CSE530 Distributed Systems
Winter 2023
Assignment 2

Deadline: Mar 29, 2023 5 pm IST

Problem Statement

Implement the following consistency protocols for a distributed key-value store supporting replication:

1. Primary-backup Blocking protocol (Remote-write)
2. Primary-backup Non-Blocking protocol (Remote-write)
3. Quorum-based protocol

Important Note: Use of gRPC for RPCs is mandatory for this assignment.

Getting Familiar

1. Sequential Consistency in Distributed Systems [Kindly also read textbook section 7.2.1 - only about sequential consistency]
2. Protocols to implement sequential consistency [Kindly also read textbook Section 7.5.1 and 7.5.2]
   a. Primary-backup protocol (Remote-write)
      i. Blocking
      ii. Non-Blocking
   b. Quorum-based protocol
3. Revise Assignment 1 Implementation as the overall setup is similar - we will have a registry server, multiple servers, multiple clients

Setup

1. For the sake of simplicity, we will deploy the entire (k,v) store on one machine. Hence, each replica will be a separate process(opened in a separate terminal) on the same machine. Additionally, each replica will have a separate file directory in the local file system for persisting data.
2. For each data object **(key, value)** in the data store, **key** is the ***uuid*** while **value** is the ***(filename, timestamp)***. The actual file (referred by ***filename***) is stored as a text file in the local file system. For example:

Python

```
    key = 03ff7ebe-bf8e-11ed-89d9-76aef1e817c5
    value = ("Formula1ChampionsList.txt", "12/03/2023 11:15:11")
```

3. Each replica stores the following:
    a. (key, value) i.e (**uuid, (filename, timestamp))** mapping is stored in memory i.e. RAM.
    b. Actual file (referred by **filename**) is stored as a text file in the file directory of the particular replica. For instance,

Python

```
    filename = “Formula1ChampionsList.txt”
    directory_replica_one = “/home/dscd-a2/replica_1/”
    full_file_path =
    “/home/dscd-a2/replica_1/Formula1ChampionsList.txt”
```

4. Any client can interact with any replica directly. There can be multiple clients concurrently interacting with the data store.
5. **Use of gRPC for RPCs is mandatory for this assignment.**
6. You may use any programming language you wish to use. We recommend using Python.

Notes:

7. *"`Server`" and "`Replica`" are used interchangeably throughout the assignment and mean the same thing.*
8. *We keep track of the different versions of a file by associating each version with a timestamp.*
9. *Each file can be assumed to be very small in size (200-500 characters)*

Important Assumptions:

All assumptions are kept only to keep the assignment simpler for the benefit of students.

1. **There are no failures in the system. That is, once the replicas and registry server start, they are always online. They never fail. If any failure happens, then we have to shutdown the entire system and restart the entire system manually.**
    a. **This assumption also means that you don’t need to implement any roll-backs of any in-progress writes.**

2. **For quorum-based protocol, assume that the required number of servers (N, N_R and N_W) are already running before any client request arrives at the registry server.**

Implementation Details

Primary Backup Blocking Protocol:

Registry server:

1. The registry server resides at a known address (ip + port)
2. The registry server stores each replica's ip address (localhost) and port number - `localhost:8888`
3. Whenever a new replica comes up, it informs the registry server of its liveness
    a. Each replica shares its ip address and port with the registry server.
    b. Registry server marks the first replica as the primary replica.
    c. In response to each replica, the registry server always sends the information (ip + port) about the primary server.
    d. The registry server also needs to tell the primary replica about the joining of a new replica (send the ip + port of the new replica to the primary replica).
4. A client should get the list of replicas (list of ip address + port) from the registry server on startup.

Client:

1. **WRITE operation**
    a. The client first generates a UUID for the new file.
        i. If it is updating an existing file, then it reuses the same UUID which was generated previously.
    b. Client then sends uuid, filename, and content of the text file to any one replica.
    c. Replica responds back with the uuid of the saved text file and version timestamp.

Python

```python
    Request Proto from the client:
        String - Name
        String - Content
        Uuid - uuid
    Response Proto from the replica:
        String - Status
        Uuid - uuid
        Timestamp - Version
'''
For example:
Client sends -
        Name : "Formula1ChampionsList"
        Content : "M. Schumacher, L. Hamilton, J.M. Fangio."
        UUID : 03ff7ebe-bf8e-11ed-89d9-76aef1e817c5
Server sends -
        Status: SUCCESS
        UUID : 03ff7ebe-bf8e-11ed-89d9-76aef1e817c5
        Version : "12/03/2023 11:15:11"
'''
```

2. **READ operation**
    a. Client sends a uuid of the text file to read from the data store.
    b. Replica responds back with the text file and timestamp corresponding to the uuid.

```python
Request Proto from the client:
        Uuid - uuid of the requested file
Response Proto from the replica:
        String - Status
        String - Name
        String - Content
        Timestamp - Version
'''
For example:
Client sends -
        UUID : "03ff7ebe-bf8e-11ed-89d9-76aef1e817c5"
Server sends -
        Status - SUCCESS
        Name : "Formula1ChampionsList"
        Content : "M. Schumacher, L. Hamilton, S. Vettel."
        Version : "12/03/2023 11:15:11"
'''
```

3. **DELETE operation**
    a. Client sends a uuid of the text file to delete from the data store.
    b. Replica responds back with the status success / failure / any readable message

```Python
Request Proto from the client:
      Uuid - uuid of the file to delete
Response Proto from the replica:
      String - Status
'''
For example:
Client sends:
     UUID : "03ff7ebe-bf8e-11ed-89d9-76aef1e817c5"
Server sends:
     Status: SUCCESS
'''
```

Replica:

1. **READ operation:**
    a. **Possible Scenarios:** For any READ request (uuid), following scenarios are possible:
        i. uuid does not exist in the replica's in-memory map. In this case, replica understands that the client is trying to read a file which does not exist. This is not allowed and this operation should FAIL. Status message should say - FILE DOES NOT EXIST. For each of the other fields, return NONE/NULL
        ii. uuid exists in the replica's in-memory map and the corresponding file also exists in the file system: In this case, replica understands that the client is trying to read an existing file. The replica can return the text file from its file directory along with timestamp and status = SUCCESS.
        iii. uuid exists in the replica's in-memory map but the corresponding file does not exist in the file system: In this case, replica understands that the client is trying to read a deleted file. This is not allowed and this operation should FAIL. Status message should say - FILE ALREADY DELETED. Also, return the timestamp of the deletion and NONE/NULL for contents.
        iv. If you feel there are any other scenarios, please feel free to handle them as you wish.

2. **WRITE operation:**
    a. **Possible Scenarios:** For any WRITE request (uuid, filename, timestamp), following scenarios are possible:
        i. uuid does not exist in the replica's in-memory map and there is no file corresponding to the filename in the file system. In this case, replica understands that this is a new file which has to be created.
        ii. uuid does not exist in the replica's in-memory map but there is already a file corresponding to the filename (given as part of the write request) in the file system. In this case, replica understands that the client is trying to create another file with the same filename as another file. This is not allowed and this operation should FAIL. Status message should say - FILE WITH THE SAME NAME ALREADY EXISTS
        iii. uuid exists in the replica's in-memory map and the corresponding file also exists in the file system: In this case, replica understands that the client is trying to update an existing file.
        iv. uuid exists in the replica's in-memory map but the corresponding file does not exist in the file system: In this case, replica understands that the client is trying to update a deleted file. This is not allowed and this operation should FAIL. Status message should say - DELETED FILE CANNOT BE UPDATED
        v. If you feel there are any other scenarios, please feel free to handle them as you wish.
    b. **Detailed Procedure:**
        i. Replica sends the filename, uuid and content of the text file to the primary replica as each write gets executed on the primary replica first.
        ii. Primary replica (PR) adds/updates the entry for the received text file in the in-memory map and then writes the text file into its local store (file directory).
            1. The replica always overwrites the entire file if the file already exists (No appends). We are keeping this assumption for the sake of keeping the assignment simpler for the benefit of students.
            2. Then the PR sends the write request (along with uuid, filename, timestamp and content of the text file) to all the other replicas (called as backup replicas). PR waits for acknowledgements from all the backup replicas (this is a blocking call)
        iii. Each backup replica adds/updates the entry for the received text file in the in-memory map and then writes the text file into its local store (file directory). Also, each backup replica sends the acknowledgement back to the PR
        iv. Once the PR receives acknowledgements from all the backup replicas, it sends the confirmation message to the replica which received the write request from the client. Then the replica sends the write completion message (along with uuid) to the client.
3. **DELETE operation:**

a. **Possible Scenarios:** For any DELETE request (uuid), following scenarios are possible:
   i. uuid does not exist in the replica's in-memory map. In this case, replica understands that the client is trying to delete a file which does not exist. This is not allowed and this operation should FAIL. Status message should say - FILE DOES NOT EXIST.
   ii. uuid exists in the replica's in-memory map and the corresponding file also exists in the file system: In this case, replica understands that the client is trying to delete an existing file. The replica can delete the text file from its file directory (and updates the uuid entry in the in-memory map as explained in detail below). Status message should say - SUCCESS.
   iii. uuid exists in the replica's in-memory map but the corresponding file does not exist in the file system: In this case, replica understands that the client is trying to delete an already deleted file. This is not allowed and this operation should FAIL. Status message should say - FILE ALREADY DELETED.
   iv. If you feel there are any other scenarios, please feel free to handle them as you wish.

b. **Detailed Procedure**:
   i. Replica sends the delete request to the primary replica as each delete gets executed on the primary replica first.
   ii. Primary replica (PR) first deletes the file from its local store(folder) and updates the entry in the in-memory map (corresponding to the uuid) with an empty file name and a new timestamp to indicate the latest deleted version.
   iii. Then the PR sends the delete request (along with the uuid and timestamp) to all the other replicas (called as backup replicas). PR waits for acknowledgements from all the replicas (blocking call)
   iv. The backup replicas delete the file from its local store(folder) and update the entry in the in-memory map (corresponding to the uuid) with an empty file name and the timestamp received from PR. Also, each backup replica sends the acknowledgement back to the primary replica.
   v. Once the PR receives acknowledgements from all the backup replicas, it sends the confirmation message to the replica which received the delete request from the client. Then the replica sends the delete completion message to the client.

c. Note: The above steps for DELETE operation mean that for each DELETE operation, each replica deletes the file from its local store but keeps the corresponding uuid entry in its in-memory map (after updating its entry in in-memory map as mentioned above).

Primary Backup Non-Blocking Protocol:

Registry server:

Same as primary backup blocking protocol

Client:

Same as primary backup blocking protocol

Replica:

1. **READ operation:**
    a. Same as primary backup blocking protocol.
2. **WRITE operation:**
    a. **Possible Scenarios:** For any WRITE request (uuid, filename, timestamp), the list of possible scenarios are the same as the primary backup blocking protocol.
    b. **Detailed Procedure**:
        i. Replica sends the uuid, filename and content of the text file to the primary replica as each write gets executed on the primary replica first.
        ii. Primary replica (PR) adds/updates the entry for the received text file in the in-memory map and then writes the text file into its local store (file directory).
            1. Then, it sends the confirmation message to the replica which received the write request from the client. This replica in turn sends the write completion message (along with uuid) to the client.
        iii. Then the PR sends the write request (along with uuid, filename, timestamp and content of the text file) to all the other replicas (called as backup replicas). PR waits for acknowledgements from all the backup replicas (this is a blocking call)
        iv. Each backup replica adds/updates the entry for the received text file in the in-memory map and then writes the text file into its local store (file directory). Also, each backup replica sends the acknowledgement back to the PR
3. **DELETE operation:**
    a. **Possible Scenarios:** For any DELETE request (uuid), the list of possible scenarios are the same as the primary backup blocking protocol.
    b. **Detailed Procedure**:
        i. Replica sends the delete request to the primary replica as each delete gets executed on the primary replica first.
        ii. Primary replica (PR) first deletes the file from its local store(folder) and updates the entry in the in-memory map (corresponding to the uuid) with an empty file name and a new timestamp to indicate the latest deleted version.

            1. Next, it sends the confirmation message to the replica which received the delete request from the client. This replica in turn sends the delete completion message to the client.
        iii. Then the PR sends the delete request (along with the uuid and timestamp) to all the other replicas (called as backup replicas). PR waits for acknowledgements from all the replicas (this is a blocking call)
        iv. The backup replicas delete the file from its local store(folder) and update the entry in the in-memory map (corresponding to the uuid) with an empty file name and the timestamp received from PR. Also, each backup replica sends the acknowledgement back to the primary replica.
    c. Note: The above steps for DELETE operation mean that for each DELETE operation, each replica deletes the file from its local store but keeps the corresponding uuid entry in its in-memory map (after updating its entry in in-memory map as mentioned above).

Quorum-Based Protocol

Registry server:

1. The registry server resides at a known address(ip + port)
2. For each replica, the registry server needs to have an ip address (localhost) + port number - `localhost:8888`
3. The registry server takes as input `N_r, N_w` and `N` on startup. The Registry server must validate the read quorum and write quorum constraints on `N_r, N_w` and `N` where `N` is the total number of replicas, `N_r` is the size of read quorum, `N_w` is the size of write quorum. If the constraints are not satisfied, it throws an error and again asks for the valid set of `N_r, N_w` and `N`.
4. Whenever a new replica comes up, it informs the registry server of its liveness
    a. Each replica shares its ip address and port with the registry server.
5. For each read request, a client should get the list of `N_r` replicas (list of ip address + port) from the registry server on request.
    a. Registry server should select the N_r replicas randomly. Use any in-built function for random selection (provided by the programming language).
6. For each write request, a client should get the list of `N_w` replicas (list of ip address + port) from the registry server on request.
    a. Registry server should select the N_w replicas randomly. Use any in-built function for random selection (provided by the programming language).
7. For each delete request, a client should get the list of `N_w` of replicas (list of ip address + port) from the registry server on request.
    a. Registry server should select the N_w replicas randomly. Use any in-built function for random selection (provided by the programming language).

8. A client is also allowed to get the list of all the N replicas (list of ip address + port) from the registry server.
    a. This may be required for debugging purposes or in test scripts where you want to check the state on each replica.

Client:

1. **WRITE operation**
    a. The client first contacts the registry server to ask for the list of `N_w` servers from the registry server
    b. The client then generates a UUID for the new file
        i. If the client is updating an existing file, then it reuses the same UUID which was generated previously.
    c. Client then sends filename, content of the text file and a uuid to all the servers(received from the registry server)
        i. Replicas respond back with the uuid of the saved text file and the timestamp.

Python

```
Request Proto from the client to each replica:
      String - Name
      String - Content
      Uuid   - uuid
Response Proto from each replica to the client:
      String - Status
      Uuid - uuid
      Timestamp - Version
'''
For example:
Client sends -
      Name : "Formula1ChampionsList"
      Content : "M. Schumacher, L. Hamilton, J.M. Fangio."
      UUID : 03ff7ebe-bf8e-11ed-89d9-76aef1e817c5
Server sends -
      Status : SUCCESS
      UUID : 03ff7ebe-bf8e-11ed-89d9-76aef1e817c5
      Version : "12/03/2023 11:15:11"
'''
```

2. **READ operation**
    a. The client first contacts the registry server to ask for the list of `N_r` replicas from the registry server
    a. Client sends a uuid of the text file to read from each of the N_r replicas.
    b. Each replica responds back with filename, file contents and timestamp corresponding to the uuid.
        i. Client compares the timestamp received from each replica, selects the replica corresponding to the latest timestamp and accepts/prints the file contents received from this replica.

Python

```
Request Proto from the client to each replica:
      Uuid - uuid of the requested file
Response Proto from each replica to client:
      String - Status
      String - Name
      String - Content
      Timestamp - Version
'''
For example:
Client sends -
     UUID : "03ff7ebe-bf8e-11ed-89d9-76aef1e817c5"
Server sends -
     Status : SUCCESS
     Name : "Formula1ChampionsList"
    Content : "M. Schumacher, L. Hamilton, S. Vettel."
    Version : "12/03/2023 11:15:11"
'''
```

3. **DELETE operation**
    a. The client first contacts the registry server to ask for the list of `N_w` servers from the registry server
    b. Client then sends a uuid of the text file to delete the file from each replica in the received list of replicas.
    a. Each replica respond back with the status success / failure

```Python
Request Proto from the client to each replica:
      Uuid - uuid of the file to delete
Response Proto from each replica to the client:
      String - Status
'''
For example
Client sends -
      UUID : "03ff7ebe-bf8e-11ed-89d9-76aef1e817c5"
Server sends -
     Status: SUCCESS
'''
```

Replica:

1. **READ operation:**
   a. Same as the primary backup blocking/non-blocking protocol.
2. **WRITE operation:**
   a. **Possible Scenarios**: For any WRITE request (uuid, filename, timestamp), the list of possible scenarios are the same as the primary backup blocking protocol.
   b. **Detailed Procedure**: Replica adds/updates the entry for the received text file in the in-memory map and then writes the text file into its local store (file directory).
3. **DELETE operation:**
   a. **Possible Scenarios**: For any DELETE request (uuid), the list of possible scenarios are the same as the primary backup blocking protocol except the following scenario:
      i. uuid does not exist in the replica's in-memory map: In this case, replica understands that the client is trying to delete a file which does not exist. In this case, the replica should create an entry in the in-memory map: (uuid, (<empty filename>, current timestamp)). This ensures that the file is definitely deleted on N_w replicas. This covers the scenario when the delete operation is sent to a replica (which doesn't have the file but the file exists on some other N_w replicas).
   b. **Detailed Procedure:** Replica first deletes the file from its local store(folder) and updates the entry in the in-memory map (corresponding to the uuid) with an empty file name and a new timestamp to indicate the latest deleted version.

c. Note: The above steps for DELETE operation mean that for each DELETE operation, each replica deletes the file from its local store but keeps the corresponding uuid entry in its in-memory map (after updating its entry in in-memory map as mentioned above).

Evaluation / Testing

Since there are so many components in a distributed system, it is always a painful job to manually start/stop each of the components and keep giving inputs. So, for this assignment, we require you to write an automated test script which creates a registry server, replicas and clients as separate processes/threads and performs a bunch of read/write/delete operations and prints out all the results. Writing automated test scripts is fun and will prepare you for your industry jobs as well. Nothing gives more satisfaction to software developers than seeing the test script pass all the test cases successfully.

1. For each protocol, write an automated test script which does the following tasks:
    a. Create/Run a registry server.
    b. Create/Run N replicas.
    c. Create/run a client
    d. Client then performs a write operation
    e. Client then reads from all the replicas one by one
        i. Print the result of read from each replica
    f. Client then again performs a write operation
    g. Client then reads from all the replicas one by one
        i. Print the result of read from each replica
    h. Client then deletes one of the files (which was written previously)
    i. Client then tries to read the deleted file from all the replicas one by one
        i. Print the result of read from each replica
    j. Feel free to add more test cases as you consider fit. TAs may also ask you to add/modify some test cases.

Some of the resources which may be useful for writing the test script:

1. Spawn multiple processes from within a process: https://docs.python.org/3/library/multiprocessing.html
2. Testing framework/library
    a. https://docs.python.org/3/library/unittest.html
    b. https://realpython.com/python-testing/
    c. https://machinelearningmastery.com/a-gentle-introduction-to-unit-testing-in-python/
    d. It is not necessary to use a testing framework/library, feel free to just use plain simple python

Deliverables

1. Submit a zipped file containing the entire code.
2. Code for each protocol should be saved in a separate folder along with the test script.