CS 6343: CLOUD COMPUTING
Final Project Review

Group A: Cloud file/DB systems

- System installations on VMs
  - Make sure that your file/DB data are external to the VMs so that your VMs can have a reasonable size and can be cloned easily
  - Create some commands for system management from a control node
    - A new data node can be created dynamically by two commands
      - One command to create and activate a new VM with the data server running
      - Second command to add the node through the corresponding file/DB system
      - The data node may be any VM, may be different from the control node
    - A data/master node can be killed by one command
      - Just kill the VM from the control node
    - A data/master node can be deleted through the file system manager, if any
- Benchmark the file/DB system as in depth as possible
  - For file systems
    - Use File Generator to populate the file systems
    - Use Request Generator to test the performance of the file systems
      - Modify Request Generator to send request to each of the file systems
  - For DB systems
    - Use YCSB to populate and test the DB systems
    - Modify YCSB to achieve some measurement goals
- Suggested performance and behavior observations
    - If it is not possible to get the data for some metrics, just report what attempts have been made and the features in the system that make the investigation difficult or impossible
  - General R/W performance with various access patterns
  - Object (file, block, or key) lookup cost (without R/W)
  - Object creation/deletion cost (overlap with write cost)
  - Performance for add/delete nodes
    - Routing table update cost
    - File transfer cost
    - Impact on performance of normal R/W operations during this period
    - …
  - Performance for load balancing
    - Create load imbalanced situations and observe load balancing performance (performance measurement would be similar to the case of add/delete nodes)
  - Inconsistency for individual operations, such as add/delete/write
    - Report the behaviors and quantitative measures
  - Inconsistency during add/delete nodes and load balancing
    - Files got transferred but routing table has not been updated fully
  - Failure handling (kill VMs to simulate node failures)
    - Observe specifically how failures are handled
    - Impact to normal operations after failures (before and after they are detected, before failure processing is done completely)
    - Impact to add/delete node operations and load balancing
      - E.g., tries to move load to or from the failed node before failure is detected
  - …

- DHT implementation
  - The basic implementation
    - DHT data structure on a single node
      - For Ceph: The basic OSD map data structure
      - For Swift and Cassandra: The basic ring-based DHT structure
      - For Redis: The basic hash node based DHT table
      - For Swift and Ceph: The physical node to virtual node mapping for facilitating load balancing
    - Single node access requests
      - Initialization: Read in the initial configuration for the DHT and load the data structure
      - Lookup: Give an object reference (file name, block id, key), find the nodes that host the data, including who is the primary and all the slaves
      - Load balancing: Issue a load balancing command and the table got updated for the specific load balancing +request
      - Add/delete nodes
  - The updated DHT implementation for distributed environment
    - Implement the regular client who may issue read and print requests
      - The print request simply prints the routing table hosted by the client, it is served locally
      - The read request will trigger a lookup request, and may trigger a table-update request, so you need to implement three requests for the read request
      - The read request first performs a local lookup, then issues the read request to the designated data server
      - The response to the read request includes whether the read request can be satisfied on the local server and the epoch number
        - The data server node checks its local routing table and decides whether the key is indeed on the local server
        - The client should send a table-update request in case its epoch is behind or the read request has failed
        - The client should reissue the read request after the local table is updated
    - Implement a control client
      - May issue add/delete-node and load-balancing requests
    - Distributed DHT for Ceph and Swift
      - Has a central server master (Monitor for Ceph and Proxy for Swift)
      - The control client sends requests to the master, master sends updates to the data servers in case there are table updates
        - May consider immediate update and collective update
      - Regular clients send their table-update requests to the master
    - Distributed DHT for Cassandra and Redis
      - Hosted without central server and table updates are propagated using a gossip protocol
        - The gossip protocol can be initiated periodically, like a heartbeat protocol, or initiated upon updates
      - For node deletion, the node being deleted will send a note to one of its neighbors to simulate the case that this neighbor detected the failure, this neighbor will update its own table
      - The control client sends requests to any node
      - A regular client sends its table-update request to the data server who has just responded to its read request that triggered the table-update request
    - Performance measurement
      - Measure the performance of table update, lookup, and read in various scenarios

        - How long does it take to finalize the update
        - The impact of incorrect information for the read performance (you may consider individual impact and statistical impact)
    - Note: Implementation should be as flexible as possible, no hardcoded configurations, use the system configuration file for initial system setup
    - Note: We may run multiple instances of the clients during demo
- Your code submission should include
    - VMs for each file system
        - Please use qemu; otherwise, it will be very difficult to submit your VMs, and you will incur point deduction
        - Upload your VMs to your Microsoft OneDrive account and open the directories for sharing
    - Source code for DHT implementation
        - Include a makefile
- Your report submission should follow the report items listed in the main document

Group B

- Study the PaaS platforms
    - Your report should include both GAE and Azure
    - Discuss their APIs that may be useful for you
    - Discuss how to set their access control policies
- Develop and deploy the SaaS RoboCode
    - User login on GAE/Azure
    - Redirect the request to the routerVM
        - You can have your own design, but the solution has to be secure
        - The Robocode server side should be able to check the validity of the redirected requests without having to keep a user-password table
        - The protocol should support later mechanism for data accesses
        - The Robocode server side should be able to easily validate the accessed data
        - You should consider confidentiality protection, integrity of the data (data are from the correct source and should not be modified during transmission), potential replay attacks, etc.
    - The routerVM redirect the request to one appVM that runs RoboCode
        - Should balance the load
        - Should show the scenarios of unbalanced loads and demonstrate your load balancing capability
        - Should be able to start appVMs based on a configuration file
    - Your RoboCode app server should
        - Allow a user to edit, save, compile, and play the robot programs
            - The system should be modified to allow users to structure their robots in a directory based concept (directory structure can be pseudo, you can design the data structure to store the relation and display them properly for convenient accesses)
        - Compute a score for all players after playing
        - Access the cloud for user login, user data storage, and user data reading
            - Should be able to handle situations such as accesses that violate the access privileges, and should clearly demonstrate the cases
        - Allow a user to assign access rights for its robots
    - Store the data in the cloud
        - Each user has a list of robots
        - Each robot has its source code, its executable, and its score
        - There should be some overall system data such as global ranking

        - Should have access control for the data stored in the cloud
    - Access control mechanism
        - The system can have a set of roles without a hierarch (because users for RoboCode probably won't have natural hierarchical relations)
        - Subjects should have roles
        - Roles should be assigned permissions to objects
        - System and user objects should be structured in a resource hierarchy to allow easy access control management
        - You should implement access control policy update routines to allow app servers to assign access rights for the newly created robots
        - There should be an interface to support the management of the system, including managing the users, managing the system data, managing the access control policies, etc.
- Your code submission should include
    - routerVM, appVM, cloud code
        - Please use qemu; otherwise, it will be very difficult to submit your VMs, and you will incur point deduction
        - Upload your VMs to your Microsoft OneDrive account and open the directories for sharing
    - Source code for RoboCode system
    - Source code for cloud side management
- Your report submission should follow the report items listed in the main document