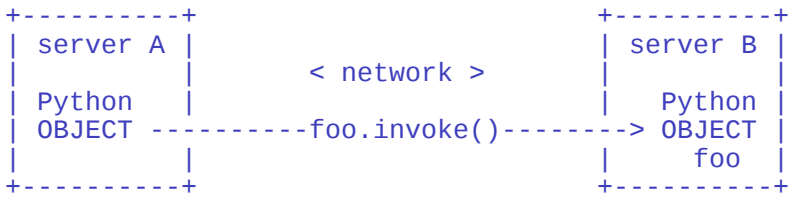
Python RMI

(Using Pyro)

1. Introduction

For a good understanding of Pyro it is necessary to know the different elements in a Pyro system. Keep in mind that in a distributed object system, the client/server style is difficult to see. Most of the time all parts of the system switch roles, one moment it's a client calling a remote object, the other moment it is itself an object that is called from other parts of the system. For a good understanding though, it is important to see that during a single method call, there are always two distinct parts of the system: the client part that initiates the method call, and the server part that accepts and executes the call. To be precise, there are actually three parts: between the client and the server is the distributed object middleware, in this case: Pyro.

Another issue is that a client can - of course! - use more than one remote object, but also that a single server can have more than one object implementation. Thus, single objects in their own right are neither a client nor a server. I'll define the executable parts of the system that contain the objects as clients and servers (depending on their actual role). For simple Pyro applications, usually the different Python modules clearly show the different parts of the system, and they are the clients and servers I'm talking about here.



2. Pyro shell scripts

Pyro provides several tools to help us during development of a Pyro application. The Pyro Name Server is also started and controlled by two of these scripts.

3. Client program

This is the part of our system that sends requests to the server program, to perform certain actions. It's the code that actually uses the remote objects by calling their methods.

Pyro client programs look suspiciously like normal Python programs. But that's the whole point of Pyro: it enables you to build distributed object systems with minimal effort. It makes the use of remote objects (almost) transparent.
Client code has to perform some initialization and setup steps. And because we're talking remote objects here, they cannot create object instances in the usual way.

They have to use a two-step mechanism:

Find the location identifier of the required object. This is done by using the Pyro Name Server.
* pyro4-ns #start the naming server
* pyro4-nsc list #check the currently registered object on the naming server

Create a special kind of object that actually calls the remote object. This is called a proxy.
* Pyro4.Proxy("PYRONAME:ayushi.warehouse") #setting up the proxy

Once it has this proxy object, the client can call it just as if it were a regular -local- Python object.

#here is our client program's
person.py

```
from __future__ import print_function
import sys

if sys.version_info < (3, 0):      #check for the python version
    input = raw_input


class Person(object):         #create the Person class with the below functionality
    def __init__(self, name):
        self.name = name

    def visit(self, warehouse):       #function to visit the warehouse
        print("This is {0}.".format(self.name)) #{0} will print the first
                                                #argument of the .format().
        self.deposit(warehouse)          #deposit item to the warehouse.
        self.retrieve(warehouse)         #retrieve item from the warehouse
        print("Thank you, come again!")

    def deposit(self, warehouse):     #function to deposit item in 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):    #function to retrieve item in 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)
```

4. Server program

The server is the home of the objects that are accessed remotely. Every object instance has to be part of a Python program, so this is it. The server has to do several things:

Create object instances using a extremely tiny bit of Pyro plumbing
Give names to those instances, and register these with the Name Server
Announce to Pyro that it has to take care of these instances
Tell Pyro to sit idle in a loop waiting for incoming method calls

#here is our server program's
warehouse.py

```
from __future__ import print_function
import Pyro4


@Pyro4.expose  #decorator on the Warehouse class definition to tell Pyro it is allowed to access
                                                                    #the class remotely
@Pyro4.behavior(instance_mode="single")
class Warehouse(object):                    #creating the warehouse class.
    def __init__(self):
        self.contents = ["chair", "bike", "flashlight", "laptop",
"couch"]

    def list_contents(self):                    #function to show list of contents.
        return self.contents

    def take(self, name, item):                 #function to remove an item from list
        self.contents.remove(item)
        print("{0} took the {1}.".format(name, item))

    def store(self, name, item):                #function to add an item to list
        self.contents.append(item)
        print("{0} stored the {1}.".format(name, item))


def main():                                     #main function to start the pyro daemon
    Pyro4.Daemon.serveSimple(
            {
                Warehouse: "ayushi.warehouse"
            },
            ns = True)  #make object available on naming server

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

Note : actually we are not having the distributed environment. So we illustrate this python remote objects in a single machine. For this we have to write an additional program which act as client in this case. It shows the persons visit the warehouse.

#here is our visitor program's
visit.py

```
import sys
import Pyro4
import Pyro4.util
from person import Person

sys.excepthook = Pyro4.util.excepthook  #It will deal with exceptions and stack
                    #trace our program produce when something wrong with the pyro object
warehouse = Pyro4.Proxy("PYRONAME:ayushi.warehouse") #It is setting the URI
for the warehouse object and registered it on the naming server.
manvi = Person("manvi")    #initializing the clients
ayushi = Person("ayushi")
manvi.visit(warehouse)    #clients access the remote object of the warehouse server
ayushi.visit(warehouse)
```

5. Remote object

Pyro object is just a regular Python object. The object doesn't know and doesn't have to know it's part of a Pyro server, and called remotely.

6. Proxy

A proxy is a special kind of object that acts as if it were the actual -remote-object. Pyro clients have to use proxies to forward method calls to the remote objects, and pass results back to the calling code.

*Dynamic proxy
This is a very special Python object provided by Pyro. It is a general proxy for all remote objects!
Dynamic proxy with attribute access support.
This allows you to access object attributes directly with normal Python syntax . Because this requires even more builtin logic, this proxy is a few percent slower than the others. You can choose to use this proxy, or one of the others. Proxies are bound to certain location identifiers, so they know on whose behalf the're running.

Proxy objects can be pickled: you can toss proxy objects around within your Pyro system.

7. Pyro daemon

It's getting more technical now. Each server has to have a way of getting remote method calls and dispatching them to the required objects. For this task Pyro provides a Daemon. A server just creates one of those and tells it to sit waiting

for incoming requests. The Daemon takes care of everything from that moment on. Every server has one Daemon that knows about all the Pyro objects the server provides.

8. Location discovery and naming

One of the most useful services a distributed object system can have is a Name Server. Such a service is a central database that knows the names and corresponding locations of all objects in the system.
Pyro has a Name Server that performs this task very well. Pyro Servers register their objects and location with the Name Server. Pyro Clients query the server for location identifiers. They do this by providing (human-readable) object names. They get a Pyro Universal Resource Identifier.
You might be wondering: how can a Pyro client find the Name Server itself?! Good question. There are three possibilities:

Rely on the broadcast capability of the underlying network. This is extremely easy to use (you don't have to do anything!) and works in most cases.
Somehow obtain the hostname of the machine the Name Server is running on. You can then directly contact this machine. The Name Server will respond with its location identifier.
Obtain the location identifier using another way such as file transfer or email. The Name Server writes its location identifier to a special file and you can read it from there. Then you can create a Name Server proxy directly, bypassing the Name Server Locator completely.

Notice: the Pyro.xxxxx namespace is reserved for Pyro itself (for instance, the Name Server is called Pyro.NameServer). Don't use it.

There are two other ways to connect to a certain object you know the name of. These are the PYRONAME:// and the PYROLOC:// URI formats, instead of the regular PYRO:// URIs. The first is essentially a shortcut to the Name Server; it will find and query the Name Server under the surface (you don't have to do anything). The second bypasses the Name Server completely and directly queries a Pyro server host for the required object.

9. Communication protocol

The communication between a client and a server basically consists of two kinds of messages:

Method call request.
This message consists of some identification of the target object, the method called, and the arguments for this call.
Return value reply.
This message is no more than the return value of the method call.

By default Pyro uses the PYRO protocol (duh!) that relies on Python's built-in pickle facility to create these messages. The transport over the network is done using TCP/IP. The way Pyro designed should make it easy to use other protocols, but for now, only the PYRO protocol is implemented.

10.Exceptions

What happens when an error occurs in your server? Pyro can't help you when your server crashes. But it does catch the Python exceptions that occur in your remote objects. They are sent back to the calling code and raised again, just as if they occurred locally. The occurrence is logged in the server log. Thus, Pyro makes no

distinction between user generated exceptions (in the remote object) or exceptions that occur because of runtime errors, such as divide by zero.

The client has no way of telling whether the exception was raised locally or in the remote object. Well, there is a way, but you should really not depend on it. It's only to facilitate debugging in case of a remote exception (the remote exception is contained within the local exception). Any errors from Pyro itself will be signaled by a PyroError exception or one of its exception subclasses.

11. Logging

A good trace facility is paramount in a complex system. Therefore Pyro provides a simple to use logger. It writes messages to a configurable logfile, annotated with a timestamp. It distinguishes errors, warnings and regular notices. Pyro uses it depending on the tracelevel you configured.

You can use the logging facility in your own code too. There is a special user version of the logger that operates independently of the Pyro system logger. You can configure the trace level and logfile location uniquely for both loggers.

By default, logging is turned off completely. You can simply turn it on during the course of your program's execution, or beforehand by setting a simple Pyro configuration option.
s