Atomic commit implementation for Git repository references

STEPAN VALIAVSKII and KIRILL CHERNYAVSKIY

To achieve strong consistency in DeGitX we need to solve a problem of atomic commit of Git references over N git repository replicas. It isn't always possible to undo changes in git, so we need to manage git transactions and provide abortion mechanism. Fortunately, some atomic commit algorithms exists, so we just need to adapt them to git references update. It is two-phase commit protocol (2PC) [1], three-phase commit protocol (3PC) [2] and Paxos-commit [3]. Theoretically, each of them could solve our problem. We have investigated how they could be implemented with git. On the git side, we can use git reference-transaction hook to handle prepare and commit states.

1 PROBLEM

DeGitX keeps N copies of every repository on different servers. It makes repository more fault-tolerant and increases availability: even in the extreme case that not all copies of a repository become unavailable simultaneously, the repository should remain readable, i.e., fetches, clones, and most of the web UI continues to work.

Each server node keeps many repositories; the repository set is distributed on different nodes, so we don't know ahead where repository replica will be located. This is why we implement replication at the application layer, rather than at the disk layer. When the replicas are N loosely-coupled Git repositories kept in sync via Git protocols, rather than identical disk images full of repositories, it gives us great flexibility to decide where to store the replicas of a repository and which replica to use for reading operations.

To split read traffic over replicas and to remain repository readable even if some of the copies become unavailable, we need to ensure that every repository replica serves same git content. Git defines three main types of objects: "blob objects" to keep git repository content (such as files, commits, etc.), "reference objects" to keep metadata of repository (such as current HEAD, branches, etc.), and "tree object" to store repository history as a tree of references. Git uses content-addressable data storage [4]. It means that data update in such kind of storage can't modify old data, only append new data, so we can safely upload blob objects asynchronously to all replicas without conflicts. It is reasonable, since blob objects could be big comparing to other git object types. The only problem we need to solve to achieve strong consistency during replication is synchronization of git reference objects updates. Some reference's updates may not have conflicts if these references are located on different tree branches, for instance, if we are pushing commits to different branches. However, a single tree reference update must cause a conflict and should be synchronized.

Git helps built-in instruments to handle such updates — hooks. We can create a hook, and git will call it on some event; using this hook, we may control some internal git processes. For example, we may create a reference-transaction hook called by any Git command that performs reference updates. This hook also allows us to implement local transaction by providing three different states of transaction: prepared, committed and aborted. The state is passed as an argument to hook executable. Each reference update command invokes this hooks starting with the prepared state when all reference updates have been queued to the transaction and references were locked on disk. Two other states could be passed when changes are committed or aborted.

As in any atomic commit protocol we have two roles of nodes:

(1) Resource Manager (RM) — a node which store git repository and can apply updates received from coordinator
(2) Transaction Manager (TM) — a coordinator of the transaction which initiates transaction and can commit it or abort.

Therefore, to do an atomic commit on distributed replicas, perform the following steps:

(1) The TM sends git pack[1] to all RMs.
(2) Each RM receives Git pack and tries to apply it.
(3) Each RM locks all reference objects that are going to be changed.
(4) Each RM verifies if reference update if Git pack could be accepted using git (prepare for commit).
(5) If all RMs are ready to commit (each RM in the prepared state), then commit, otherwise, abort the commit.

2 RELATED WORKS

2.1 GitHub Spokes

Spokes on github.blog.

Spokes uses the 3PC protocol to update the replicas. All in all, this costs four round-trips to the distant replicas; expensive, but not prohibitive. (Spokes has plans to reduce the number of round trips through the use of a more advanced consensus algorithm.)

As much as possible, Spokes also makes use of the time spent waiting on the network to get other work done. For example, while one replica is acquiring its lock, another replica might be computing a checksum[2] (To check that replicas are in sync, after every update Spoke computes checksum for every replica over the list of all of its references and their values, plus a few other things).

2.2 Gitaly Cluster

Gitaly HA — not yet ready.

[1]Git pack is a compressed blobs and references update
[2]https://github.blog/2017-10-13-stretching-spokes/#using-checksums-to-compare-replicas

Gitaly Cluster allows Git repositories to be replicated on multiple warm Gitaly nodes. This improves fault tolerance by removing single points of failure. Reference transactions, introduced in GitLab 13.3, causes changes to be broadcast to all replicas. If all the replica nodes dissented, only one copy of the change would be persisted to disk, creating a single point of failure until asynchronous replication completed. To avoid it Gitaly introduced quorum-based voting.

Quorum-based voting improves fault tolerance by requiring a majority of nodes to agree before persisting changes to disk. When the feature flag is enabled, writes must succeed on multiple nodes. Dissenting nodes are automatically brought in sync by asynchronous replication from the nodes that formed the quorum.

Voting protocol will start as soon as a first"TX" message is received on the Praefect(coordinator) node. Each of the pre-receive hooks will block until it receives a message from Praefect telling to to either go on with the update or to abort. In case the vote was successful, the hook will exit with 0 to indicate success, otherwise it will return an error code and thus abort the reference update[3]. Voting strategy decides whether the vote was successful. Default voting strategy is quorum based and requires primary node to be a part of quorum. Instead of requiring all nodes to agree, only the primary and half of the secondaries need to agree. It doesn't ensure strong consistency. Dissenting nodes are automatically brought in sync by asynchronous replication from the nodes that formed the quorum. This strategy is enabled by default since GitLab 13.4

Strong consistency is currently in alpha and not enabled by default. If enabled, transactions are only available for a subset of RPCs.

Actually, if some nodes are synced in background after update, then it's not a strong consistency, but eventual consistency. Atomic commit protocol ensures that all resource-managers are agree on transaction before committing it. Gitaly still don't want to be blocked if only one replica is not available[4].

However, it's possible to change voting strategy to "all nodes need to agree".

2.3 Conclusion

A 2PC protocol cannot dependably recover from a failure of both the coordinator and a cohort member during the Commit phase. If only the coordinator had failed, and no cohort members had received a commit message, we could safely be inferred that no commit had happened. If, however,both the coordinator and a cohort member failed, it is possible that the failed cohort member was the first to be notified, and had actually done the commit. Even if a new coordinator is selected, it cannot confidently proceed with the operation until it has received an agreement from all cohort members. Hence, it must block until all cohort members respond.

The 3PC commit protocol eliminates this problem by introducing the Prepared to commit state. If the coordinator fails before sending pre-commit messages, the cohort will unanimously agree that the operation was aborted. The coordinator will not send out a do-commit message

[3]https://gitlab.com/gitlab-org/gitaly/-/issues/2635
[4]https://gitlab.com/gitlab-org/gitaly/-/merge_requests/2476

until all cohort members have ACKed that they are Prepared to commit. This eliminates the possibility that any cohort member actually completed the transaction before all cohort members were aware of the decision to do so (an ambiguity that necessitated indefinite blocking in the two-phase commit protocol). Both GitHub Spokes and Gitaly Cluster have chosen 3PC for some reason.

3PC selects new TM if the first fails. However, if a cohort receives messages from two different processes, both claiming to be the current TM, it could lead to an inconsistent state. In contrast, two cohorts could accept different decisions from different processes. Guaranteeing that this situation cannot arise is a problem that is as difficult as implementing a transaction commit protocol.

3PC avoids blocking problem of 2PC if TM or cohort fails, but the partitioning of the network still may lead to blocking or inconsistency.

Both algorithms, Two-phase and Three-phase commit assume a network with bounded delay and nodes with bounded response times; In most practical systems with unbounded network delay and process pauses, it cannot guarantee atomicity[5]. Therefore they cannot work in systems with asynchronous messaging model.

3 SOLUTION

The git itself has an internal transaction mechanism and hooks to manage it, the workflow for git transaction is:

(1) Git receives a pack to apply
(2) Git begins a new transaction and locks references for new update by writing to a lock-file
(3) Git performs checks that the pack could be applied by validatin it and comparing locked references — new reference-update should fit well into git tree
(4) In case if validation passed successfully, git calls reference-transaction hook with `prepared` message and passes references hash-sums as parameters; if the pack could not be applied, then git calls this hook with `aborted` message.
(5) If hook exited successfully after prepare (with `0` exit code), then git commit the transaction, calls reference transaction hook with `commited` message unlock references by removing lock-file; in other case, if the hook decided to exit with error (using non-zero exit code), then git aborts the transaction, removes prepared changes, calls hook with `aborted` message and unlock the reference.

This workflow allows to control the transaction flow by introducing transaction manager (TM): we can handle `prepared` ad `aborted` states and send it to the TM, then wait for response with a decision, if the TM decides to commit the prepared transaction, the we continue prepared transaction hook and exit it with zero status, otherwise, if the TM decides to abort the transaction, we exit from prepared state with error code making the local git transaction aborted.

[5]https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5353152&tag=1

This scenario fits into districuted atomic commit protocol, where we run git transaction hook on each replica back-ends and collect votes (prepared or aborted messages) on the front-end: the front-end node acts as a transaction manager (TM) in this scheme and back-end replicas as a resource managers (RM). The workflow of the distributed transaction will be:

(1) The front-end sends a git-pack to all back-end storages
(2) The git repository on each back-end node calls reference-transaction hook, sends the vote to the RM of the node, and waits until a signal from the RM
(3) The RM module on each back-end node broadcasts the vote from the hook to the TM
(4) If all RMs are prepared (TM receives prepared votes from each RM), then the TM decides to commit, and sends commit message to each RM. Otherwise, if any of the RM aborted (TM receives at least one aborted vote), then it decides to abort the transaction and sends abort message to each RM
(5) Each RM receives the decision from TM. If the decision is `commit`, then RM signals to reference-transaction hook to continue; if the decision is `abort`, then the RM signals to transaction hook to exit with error code.
(6) If hook continued with zero-code, then git applies the transaction and makes it visible to git repository; If it exited with error code, then git revert this transaction. In any case, git unlock the reference

NOTE: git repository may contain other git hooks, such as `pre-receive`, `post-receive`, etc. These hooks may contains some code which should be performed only once, e.g. sending a email. It means that only one replica node should execute this hook, so the front-end node should send a flag to one random node to make it hooks-executable. The hooks-executable node runs all git hooks in proper order, these hooks may affect reference transaction status, e.g. if `pre-receive` hook fails, then the transaction will be aborted, so hooks status affects the whole transaction, since if hooks-executable node fails, then the TM decides to abort.

To make the system fault-tolerant, we need to have secondary TMs and distributed voting system for RMs. Paxos-commit [**?**] solves this problem by introducing Paxos-simple consensus algorithm for RMs voting: each RM has a "proposer" module to broadcast the vote to "acceptors", making the vote available after the crash of RM or TM. For optimization reasons, the "acceptor"s could be deployed to the same back-end nodes as RMs.

The successfully transaction flow could be described in 14 steps, see 1 and 2 for visualisation. These diagrams includes one actor (e.g. Gitlab workhorse or shell) — somebody who's sending git pack to the DeGitX front-end; two deployments: DeGitX front-end and back-end nodes, the transaction may be performed on different deployment's scale, but it's recommended to have one primary front-end for actor communication, one secondary to perform transaction management in case of primary failure, and three back-ends for resource management. The front-end deployment consist of two primary modules:

LB Load balancer redirects the traffic for git pack uploading to back-end nodes' endpoints, chooses hooks executable node.

TM Transaction manager resonsible for managing the transaction state, collect votes from resource managers and make decision when to commit or abort the transaction.

Back-end deployment has five primary components related to transaction handling:

git git processor, it could be separate git executable process or gitlib library.

hook git reference transaction hook trigered at reference transaction state changes, responsible for communication with RM and waiting for signals from RM to continue or abort the transaction.

RM Resource manager receives transaction notification updates from hook, uses Paxos-commit protocol to broadcast git transaction changes to acceptors, begin the transaction using TM, handle transaction commands from TM to commit or abort the transaction, notify TM on finish.

Proposer part of Paxos-commit protocol, propose RM votes to all acceptors in transaction scope, broadcasts the vote to acceptors on other nodes. Each RM has it's own Paxos-instance embedded into transaction scope. Proposer on each back-end node propagates votes only for associated RM.

Acceptor each back-end node manages a set of acceptors for each Paxos instance in scope of current transaction. E.g. if a transaction has 3 Paxos instances (3 RM with associated proposers), then each back-end node has 3 acceptors for each instance.

The flow steps are:

(1) Actor sends git pack to the front-end, the request is handled by load balancer.
(2) Load balancer gets back-end nodes for git repository in the request, chooses hooks executable node, and sends the request to each node. The front-end attaches the metadata related to the transaction scope, such as acceptor addresses and secondary TM nodes.
(3) Back end node receives git pack and pass it to git component to process it. Git applies changes and starts a new transaction, if references could be updated, then git changing the transaction state into `prepared`, locks references using lock-file, and calls transaction-hook with `prepared` message and references to be updated.
(4) Hook uses reference hash-sums as transaction ID, send it to RM as a vote, and blocks git execution until the signal from the RM.
(5) RM uses the vote from hook to broadcast it to all Paxos instances in this transaction. It asks a proposer to distribute a vote value to all acceptors.
(6) Proposer connects to all acceptors and perform Paxos-simple flow starting with 2A message to accept a vote as value (first stage is not needed at the beginning, since we always have the only proposer for Paxos instance).
(7) When the value successfully proposed, the proposer notifies the RM.
(8) RM sync votes state with the acceptors for each Paxos instance located at the same node, and gets the voting table for transaction voting (with RM id as rows and acceptor id as columns).
(9) RM begins a transaction on TM, since it can't be sure that transaction for this particular transaction ID was started alreay — TM doesn't know the transaction ID prior to first

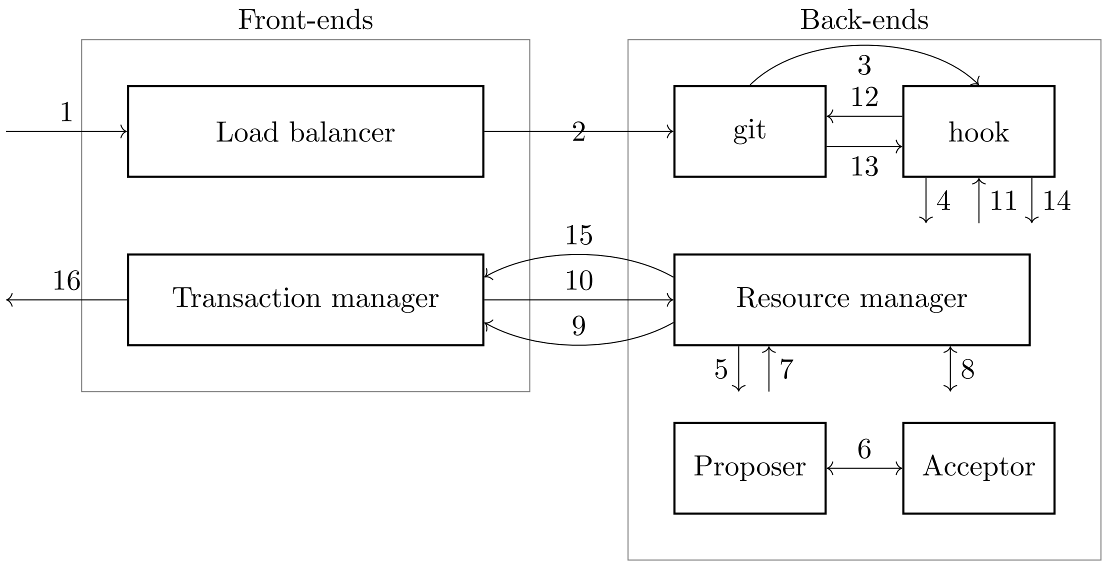

Fig. 1

`begin` call from RM. RM attaches voting table into the begin-transaction message to update transaction state on TM. If the front attaches a secondary TM node, then RM duplicates this message to each secondary TM.

(10) TM receives a begin call. If it's a first message received for this transaction ID, it starts a transaction and saves voting table received from RM as initial transaction state. It waits for other RMs to send the begin call, unti the table will be full enough: if TM has a quorum of "prepared" votes for each RM, then TM sends a "commit" message to each RM; If TM finds in table that at least one RM has a quorum of "abort" votes, then it sends an "abort" message to each RM.

(11) RM receives the TM decision: either "commit" or "abort" and notifies git reference-transaction hook with "continue" or "error" signals repspectively.

(12) If hook receives "continue" signal from RM, it continue execution and exit from prepare state with success zero status, then git commit the reference-transaction; in case of "error" signal, hook exits with error code and git aborts the transaction.

(13) Git commits or abort the reference transaction locally and call the hook on complete with `commited` or `aborted` repspectively.

(14) Hook notifies RM that transaction finished and immediately exit

(15) RM notifies TM that the transaction was finished.

(16) TM waits for all RM to finish transaction, and sends the response to the actor with success or error code based on final transaction decision (commited or aborted).

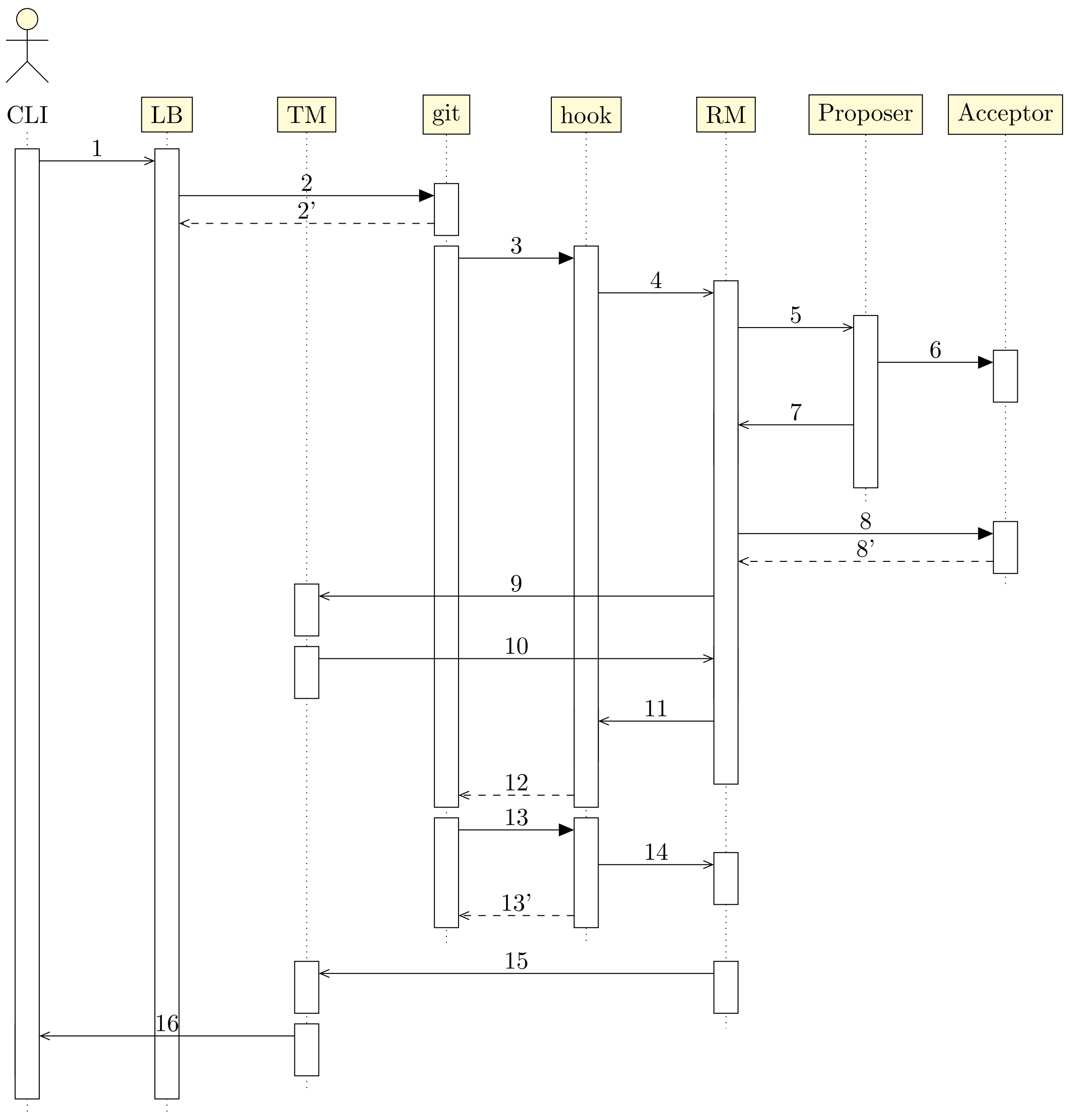

Fig. 2. Transaction commit sequence diagram

4 FUTURE WORK

(1) Implement a prototype and do benchmarking.
(2) Create formal TLA+ specification for this algorithm.
(3) Design and implement transactions.

(4) Research for Paxos performance optimizations.

5 CONCLUSION

Now DeGitX team has a vision how to achieve strong consistency.

To simplify, we decided to use Paxos-commit with Git reference-transaction hooks. It solves atomic commit problems and doesn't hurt performance due to asynchronous blobs update and simultaneous reference objects updates on different tree paths. It has hood fault-tolerance, works in the partially synchronous system model, handles non-byzantine node failures. It satisfies all atomic-commit requirements:

(1) Stability - once an RM has entered the committed or aborted state, it remains in that state forever.
(2) Consistency - it is impossible for one RM to be in the committed state and another to be in the aborted state.
(3) Non-triviality - if the entire network is non-faulty throughout the execution of the protocol, then (a) if all RMs reach the prepared state, then all RMs eventually reach the committed state, and (b) if some RM reaches the aborted state, then all RMs eventually reach the aborted state.
(4) Non-blocking - if, at any time, a sufficiently large network of nodes is non-faulty for long enough, then every RM executed on those nodes will eventually reach either the committed or aborted state.

REFERENCES

[1] J. Gray, "Notes on data base operating systems," 1978.
[2] D. Skeen, "A quorum-based commit protocol," 1982.
[3] J. Gray and L. Lamport, "Consensus on transaction commit," *ACM Trans. Database Syst.*, vol. 31, p. 133–160, Mar. 2006.
[4] O. G. Arthur Baars, "Content-addressable data storage," November 2017.