Section 13: Two Phase Commit

CS162

April 19–20, 2016

Contents

1 Warmup

Suppose you had a remote storage system composed of a client (you), a single master server, and a single slave server. All units are separated from each other and communicate using RPC. There is no caching or local memory; all requests are eventually serviced using the backing store (disk) of the slave. The slave guards itself against failure by committing entries to a non-volatile log that never gets deleted in the event of a crash. The system only understands PUT(VALUE) and DEL(VALUE) commands, where VALUE is an arbitrary string. Calls to DEL on values that dont exist cause the slave to VOTE-ABORT.

Suppose you issue the following sequence of commands. Recall that the correct sequence of message passing is CLIENT - MASTER - SLAVE - MASTER - CLIENT. Calls to PUT on values that already exist cause VOTE-ABORT.

- PUT(I LOVE)

- PUT(OPERATING SYSTEMS)

- DEL(I LOVE)

- DEL(I LOVE)

- PUT(GOBEARS)

What is the sequence of messages sent and received by the MASTER server? List communications with the slave only. Your answer should be a list of the form:

SEND: PUT(XXX)

RECIEVE: VOTE-XXX

SEND: DEL(XXX)

...

```
SEND: PUT(I LOVE)
RECEIVE: VOTE-COMMIT
SEND: GLOBAL-COMMIT
RECEIVE: ACK
SEND: PUT(OPERATING SYSTEMS)
RECEIVE: VOTE-COMMIT
SEND: GLOBAL-COMMIT
RECEIVE: ACK
SEND: DEL(I LOVE)
RECEIVE: VOTE-COMMIT
SEND: GLOBAL-COMMIT
RECEIVE: ACK
SEND: DEL(I LOVE)
RECEIVE: VOTE-ABORT
SEND: GLOBAL-ABORT
RECEIVE: ACK
```

```
SEND: PUT(GOBEARS)
RECEIVE: VOTE-COMMIT
SEND: GLOBAL-COMMIT
RECEIVE: ACK
```

What is the sequence of messages committed to the log of the slave?

```
PUT(I LOVE)
COMMIT
PUT(OPERATING SYSTEMS)
COMMIT
DEL(I LOVE)
COMMIT
DEL(I LOVE)
ABORT
PUT(GOBEARS)
COMMIT
* notice that there are no read commands (i.e. GET) in this toy example,
but there would be no need to commit reads anyways as they do not modify
the state of the server.
```

Why dont you simply log the state of the slave instead of the action sequence?

In this basic toy example, it doesnt really make a difference because there is only one client and everything occurs in serial. However, in situations where there are multiple concurrent requests submitted to the master, it is important for the slave to keep track of not its instantaneous state but the sequence of actions it has serviced. Concurrent writes are not guaranteed to be idempotent and logging only the state of your slave gives it no ability to distinguish between duplicate requests in the event it dies AFTER writing the GLOBAL-COMMIT to the log in phase 2 but BEFORE it sends an ack back to the master. (Recall that this scenario will cause the master to continuously poke the slave with GLOBAL-COMMIT commands until a response is received. You should not honor the commit requests after the first one and you have no way to track that if only the state is logged.)

2 Vocabulary

- **Transaction** - A transaction is an indivisible set of data operations that must either succeed or fail as a single unit, that takes data from one valid state to another valid state. Commonly, reliable transactions are said to satisfy the ACID properties.
- **ACID** - An acronym standing for the four key properties of a reliable transaction.
  Atomicity - the transaction must either occur in its entirety, or not at all.
  Consistency - transactions must take data from one consistent state to another, and cannot compromise data integrity or leave data in an intermediate state.
  Isolation - concurrent transactions should not interfere with each other; it should appear as if all transactions are serialized.
  Durability - the effect of a committed transaction should persist despite crashes
- **Idempotent** - Idempotency is the property on operations that defines the ability of an operation to be repeated without effect beyond the first occurrence. If an operation can be repeated multiple times, but only the first such operation changes the outcome, the operation is said to be idempotent.

- **Logging file system** - A logging file system is a file system in which all modifications are done via transactions. Instead of modifying the disk directly, intended changes are written to an append-only log. Once the transaction is committed, it becomes safe to copy those changes onto disk, since in the event of a crash, transactions committed to the log can safely be re-copied onto disk.
- **TPC/2PC** - Two Phase Commit is an algorithm that coordinates transactions between one coordinator (Master) and many slaves. Transactions that change the state of the slave are considered TPC transactions and must be logged and tracked according to the TPC algorithm. TPC ensures atomicity and durability by ensuring that a write happens across ALL replicas or NONE of them. The replication factor indicates how many different slaves a particular entry is copied among. The sequence of message passing is as follows:

  ```
  for every slave replica and an ACTION from the master,
  origin [MESSAGE] -> dest :
  ---
  MASTER [VOTE-REQUEST(ACTION)] -> SLAVE
  SLAVE [VOTE-ABORT/COMMIT] -> MASTER
  MASTER [GLOBAL-COMMIT/ABORT] -> SLAVE
  SLAVE [ACK] -> MASTER
  ```

  If at least one slave votes to abort, the master sends a GLOBAL-ABORT. If all slaves vote to commit, the master sends GLOBAL-COMMIT. Whenever a master receives a response from a slave, it may assume that the previous request has been recognized and committed to log and is therefore fault tolerant. (If the master receives a VOTE, the master can assume that the slave has logged the action it is voting on. If the master receives an ACK for a GLOBAL-COMMIT, it can assume that action has been executed, saved, and logged such that it will remain consistent even if the slave dies and rebuilds.)

3 Problems

3.1 Delay of Game

Consider a system with three worker nodes and a master node. A client connects and performs an action which triggers a two-phase commit. The one-way latency between the client and the master is 100ms. The latency between the master and each of the slaves is 10ms. Let's assume that it takes each slave 20ms to update its log and/or commit, and all other processing/transmission time is negligible.

For each scenario, please give the amount of time that passes from when the client first sends its request to the time the slaves receive the final GLOBAL-COMMIT or GLOBAL-ABORT message AND draw the diagram corresponding to the communications sent between the master and the workers.

a.) Every transmission occurs correctly (and no nodes fail).

100ms (client send)
+ 10ms (VOTE_REQ)
+ 20ms (workers advance to READY state, write to log, send VOTE_COMMIT)
+ 10ms (master receive VOTE_COMMIT messages)
+ 10ms (master send GLOBAL_COMMIT messages)
= 150ms
(diagram not shown)

b.) One worker node disconnects and misses the VOTE-REQ, and does not reconnect fast enough to stop the master from timing out. The master times out in 5s. The worker comes back online just as the master times out, for simplicity.

100ms (client send)
+ 5000ms (master timeout)
+ 10ms (GLOBAL_ABORT)
= 5110ms or 5.11s

3.2 Two Pigs with One Bird

You are playing a game of Irascible Avians, a turn based game of loose physics simulations. The game state is hosted entirely on remote machines located somewhere underground in Switzerland. Your machine communicates with the host server via RPC. Because of the high popularity of the game, the master server does not perform any computations or host any storage and offloads those tasks to worker nodes. The master and worker nodes are synchronized via Two Phase Commit.

The only legal message you may send to the remote master is SHOOT(ANGLE, POWER). Because the developers are obsessive about data integrity, (for competitive leaderboards and such) your particular game session is replicated across two different worker servers. When you (the client) make a move in the game, the master forwards the SHOOT message to BOTH workers. The workers then compute the resulting game state at the end of your turn and forward the resulting STATE to the master. The master then communicates the new state of the game back to you. (Note that this is an example of a recursive query.)

If the action fails to commit on at least one of the workers, the worker propagates an ERROR back to the user and the game behaves as if the current turn did not occur. Each worker knows the initial state of its game session and logs the sequence of committed actions specified by the user. The log is non-volatile and invincible to all known disasters. A crashed worker can recreate the current state of the game by executing the series of logged actions on its initial state (the game is deterministic).

(a) First, draw and label a diagram of the setup. Use boxes to denote machines/nodes and arrows to denote remote communications for a successful turn. Label each arrow with numbers to denote the order of message passing in a complete sequence. (Remember that the master coordinates the workers with 2PC.) The master contacts each slave in serial. The game session is unique to you (your messages will not be confused with other players) and only one message from the client may be processed at a time (the game is discrete/turn based). On a successful commit, the worker sends the game state data with its final ACK. Also, CIRCLE the commands that the slave needs to save to its log.

```
1. client -> master: SHOOT(ANGLE, POWER)
2. master -> slave1: VOTE_REQ( SHOOT(ANGLE, POWER) ) [LOG THIS]
3. slave1 -> master: VOTE_COMMIT
4. master -> slave2: VOTE_REQ( SHOOT(ANGLE, POWER) ) [LOG THIS]
5. slave2 -> master: VOTE_COMMIT
5. master -> slave1: GLOBAL_COMMIT [LOG THIS]
6. slave1 -> master: ACK + GAME_STATE
7. master -> slave2: GLOBAL_COMMIT [LOG THIS]
8. slave2 -> master: ACK + GAME_STATE
9. master -> client: GAME_STATE

draw something pretty here.
```

(b) Suppose the second worker dies during PHASE 1 of 2PC and does not rebuild until after the master times out. What happens if worker death occurs before the phase 1 action is written to the log?

How about after? List the sequence of actions that occurs for both cases and indicate which actions are logged. Your answer should be of the form:

```
1. client -> master: SHOOT(POWER, ANGLE)
2. master -> slave1: <MESSAGE> + LOG?
...
```

Theres no difference in either case in behavior. The only difference is that the slave will either log an ACTION paired with an ABORT, or simply an ABORT with no pair (which should be ignored on rebuild). Dying in phase 1 (defined as dying before the slave sends a vote) and causing a timeout will always cause the master to send a GlOBAL-ABORT and roll back the last action.

```
1. client -> master: SHOOT(ANGLE, POWER)
2. master -> slave1: VOTE_REQ( SHOOT(ANGLE, POWER) ) [LOG]
3. slave1 -> master: VOTE_COMMIT
4. master -> slave2: VOTE_REQ( SHOOT(ANGLE, POWER) ) [LOG]

-slave2 dies-, master times out

5. master -> slave1: GLOBAL_ABORT [LOG]
6. slave1 -> master: ACK + SERVER ERROR
7. master -> slave2: GLOBAL_ABORT [LOG]
8. slave2 -> master: ACK + SERVER ERROR
9. master -> client: ERROR
```

(c) Now repeat the last question assuming that the same worker dies in PHASE 2. (Both before and after logging the phase 2 command) What edge case do you need to guard against to ensure proper worker behavior in this case?

In both cases the action should successfully complete (2PC guarantees that once phase 2 is reached, the action MUST complete successfully on all copies). The result is the same in both cases. If the slave dies before the COMMIT is logged, it will eventually come back, log and acknowledge the commit, and send the proper game state to the master. If the slave dies after it has already logged the commit but before an ACK is sent, however, the master will keep on poking it with duplicate COMMIT commands. The slave should know not to execute a command it has already processed as this will incorrectly change the state of the game. The system should be IDEMPOTENT.

```
IF BEFORE:
1. client -> master: SHOOT(ANGLE, POWER)
2. master -> slave1: VOTE_REQ( SHOOT(ANGLE, POWER) ) [LOG THIS]
3. slave1 -> master: VOTE_COMMIT
4. master -> slave2: VOTE_REQ( SHOOT(ANGLE, POWER) ) [LOG THIS]
5. slave2 -> master: VOTE_COMMIT
5. master -> slave1: GLOBAL_COMMIT [LOG THIS]
6. slave1 -> master: ACK + GAME_STATE
-slave2 dies, master times out -
repeat every TIMEOUT seconds until acked:
7. master -> slave2: GLOBAL_COMMIT [LOG THIS ONLY ONCE]
...
...
```

```
8. slave2 -> master: ACK + GAME_STATE
9. master -> client: GAME_STATE

IF AFTER:
1. client -> master: SHOOT(ANGLE, POWER)
2. master -> slave1: VOTE_REQ( SHOOT(ANGLE, POWER) ) [LOG THIS]
3. slave1 -> master: VOTE_COMMIT
4. master -> slave2: VOTE_REQ( SHOOT(ANGLE, POWER) ) [LOG THIS]
5. slave2 -> master: VOTE_COMMIT
5. master -> slave1: GLOBAL_COMMIT [LOG THIS]
6. slave1 -> master: ACK + GAME_STATE
7. master -> slave2: GLOBAL_COMMIT [LOG THIS ONLY ONCE]
-slave2 dies, master times out -
repeat every TIMEOUT seconds until acked:
7. master -> slave2: GLOBAL_COMMIT [IGNORE THESE
...
...
8. slave2 -> master: ACK + GAME_STATE
9. master -> client: GAME_STATE
```