pyro. Also see the new 'socketpair' example.
- dropped support for Python 3.3 (which has reached end-of-life status). Supported Python versions are now 2.7, and 3.4 or newer. (the life cycle status of the Python versions can be seen here https://devguide.python.org/#status-of-python-branches)

Pyro 4.63

- fixed bug in autoproxy logic where it registered the wrong type if daemon.register() was called with a class instead of an object (internal register_type_replacement method)
- added check in @expose method to validate the order of decorators on a method (@expose should come last, after @classmethod or @staticmethod).
- added resource tracking feature (see 'Automatically freeing resources when client connection gets closed' in the Tips & Tricks chapter)
- the warning about a class not exposing anything now actually tells you the correct class

Pyro 4.62

- **major new feature: SSL/TLS support added** - a handful of new config items ('SSL' prefixed), supports server-only certificate and also 2-way-ssl (server+client certificates). For testing purposes, self-signed server and client certificates are available in the 'certs' directory. SSL/TLS in Pyro is supported on Python 2.7.11+ or Python 3.4.4+ (these versions have various important security related changes such as disabling vulnerable cyphers or protocols by default)
- added SSL example that shows how to configure 2-way-SSL in Pyro and how to do certificate verification on both sides.
- added cloudpickle serialization support (https://github.com/cloudpipe/cloudpickle/)

- added a small extended-pickle example that shows what dill and cloudpickle can do (send actual functions)
- daemon is now more resilient to exceptions occurring with socket communications (it logs them but is otherwise not interrupted) (this was required to avoid errors occurring in the SSL layer stopping the server)
- some small bugs fixed (crash when logging certain errors in thread server, invalid protected members showing up on pypy3)
- the `raise data` line in a traceback coming from Pyro now has a comment after it, telling you that you probably should inspect the remote traceback as well.
- *note*: if you're using Python 3 only and are interested in a modernized version of Pyro, have a look at Pyro5: https://github.com/irmen/Pyro5 It's experimental work in progress, but it works pretty well.
- *note*: Pyro4 is reaching a state where I consider it "feature complete": I'm considering not adding more new features but only doing bug-fixes. New features (if any) will then appear only in Pyro5.

Pyro 4.61

- serpent 1.23 library now required.
- Pyro4.utils.flame.connect now has an optional `hmac_key` argument. You can now use this utility function to connect to a flame server running with a hmac_key. (Previously it didn't let you specify the client hmac_key so you had to create a flame proxy manually, on which you then had to set the _pyroHmacKey property).
- main documentation is now http://pyro4.readthedocs.io instead of http://pythonhosted.org/Pyro4/

Pyro 4.60

- `Pyro4.core.async()` and `proxy._pyroAsync()` now return `None`, instead of the proxy object. This means you'll have to change your code that expects a proxy as return value, for instance by creating a copy of the proxy yourself first. This change was done to avoid subtle errors where older code still assumed it got a *copy* of the proxy, but since 4.57 that is no longer done and it is handed back the same proxy. By returning `None` now, at least the old code will now crash with a clear error, instead of silently continuing with the possibility of failing in weird ways later.

Pyro 4.59

- Fixed pyro4-check-config script.

Pyro 4.58

- Added feature to be able to pass through serialized arguments unchanged via `Pyro4.core.SerializedBlob`, see example 'blob-dispatch'
- Fixed a fair amount of typos in the manual and readme texts.
- The stockquotes tutorial example now also has a 'phase 3' just like the warehouse tutorial example, to show how to run it on different machines.

Pyro 4.57

- Pyro4.core.async() and proxy._pyroAsync() no longer return a copy of the proxy but rather modify the proxy itself, in an attempt to reduce the number of socket connections to a server. They still return the proxy object for api compatibility reasons.
- async result now internally retries connection after a short delay, if it finds that the server has no free worker threads to accept the connection. If COMMTIMEOUT has been set, it retries until the timeout is exceeded. Otherwise it retries indefinitely util it gets a connection.
- _StreamResultIterator now stops all communication as soon as StopIteration occurred, this avoids unnecessary close calls to remote iterators.

Pyro 4.56

- optional msgpack serializer added (requires msgpack library, see https://pypi.python.org/pypi/msgpack)
- fixed possible crash in closing of remote iterators (they could crash the proxy by screwing up the internal sequence number).
- json serializer can now serialize uuid.UUID, datetime and decimal objects (into strings, like serpent does)
- serializers can now deal with memoryview and bytearray serialized data input types.
- serpent library dependency updated to 1.19 to be able to deal with memoryview and bytearray inputs.
- added `response_annotations` on the call context object to be able to access annotations more easily than having to subclass Proxy or Daemon.
- `Proxy._pyroAnnotations` and `Daemon.annotations` no longer needs to call super, the annotations you return here are now automatically merged with whatever Pyro uses internally.
- Proxy and Daemon now contain the ip address family in their repr string.
- Pyro now logs the ip address family for proxy or daemon socket connections.
- ipv6 doesn't have broadcasts, so Pyro no longer uses them when ipv6 is in use.
- improved the docs about binary data transfer a bit.
- documentation is now also available on ReadTheDocs: http://pyro4.readthedocs.io/
- fixed various examples

Pyro 4.55

- *CRITICAL FIX:* serpent library dependency updated to 1.17 to fix issues with encoding and parsing strings containing 0-bytes. Note that if you don't want to upgrade Pyro itself yet, you should manually upgrade the serpent library to get this fix.
- Prefer selectors2 over selectors34 if it is available (Python 3.4 or older, to have better semantics of failing syscalls)
- Removed THREADING2 config item and Pyro4.threadutil module. (the threading2 third party module is old and seems unmaintained and wasn't useful for Pyro anyway)
- Improved module structure; fixed various circular import dependencies. This also fixed the RuntimeWarning about sys.modules, when starting the name server.
- To achieve the previous item, had to move `resolve` and `locateNS` from `Pyro4.naming` to `Pyro4.core`. They're still available on their old location for backwards compatibility for now. Of course, they're also still on their old "shortcut" location in `Pyro4` directly.
- Removed the publicly visible serializer id numbers from the message module. They're internal protocol details, user code should always refer to serializers by their name.
- When a connection cannot be made, the address Pyro tries to connect to is now also included in the error message.
- Added overridable `Daemon.housekeeping()` method.
- Improved error message in case of invalid ipv6 uri.
- Fixed various examples, and made the Pyro4 main api package documentation page complete again.

Pyro 4.54

- Serpent serializer: floats with value NaN will now be properly serialized and deserialized into a float again, instead of the class dict `{'__class__':'float', 'value':'nan'}` Note that you can achieve the same for older versions of Pyro by manually registering a custom converter: `Pyro4.util.SerializerBase.register_dict_to_class("float", lambda _, d:  float(d["value"]))`

- Removed platform checks when using dill serializer, latest Pypy version + latest dill (0.2.6) should work again. Other platforms might still expose problems when trying to use dill (IronPython), but they are now considered to be the user's problem if they attempt to use this combination.
- Applied version detection patch from Debian package to contrib/init.d/pyro4-nsd
- Don't crash immediately at importing Pyro4 when the 'selectors' or 'selectors34' module is not available. Rationale: This is normally a required dependency so the situation should usually not occur at all. But it can be problematic on Debian (and perhaps other distributions) at this time, because this module may not be packaged/not be available. So we now raise a proper error message, but only when an attempt is made to actually create a multiplex server (all other parts of Pyro4 are still usable just fine in this case). The selectors module is available automatically on Python 3.4 or newer, for older Pythons you have to install it manually or via the python2-selectors34 package if that is available.
- Fixed crash when trying to print the repr or string form of a Daemon that was serialized.
- Changed uuid.uuid1() calls to uuid.uuid4() because of potential issues with uuid1 (obscure resource leak on file descriptors on /var/lib/libuuid/clock.txt). Pyro4 already used uuid4() for certain things, it now exclusively uses uuid4().
- Fixed a few IronPython issues with several unit tests.
- Improved the installation chapter in the docs.

Pyro 4.53

- *CRITICAL FIX:* serpent library dependency updated to 1.16 to fix floating point precision loss error on older python versions. Note that if you don't want to upgrade Pyro itself yet, you should manually upgrade the serpent library to get this fix.
- added unittest to check that float precision is maintained in the serializers
- fixed some typos in docs and docstrings, improved daemon metadata doc.
- mailing list (`pyro@freelists.org`) has been discontinued.

Earlier versions

Change history for earlier versions is available by looking at older versions of this documentation. One way to do that is looking at previous versions in the Github source repository.

2.17 Software License and Disclaimer

Pyro - Python Remote Objects - version 4.x

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.

This is the MIT Software License which is OSI-certified, and GPL-compatible.

2.18 Indices and tables

- genindex
- search

Python Module Index

p

Index

Symbols

A

B

C

D

R

S

T

U

V

W

Y