PyDO: Python Distributed Objects

Manthan Thakar, Tirthraj Parmar, Alankrit Joshi

12/12/2017

Abstract

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

Python Distributed Object (PyDo), provides an easy-to-use abstraction for distributing objects across nodes. 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. With this system, objects become reusable, extensible and interoperable across nodes. Moreover, developers can easily distribute load across multiple nodes. This system allows extension of Object Oriented paradigm across network boundaries.

Background

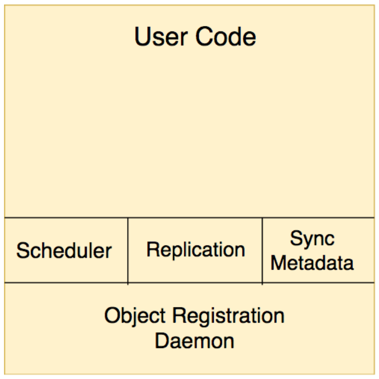
Implicit - explicit bakchodi.

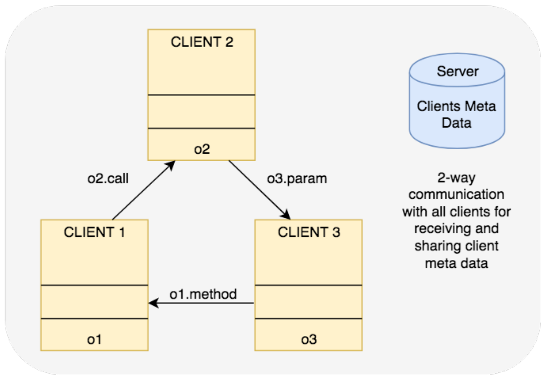
2. Novelty

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 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 the 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.

3. Design




CLIENT 2
o2
o2.call
o3.param
CLIENT 1
CLIENT 3
o1.method
o1
o3
Server
Clients Meta Data
2-way communication with all clients for receiving and sharing client meta data

System

4. 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
```

Challenges

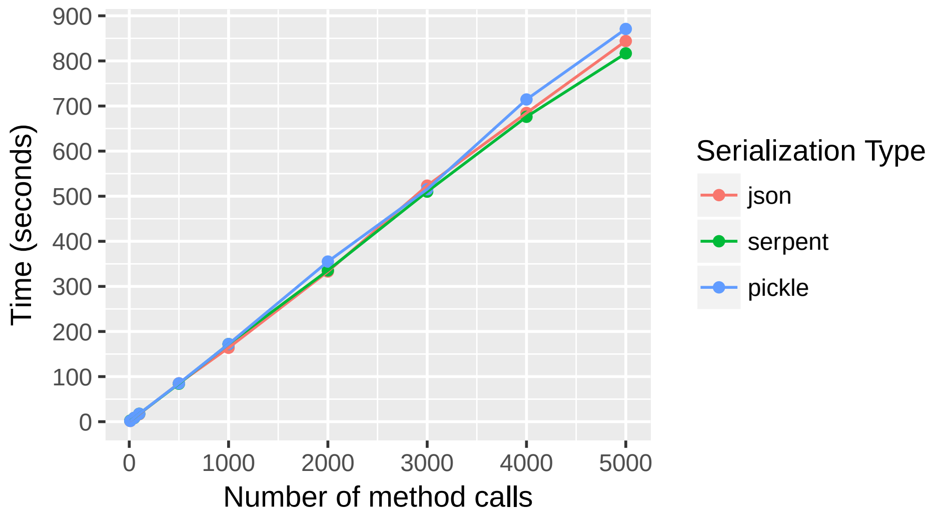
5. Evaluation

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 3. 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 3. Evaluation of different serialization Methods



6. 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.