Project – BerkeleyDDB

ECSE 420 - Parallel Computing

Fall 2016

Final Report

Group 16

Sangmoon Hwang (#260569675)
Shivan Sahib (#260512593)

Electrical, Computer and Software Engineering Department
Trottier Bldg., 3630 University St, Montreal

Released date: October 13th 2016
Due date: December 12th 2016

Table of Contents

Introduction

There are growing demands in the field of software engineering for development tools that are easy and intuitive to use. Many present database systems are designed with the intention of providing a simple interface to the developer, such as MongoDB. However, distributed computing as a field does not yet offer many simple-to-use tools. Our project springs from this philosophy - that there's a market for simple, intuitive distributed database systems.

The purpose of our project is to use BerkeleyDB - an embedded key-value store (for among other languages, Java) - to create a distributed database system supporting tables and capable of performing load-balancing, server pushed look-ups, replication, and guaranteed transactions.

The repository for our code is here: https://github.com/ShivanKaul/BerkeleyDDB

Methodology

Running instructions

To compile and run the software:

1. Run *compile.sh* script. (Do it on each host if using multiple hosts)
2. Go to the directory */tmp/berkeleydb/*
3. On the master host, run *runServer.sh* script.
4. On each worker host, run *runWorker.sh <Database_Name> <IP_of_Master>*.
    - It is normal to see an exception about the already used port for rmi registry if more than one workers or the master is running on the same machine. However if the exception does not allow the program to start, please run *killrmis.sh* script and try the step 4 again. If this doesn't work, please manually run `ps aux | grep rmiregistry` and `kill -9` the process.
5. On each client, run *runClient.sh <IP_of_Master>*.
6. Have fun!
7. To turn off the system, please use *Ctrl + c* on each host running one of the *runXX.sh* scripts. For killing rmi registries, please run *killrmis.sh* script.

Operation and Testing

The server can be started up by navigating to /tmp/berkeleyddb and running the runServer script.

```
/tmp/berkeleydb sh runServer.sh
-e Process running rmi registry:
75719 ttys003    0:00.00 rmiregistry
75721 ttys003    0:00.00 grep rmiregistry
Server ready
```

Figure 1: Master start

Now, we can connect our client to it. The client can live anywhere, but if it is not on the same machine as the Master, then it must supply the IP address of the Master. We showcased how the client can connect to the Master from a different machine in the class demo - here, we assume that the client is on the same machine.

```
/tmp/berkeleydb sh runClient.sh
--------------------------
 What would you like to do?
 1. `get <table> <key>`
 2. `set <table> <key> <value>`
 3. exit
--------------------------
```

Figure 2: Client start

We can now provision multiple workers for multiple databases. We have shown the operation of one Employee table/worker and two Customer tables/workers. Note that one worker can only serve one table. But of course, one table can be supported by multiple workers (for load balancing). Similar to client, we showcased how a worker can connect to the Master from a different machine in the class demo - here, we assume that the worker is on the same machine, and hence don't supply an IP address.

```
/tmp/berkeleydb sh runWorker.sh employee
java.rmi.server.ExportException: Port already in use: 1099; nested exception is:
        java.net.BindException: Address already in use
        at sun.rmi.transport.tcp.TCPTransport.listen(TCPTransport.java:341)
        at sun.rmi.transport.tcp.TCPTransport.exportObject(TCPTransport.java:249)
        at sun.rmi.transport.tcp.TCPEndpoint.exportObject(TCPEndpoint.java:411)
        at sun.rmi.transport.LiveRef.exportObject(LiveRef.java:147)
        at sun.rmi.server.UnicastServerRef.exportObject(UnicastServerRef.java:208)
        at sun.rmi.registry.RegistryImpl.setup(RegistryImpl.java:152)
        at sun.rmi.registry.RegistryImpl.access$100(RegistryImpl.java:73)
        at sun.rmi.registry.RegistryImpl$2.run(RegistryImpl.java:128)
        at sun.rmi.registry.RegistryImpl$2.run(RegistryImpl.java:125)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.security.AccessController.doPrivileged(AccessController.java:713)
        at sun.rmi.registry.RegistryImpl.<init>(RegistryImpl.java:125)
        at sun.rmi.registry.RegistryImpl$5.run(RegistryImpl.java:383)
        at sun.rmi.registry.RegistryImpl$5.run(RegistryImpl.java:381)
        at java.security.AccessController.doPrivileged(Native Method)
        at sun.rmi.registry.RegistryImpl.main(RegistryImpl.java:380)
Caused by: java.net.BindException: Address already in use
        at java.net.PlainSocketImpl.socketBind(Native Method)
        at java.net.AbstractPlainSocketImpl.bind(AbstractPlainSocketImpl.java:382)
        at java.net.ServerSocket.bind(ServerSocket.java:375)
        at java.net.ServerSocket.<init>(ServerSocket.java:237)
        at java.net.ServerSocket.<init>(ServerSocket.java:128)
        at sun.rmi.transport.proxy.RMIDirectSocketFactory.createServerSocket(RMIDirectSocketFactory.java:45)
        at sun.rmi.transport.proxy.RMIMasterSocketFactory.createServerSocket(RMIMasterSocketFactory.java:345)
        at sun.rmi.transport.tcp.TCPEndpoint.newServerSocket(TCPEndpoint.java:666)
        at sun.rmi.transport.tcp.TCPTransport.listen(TCPTransport.java:330)
        ... 15 more
Worker ready
```

Figure 3: Worker Employee start

```
/tmp/berkeleydb sh runWorker.sh customer
java.rmi.server.ExportException: Port already in use: 1099; nested exception is:
        java.net.BindException: Address already in use
        at sun.rmi.transport.tcp.TCPTransport.listen(TCPTransport.java:341)
        at sun.rmi.transport.tcp.TCPTransport.exportObject(TCPTransport.java:249)
        at sun.rmi.transport.tcp.TCPEndpoint.exportObject(TCPEndpoint.java:411)
        at sun.rmi.transport.LiveRef.exportObject(LiveRef.java:147)
        at sun.rmi.server.UnicastServerRef.exportObject(UnicastServerRef.java:208)
        at sun.rmi.registry.RegistryImpl.setup(RegistryImpl.java:152)
        at sun.rmi.registry.RegistryImpl.access$100(RegistryImpl.java:73)
        at sun.rmi.registry.RegistryImpl$2.run(RegistryImpl.java:128)
        at sun.rmi.registry.RegistryImpl$2.run(RegistryImpl.java:125)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.security.AccessController.doPrivileged(AccessController.java:713)
        at sun.rmi.registry.RegistryImpl.<init>(RegistryImpl.java:125)
        at sun.rmi.registry.RegistryImpl$5.run(RegistryImpl.java:383)
        at sun.rmi.registry.RegistryImpl$5.run(RegistryImpl.java:381)
        at java.security.AccessController.doPrivileged(Native Method)
        at sun.rmi.registry.RegistryImpl.main(RegistryImpl.java:380)
Caused by: java.net.BindException: Address already in use
        at java.net.PlainSocketImpl.socketBind(Native Method)
        at java.net.AbstractPlainSocketImpl.bind(AbstractPlainSocketImpl.java:382)
        at java.net.ServerSocket.bind(ServerSocket.java:375)
        at java.net.ServerSocket.<init>(ServerSocket.java:237)
        at java.net.ServerSocket.<init>(ServerSocket.java:128)
        at sun.rmi.transport.proxy.RMIDirectSocketFactory.createServerSocket(RMIDirectSocketFactory.java:45)
        at sun.rmi.transport.proxy.RMIMasterSocketFactory.createServerSocket(RMIMasterSocketFactory.java:345)
        at sun.rmi.transport.tcp.TCPEndpoint.newServerSocket(TCPEndpoint.java:666)
        at sun.rmi.transport.tcp.TCPTransport.listen(TCPTransport.java:330)
        ... 15 more
Worker ready
```

Figure 4: Worker Customer start

We can start another Customer worker (not shown because it would look exactly the same). We can now look at Master to see its responses.

```
/tmp/berkeleydb sh runServer.sh
-e Process running rmi registry:
75719 ttys003    0:00.00 rmiregistry
75721 ttys003    0:00.00 grep rmiregistry
Server ready
Received request from a worker to add table employee for ip address 192.168.2.14
Table employee does not exist, adding...
Hashing ring for table employee looks like: [{"address":"192.168.2.14","id":5384}]
Received request from a worker to add table customer for ip address 192.168.2.14
Table customer does not exist, adding...
Hashing ring for table customer looks like: [{"address":"192.168.2.14","id":4134}]
Received request from a worker to add table customer for ip address 192.168.2.14
Table exists, appending...
Hashing ring for table customer looks like: [{"address":"192.168.2.14","id":4134},{"address":"192.168.2.14","id":9291}]
```

Figure 5: Master receives join requests from workers

On startup, a worker sends a join request to the Master along with its IP address and a randomly generated UUID. When we add the Employee worker, at that time the Master does not have any reference to any other workers supporting the Employee table. It initializes a hashing ring for that table and prints it. A hashing ring consists of tuples of IP address and UUID (here they are modulo'ed over 10,000 - this depends on the range of the hashing function used). We see that when the second Customer worker is added, the hashing ring now contains two tuples. Note that even though the IP address for both Customer is same, the data is stored in the correct db because on the filesystem, the data is stored in /tmp/worker_<tablename>_<uuid>.

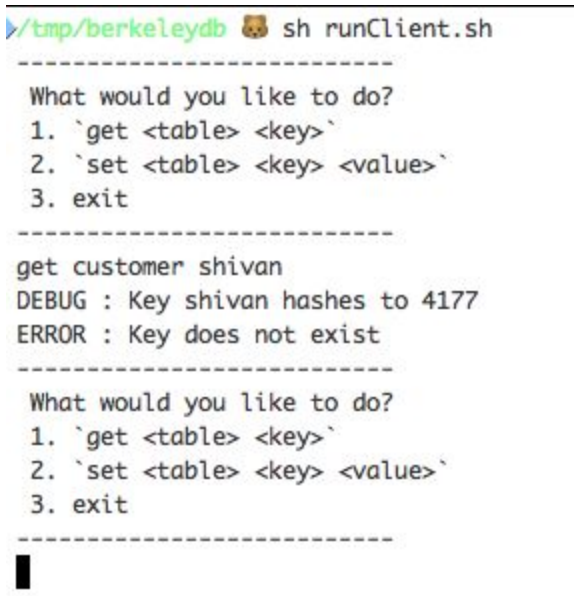

Figure 6: Client faulty get

Here we see that the client tried to get key Shivan in table Customer without setting it first, and the worker responds with "key does not exist". client displays this error. What happens is that the client fetches the WorkerLookupComputation from Master and runs the lookUpWorker() method. Key shivan deterministically hashes to 4177, which places it in the bucket for the first Customer worker.

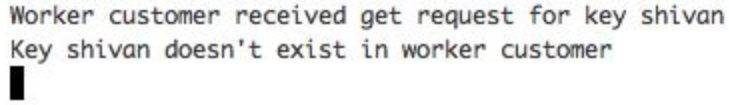

Figure 7: Worker Customer's faulty get

Now we can do a set and a get.

```
--------------------------
 What would you like to do?
 1. `get <table> <key>`
 2. `set <table> <key> <value>`
 3. exit
--------------------------
set customer shivan kaul
DEBUG : Key shivan hashes to 4177
SUCCESS : set!
--------------------------
 What would you like to do?
 1. `get <table> <key>`
 2. `set <table> <key> <value>`
 3. exit
--------------------------
get customer shivan
DEBUG : Composite key customer_shivan exists in cache, fetching directly from Worker
SUCCESS : Response: kaul
--------------------------
 What would you like to do?
 1. `get <table> <key>`
 2. `set <table> <key> <value>`
 3. exit
--------------------------
```

Figure 8: Client's set and get showing operation of cache

Here we see that we do a set for table Customer for key shivan with value kaul. The key shivan again hashes to 4177, and is successfully set on the same worker as before. Note that we did not use the cache because the previous operation did not complete successfully. Now we do a get for shivan on Customer table and we see that the composite key exists in the cache from the previous set operation, so we already know which worker to talk to for this key and this table. We get the correct response as we can also see on the worker.

```
Worker customer received set request for key shivan with value kaul
Worker customer received get request for key shivan
```

Figure 9: Worker Customer's successful set and get

We can set shivan to be something else on a different table, and we'll see that we get the correct response.

```
--------------------------
 What would you like to do?
 1. `get <table> <key>`
 2. `set <table> <key> <value>`
 3. exit
--------------------------
set employee shivan sahib
DEBUG : Key shivan hashes to 4177
SUCCESS : set!
--------------------------
 What would you like to do?
 1. `get <table> <key>`
 2. `set <table> <key> <value>`
 3. exit
--------------------------
get employee shivan
DEBUG : Composite key employee_shivan exists in cache, fetching directly from Worker
SUCCESS : Response: sahib
--------------------------
 What would you like to do?
 1. `get <table> <key>`
 2. `set <table> <key> <value>`
 3. exit
--------------------------
```

Figure 10: Client's successful set and get for different table

Note the different caching (composite) key used.

Discussion

BerkeleyDB

BerkeleyDB is a simple embedded key-value database that has bindings for many different languages. We were using the Java edition. It's widely used as a building block in industry applications. Most pertinently, it's used also in Project Voldemort - a fact we realized only recently, which further retrospectively validates our decision to use it.

We use BerkeleyDB as the base store for our Worker and Master nodes. The bulk of our project involved writing wrappers around BerkeleyDB instances and implementing access and data distribution strategies over it.

RMI

We made heavy use of Remote Method Invocation. Remote Method Invocation is the Object Oriented counterpart of Remote Procedure Call (RPC), and is in a similar vein as CORBA. Java RMI is a Java API performs RMI, by allowing an object running in one JVM to call a method on an object running in a physically separate JVM.

RMI requires the the definition of an interface for each remote object. This interface is then used by whichever object wants to call a method. In our codebase, we call the Interface the thing itself (for example Thing), and suffix an 'Impl' to the implementing class (for example ThingImpl).

RMI also requires a registry. A registry manages the TCP connections between remote objects. An object registers itself with a registry, and then any object that wants access to that object refers to it by name and gets a handle to it via the registry. We could have had a single registry for the entire application, however, because of security restrictions we decided to go with having registries for every single component. When access is needed, we get the right registry by using the IP address.

```
private static Worker getWorkerFromAddress(String workerAddress) throws RemoteException, NotBoundException {
  registry = LocateRegistry.getRegistry(workerAddress);
  return (Worker) registry.lookup("Worker");
}
```

Figure 11: getWorkerFromAddress method

Architecture

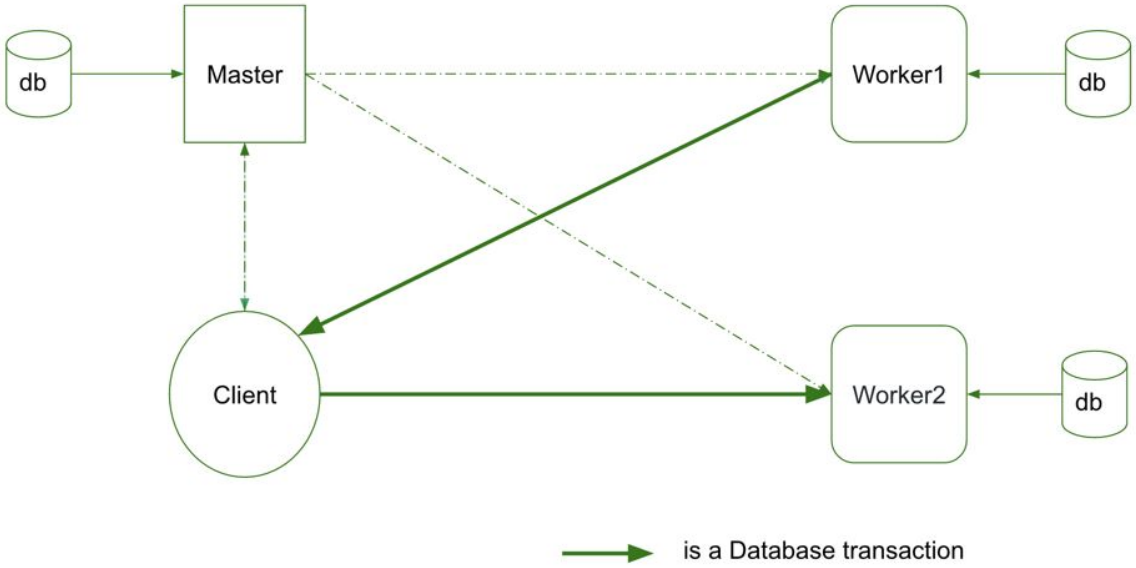

Figure 12: Architecture of system

The data in the system is distributed across the Worker nodes. We created a sample client application to show how the system can be used. Essentially, the client queries the Master with a key and a table it wants to put the key in. The Master responds with information on how to look up the Worker, then the client can use Master's response to execute get and put directly on the relevant Worker. Pushing the worker lookup information to the client instead of fetching and putting the data itself prevents the Master from becoming a bottleneck for the system. While this adds a network hop for the client, the tradeoff is worth it. To mitigate this delay, the client caches the Worker information for a composite key in-memory, using a hash map. Now, if client receives an error from the Worker (for example if it went down, or if it does not have the right data), then at that point the client considers its cache for that composite key invalidated and communicates with the master.

Master and MasterImpl

The Master node serves as a receptionist to clients looking for a right worker to communicate for data transactions. In other words, when a client would like to send a request with a data, there are two scenarios that the client would send a request to the Master:

1. The client has not cached which Worker to connect to given the data
2. The client has failed to connect to the Worker which had already been cached

On startup, a worker talks to the Master (given the IP address, it looks up the registry and uses the name "Master" to fetch a remote handle) and registers itself. The Master stores workers by adding the table they support and a UUID to a hashing ring. A hashing ring is particular for a table. It would look something like this:

```
Received request from a worker to add table employee for ip address 192.168.2.14
Table employee does not exist, adding...
Hashing ring for table employee looks like: [{"address":"192.168.2.14","id":5384}]
Received request from a worker to add table customer for ip address 192.168.2.14
Table customer does not exist, adding...
Hashing ring for table customer looks like: [{"address":"192.168.2.14","id":4134}]
Received request from a worker to add table customer for ip address 192.168.2.14
Table exists, appending...
Hashing ring for table customer looks like: [{"address":"192.168.2.14","id":4134},{"address":"192.168.2.14","id":9291}]
```

Figure 13: Master's hashing rings for different tables

This hashing ring is used for distributing the load across the workers and for performing lookups. More details in the *Load Balancing* section.

When a client wants to know which worker it should talk to for a particular table and a particular key, it provides the table name and key to the Master, and the Master simply sends across a computation that holds the hashing ring for the table requested and the key requested. This logic is encapsulated in WorkerLookupComputation. Now, it is the client's responsibility to run

the computation and figure out which worker to talk to. This scheme is followed so as to minimize the amount of work that the Master has to do, in order to prevent it from becoming a bottleneck - an important requirement in distributed systems. The client takes the performance hit for computing which worker to talk to. Note that we could have multiple Masters behind a load balancer like Nginx or similar, in which case they would need to maintain data consistency between themselves.

Worker and WorkerImpl

A worker registers itself with the Master on startup. It generates a UUID for itself and sends across this plus its IP address to the Master, and the Master stores this as described in previous section. A worker is completely independent of anything else in the system, which was an important requirement. It simply provides get and set remote methods to the client for its data.

In order to ensure the persistence, the worker makes sure to flush any cached information for this database to disk by calling *sync()* method after each *set* request from a client. [1] [Snippet 1]

```
public Result set(Stringkey, String value) throws DatabaseException,
UnsupportedEncodingException {
                                .
                                .
                                .
        workerDb.sync();
        return new Result(opStatusb, true, null);
}
```

Figure 14: Worker sync snippet

Client

We wrote a sample client application to demonstrate the use of the system. The client program takes in the IP address of the Master node (defaults to localhost). After compilation using *compile.sh,* it can be called thusly (note that the UI has changed superficially now):

```
/tmp/berkeleydb ./runClient.sh 142.157.43.210
What would you like to do?
 1. `get <table> <key>`
 2. `set <table> <key> <value>`
 3. exit
```

The UI is simple and demonstrates what can be done with the system.

The client uses the IP address to fetch the registry of the Master, and gets a stub. A stub in RMI parlance refers to an object that holds the interface. This stub can be used to call methods on Master. Once we have the stub, we enter a prompt loop which handles the user input.

```
// prompt loop
boolean loop = true;
while (loop) {
  try {
    PromptResult promptResult = prompt(masterStub);
    if (!promptResult.continueLoop()) {
      loop = false;
      continue;
    }

    ...
```

Figure 15: prompt loop

There's a prompt method that takes in the Master stub and parses the input to understand what the user wants to do, and then executes the handleSet() and handleGet() methods.

We use the concept of a composite key. A composite key is a key made up of the table name and the key specified by the user. A composite key is used for the caching.

Caching

```
private static Result handleGet(String table, String key, Master master_stub) throws Exception {
  // Get a handle to the worker from the master
  // Talk to the worker directly
  String compositeKey = table + "_" + key;
  if (!cache.containsKey(compositeKey)) {
    Result response = master_stub.getWorkerHost(compositeKey);
    if (response.noErrors()) {
      String workerAddress = response.returnedValue;
      Worker worker = getWorkerFromAddress(workerAddress);
      // Put in cache
      cache.put(compositeKey, workerAddress);
      return worker.get(key);
    } else return response;
  } else { // cache does contain key
    String workerAddress = cache.get(compositeKey);
    Worker worker;
    try {
      worker = getWorkerFromAddress(workerAddress);
    } catch (Exception e) {
      // failed, just get key from master
      Result response = master_stub.getWorkerHost(compositeKey);
      if (response.noErrors()) {
        workerAddress = response.returnedValue;
        worker = getWorkerFromAddress(workerAddress);
        // Put in cache
        cache.put(compositeKey, workerAddress);
        return worker.get(key);
      } else return response;
    }
    return worker.get(key);
  }
}
```

Figure 16: Caching logic

Here we can see a snippet of how we implemented client-side caching of Master responses. If the cache does not contain the composite key, then we talk to Master and put the response in the cache. If the cache does contain the composite key, then we use that to get the worker. If however this fails, we simply do what we did previously and get the Worker address from Master.

Load Distribution

Based on Shivan's experience with consistent hashing at an internship, we went with this scheme for distributing the load for a single table across multiple workers, and for performing worker lookups [2].

As mentioned previously, the Master maintains a hashing ring for each table. When the client requests the hashing ring for a certain table and key, the Master initializes WorkerLookupComputation (a computation wrapper we wrote) with the hashing ring and the key and sends it across to the client. Now, the client calls lookUpWorker() on the WorkerLookupComputation instance, which returns a ComputationResult. What happens in the lookUpWorker method is that given a ring of workers (in the form of hash values) and a hashed key, we figure out which bucket it should belong to. So if the hashing ring for table Customer is [{ipAddress : "xx", id: 4134}, {ipAddress : "yx", id: 4200} {ipAddress : "xy", id: 9291}, {ipAddress : "yy", id: 1004}], then a key that hashes to 4250 would go to worker with IP address yx. This is how the client knows which worker to talk to. And as a good hash function is deterministic and uniformly-distributed, we are guaranteed well-balanced load. If a worker goes down, for example if our hashing ring now becomes [{ipAddress : "yx", id: 4134} {ipAddress : "xy", id: 9291}, {ipAddress : "yy", id: 1004}], now key 4250 will slot into server with IP address yx. Only K/n keys will need to be redistributed, where K is total number of keys and n is number of workers, instead of all the keys. However, our system has not been tested to work with failure. For this we would need to implement replication, which we will work on in the next phase.

Acknowledgements

We'd like to thank Joseph D'silva, a PhD student in the Distributed Information Systems Lab. He gave us immense guidance on building distributed systems and help with getting started. Also we'd like to thank Prof Bettina Kemme who heads the Distributed Information Systems Lab for allowing us to work on this project.

Future Work

We would like to implement replication, for the workers as well as Master. Consistency between replicas is a hard problem and we would have to think carefully about the tradeoffs for having an instantaneous set-triggered replication. Another thing worth having, but fairly difficult to do, would be to have transactional guarantees for the entire system. Right now every individual instance running on a node has transactional guarantee, but implementing that for the entire system as a whole, from the point of view of the clients would be a worthy goal.

Conclusion

Overall, we made good progress this semester on BerkeleyDDB. We were able to write an MVP, and were able to get all the essential components in place. Apart from this, we also implemented server pushed lookups, client-side caching, and consistent hashing for load balancing. We'd like to continue working on it and add additional features as talked about in the Future Work section. The project is open source and is hosted on GitHub.

One interesting thing to note is that when we began, we were only dimly aware of Project Voldemort. We used it more as an example of the fact that such a system can be useful. However, after digging deeper into their implementation, it turns out that their implementation looks similar to ours, including operational semantics. They are using BerkeleyDB as well for their underlying data store, which is extremely interesting, as we had no idea that they were doing so when we started our project. The major implementation difference between BerkeleyDDB (our project) and Project Voldemort is that they don't seem to be using RMI for distributed method calling. Instead they offer support for pluggable serialization techniques such as Thrift and Protobuf. The major application difference is Project Voldemort is essentially a giant, distributed hash map. Our project BerkeleyDDB, however, also offers table support. We let the user of the system decide which tables they want to support and which table a particular worker should hold. While not nearly relational, this table support allows for more interesting application logic to be built on top of the system.

References

[1]"Oracle - Berkeley DB Java Edition API", Docs.oracle.com, 2016. [Online]. Available: http://docs.oracle.com/cd/E17277_02/html/java/index.html. [Accessed: 05- Dec- 2016].

[2] Karger, David et al. "Web Caching With Consistent Hashing". Computer Networks 31.11-16 (1999): 1203-1213. Web.