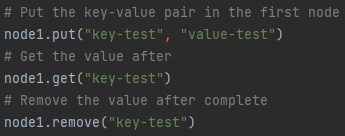
Christopher DeRoche – Distributed Key/Value Store

* Command Line Run Options
  + Main file to run is **driver.py**
    - Requires command line options to select and run consistencies
  + -e 🡪 Eventual Consistency
  + -s 🡪 Sequential Consistency
  + -l 🡪 Linearizable Consistency
  + -a 🡪 Run all consistencies
  + -v 🡪 Verbose logging to show node’s output as well as client’s output
    - Default only shows client output (Used for debugging and showing the consistencies working)
* Configuration Files
  + clients\_config.py
    - Preconfigured 2 clients using a list of dictionaries
  + nodes\_config.py
    - Preconfigured 3 nodes using a list of dictionaries
* Overall Design
  + driver.py 🡪 Spawns all the client and node processes using command line arguments and multiprocessing
    - kv\_node.py 🡪 Contains the classes for each consistency along with XML RPC instance setup
      * nodes\_config.py 🡪 Configuration for each node
    - client.py 🡪 Tests for each of the 2 preconfigured clients along with connections with xml clients to the nodes
      * clients\_config.py 🡪 Configuration number and file for test data
* Additional Files Included
  + Raw Test File Data
    - raw\_test\_data\_0.txt
    - raw\_test\_data\_1.txt
    - test\_data\_0.txt
    - test\_data\_1.txt
  + Example Output File PDFs
    - Eventual\_Output.pdf
    - Eventual\_Output\_Verbose.pdf
    - Sequential\_Output.pdf
    - Sequential\_Output\_Verbose.pdf
    - Linearizable\_Output.pdf
* What lower-level key-value store you have used, and why?
  + I decided to use Python Dictionaries within classes to act as each node’s Key-Value store. This allowed the simplicity of putting and getting values from the built-in Python data structure while being able to design the rest of the project. I decided this because it was in memory so hopefully performance was a bit better, and I did not have to deal with file reading and writing consistency when testing. It also made testing easier because I did not have to clear out files every time I wanted to test because when the program stops, everything clears out automatically. Connecting between nodes and connecting to nodes is done over the Python XML RPC library.
* How is each consistency model is implemented?
  + Overall Implementation of Nodes
    - Each node is considered a class and registered instance of XML RPC. Each class then has its respective Key-Value store methods like put, get, and remove. There are also update and update\_remove methods that are only used by inter-node communication for consistency updates. Any worker threads, which does the consistency updating, are implemented through static methods which run inside of the node process. Each update method has a small random sleep function that will add some level of “real” world update times for network and drive latency.
  + Eventual Consistency
    - Every time there is a put or remove, there is a new thread created. This background worker thread will handle sending out the updates to the other nodes. Again, there is a small delay to emulate real-world timing. Since XML RPC only allows single connections and is not generally mutli-threaded each update is retried until success. So, this should achieve an eventual update on the other nodes.
  + Sequential Consistency
    - When nodes are running the have an extra update thread always running in the background. Every time a put or remove is given to a node they are added to a queue. When there are updates in the update queue then the background worker thread will go through in a FIFO to achieve sequential consistency for updates. The same is done when remove is called and it is added to the queue.
  + Linearizable
    - The implementation is somewhat identical to Sequential Consistency because we want to retain order of operations like put and remove. The main difference between Sequential and Linearizable in my implementation is the fact after a put or remove there is a confirmation get request to the other node before moving on. This ensures that the value in the node is completely up to date with the node that was given the update.
* What were the main challenges?
  + Sometimes using XML RPC in Python there is the issue of it not being multi-threaded and accepting multiple connections at once. So, there is the issue of a client doing an update, and another node being updated. Adding some while looping and try-except clauses allowed me to overcome this issue.
  + Understanding and implementing the consistencies. I did a few iterations of Sequential and Linearizable before deciding on a final to turn it in. Eventual was somewhat easy to complete.
  + The timings of everything posed a challenge since everything was running on separate threads and processes. When testing I had to make sure that processes were finished before killing them to see my results. So, there are some sleep functions to ensure that everything is complete.
  + Getting some test key-value data. I ended up finding generator for words that could be separated by a colon. Then I had to create a Python script to clean up the data to be used for testing.
  + Creating tests
  + Challenges in multiprocessing and threading running multiple things going on at the same time.
* What is your client API?
  + My client API is somewhat simple actually and would mimic what would happen in a local Python Dictionary or Java HashMap with simply put, get, and remove functions. Since I used XML RPC creating a connection is very simple for a node and calling the functions.
  + Creating the connection to the node to do all the requests   
    
  + Key Value Function calls, this can also be done with any other node connection.  
    
    - Put returns nothing
    - Get returns the value from the entered key or “NULL” if not there
    - Remove returns the value from the entered key or “NULL” if not there
* Some performance evaluation
  + All key and values are stored in Python Dictionaries in each node. So, they are limited to likely O(n) and O(1) because Python Dictionaries are based on HashMap.
  + The speed of the test data is read into the clients from text files by each line. Then put into a faster dictionary.
  + There is likely some network delay because of the limitations of XML RPC and http requests.
  + All node updates are called through XML RPC and then those functions utilize the Python Dictionary.
  + The overall performance is quite fast so to allow more realistic testing there are random time.sleep() function calls usually between 0.2 and 1 seconds before every node update. There is also added delays before each put function call from the clients to allow for better testing.