Distributed Transactions

Team Details

- Cluster 45
- Git Repo: https://gitlab.engr.illinois.edu/neilk3/cs425-mp3-neilk3-ngogate2/
- Git Revision: 98b00a18fa09a1f28faf330e83d85089ffe3d25d

Group Members

- Neha Gogate (ngogate2)
- Neil Kaushikkar (neilk3)

Build Instructions:

1. You must first have the Rust compiler (rustc) and Cargo installed. You can either run `curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh` to install these or visit https://www.rust-lang.org/tools/install for more installation options. Follow the instructions for the default installation of Rust when prompted.
2. If you are installing Rust for the first time, you will also need to run the following command to add `cargo` to your path for the current shell/session: `source "$HOME/.cargo/env"`
3. Run `make` in the project root directory. This will build 2 executables `./server` and `./client`.

Running Instructions:

1. To start up a server, run `./server [node id] [path to config]`. Our system requires that `[node id]` be a single character (as described in the documentation). The system will repeatedly attempt to connect all nodes within 60 seconds. If the nodes do not join within that time, the server executable will exit.
2. To start a client, run `./client [client id] [path to config]`. Our system DOES NOT use the `[client id]` field but it is kept in place to comply with the requirements. Thus, the `[client id]` field does not need to be unique across different transactions. The client will generate error message logs when it encounters TCP connection errors or when it receives invalid input. To prevent these errors from printing to the terminal, run the client executable as follows: `./client [client id] [path to config] 2> /dev/null`.

Design:

Our system uses timestamped ordering to enforce an optimistic concurrency control. The system maintains a set of read timestamps and a set of tentative writes and their timestamps for each account on a server. The system then partitions the accounts by the first letter (i.e. an account starting with `A` will be stored on server `A`). Our system also uses 2-phased commit to ensure consistency across all servers (more on this below). Our system generates timestamps using the timestamp at the instant the client begins a transaction and connects to a coordinator. Timestamp ties across servers are broken by the node id of the server (coordinator). Local timestamp ties are broken by simply keeping track of the last timestamp generated and setting the newly generated timestamp to the last timestamp plus 1 if the newly generated timestamp is less than or equal to the last timestamp generated.

Data Structures

Our implementation uses locks to allow the system to process concurrent client requests on a server. However, the server will never encounter a deadlock since it enforces timestamped ordering rules (i.e. older transactions will never wait on newer transactions). Each object maintains an ordered set of read timestamps, an ordered map of tentative writes (ordered by timestamp), the timestamp of the last commit to the object, and the value of the object itself. We use the ordered set and map so we can easily check if some transaction must wait for an older transaction to commit or abort before committing.

Representing Tentative Writes

Our system uses a notion of 'diffs' to represent writes (similar to differntials from calculus). We represent a withdraw request for some amount `w` with a diff of `-w` and a deposit request for some amount `d` with a diff of `d`. That is to say, we represent deposits by a positive diff and withdrawals with negative diffs. We commit writes by applying diffs to the value of an object. In our case, applying a diff to an object's value means adding the diff to the object value. The only difference with this approach and simply storing the sum of the amount withdrawn or deposited and the account balance at the time of the write upon any tentative write is that we do not encounter and never need to deal with any stale/incorrect balances in newer transactions whenever an older transaction is aborted.

We require that diffs be 'updateable'. Multiple diffs should be able to be combined into a single diff that can be applied upon a commit. This allows multiple diff (write) in a single transaction to be flattened into a single diff (write) that is applied to an object upon a commit. We could have removed this requirement of combining diffs and simply applied the sequence of diffs (writes) from a single transaction upon a commit, but requiring the diffs to be combineable makes the implementation (specifically complying with Rust's ownership rules) a lot easier. In our system, we combine diffs by simply adding their values. There are no issues with this under our account balance implemenation since addition is both commutative and associative.

Waiting for Older Transactions

Certain conflicting operations from newer transactions may need to wait for older transactions to either be committed or aborted before being able to proceed. Each server maintains a notification list for each transaction that the entire system encounters. Each server maintains a task (also known as a green thread) for each client it is connected to. We also maintain a task for each request issued from another server in the system. These requests are from coordinators forwarding a client request to other servers when the coordinator server does not own the object referenced in the request. We can block any task whenver it issues a conflicting operation that needs to wait for another transaction to complete without blocking the entire system. Whenever a task needs to block, it will subscribe to the notification list of the transaction it must wait for. When any transaction commits or aborts, the server will notify all other tasks with blocked conflicting operations that are subscribed to the notification list associated with the transaction. The blocked tasks can then re-attempt the conflicting operation. This notification list approach is similar to conditional variables in system programming.

Local Locks

Our system uses a lock for each account and a lock on the map storing all accounts to prevent concurrency bugs in our implementation of the timestamped ordering rules since we have many concurrent tasks attempting to access the map of all objects on a server and each object itself. However, our approach differs in that locks are only held for short amounts of time – just enough to apply a read or write rule for a transaction's operation on an object. The read and write rules are simple checks that will potentially update an underlying data structure. No task will hold a lock for large amounts of time while waiting for another transaction to complete.

Handling Timestamp Ordering Conflicts

Whenever an older transaction attemps to read or write an object that has been committed to by a newer transaction, the server servicing the request will notify the coordinator (or the coordinator itself was attempting to service the request and will notify itself) and the coordinator will abort the transaction. Whenever a server encounters a request for an object that does not exist, it will notify the coordinator (or again the coordinator will notify itself) and the coordinator will abort the transaction. Likewise for withdraw requests or read requests to non-existent objects.

Handling Aborts & Avoiding Stale Data

On an abort, the coordinator will notify all other servers to abort the aborted transaction. The server will then notify each object to abort the transaction. Each object will purge any tentative writes from its queue of tentative writes. Then the server will send a notification to all tasks subscribed to the transaction's notification list, as described above. Since our system uses a notion of 'diffs' (described above), we do not need to modify any other tenative writes or the value itself on an abort. We simply discard the diff, and the system will proceed as if the write associated with the diff never occurred. We are not using concrete values in our tentative writes so no other transaction will encounter the diff until the diff is successfully committed.

Handling Commits

When a client issues a request to commit its transaction, the coordinator that is servicing the client will ask all other servers to begin a consistency check. Each server will perform a consistency check on each object it owns and notify the coordinator if whether all consistency checks passed or if any consistency check failed. If any consistency check on any server fails, the coordinator will issue an abort command to all servers and the system will proceed with the abort protocol described above. If all consistency checks at all servers pass, then the coordinator will issue another request to commit the transaction. Each server will direct each object to commit the diff associated with the committing transaction (if such a diff exists). Each object will apply the diff if there were any tenative writes on the object, update the value and committed timestamp of the object, and remove the tentative write from the object's ordered map. The coordinator will go ahead and notify the client that the commit succeeded once it received the result of the consistency checks from all servers. The coordinator will notify the client that a commit failed once it receives just one response from a server indicating the consistency check failed.

Committing Newer Transactions

When a server is performing the consistency check and the transaction attempting to commit must wait for another transaction to commit if an older transaction also performed a tentative write without committing, the server will use the same notification list subscription technique described above to wait for the older transaction to complete before performing the consistency check on each object.