PyDO: Python Distributed Objects

Manthan Thakar, Tirthraj Parmar, Alankrit Joshi

12/12/2017

Abstract

Python Distributed Object (PyDO) is a distributed object system for Python, which allows users to distribute objects without having to worry about the underlying details of network and distribution. With PyDO, distributing objects is as simple as extending a user-defined class with PyDO's `SharedObject` class. Traditionally, distributed objects are either implemented by the means of Remote Method Invocation or Distributed Shared Memory. Both of these approaches have advantages and disadvantages. PyDO takes relevant ideas from both of these approaches and builds a system that is easy-to-use and efficient at the same time.

Contributions

1. Introduction

Having a simple abstraction to make any kind of object distributed across multiple nodes comes with many advantages to a programmer. It enables programmer to focus on the core problem at hand without worrying about low-level details of efficiently distributing it across nodes. However, such an abstraction has its own set of benefits and drawbacks. Two central parameters to object distribution are performance and transparency. Better abstraction comes at a price of performance, whereas better transparency comes at a price of performance. With PyDO, we try to reach a common ground to provide better parts of both worlds while keeping minimal overhead.

As the name suggests, PyDO is developed for Python. It allows user to extend object parameters and methods to distributed nodes such that they can be operated on as if they were on local node. Internally, PyDO handles object storage, distribution, serialization/deserialization, network transfer, access, update, reliability and weak consistency. It makes objects reusable, extensible and inter-operable across nodes. Moreover, developers can easily distribute load across multiple nodes. This allows extension of Object Oriented paradigm across network boundaries.

2. Background

Under the hood, all distributed object implementations boil down to *message parsing*. What matters after that is how much overhead is introduced by the system. Mainly there are two kinds of abstraction approaches prevalent for object distribution; Explicit Approach and Implicit Approach. Here we talk about both approaches briefly.

Explicit Approach

In this approach, the programmer needs to explicitly specify in code about the remote communication. From there on, the system take care of the invocation, communication and serialization/deserialization. Since the programmer is the one responsible to provide the information on distribution nodes, it gives good performance. However, this doesn't give desired level of transparency. Some implementations may require the programmer to learn its own domain specific language. For example, CORBA requires the programmer to describe the object's interface in CORBA IDL (Interface Definition Language).

Implicit Approach

Contrary to explicit approach, the responsibility of object distribution is on execution environment rather than the program. This gives a complete transparency to the programmer. However, this abstraction comes with a price of lesser efficiency since distribution is handled below the programming language. A popular implementation of this approach is *Distributed Shared Memory (DSM)*. In DSM, programs are given a shared memory space, which is transparently realized via message passing and data caching. Here the shared memory access becomes quite expensive. To gain better efficiency, such systems require to introduce relaxed consistency.

3. Novelty

Although the traditional approaches to distributed objects are well defined and thoroughly explored, they suffer some disadvantages. Sometimes these disadvantages cost usability of these systems and sometimes there are performance concerns. For example, explicit approaches like Java RMI are not developer-friendly. Although these approaches are more efficient, they require the developer to specify where to distribute objects and how to retrieve them. Conversely, DSM based approaches abstract away all the network concerns from the developer at the cost of extra overhead.

PyDO aims to take advantages from both implicit and explicit approaches and build a system that's both performant and easy-to-use. In order to attain that goal, we combine remote invocation with DSM's dynamic placement of objects.

4. System Design

Fig 1: A single node (left) and remote access in network (right)

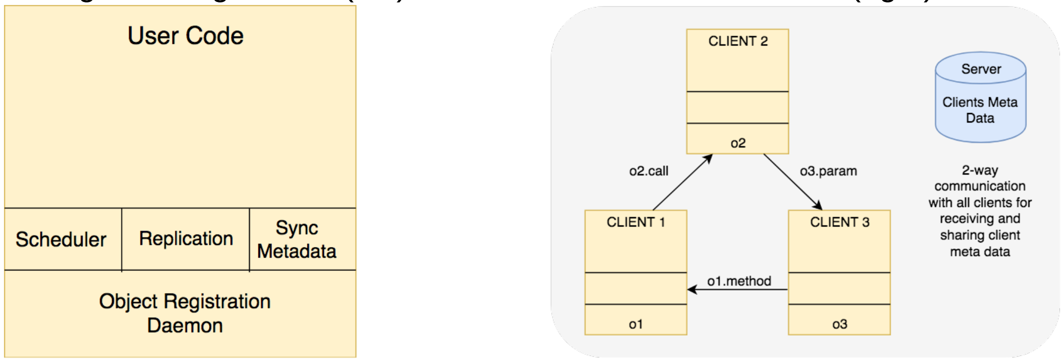


Fig 1 shows the structure of a single node in the network and how they communicate by calling methods on remote objects. As shown, there are a few important components in a single node that facilitates distributed objects. They are discussed briefly below:

Central Server

The central server is a registry for the nodes in the network. It stores their location and metadata about the nodes.

Object Registration Daemon

Object registration daemon is at the lowest level in a node. This daemon is responsible for taking requests for newly instantiated objects. When a user creates a new object from a class that is derived from PyDO's `SharedObject` class, `SharedObject` class registers this user-defined class to this daemon but before that it has to go through the layer above it, which consists of Scheduler, Replication and metadata Sync with central server.

Scheduler

This component is responsible for object placement, which right now chooses one node from the network using round-robin method. Each separate node gets the information about the nodes in the network from the central server and caches it. This information is then periodically updates allowing updates in network to be propogated to each node. The scheduling decision is made locally on each node which makes object distribution faster, since central server is not contacted for each newly created object.

Moreover, this design also allows the system to continue function in the absence of central server. Due to this, when the central server is down, it's not fatal since object scheduling continues with its cache of existing nodes.

Replication

This component is responsible for replicating the objects to multiple other nodes, which allows objects to be found in the network if the primary host of the object fails. This feature combined with decentralized nodes make our system fault-tolerant.

Metadata Sync

This component is responsible for periodically fetching information about nodes from the network. This includes newly added or removed nodes as well as the resource allocation on them.

Communication Mechanism

As mentioned above, for communication mechanism we use remote invocation which allows direct access to the object and methods can be invoked on the address without having to make many requests (in most cases just one request).

5. Implementation

API

PyDO exposes a very simple API for developers. In order to make any user-defined object distributed, the class has to be extended from PyDO's `SharedObject` class. `SharedObject` class then becomes responsible for distributing the object. This allows the system to be very developer-friendly and the underlying distribution is transparent to the user.

```
from PyDO import SharedObject

class Counter(SharedObject):

  def __init__(self, count):
    self.count = count

  def increment(self):
    self.count += 1
```

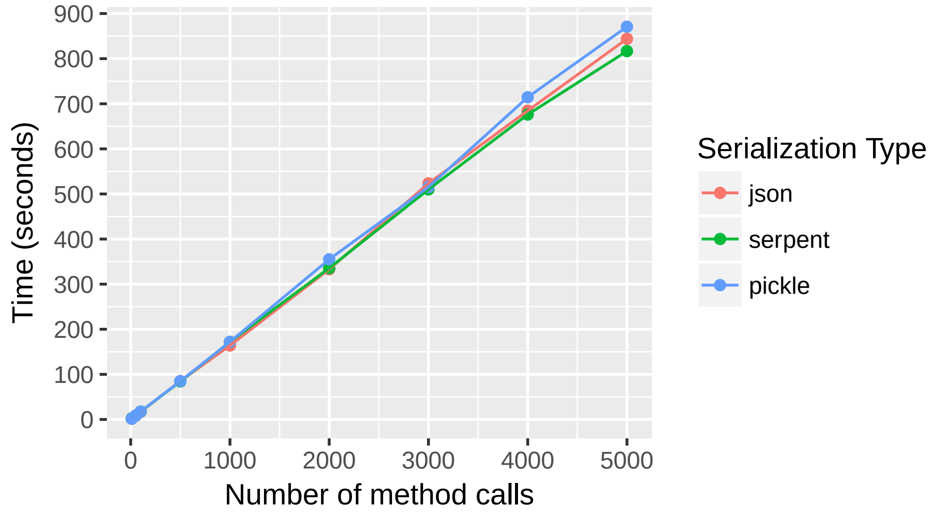
6. Evaluation

As mentioned, PyDO employs remote invocation as the communication mechanism whereby parameters to methods for remote objects are serialized from the caller node and deserialized at callee node. Since, method invocation is the fundamental message passing mechanism in object oriented programming, serialization then becomes a very important aspect of our system. Therefore, we test the method invocation with different serializers to achieve better efficiency.

Fig 2. shows execution times for three serialization methods Pickle, Serpent and JSON. The execution times for each one of the methods are recorded for increasing numbers of method calls. Note that, for all the benchmarks same number of arguments and same object is used to avoid noise.

For very small number of method calls, all the serializers seem to show similar performance. As the number of method calls increase, pickle performs worse than both json and serpent. For our purposes, we use serpent as our serializer since it is similar to json when pythnon dictionaries are passed as arguments, but performs better for other types of arguments.

Fig 2. Evaluation of different serialization Methods



7. Conclusion and Future Work

In summary, we have achieved a decentralized object sharing system with PyDO. There were challenges involved with how the communication with nodes should proceed, but with remote invocation it becomes less cumbersome. Nonetheless, PyDO provides a very easy to use system with promises of lesser overhead than DSM systems.

To make the system more robust, we'd like to explore different placement strategies for objects e.g. based on node resources or priorities. Moreover, it'd be valuable to provide asynchronous proxies for objects so that method calls to remote objects are non-blocking. There still remains a single point of failure of the central server, which isn't fatal for the system but stops other nodes from joining the PyDO network. We'd like to remove that failure. Moreover, sometimes it could prove more performant to place certain objects together on the same node based on their usage. For this purpose, we'd like to explore program analysis approaches for object placement.

Reference

1. Distributed Object Systems: An Evolutionary Perspective (http://www.diss.fu-berlin.de/diss/servlets/MCRFileNodeServlet/FUDISS_derivate_000000000988/2_chapter2.pdf?hosts=)