EECE411.2015.A3

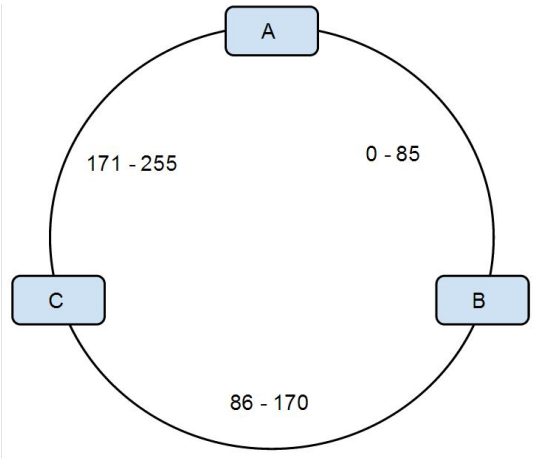
Project: Build a distributed key-value store based on consistent hashing - KVStore.

Deadlines for various phases: see course website.

Late policy: 33% point deduction for each late day.

Background: Consistent hashing is based on mapping each key and participating node to a point on a circular space (or equivalently, mapping each object to a real angle). The system maps each available machine to many pseudo-randomly distributed points on the edge of the same circle.

To find where an object should be placed, the system finds the location the key on the circular space; then walks around the circle until falling into the first machine it encounters (or equivalently, the first available machine with a higher angle). The result is that each machine contains all the resources located between its point and the previous machine point.



For instance, in the figure above there are three machines: A, B, C, and the whole hash space is from 0 to 255. Assume A is responsible for items whose hash value is from 171 to 255, B is responsible for [0, 85], and C is responsible for [86, 170]. When one put one key-value pair in the system, if hash(key) = 40, then the pair should be stored on machine B.

When replication is used, one item will be stored on multiple machines. When one machine fails, the replicas on it should be propagated to other machines.

You will use the PlanetLab machines and implement KVStore based on consistent hashing.
Your key-value store should have the following operations:

1. *put(key, value)*: Puts some value into the store. The value can be later retrieved using the key. If there is already a value corresponding to the key then the value is overwritten.
2. *get(key):* Returns the value that is associated with the key. If there is no such key in your store, the store should return error - not found.
3. *remove(key):* Removes the value that is associated with the key. If there is no such key the store should return error. Otherwise the value should be removed (get calls issued later for the same key should return error - not found).

Wire protocol: This is implemented on top of the request/reply protocol you developed for A1 (i.e., using UDP, the process for unique IDs, timeouts, etc). The following describes the format of the application level messages exchanged using that protocol.

Requests:

command (1 byte)	key (32 bytes)	value-length (integer, 2 bytes, little endian) (only for put operation)	value (up to 15,000 bytes)

1. Command is 1 byte long. It can be:
   - 0x01. This is a *put* operation.
   - 0x02. This is a *get* operation.
   - 0x03. This is a *remove* operation.
   - *[Note: We may add some management operations, the message format will stay the same. For example:]*
     0x04. Command to shutdown the node (think of this as an announced failure). The operation is asynchronous: returning success means that the node acknowledges receiving the command at it will shutdown as soon as possible.
   - anything > 0x20. Your own commands if you want.
2. The key is 32 bytes long. This is the identification of the value.
3. The length of the value: integer represented on two bytes. Maximum value 15,000. Only used for put operation.
4. Value. Byte array. Only used for put operation.

Replies: Have the following following format:

response code (1 byte)	value-length and value (same as in requests) (only for get operation)

1. The code is 1 byte long. It can be:
   - 0x00. This means the operation is successful.
   - 0x01. Non-existent key requested in a get or delete operation
   - 0x02. Out of space (returned when there is no space left for a put).
   - 0x03. System overload.
   - 0x04. Internal KVStore failure
   - 0x05. Unrecognized command.
   - *[possibly more standard codes will get defined here]*
   - anything > 0x20. Your own error codes. *[Define them in your Readme]*

Planned testing

At each of the intermediate phases (described below) we will test for aspects related to correctness, service robustness, data durability and availability, scale, and performance (response time and throughput).

- *Scale:* By the end of this project you should deploy your system on at least 150 nodes.
- *Correctness:* We will issue *put* commands on multiple machines of your key-value store, and verify that the key-value pairs are correctly stored by issuing *get* commands on the machines that are different from the ones used for put commands. We will also test *remove* commands followed by *get* commands to see if the *remove* functionality is correct.
- *Service robustness:* Your service should be able to continue to operate in spite of node failures.
- *Data durability.* Your system should not lose data to node failures (announced or unannounced). Most likely we will extend the protocol to mimic and 'announced failure' - i.e., we'll have a message to request a node to shut down. Your system should resist without losing data on for at least 24h.
- *Performance*: A testing framework will stress-test your system for performance and evaluate both its responsiveness (time to reply to requests) and throughput.
- *Data availability / Timeouts*: the testing harness will have a timeout of 5 seconds - if a request is not served in 5s then it is considered failed.

More notes:

- *Persistence:* You do not need to persist data on disk.

- *Space usage*: Consider that space usage on each node is limited by at most 100,000 key/value pairs or at most 64MB of memory space. (this includes all replicas you may create). This is a requirement introduced to play nicely in the shared PlanetLab environment - obviously each node in a real system will store many more keys. Please see also the note on resource usage below.
- *Key randomness*: You should NOT assume that the keys have been randomly generated: i.e., the keys have NOT been passed through a one-way hash function by the time they reach your service.

Assignment 3: Single-node mock system. Implement a mock system that works on one node and follows the protocol defined above, deploy it on PlanetLab, and get it tested for functionality and performance.

Deliverables: Send to the usual email address:

- 3 (three) locations where you have launched independently your service (node:port). We will run correctness and performance tests against these nodes. You'll do better if you choose nodes at or close to UBC.
- An archive with your source code (we may not try to execute but we will inspect it). This should include both a server and a client/testing code
- Your README file should include:
    - brief description of design choices,
    - a complete description of the tests you have executed,
    - performance characterization summary - figures are encouraged

Assignment 4: Correct routing. Your system should now be able to operate at scale - 50+ nodes, and (although it may lose data) it should be able to correctly route messages and retrieve data to survive after individual node failures and node joins. While the client protocol is fixed as described above, you can use any protocol you want internally.

You can assume that, at startup, each node knows the identities (host:port) of all (or a group) of other nodes that may participate in the system. Obviously - some nodes may fail to start and join the system, some nodes may fail along the way, some nodes may (re)join late.

To enable testing for node failures, we have added a new operation that asks a node to shutdown (please see above, opcode 0x04). Your service can NOT take any measures before actual shutdown -- it should immediately simulate a crash, independent of the current state of the program.

Deliverables: Send to the usual email address:

- The sets of locations where you have launched your service (as files containing a list of node:port, one per line). We will run correctness, performance, and robustness tests against these nodes.
    - SIX sets with THREE nodes communicating with only each other.
    - ONE set with FIFTY nodes communicating with only each other.
        - All nodes should be unique and should only be in one set.
- An archive with your source code (we may not try to execute but we will inspect it).
- A README file including a description of your solution, the major design decisions you have made, and the performance benchmarks you have run and their results.
    - You should certainly describe: the overall design, what happens during normal operation (how requests are routed, how membership updates are propagated), what happens after a node failure (and how this gets detected), what happens after a node join.
    - You should also describe in detail your testing strategy and the test cases. You description should clearly specify the property you are testing (correctness, reliability, performance, etc); the success metric you are using; an the test itself.
    - Finally you should describe your performance benchmarking and their results

Assignment 5: Data replication. To provide data durability, improve availability and overall performance, the system can create multiple replicas of a (key,value) pair and place them on different nodes.

Replication factor and replica placement: Your system should use a replication factor of 3 (i.e., maintain 3 copies of each value pair). Your design has full flexibility in terms of choosing a replica placement strategy (a possible solution is to keep store replicas on 'successor' nodes) and on choosing which replica replies to a request.

Consistency model: Eventual consistency (a.k.a. best effort). Your design will favour availability and fault-tolerance over consistency - thus you have a consistency model that does not offer any formal guarantee: eventual consistency. The client applications expect, however, that a majority of the time, the system will appear as sequentially consistent at the level of a single (key,value).

Target scale: 150 nodes

Deliverables: Send to the usual email address:

- The sets of locations where you have launched your service (**as files containing a list of node:port, one per line**). We will run correctness, performance, and robustness tests against these nodes.
  - THREE sets with THREE nodes communicating with only each other.
  - ONE set with ONE HUNDRED FIFTY nodes communicating with only each other.
- An archive with your source code (we may not try to execute but we will inspect it).
- A README file including a description of your solution, the major design decisions you have made, and the performance benchmarks you have run and their results.
  - You should certainly describe: the overall design, what happens during normal operation (how requests are routed, how membership updates are propagated), what happens after a node failure, what happens after a node join.
  - You should also describe in detail your testing strategy and the test cases. You description should clearly specify the property you are testing (correctness, reliability, performance, etc); the success metric you are using; and the test itself.
  - Finally you should describe your performance benchmarking and their results.

Assignment 6: Performance optimizations and scaling up. Anything you can think of to improve the characteristics of your system: note that we test for performance (throughput, response time for various requests), availability, correctness and overall reliability.

One possible optimization, for example, is to flexibly route 'get' requests to the replica with the best predicted response time rather than using a fixed routing. Or you could try better load-balancing, or better dealing with temporary node failures.

There are many places to look if you do not have optimization ideas. In the list of papers here, for example, I would look at: *Fixing the Embarrassing Slowness of OpenDHT on PlanetLab* and, even better at: *Handling Churn in a DHT*.

I encourage you to discuss your ideas with the TAs or with me.

Target scale: 150 nodes (or more).

Deliverables: Same as for the previous phase plus a description of your optimization(s).

Important note on resource usage:

To reduce the stress we generate on the PlanetLab platform, and, importantly, to make results comparable across projects your implementation and deployment should follow the following

constraints (if you pass the resource limits by much PlanetLab will kick you out anyway, and possibly disable your slice):

- *Memory usage*: The nodes should limit their memory use to at most 64MB. This limit can be enforced by setting the JVM maximum heap size at startup by using:

  java –Xmx64m <your-command-line>

- *Number of concurrent open TCP sockets*: the nodes participating in the system should never have more than 100 open TCP connections at any point in time (if you are using TCP anywhere). Do not open connections to all nodes in the system - this will certainly not scale.
- *Network port usage:* Be conservative in the number of ports you use - the project can be comfortably implemented with 1 - 5 ports per node.
- *Deployment:* all components of your system should run on PlanetLab

Submission details:
Send your submissions to 202020m@gmail.com