MP3: Distributed transactions

Due: May 1, 11:55pm

In this MP you will be implementing a distributed transaction system. You goal is to support transactions that read and write to distributed objects while ensuring full ACI(D) properties. (The D is in parentheses because you are not required to store the values in durable storage or implement crash recovery.)

Clients, Servers, and Objects

Your system should support clients and servers. A server stores objects and makes updates to them, while a client interacts with the user to execute transactions on the server. Your system should support five servers (named A, B, C, D, E) and up to 10 clients. You may have additional nodes acting as coordinator(s) for the transaction.

Unlike the previous MPs, you *do not have to handle failures* and can assume that all the servers (and clients) remain up for the duration of the demo.

Objects are named with the server identifier and the object name; e.g., `A.x` is object `x` stored on server A. Note that the name spaces on each server are separate; i.e., A.x and B.x are separate values. The object names consist of non-whitespace characters; object values are strings that do not include the return character.

Client Interface

At start up, the client should automatically connect to all the necessary servers and start accepting commands typed in by the user. The user will execute the following commands:

- `BEGIN`: Open a new transaction, and reply with "OK".
- `SET server.key value`: Set the value of an object with the named key stored on the named server. E.g.:

```
SET A.x 1
OK
SET B.y 2
OK
```

You should either create a new object, or, if one exists, update the value of an existing object with the given name, and reply with `OK`.

- `GET server.key`: Get the value of the object named by the key on the named server. Reply with: `server.key = value` on a separate line. E.g.:

```
GET A.x
A.x = 1
```

If a query is made to an object that has not previously been SET, you should return `NOT FOUND` and abort the transaction.

- `COMMIT`: Commit the transaction, making its results visible to other transactions. The client should reply either with `COMMIT OK` or `ABORTED`, in the case that the transaction had to be aborted during the commit process.
- `ABORT`: Abort the transaction. All updates made during the transaction must be rolled back. The client should reply with `ABORTED` to confirm that the transaction was aborted.

Notes:

- You do not need to support `GET` or `SET` commands outside of a transaction
- A transaction should see its own tentative updates; e.g., if I `SET A.x` to 1 in a transaction, then call `GET` on `A.x` in the same transaction, I should see the value 1. Whether updates from other transactions are seen depends on whether those transactions are committed and isolation properties, discussed below.
- If a `GET` or `SET` operation requires a lock, you should not reply until the lock has been acquired. However, a client may send `ABORT` before getting a response and you must abort the transaction.

Concurrency and Isolation

Clients may execute transactions concurrently. You should guarantee the serializability of the executed transactions. This means that the results should be equivalent to a serial execution of all committed transactions. (Aborted transactions should have no impact on other transactions.) You may want to use two-phase locking to achieve this, though this is not a strict requirement. (E.g., you can implement optimistic concurrency instead.)

You *must* support concurrency between transactions that do not interfere with each other. E.g., if T1 on client 1 executes `SET A.x, GET B.y` and then T2 on client 2 executes `SET A.w, GET B.z`, the transactions should both proceed without waiting for each other. In particular, using a single global lock (or one lock per server) will not satisfy the concurrency requirements of this MP. You should support read sharing as well, so `GET A.x` executed by two transactions should not be considered interfering.

On the other hand, if T1 executes `SET A.x` and T2 executes `GET A.x`, you may delay the execution of one of the transactions while waiting for the other to complete; e.g., `GET A.x` in T2 may wait to return a response until T1 is committed or aborted.

Deadlock Detection

Depending on your locking strategy, you may need to detect and resolve deadlocks. This may require aborting some existing transactions. Your design should minimize the number of transactions aborted. You should notify a client that a transaction has been aborted by returning `ABORTED` as the result of a `GET` or `SET` command (instead of `OK` or a value). Remember that deadlocks may span multiple servers and clients.

You should not use timeouts as your deadlock detection strategy because transactions will be executed interactively and this will therefore result in too many false positives.

Note: you should think about deadlock detection in your design process, but we strongly recommend that you complete the rest of the functionality first before implementing deadlock detection. In testing the rest of the functionality, we will ensure to access variables in a way that induces a lock order, avoiding any deadlocks, so you can still get credit for most of the MP even if your deadlock detection doesn't work.

Some more examples

Examples of full transaction execution. Normal transaction:

.

```
BEGIN
OK
SET A.x 1
OK
SET A.y 2
OK
COMMIT
COMMIT OK
```

Aborted transaction:

```
BEGIN
OK
SET A.x 1
OK
SET A.y 2
OK
ABORT
ABORTED
```

Transaction that was aborted by the client while waiting for lock for `A.y`

```
BEGIN
OK
SET A.x 1
OK
SET A.y 2
ABORT
ABORTED
```

Transaction that was aborted by the server while waiting for lock for `A.y`

```
BEGIN
OK
SET A.x 1
OK
SET A.y 2
ABORTED
```