Summary of My Explorative Work on the RPC Library

State of Development and High Level Overview

I didn't get nearly as far as I thought that I would on this, due to having far more difficulty developing in zmq than expected. Overall, thought, I did manage to get through the harder part of making this work, and the work I did can be used to fully develop an rpc solution. Whether it should be is for a different section...

Locks

locks.py is a library that was licensed for open source usage. It provides a locking object called a RWLock() that can be used to handle reader-writer problems. The writer is able to prevent new readers from entering a critical section while allowing currently reading processes to finish, before writing to a file. Overall, it a powerful tool, and is used liberally in my work.

Publisher and Subscriber

These are simple classes that are designed to abstract away the logic of setting up a pub sub server. The Subscriber is blocking, so it can be run in a thread, and will block until it receives data. Overall, these classes work at a basic level, but have not been tested with a unit test yet.

RPCClient and RPCServer

These classes serve as an abstraction for creating a client server model socket server. The server is pretty simple, but still needs simple unit tests. The client has a lot going on. Because the normal client will block until it receives responses from its requests, it is unsafe to use in a distributed environment. If the server goes down without responding to the client's request, then the thread will be blocked forever. To avoid this issue, I used a lazy polling implementation taken from the zmq documentation (http://zguide.zeromq.org/py:lpclient). Instead of suspending and waiting for a response, it will poll the server for a set amount of time. If that fails, then it will reset its connection to the server and poll again up to a set number of times. An exponential timeout system can easily be added to this, but hasnt yet. Ultimately, the logic of this needs testing, and is not yet 100% bulletproof. Particularly, I think that testing if it can get a response after reconnecting is imporant.

MasterServer

The Master Server is dependant on all the classes described before. This is where the ring resizing protocol is implemented.

Overall, that protocol can be described as follows:

Protocol Phases: 0: The server is running normally 1: Initialize a ring resizing 2: Normalize the cluster to a Synchronized State 3: Notify Servers to return to phase 0

Protocol UML Diagaram: Client Server ------ ------ Begin Phase 1: [ ] <------------------SYNC = { "type":"sync", "name": "Master", "phase": 1, "data":{} }

```
SYNC = {  ------------------------->     [    ]
    "type":"sync",
    "name": "client_name",
    "phase": 1,
    "data":{
        "current_job": job id
    }
}

Begin Phase 2:
[    ]  <------------------  SYNC = {
                                 "type":"sync",
                                 "name": "Master",
                                 "phase": 2,
                                 "data":{
                                     "sync_to": job id,
                                     "members": []
                                 }
                             }

SYNC = {  ------------------------->     [    ]
    "type":"sync",
    "name": "client_name",
    "phase": 2,
    "data":{
        "ack": "ok"
    }
}

Begin Phase 3:
[    ]  <------------------  SYNC = {
                                 "type":"sync",
                                 "name": "Master",
                                 "phase": 3,
                                 "data":{}
                             }

SYNC = {  ------------------------->     [    ]
    "type":"sync",
    "name": "client_name",
    "phase": 3,
    "data":{
        "ack": "ok"
    }
}
```

The way it is configured right now, it waits for all the members to respond before advancing phases, however, this is a bad way to set this up. Only the first phase needs to be managed this way. This only needs to be done up to phase 2. Once every server is stopped and synchronized, then the server should resume normal operation.

There is one glaring issue that I didnt have time to engineer a solution to: what happens when a server dies during the resizing process. Ultimately, this can be implemented in a few ways. First, we can use the health checks to prevent any deadlocks in this protocol. But this would be a slow option. So, we can also send periodic pings to all clients to check their livelihood. Both of these are easy to implement.

Client

Barely started 😐

Take Away: Should you use this?

At the end of the day, I think that this is a viable option. That being said, I think that the amount of development time needed to make this work as reliably as something like rabit or just a flask server is pretty high. The overall pros and cons for using zmq are below, in my opinion:

For

- very lightweight
- Part of the work done
- good doccumentation

Against

- Blocking (the client/server model is very deadlock prone)
- Single threaded by default
- annoying to multithread
- difficult to write code for

In my opinion, I think you could accomplish the same amount of work with flask in less time, and it would be more stable.

RPC Protocol

After giving it a good amount of thought, it seems like the garuntees that are needed to make this work are consistent with that of a consensus algorithm. I know I have gotten push back for this before, so I wanted to take the time to explain how that would work, and why you should consider something similar to or based on raft for your protocol. First off, you dont want to lose any jobs. In raft you can replicate each delta (change in state) with up to n redundant versions. You can define the n replication rate, allowing you to choose between consistency and performance. Overall, it promises that each operation that is executed is not lost, or duplicated. The second reason I think RAFT would be attractive to this use case is because it maintains a log of deltas that have been executed across the system. These deltas represent the state of the system, and can be used to recover failed systems very quickly. Since it can be used to keep track of each nodes progress as well, it can be used to more accurately re-assign work to nodes in the case of a node failing, resulting in quicker, and less wasteful disaster recovery.