Assignment #5
A Replicated Distributed File System

Objective

It is required to implement a replicated file system. There will be one main server (master) and, data will be partitioned and replicated on multiple replicaServers. This file system allows its concurrent users to perform transactions, while guaranteeing ACID properties. This means that the following need to be ensured:

- The master server maintains metadata about the replicas and their locations.
- The user can submit multiple operations that are performed atomically on the shared objects stored in the file system.
- Files are not partitioned.
- Assumption: each transaction access only one file.
- Each file stored on the distributed file system has a primary replica.
- After performing a set of operations, it is guaranteed that the file system will remain in consistent state.
- A lock manager is maintained at each replicaServer.
- Once any transaction is committed, its mutations to objects stored in the file system are required to be durable.

The characteristics of the file system

1. **Reads from the file system**.
   Files are read entirely. When a client sends a file name to the server, the entire file is returned to the client.
2. **Writes to files stored on the file system**.
   The following procedure is followed:
   1) The client requests a new transaction ID from the server. The request includes the name of the file to be muted during the transaction.
   2) The server generates a unique transaction ID and returns it, a timestamp, and the location of the primary replica of that file to the client in an acknowledgment to the client's file update request.
   3) If the file specified by the client does not exist, the server creates the file on the replicaServers and initializes its metadata.
   4) All subsequent client messages corresponding to a transaction will be directed to the replicaServer with the primary replica contain the ID of that transaction.
   5) The client sends to the replicaServer a series of write requests to the file specified in the transaction. Each request has a unique serial number. The server appends all writes sent by the client to the file. Updates are also propagated in the same order to other replicaServers.

6) The server must keep track of all messages received from the client as part of each transaction. The server must also apply file mutations based on the correct order of the transactions.
7) At the end of the transaction, the client issues a commit request. This request guarantees that the file is written on all the replicaServer disks. Therefore, each replicaServer flushes the file data to disk and sends an acknowledgement of the committed transaction to the primary replicaServer for that file. Once the primary replicaServer receives acknowledgements from all replicas, it sends an acknowledgement to the client.
8) The new file must not be seen on the file system until the transaction commits. That is a read request to a file that is being updated by an uncommitted transaction must generate an error.

3. Client specification.

1) Clients read and write data to the distributed file system.
2) Each client has a file in its local directory that specifies the main server IP address.

4. Main Server specification.

1) The server should communicate with clients through the given RMI interface.
2) The server need to be invoked using the following command.
    a. server -ip [ip_address_string] -port [port_number] -dir <directory_path>
    b. where:
        - **ip_address_string** is the ip address of the server. The default value is 127.0.0.1.
        - **port_number** is the port number at which the server will be listening to messages. The default is 8080.
        - **directory_path** is the directory where files are created and written by client transactions. This directory is the same for all replicaServers. At the server starts up, that directory may already contain some files (we could put some files there for testing). You must ensure that the files in that directory are not deleted after the server exits. We will use the contents in those files to check the correctness of your system.
3) When the client asks the primary replicaServer to commit a transaction, the server must ensure that the transaction data is flushed to disk on the all the replicaServers local file systems. Using the write is not enough to accomplish this because it puts the data into file system buffer cache, but does not force it to disk. Therefore, you must explicitly flush the data to disk when committing client transactions or when writing important housekeeping messages that must survive crashes. To flush the data to disk, use flush() and close() methods of the FileWriter in Java.

5. **ReplicaServer specification**.
   1) Each file is replicated on there replicaServers. The IP addresses of all replicaServers are specified in a file called "repServers.txt". The master keeps heartbeats with these servers.
   2) Acknowledgements are sent to the client when data is written to all the replicaServers that contain replicas of a certain file.
   3) You will need to implement one of the primary-based consistency protocols studied in class.

6. **Concurrency**.
   Multiple clients may be executing transactions against the same file. Isolation and concurrency control properties should be guaranteed by the server.

7. **Handling aborting a transaction**.
   A client can decide to abort the transaction after it has started. Note, that the client might have already sent write requests to the server. In this case, the client requests transaction abort from the server. The client's abort request is handled as follows:
   1) The primary replicaServer ensures that no data that the client had sent as part of transaction is written to the file on the disk of any of the replicaServers.
   2) If a new file was created as part of that transaction, the master server deletes the file from its metadata and the file is deleted from all replicaServers.
   3) The primary replicaServer acknowledges the client's abort.

Testing

You will need to follow the above specifications to allow us to test your file system using our client. These are some of the test cases that will be run to evaluate the correctness of your system.

- Basic client read and write operations to the file system. This includes: (1) the file updated by the client contains the correct written data after committing the transaction, (2) partial updates (before transaction commits) are not seen by other clients or other transactions by the same client, and (3) a new file created by a transaction is not visible in the file system until the transaction commits.
- In case of aborted transactions, we will double check that any mutations to a file as part of this transaction is not rolled back after the transaction is aborted. We will also check that any file that was created as part of the aborted transaction does not exist after the transaction aborts.

- Bonus: We will intentionally have the client omit some messages to test how the server handles undelivered messages.
- We will test multiple concurrent clients accessing the same file and check if the mutations to the file are done as if the clients' transactions were performed sequentially.
- Bonus: We will also test failing a client before committing its transaction.
- Bonus: The server failure while sending a transaction will also be tested. If there is only one server, we expect it to continue its transactions when it is restarted. If there are multiple servers, killing one of them means that the other servers should resume processing the transactions.

Bonus (15%):

- It is guaranteed that the data stored in the file system will survive any potential failures of the servers and clients.

  Server:

  1) The servers need to be stateful, so it needs to remember all of the transactions that are being processed by each client. Therefore, the server needs to keep a log or records to remember its state. Yet, these logs should not be visible in the server's file system. Log files should also be deleted when the server exits gracefully.
  2) The master needs to keep heartbeat of replicaServers and it needs to force replication if some of the replicaServers fail.

  Handling loss of messages.

  Each message sent within a transaction has a serial number. If some of the messages that are part of a transaction are lost, the server should not commit until it receives them. This can be handled as follows:

  1) When the transaction commit request is sent by the client, it should include the total number of messages that the server should have received.
  2) If the server realizes that it has not received all of the messages that the client claims to have sent, it must ask the client to retransmit the missing messages. The retransmission request must include the transaction ID as well as the missing message number.
  3) A separate retransmission request is sent for each missing message.

  Moreover, the client may lose the server's acknowledgement of the committed transaction. In that case, it will retransmit the message asking the server to commit the transaction. The server must remember that the transaction has been committed and re-send the acknowledgement to the client.

  Client failure.

  If a client crashes before committing the transaction, the transaction must never be committed. It is the responsibility of the server to decide how long to keep track of unfinished transactions by setting a timeout

Master server failure.

1) The replicaServers will not terminate unless explicitly terminated. Therefore, there is a chance that some transactions are resumed in the presence of master failure.
2) When the masterServer is restarted, the whole system is restarted. Metadata about the file system is constructed through exchanged heartbeats with the replicaServers.

ReplicaServer failure

1) The server should resume any uncommitted transactions after it is rebooted.
2) It should also remember committed transactions in case a client resends a message related to one of them.
3) If a transaction has committed but failure happens before acknowledging it, the server should remember that transaction and can send an ack if the client requested that.
4) The files need to be in consistent sate even if a crash happens. If a file is acknowledged to be committed then it is guaranteed to be committed even if a crash occurs.

Deliverables

- Client and server (main server and repServer) modules that follow the above specifications.
- A report that describes the details of the components of your file system, any assumptions that you have made, and the role of each of the team members.

Remark: This assignment is loosely based on lab assignments given at Simon Fraser University, Canada.