Distributed Lock Manager

INFR11022: Distributed Systems, Autumn 2024

VADIM ZARIPOV

1 ARCHITECTURE

This distributed lock manager is implemented in C and consists of two parts: client library and server application. Clients are able to acquire and release the distributed lock and also write to the server's files. The application is fault-tolerant: it can handle client, server, and network failures. It uses Raft algorithm to replicate data in a cluster of servers.

Server uses the following modules:

(1) `server_rpc.h` – used to answer clients' RPC requests; handles network failures.
(2) `raft.h` – implementation of the Raft algorithm; manages leader election, lock replication, log compaction, etc. Also uses additional files `raft_leader.h`, `raft_follower.h`, `raft_candidate.h`, `raft_utils.h`, and `raft_storage_manager.h`.
(3) `udp.h` – UDP managing module provided in a coursework.
(4) `spinlock.h` – spinlock implementation.
(5) `tmdspinlock.h` – spinlock with the built-in timer; used to handle client failures. Uses a timer `timer.h`

Client uses the `client_rpc.h` library to communicate with the server.

2 RPC IMPLEMENTATION

2.1 Client side

RPC's were implemented using UDP sockets. `client_rpc.h` handles network failures by resending the request to the server if it has not received a response after a certain timeout (`RPC_READ_TIMEOUT = 100`). The system distinguishes between different requests and the duplicates of the same request by adding the clients `vtime` to the request RPC – `vtime` is updated between separate requests, but not between resending of the same request. Moreover, when the client receives no response after `RPC_RETRY_LIMIT = 10` requests, it concludes that the server is down and attempts to contact a different server in the Raft cluster.

Applications using `client_rpc.h` must ensure that the following requirements are satisfied:

- Each client issues RPC requests synchronously, e.g. a new RPC can be issued only after the previous one has returned. This implies that a client has to get the response from the server before it can issue the next request.
- The first request is always `RPC_init` and each client has a unique client id passed to `RPC_init`. No client with this id can be initialized before this client runs `RPC_close`.

If these requirements are satisifed, `server_rpc.h` can ensure some invariants described below.

2.2 Server side

Server has the following data structures:

(1) `client_process_data_t` – has information about a client connection: client's id, vtime, state, and last response. Protected by its own lock.
(2) `server_rpc_conn_t` – server RPC connection. Contains the client table, raft state (to check that we respond only of we are the leader), and function pointers to RPC handlers.

The only thing that `server_rpc.h` does is that it processes incoming RPC and calls the corresponding handlers. If clients behave as expected, it guarantees the following properties:

- For each client, at most one request handler is running in every moment. If the server is already handling some client request and this client sends another request (or repetition of the same one), `server_rpc.h` will return `E_PROCESSING` without invoking a handler.
- `server_rpc.h` stores the last response for each client, meaning that if the client repeats a request that was just answered, the library resends the response once again, not invoking a handler.
- If a server is not a leader of the Raft group, but still receives a use request, `server_rpc.h` returns `E_FOLLOWER` or `E_ELECTION` without invoking a handler.

Therefore, if `server.c`, that uses `server_rpc.h`, has its user request handler invoked by `server_rpc.h`, it can be sure that (1) this server is a leader, (2) this is the first and the last time this request is called on this server, and (3) no handlers for this client run on any of the other threads. This way, RPC libraries handle all network failures between clients and servers.

3 CONSISTENCY MODEL

This design uses the **bounded consistency model**: updates are replicated after every lock release. This implies that a single **transaction** consists of all updates to the file system that a single client issued between acquiring and releasing the lock. This is justified because it guarantees the clients that all the changes to different files within a single session between getting and releasing the lock will be added atomically – either they all will be applied together or none will be applied. In terms of performance, this is more optimal than replicating each individual request.

The disadvantage of this approach is that if the server crashes before the client releases the lock, all the client `AppendFile` requests after it has acquired the lock will not be applied, so the client might need to repeat them.

In order to give clients certainty about whether or not its changes were committed, there is one important invariant that the system maintains: *when the* `ReleaseLock` *RPC returns successfully, the client is* ***guaranteed*** *that the corresponding transaction is committed and replicated, and when* `ReleaseLock` *returns an error, the client is* ***guaranteed*** *that the corresponding transaction was lost and will never be applied.* This implies that before responding to the `ReleaseLock` RPC, the server waits until it knows for sure that the transaction is committed or will never be committed.

4 REPLICATION STRATEGY

This system uses Raft consensus algorithm to replicate log entries. `raft.h` declares the following structures used by Raft:

(1) `raft_configuration_t` – configuration of a raft cluster. Contains a list of `raft_server_configuration_t` objects that specify the socket ports, ids, and file directories of each of the servers in a cluster. Clients need to know this configuration to find a leader, and servers need to know it to communicate with each other.
(2) `raft_log_entry_t` – raft log entry. Stores the entry's term, id, type, client id, and the list of `raft_transaction_entry_t` objects, which contain the information about filename and buffer of each file write. There are maximum `MAX_TRANSACTION_ENTRIES = 10` transaction entries, meaning that during each acquire-release session a client can only write to 10 files.
(3) `raft_state_t` – main raft structure containing the state of the server. Stores the current log, term number, commit index, etc. Raft state is protected by its own spinlock.
(4) `raft_packet_t` – raft packet.

4.1 Leader election

Leader election is done precisely as in the Raft paper, with only one exception: as soon as the leader is elected, it adds to the log one artificial entry of its own term, attempting to commit it. This ensures that even with no client requests, there will be a committed log entry of the new term very soon after a leader is elected. This allows to determine whether a log entry of a previous term is committed or will never be committed soon after an election, which allows to answer `LockRelease` requests for previous term transactions without waiting for new log entries.

4.2 Log compaction

The log size used in the system is `LOG_SIZE = 100`, which is not sufficient in the long term. Therefore, each server periodically saves snapshots of the file system up to a certain committed log entries. Each snapshot is saved in a separate directory, and upon a creation of a new snapshot the previous one is deleted. The snapshot is created independently on each server when there are `COMMITS_TO_SNAPSHOT = 60` committed log entries in the log. However, it is not efficient to snapshot all committed transaction, since there is a high chance what we will need to send some of the last committed entries to the servers lagging behind. Thus, we only snapshot `SNAPSHOT_SIZE = 50` first committed log entries.

When a server is lagging behind, the leader might need to send the whole snapshot to it, which is also supported by the Raft module. Leaders use `snapshot_iterator_t` object contained in `raft_storage_manager.h`, which allows to split the snapshot into chunks of data that could be sent over `InstallSnapshot` RPC.

5 TESTING

Binary files are saved to `./bin` directory; object files are saved to `./build` directiory. Run `make clean` to clear them. `./raft_config` contains the raft configuration. According to this configuration, servers use socket ports `10000`, ..., `10004` for client requests, and `30000`, ..., `30004` for Raft requests, and save files to directories `./server_files_1`, ..., `./server_files_5`. Run `make clean_files` to clean all data files of all servers, and run `make clean_snapshots` to remove all snapshots saved on each server.

Run `make run_<test name>` to run a test. You can view available tests in the `./tests` directory. For example, `make run_test1_packet_delay` tests system's behavior when there is a packet delay. If you want the system to keep all previous data (e.g. not clear the state of each server before the test), include the `use-backup` flag (e.g. `make run_test1_packet_delay use-backup`).

If you want to just run a server (not a test), run `make run_server ./raft_config <server id>`, or, if you want to use a backup, `make run_server ./raft_config <server id> use-backup`. There are several test client programs in `./test_clients`, you can run any of them using `make run_<client name> ./raft_config <id> <port>`. Also `test_clients.c` tests their behavior.