Distributed File system with Transactional Semantics

Your goal is to implement a distributed file system with transactional semantics. A transaction is a sequence of file operations that follow the properties of (A)tomicity, (C)consistency, (I)solation and (D)urability (ACID). That is, each sequence of file operations must appear atomic, leave the file system consistent, work in face of concurrency and survive server or client failures.

The file system supports both reads and writes, but semantics of read are quite simple: files are always read in their entirety. There are certain semantic rules that govern correct write operations submitted the file server, which we describe below. In addition, your system should implement the specified message format for the wire protocol, so that we can test your system using our own client. During the course of the assignment you will implement the file server and a client that you will use to test your server. You will submit to us the code for the file server; we will test it using our own client. The assignment can be done in any programming language, as long as your server is designed to accept messages over a TCP connection.

Your system will be required to handle omission, byzantine and failstop failures on the client and failstop failures on the server. The server must be stateful: it must remember about the operations it was doing before it crashed, as described by the rules below.

We now describe the rules for the system. First we describe how the client and server must interact and handle failures. Then we describe the message format for the client-server protocol.

Client-Server Interaction

We describe several modes of interactions between the client and the server, such interaction in the common-case (no failures) and interaction in face of failures.

Common case interaction (no failures, transaction is committed)

- Reads are quite simple. The client sends the name of the file to the server, and the server returns the file in its entirety. The rest of the section talks about writes.
- The client requests a new transaction ID from the server, at the same time specifying the file that will be written during that transaction
- The server generates a unique transaction ID and returns it to the client. If the file specified by the client in the message requesting transaction ID does not exist, the server creates the file.
- All subsequent client/server messages corresponding to a transaction will contain the ID of that transaction
- The client sends to the server a series of write requests to the transaction's file. Each request is numbered.
- The client asks the server to commit the transaction, that is to ensure that all write requests are written to the file on the server's disk.
- The server appends all writes sent by the client to the file, flushes the file data to disk and sends an acknowledgement of the committed transaction to the client.

- •The new file must not be seen on the file system until the transaction commits. That is a read request for a file that was created as part of an uncommitted transaction must generate an error.
- •The server must acknowledge the client's message requesting a new transaction. This acknowledgement message will have the transaction ID. Write requests can be acknowledged selectively.

We will test the correctness of common case interaction by checking that after the transaction commits, the file written by the client has the contents written during the transaction. We will also check that the data written to the file as part of the transaction is not seen in the file until the client has committed the transaction. We will also check that if a new file is created as part of the transaction, the file is not seen on the server's local file system until the transaction commits -- that is we will check that your server correctly returns an error when the client requests to read a file that is part of an uncommitted transaction.

Interaction in face of aborted transactions

- •A client may decide to abort the transaction after sending a few write requests to the server
- •In that case the client sends an "abort" request to the server
- •The server ensures that no data that the client had sent as part of transaction is written to the file on the disk
- •The server acknowledges the client's abort message
- •If a new file was created as part of that transaction, the server deletes the file
- •If the aborted transaction was about to create a new file, that file must not be seen on the file system at any point, because the enclosing transaction aborted. That is a read request for a file that was created as part of an uncommitted transaction must generate an error.

We will test the correctness of your system in face of aborted transactions by making sure that the data from the aborted client transaction is never to be seen in the file and the file that was created as part of the transaction is not in the file system if the transaction was aborted.

Interaction in face of client omission failures

- •A client may lose some of the messages sent as part of a transaction
- •The server must not commit a transaction unless it has received all write requests that were a part of this transaction
- •Therefore, the transaction commit request sent by the client includes the total number of messages that were sent by the client as part of that transaction
- •The server must keep track of all messages received from the client as part of each transaction
- •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
- •When asking for retransmission the server needs to tell the client the sequence number of the missing message (the server should be able to figure this out if it has kept track of message sequence numbers it received from the client)
- •If more than one message is missing, the server may ask for a retransmission multiple times
- •A 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.

We will test the correctness of your system in face of client omission failures by making the client deliberately omit messages and making sure that a transaction with missing messages is never committed to the file until the server asks for and receives all the missing messages.

Interaction in face of client failstop failures

- If a client crashes before commiting the transaction, the transaction must never be committed
- The server decides how long to keep track of unfinished transactions by setting a timeout

We will test the correctness of your system by making sure that the data from transactions that are not committed never appears in the file. This failure is similar to the case when the client aborts a transaction.

Interaction in face of Byzantine failures on the client

- Your server will be required to handle incorrect behavior by the client; that is when the client sends messages that do not follow the rules of the system
- For example, the client might send a message with a transaction ID that it never received from the server; the server must never commit a transaction with a fake ID.the server must continue operating correctly if this happens.
- If the sever detects incorrect operation, it should return error message to the client, describing the problem as specifically as possible.

We will test the correctness of your server in face of Byzantine client failures by generating messages with fake transaction IDs on the client and making sure that those transactions are never committed. We will also have the client generate incorrect messages and check that the server responds to the client with an appropriate error code.

Interaction in face of concurrency

- Multiple clients may be executing transactions against the same file
- The server must ensure that those transactions are isolated, that is, it must create the illusion that the clients' transactions committed sequentially, even if the clients executed them concurrently
- If two clients commit their transactions on the same file simultaneously, it is up to the server to decide whose data gets committed to the file first.

We will test the correctness of your server in face of concurrency by having many concurrent clients executing transactions against the same file and making sure that the messages written to the file by concurrent transactions appear in the file as they would had the transactions occurred sequentially.

*We will also enforce certain **concurrency requirements on your server.** Your server must process requests from multiple clients concurrently. We will test this by timing your server on a multiprocessor system. We expect that N concurrent transactions executed by N clients take measurably less time to complete than the same N transactions performed sequentially.*

Interaction in case of failstop server failures

•If the server crashes ***before*** the client has asked it to commit the transaction, it must be able to continue processing the transaction after it reboots. The server must be able to continue from the point where it last sent an acknowledgement to the client.
•If the server crashes ***after*** committing the transaction but before sending the acknowledgement of that fact to the client, it must remember that transaction had been committed. Once the server reboots it should be able to re-send the acknowledgement to the client if the client retransmits the request to commit that transaction
•If the server crashes ***while*** the transaction is being committed to the disk, it must ensure that the file is left in the consistent state after it reboots; that is, upon reboot it must ensure that no data from partially committed transactions is in the file. If there is such data, the server must fix the file.
•The server must remember about all transactions committed before the crash, in case a client retransmits a transaction commit request after the server has crashed and rebooted.
•If the server has acknowledged a committed transaction to the client, the transaction data must be in the file as soon as the client receives the acknowledgement, no matter what happens with the server.

We will test the correctness of your system in face of failstop server failures by killing your server while it is running, then manually restarting it, and making sure that it continues correct operation and cleans up the file system after the restart.

Wire Protocol

We will use a message format similar to HTTP (recall Lecture V). The message format is as follows:

Request message format:

Method	*Transaction ID*	*Message sequence number*	*Content length*	*Data*
WRITE	67861	2		

Method field contains the type of operation (all methods are listed bellow)

Transaction ID field specifies the ID of the transaction to which this message relates. In the "NEW_TXN" message, transaction must be set to any id, such as "-1". The transaction ID in the "NEW_TXN" message must be ignored by the server

Message sequence number field identifies the number of the current file operation in the current transaction. Each transaction starts with message sequence number 0, so the "NEW_TXN" message will have sequence number 0.

Content length field specifies the length of data (in bytes)

Data field contains the data to be written to the file (if the method is WRITE) or the file name is the method is (NEW_TXN)

•The first four fields of the message constitute a ***message header***

•The request header is followed by a single blank line (a "\r\n\r\n" sequence) if the message contains the data field. The data follows that blank line.

•The request header is followed by two blank lines (a "\r\n\r\n\r\n" sequence) if the message contains no data field (as with COMMIT method)

***Response message format**:*

Method	*Transaction ID*	*Message sequence number*	*Error code*	*Content length*	*Reason*
ERROR	67861	2			

Method field contains the type of operation (all methods are listed bellow)

Transaction ID field specifies the ID of the transaction to which this message relates

Message sequence number field in the context of response message specifies the sequence number of the message that must be retransmitted. This field only makes sense if the method is ASK_RESEND.

Error code field specifies the error code if the method is ERROR (all mandatory error codes are listed below)

Content length field specifies the length of reason (in bytes)

Reason field contains a human readable string specifying the reason for error

- The first five fields of the message constitute a ***message header***

•A response header is followed by two blank lines (a "\r\n\r\n\r\n" sequence) if the message does not include the "reason" field.
•If the message includes the "reason" field, the response header is followed by one blank line (a "\r\n\r\n" sequence)

Example messages

An example write request message (tags ==>begin and ==>end are not part of the message):
==>begin
WRITE 35551 1 35

Here is my data that goes into file
==>end
An example commit request message (tags ==>begin and ==>end are not part of the message):
==>begin
COMMIT 35551 8 0

==>end

Separation of header fields:

Fields in the message header must be separated by exactly one white space (ASCII character 32). No other characters may be used for field separation.

Methods:

READ - the client reads the file from the server. In that case, the transaction ID can be set to any number, as it will not be interpreted by the server: reads are not part of any transaction. The content-length field will contain the length of the file name. The data field will contain the file name itself. For this assignment you should assume that file names contain no spaces.

NEW_TXN - the client asks the server to begin a new transaction

WRITE - the client asks the server to write data as part of an existing transaction

COMMIT - the client asks the server to commit the transaction. In this case, the message sequence number field includes the total number of writes that were sent by the client as part of this transaction.

ABORT - the client asks the server to abort the transaction

ACK - the server acknowledges the committed transaction to the client or indicates a successful response to the client's NEW_TXN request.

ASK_RESEND - the server asks the client to resend a message for the given transaction whose sequence number is specified in the "message sequence number" field

ERROR - the server reports an error to the client. The error code and the ID of the transaction that generated the error must be included in the appropriate fields in the message

Error codes:

The following error codes are mandatory. You may implement more error codes, if you wish to enhance the client experience.

201 - Invalid transaction ID. Sent by the server if the client had sent a message that included an invalid transaction ID, i.e., a transaction ID that the server does not remember

202 - Invalid operation. Sent by the server if the client attemtps to execute an invalid operation - i.e., write as part of a transaction that had been committed

205 - File I/O error

206 - File not found

Implementation Rules and Tips

Starting your server Your server must accept three command line arguments:

- the IP address on which it will listen for client connections - we provide this in case the host where your server is running has multiple network interface cards and so it can listen on several different IP addresses.
- the port

- the directory where it will keep the files created by the clients

The file system Your server will store files created and written by client transactions in the directory specified on startup as a command line argument. When 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.

Since your server is statefull, it probably needs to keep a log or records helping it to remember its state. Make sure that those housekeeping logs are hidden from view on the server's file system (recall how Unix implements hidden files). Those files must be deleted every time the server exits gracefully. (If the server exits gracefully it is okay for it to forget about any transactions executed in the past.)

Committing file data to stable storage When the client asks the server to commit a transaction, the server must ensure that the transaction data is flushed to disk on the local file system. Using the"write" system call (if you are programming in C) is not enough to accomplish this. If you remember from your operating systems course, the "write" system call 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 thedata to disk, use the fsync (or fflush) system call. To tell you the truth, even fsync or fflush will not guarantee that the data is forced to disk: many operating systems return from fsync before the data is flushed to disk, while flushing the data on the background. To *really* flush the data to disk, you must call fsync (or fflush) twice.

If you are programming in a language other than C or C++ you must find out the equivalent to the fsync system call.