[DRAFT] White-paper: DeGit - distributed git repository manager

KIRILL CHERNYAVSKIY

DeGitX is a distributed git repository manager. It provides a front-end interface for git operations by exposing one of the supported endpoints for git clients, and hides the distributed nature of git storage located on back-end nodes. The back-end keeps git repository replicas simultaneously on multiple different nodes to scale up the read capacity, increase durability and provides better availability, especially for different geographically distributed regions.

1 INTRODUCTION

Providing efficient access to code repositories is a significant challenge that large software companies or open-source communities may face when the number of code repositories used by software developers counts on millions. This challenge becomes especially critical for companies that have distributed teams around the world. In this case, using a single git server does not provide even nearly appropriate performance. On the other hand, deploying multiple servers with load balancing is not feasible because storage I/O scaling is impossible for reading operations.

This article describes how to achieve significant performance improvement by configuring a distributed git repository storage that replicates repositories across nodes. The result can be achieved by implementing a distributed Git repository manager DeGit with strong-consistent replica nodes, guaranteeing high availability for reading operations, as well as availability across different regions. Such a solution also provides durability by replicating repositories on different server racks and read scalability by routing fetch traffic to different nodes. There is a number of technical problems related to scalability, dispatching, and consistency, which need to be solved when implementing the distributed Git repository manager with strong-consistent replica nodes. In addition to describing the suggested solution's major work principles and functionality, this article also addresses these issues. The DeGit architecture and components, including the main system protocols, are also explained. These enable the system to identify the node, establish the node's connection, lookup at the node's real address, and map a git repository to the storage node. The potential presence of malicious nodes can misroute, corrupt, or drop messages and routing information in the system. This article reviews the security-related methods with which the adopted DeGit approach uses to protect the system infrastructure from potential attacks. In this paper, you can find detailed instructions on creating a new repository. The article also explains how a node finds a replica with a git repository and provides descriptions of the node discovery workflow and metadata exchange components. In this solution, the git communicates with the data storage via a front-end. The document describes communication between the git and the front-end and explains how data is passed between the front-end and the repository. The functional and non-functional requirements for building a distributed git repository storage and the metrics expected for large enterprises are addressed further in the document. Readers can also find helpful information regarding the comparison of DeGit with other solutions.

2 TECHNICAL PROBLEMS

There are three main problems to solve for the distributed git repository manager (DGRM):

scalability — one repository may be used actively by many users simultaneously. Storage disk has input-output (IO) operation limits and cannot perform a lot of `fetch` operations in a short period of time. The git repository storage should be replicated on different system nodes to solve this problem. TODO: REF find references about storage scaling to improve read capacity in a distributed system.

dispatching — the user does not know the network addresses of git repository storages. Also, according to the previous problem, one repository can be located on multiple replicas. Hence, the system should be able to locate and redirect the user's requests from git client[1] to correct git repository storage. It should lookup repository nodes in the system and load-balance requests to different storage replicas.

consistency — the user may `push` to the repository and `fetch` then; in that case it must receive the same or newer data that user `push`ed previously, even if the user `fetch`ed data from another replica of this repository. The linearizability of every single repository in a system is a must-have option.

2.1 Scalability

Big software companies or large developer communities have millions of repositories distributed worldwide, and big teams located in different countries (regions). On a high load of `fetch` requests, git repository storage disks may go above IO operation limits. Usually, repositories are accessed in different ways:

Developer personal activity — programmers access git repositories via `git` client and performs `fetch`/`push` actions. They may also use web UI interface to upload files, edit files in a browser, merge the pull requests, etc. The number of requests for this activity is not very large, according to statistics.

API access — many services depend on repository management services: API robots collecting developers' activity, background code analyzers of a repository, IDEs communicating with repositories and reading some historical data. The number of such requests is approximately a few thousand requests per minute for every million repositories.

CI systems — many events may trigger a CI workflow; most common events are: new commit pushed, new pull (merge) request created, new tag pushed, etc. On each action, the CI system downloads the whole repository to run some workflows, the download (`clone`) operation it typically accompanied by huge bandwidth consumption. In addition,

[1]Git client is a software that performs git fetch and push operations; it can be a command-line tool, IDE or any graphical user interface communicating with DGRM

some repositories may include submodules; this leads to submodules cloning by the CI system. According to statistics, the number of such operations is on the scale of ten thousand requests per minute for every million repositories.

Even in cases when git repository storage is distributed, and clients are routed to storage with the correct git repository, a large amount of `fetch` traffic for one repository still able to make disk go above input-output operations (IOP) limits, and repository storage will stop serving requests.

Asynchronous replication of git repository data was one approach attempting to address such scalability problems; the DGRM was a proof-of-concept, which aimed to eventually consistently guarantee that one repository was replicated asynchronously after any update operation to multiple git repository storages. Therefore, clients were load balanced to different git repository storages. TODO: REF find references about comparing eventual consistency with strong consistency in the context of bias between Read and Write operations in distributed storages. However, it did not succeed because of two reasons:

(1) fetch traffic was correlated with push frequency because of the huge amount of CI systems and API robots involved in the development process: each update event usually triggers a CI build immediately, and CI clones the repository. CI was able to clone only the primary repository (repository with actual data) because the replication has not been completed before CI cloning started.
(2) `push` and `fetch` frequency was not distributed uniformly over time — in each repository team members may have different responsibilities for the review and merge process, while the project technical lead can merge all approved pull-requests in a short period of time which causes frequent push operations in a git repo.

These two conditions lead to high fetch traffic peaks for git repositories: Frequent push operations turn replication storages into an inconsistent state, and lead to high fetch traffic to primary repository storages from CI systems, which clones these repositories. This causes the same problems as were common for non-replicated system — the primary node goes beyond IOPS limits and rejects new `fetch` requests.

Furthermore, `merge` is not the only way to "update" the repository. Many of other activities can do that as well, such as:

branches changing `create/delete/update` branches by git push or by using web page.
tags changing `create/delete/update` tags by git push or by releasing new version on web page.
special refs changing when new merge request is created, a new *`refs/merge-requests/IID`*[2]*`/head`* named ref is created. When source branch of the merge request is updated, the ref is also updated.

[2]Internal id of a project on gitlab

migrations sometimes repository administrator can migrate the repository to another physical device or another region node; it also could be treated as an update. Also, deleting a repository could be treated as migrating it to a trash area.

2.2 Dispatching

Git client does not know where a repository replica is located in the system. The dispatching algorithm should be able to locate the correct storage nodes (distributed system nodes) of the repository and redirect each client's request to one of these nodes. Moreover, it is not enough just to redirect the client's request. The dispatching endpoint should load-balance all requests for each particular repository to distribute storage load over time. Round-robin load-balancing solves this problem since the scalability problems occur only on peaks of high read traffic. Hence, statistically redirecting each next read request to a different storage node reduces disk load by `n`-times, where `n` is the number of storage replicas.

2.3 Consistency

TODO: Describe the problem

3 SOLUTION

The proposed solution is a distributed Git repository manager with strong consistent replica nodes. It consists of two parts: the back-end and front-end (see Figure 1 diagram):

The back end — (core network, storage), P2P[3] system which stores git data and metadata[4]. It is responsible for replication of repositories and guarantees strong consistency of git storage across replicas. It exposes internal RPC API to accept intermediate git requests, which can modify the repository state and provide endpoints to read (`fetch`) repository data.

The front end — (multiplexer, load balancer). It exposes public API for all git operations, translates all operations request into intermediate RPC language, routes requests to proper storage nodes and load-balancing read (`fetch`) requests to different replicas of the same repository.

The front end exposes public API for well-known protocol, e.g. one front-end implementation provides Gitaly gRPC interfaces for GitLab instance set[5] (see Figure 2 diagram). GitLab is a client of DeGitX front-end; it sends gRPC requests to the front-end to modify and fetch git data.

[3]peer to peer

[4]Here git data referenced to git objects, and git metadata to git references. In this document, git data usually refer to both objects and references unless otherwise stated

[5]In this document GitLab instance set is a set of Shell, Workhorse and web components of GitLab

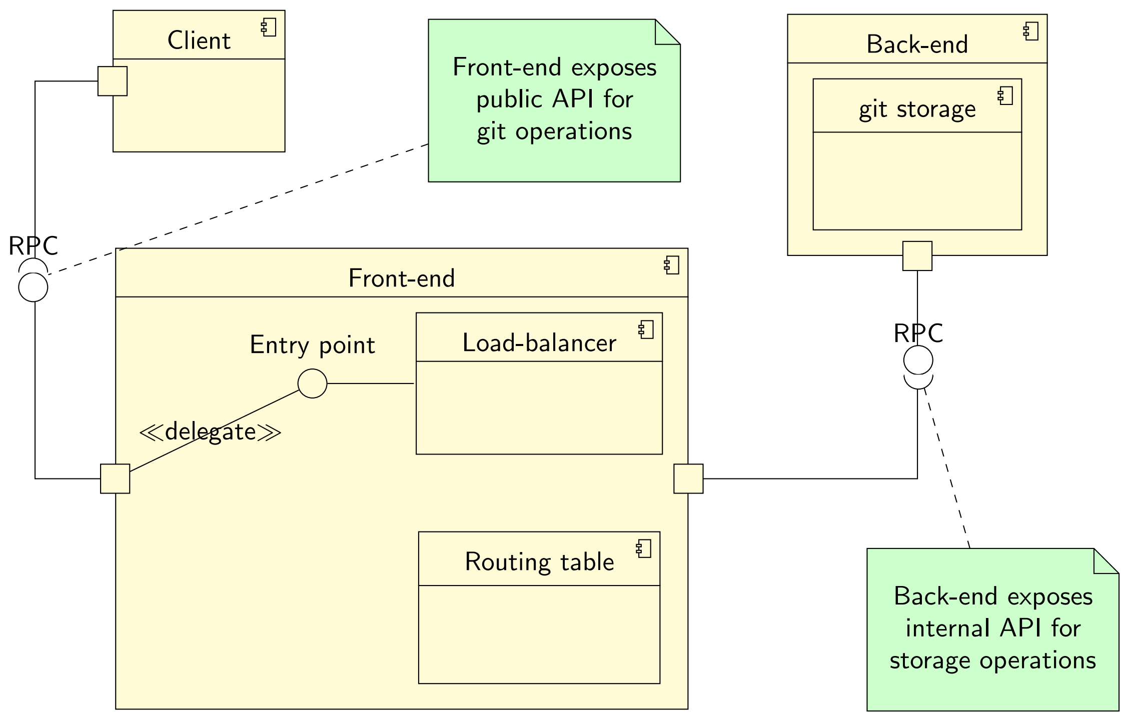

Client
Front-end exposes public API for git operations
Back-end
git storage
RPC
Front-end
Entry point
Load-balancer
≪delegate≫
RPC
Routing table
Back-end exposes internal API for storage operations

Fig. 1. Components architecture overview: Each client C connected to DeGit front-end component, the front-end has load-balancer to proxy incoming requests to back-en. It finds back-end nodes using the routing table.

DeGit metadata is the communication layer of the DeGit front end and back end. It consists of a repository hash to the storage node locator. The front-end can be configured differently depending on its setup; it can query metadata from the database; it can support lookup queries via DHT[6] or it can receive metadata updates broadcast from storage node peers in local network via UDP protocol, see 3.2.

When a client (e.g., GitLab-shell) writes git data to the system (via `push`), the request is routed by the front-end load balancer to one of the storage replicas. The storage back-end node starts leader election with other repository replicas and updates the log[7] of repository holders (replicas) consensus, see 3.4.

The system automatically rebalances repository storage: when the repository is not actively used for a long time, the node can remove it from storage, as long as 3 replicas of this repository exist on other nodes. If some node has a lot of free storage space and another node's storage

[6]Distributed hash table

[7]Here: the node log or back-end log is a distributed replicated log of a state machine associated with specific git repository storage

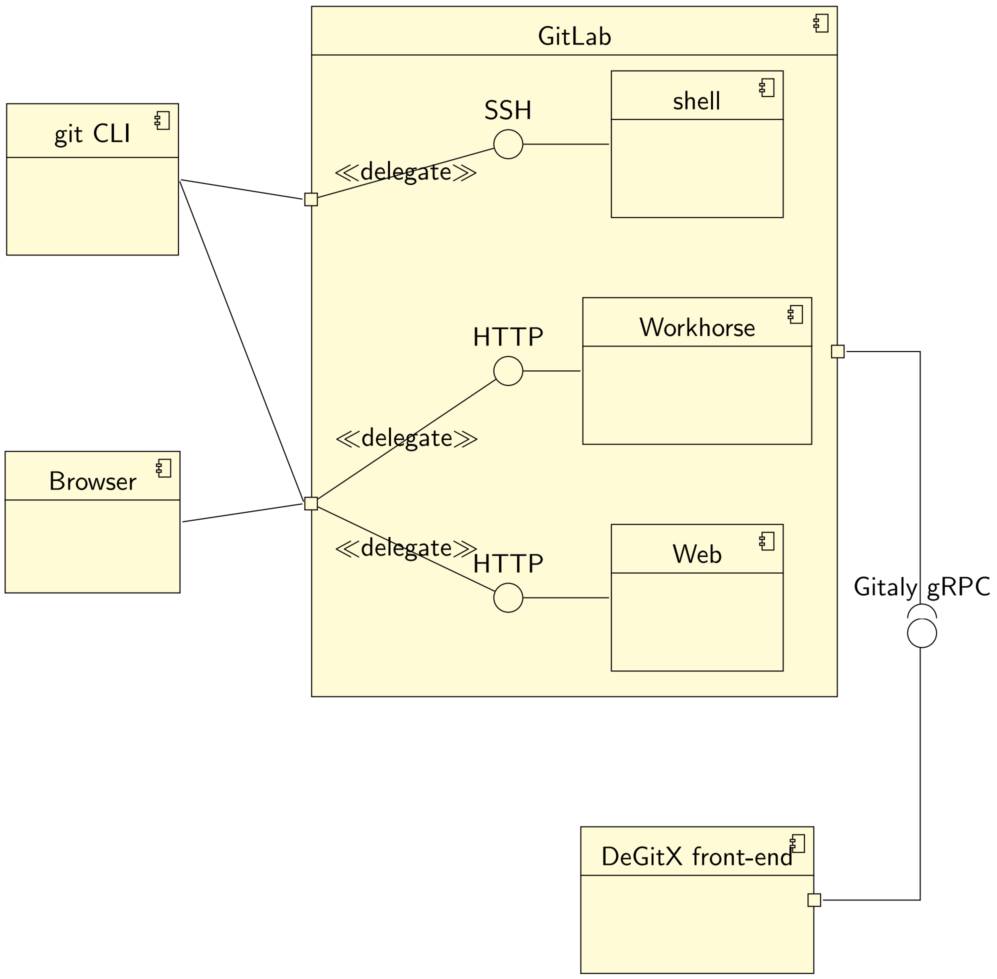


Fig. 2. GitLab set and Gitaly front-end: git client communicates with GitLab-shell via SSH or with GitLab-workhorse via HTTP(S). The browser uses GitLab-web. All three GitLab components communicate with git storage via DeGitX load balancer (front-end) via gRPC protocol defined by Gitaly component of GitLab.

is almost full, the full node can transfer (move) some repositories to another node. IOPS of a storage device is also a measure of importance.

The system can accept new nodes, automatically fill them with a replica repository and move some repository to a new node. The same is true for disconnecting: if a node was disconnected from the system or has crashed, it creates additional replicas on nodes to have at least 3 replicas for each repository.

3.1 Protocol

Main system protocols are:

Location protocol — unique node identity.
Network protocols — nodes communication protocols.
Discovery protocol — lookup of a node real address by locator ID.
Metadata exchange protocol — mapping of git repository to storage back-end node.
Data exchange protocol — Git objects and references exchange protocol, commands to add new objects and update references, linearizability guarantee.

3.1.1 Locators. Network addresses are not stable; a back-end node may get its network address via Dynamic Host Configuration Protocol (DHTP), or node owner, may move it from one server to another. As a result, we cannot rely on network addresses when working with back-end nodes. Instead, we need overlay networks and unique identifiers for each node. To identify a node, DeGit uses public-key cryptography: the node owner generates a private and public key pair using one of the supported crypto algorithms TODO: which exactly?. These keys will uniquely identify a back-end node. A cryptographic hash TODO: choose algorithm of the public key is used as a node locator ID, as described in TODO: REF add cite to "A Practicable Approach Towards Secure Key-Based Routing". Nodes use locator IDs to introduce themself to the system and build an overlay network on top of the real network. DeGit uses Multihash[8] format to encode locator IDs. TODO: REF cite IPFS research https://raw.githubusercontent.com/ipfs-inactive/papers/master/ipfs-cap2pfs/ipfs-p2p-file-system.pdf since they use a similar approach for introducing node IDs in the P2P system.

Private and public keys can be used to sign requests to other nodes or to build a trusted discovery (see 3.1.3) point in a system: when a new system is created, the administrator can create certification authority (CA) for issuing digital certificates for node public keys, so each node will be able to verify the certificate of any other node when using seed list URLs or other peers exchange algorithms. Node instances use locators to talk with each other. The real network address of nodes can be found using this locator ID. Also, the system uses locators for node-repository mapping (see 3.4), where an item of mapping table consists of a locator ID and a unique repository name hash.

3.1.2 Network. DeGitX uses TCP or UDP protocols for transport layer; TCP is used by default. It uses IPv4 or IPv6 for addressing, storing addresses in multiaddr format, for instance: `/ipv4/1.2.3.4/udp/4444` evaluates to `4444` UDP port on `1.2.3.4` IPv4 address. DeGitX uses a routing system (see 3.1.3) to find addresses of nodes via locators.

3.1.3 Discovery. By default, a node does not know the network addresses of other nodes, only their locator IDs. Various discovery techniques are used for node lookup. They can be configured independently by node administrator or used together:

[8]https://multiformats.io/multihash/

LPD — local peer discovery: each peer sends UDP messages containing the locator ID. This technique is cheap and fast for the network layer (due to UDP messages). Other system components, such as front-ends and back-ends receive these messages and update a local cache of node locators. This approach dramatically improves lookup performance in local networks, e.g. when most of the communications are taking place in one local region, and all regional nodes are connected via a local network. This protocol is supposed to be used with other discovery techniques. For security reasons, in untrusted networks, the broadcast messages may be optionally signed with node private keys to be verified by the receiver using the CA public certificate. For optimization reasons, this protocol is reused by a metadata exchange protocol for repository hash table propagation (see 3.2). TODO: lookup for researches. There are two common ways to implement LDP: bep 14 and bep 26. TODO: REF use cites to bittorent and zeroconf researches instead of URL links here. Bep 14 is SSDP-like style and Bep 26 is zeroconf style LDP. To implement Bep 26 each host is required to run a zeroconf service discovery daemon. There is a popular go zeroconf implementation that could be used. BitTorrent uses following multicast groups: A) `239.192.152.143:6771` (org-local) and B) `[ff15::efc0:988f]:6771` (site-local) for Bep 14 implementation.

- Site-Local scope is intended to span a single site.
- Organization-Local scope is intended to span multiple sites belonging to a single organization.

They've chosen such IPs because `239.192.0.0/14` is defined to be the IPv4 Organization Local Scope, and is the space from which an organization should allocate sub- ranges when defining scopes for private use. `ff15::efc0:988f` also comes from IPv6 spec and means:

- FF == Multicast
- 1 == 'Flags' — where 1 indicates a nonpermanently assigned ("transient" or "dynamically" assigned) multicast address.
- 5 == Site-Local scope
- efc0:988f — the hex representation of 239.192.152.143

We could easily implement BEP 14 as described and take this implementation as an example.

Distributed DB — network addresses could be propagated to the system using a supplementary distributed database, e.g., etcd or others. On startup, the back-end node registers itself in the database, and other system nodes (both front-ends and back-ends) use this database for lookups. It requires additional system configuration but delegates some responsibilities to third-party services. TODO: REF add cites for comparing distributed key-value database in the context of metadata storage for cross-region access.

Seed hosts — nodes can use other nodes as seed hosts for caching optimization; some nodes will be responsible for caching lookup results. In other words, these seed nodes represent a similar abstraction to Content Delivery Network (CDN) in web caching.

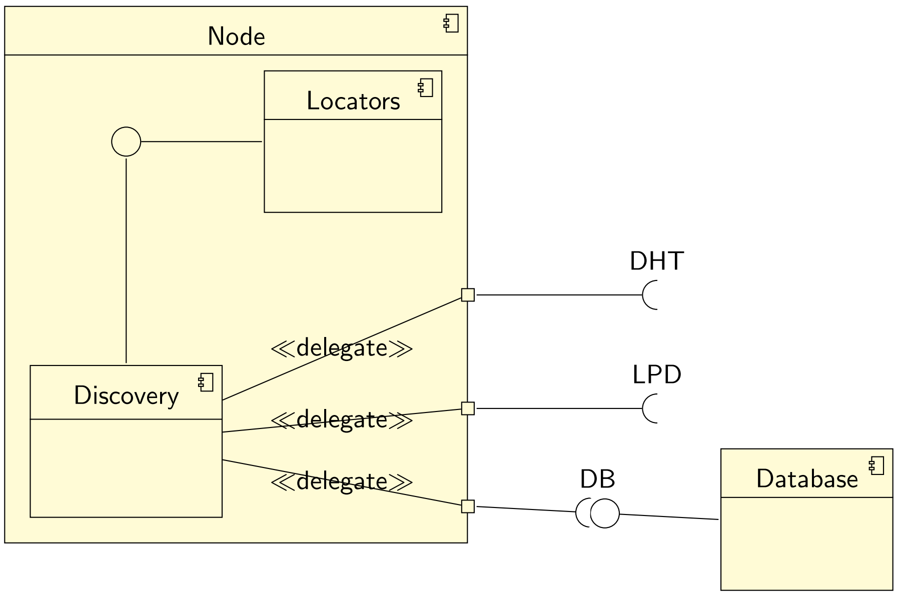


Fig. 3. Doscovery protocols depend on Locators protocol, and uses multiple discovery interfaces: LPD on local network, regional database to speedup the lookup, DHT for global lookup in all system.

DHT — distributed hash tables use different metrics to compare the distance of nodes in the overlay network. For example Kademlia[9] uses the `XOR` metric of node ID for distance measurement. It requires node IDs to be widely distributed — DeGit locators satisfy these requirements, being generated as a cryptographic hash function from the public key. In the Kademlia lookup system, each node stores references in K-buckets, where each K-bucket contains node addresses with the same ID prefix. TODO: REF add cites to Kademlia and Chord DHT papers.

Depending on the configuration, the node may or may not use some discovery protocols. They are performed in a well-specified order: first, the node performs a lookup for locator ID in the local cache; the cache is updated by LPD broadcasts; in case if it is not found, the node queries the discovery database, then seed hosts, and DHT as a last resort.

The structure of discovery entries is a mapping of locator ID (which is cryptographic hash by design) to network address in Multiaddr format (as described in network section: 3.1.2).

Example of a routing table:

Locator ID	Node address
`122041dd7b6443...0022ab11d2589a8`	`/ipv4/192.168.1.42/tcp/9031`
`122041dd7b6443...0022ab11d2589a8`	`/ipv4/192.168.1.33/tcp/8011`
`132052eb4dd19f...6f8c7d235eef5f4`	`/ipv4/172.18.11.22/tcp/9031`

[9] pdos.csail.mit.edu/ petar/papers/maymounkov-kademlia-lncs.pdf

When a new node starts, it requires discovery protocols to be configured, depends on configuration, it starts corresponding services to register itself in a system, then other nodes will be able to find a new node using this registration information. In case of distributed database, the node just puts new value with a network address for node locator ID as a key.

3.2 Metadata

DeGit peers don't know where to find repository storage by default. The system introduces metadata layer to exchange repository coordinates to locator IDs. The structure of metadata is a many-to-many relation of repository cryptographic hash to storage locator ID.

For instance, below is an example of metadata of two repositories located at two nodes: repository `repo1` is located on both nodes `node1` and `node2`, while repository `repo2` is located only on node `node2`:

Repository hash	Locator ID
`hash(repo1)`	`locator(node1)`
`hash(repo1)`	`locator(node2)`
`hash(epo2)`	`locator(node2)`

The repository hash is encoded in Multihash format. Metadata exchange protocol partially reuses discovery protocol for network optimizations. Peers send local broadcasts with locator IDs and known repository hashes, and the discovery database may keep (if configured) the repository hash map to node locator ID relations (see 4). DHT keeps locator IDs as a value for repository hash keys 5; it keeps all metadata of the whole system (???). The metadata lookup process is similar to the discovery protocol: first, the peer looks for metadata in the local cache (populated with network broadcasts); then, it checks the region database. Finally, as a last resort, it performs a query lookup for global DHT.

When a new node starts replicating some repository, it first synchronizes with other replicas, and then updates metadata asynchronously. The metadata is updated as follows:

(1) A new node wants to replicate some repository
(2) The node finds current repository holders (replicas) in existing metadata and choose a random node from this list
(3) The node sends a request to the selected node to add itself to the replica list
(4) Receiver node starts leader election and updates node log to add a new node to the replica list
(5) The consensus accepts new node and stores it in local persistent storage
(6) The leader notifies a new node that it becomes a part of the replicas and consider it when consensus is required
(7) New node replicates the state of consensus, holds the repository and sends UDP broadcast to local peers on success
(8) Leader node propagates replicas change in metadata storages asynchronously, updating DHT or database storage (as configured)

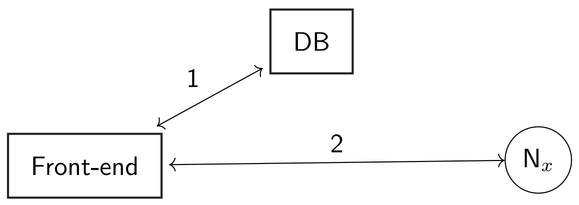


Fig. 4. Repository lookup in database: DB — Database with metadata for current region. 1 step — front-end load balancer query database for repository metadata. 2 step — front-end access node N_x with the required repository.

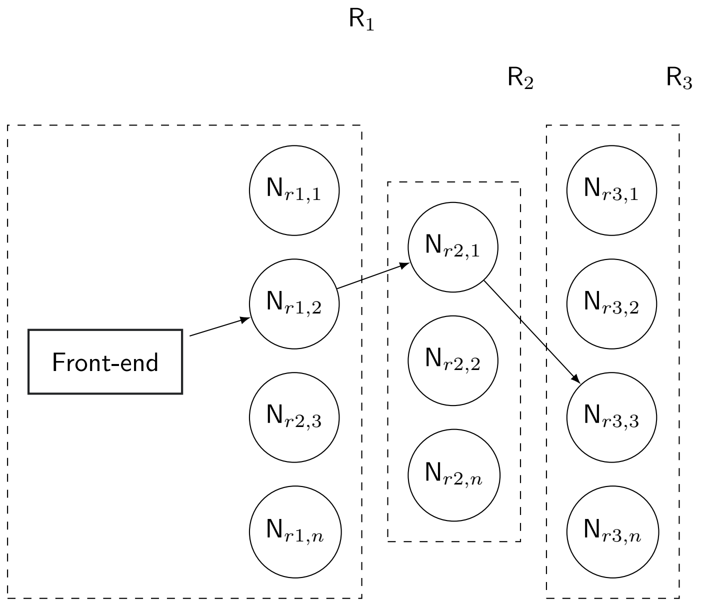


Fig. 5. Repository lookup via DHT: When a front-end in Region R_1 is unable to find repository located on node $N_{r3,3}$ in same region, it queries the nearest known node $N_{r2,1}$. The node $N_{r2,1}$ knows where repository is located and redirects the query to node $N_{r3,3}$ with repository.

Repository mapping could be stored in a distributed hash table (e.g., Kademlia), database cache, or broadcasted via local network broadcasts. Therefore, for multi-region cluster setup, the front-end load balancer may look for a node in the local cache populated by local network broadcasts, then check region database where all repositories in the same region are registered, and if not found perform DHT query lookup in different regions, see Figures 4 and 5.

3.3 Security

A small fraction of malicious nodes can prevent correct message delivery throughout the overlay.

Such nodes may misroute, corrupt, or drop messages and routing information. Additionally, they may attempt to assume other nodes' identity and corrupt or delete objects they are supposed to store on behalf of the system.

All attacks are based on the presence of malicious nodes. There are two ways:

- Implement techniques that allow nodes to join the overlay, maintain routing state, and forward messages securely in malicious nodes.
- Make the appearance of malicious nodes impossible.

Degitx is not an open peer-to-peer system where resource pooling without preexisting trusted relationships is possible.

nodes aren't allowed to join, and all network members are trusted not to cheat.

To be sure that the node is trusted, its nodeId certificates should be signed by trusted CAs. Then each node rejects all unsigned requests.

Certified nodeIds work well when nodes have fixed nodeIds. This condition is met while the node uses a cryptographic hash of public key as nodeId.

These certificates give the overlay a public key infrastructure suitable for establishing encrypted and authenticated channels between nodes. Nodes with valid nodeId certificates can join the overlay, route messages, and repeatedly leave without the involvement of the CAs.

When the membership of a peer-to-peer system is a constraint and all nodes are trusted as in DeGitX (??????), CAs could solve security issues since according to the TODO: REF insert a cite for "Secure routing for structured peer-to-peer overlay networks" paper, all attacks are based on the presence of malicious nodes.

3.4 Git data exchange

Github Spokes and Gitaly HA achieve strong consistency via *3PC protocol*. Spokes sources are closed, but they mentioned the protocol in their *blog*. So we only know they use it somehow. On the other hand, Gitaly team, which had faced the same issue, has performed open research and a couple of POC to decide how to implement 3PC protocol. After all experiments: *Implement 3PC for WriteRef RPC 3PC git-update-ref experiment 2PC via pre-receive hook*, they concluded, that the best and only way is to push transaction handling as far down as possible, which is the git client itself and, more specifically, its ref transaction handling. The only problem is that they need a place in git, where all write operations could be captured, checked and committed or safely aborted. There was no such place to cover all write operation, and new *reference transaction hook* was introduced by the Gitaly team.

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.

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 [1] 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 6 and 7 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 `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).

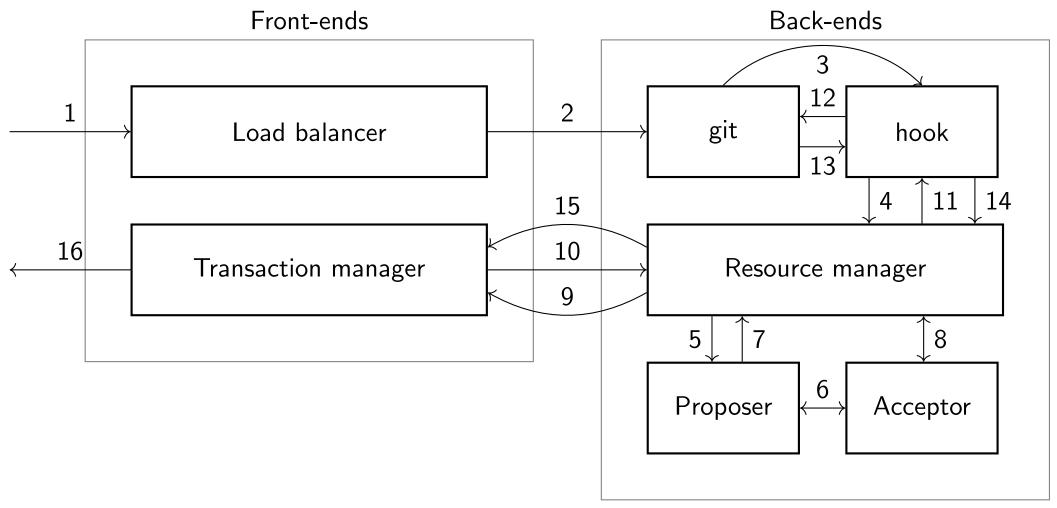

Front-ends
Back-ends
Load balancer
Transaction manager
git
hook
Resource manager
Proposer
Acceptor
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16

Fig. 6

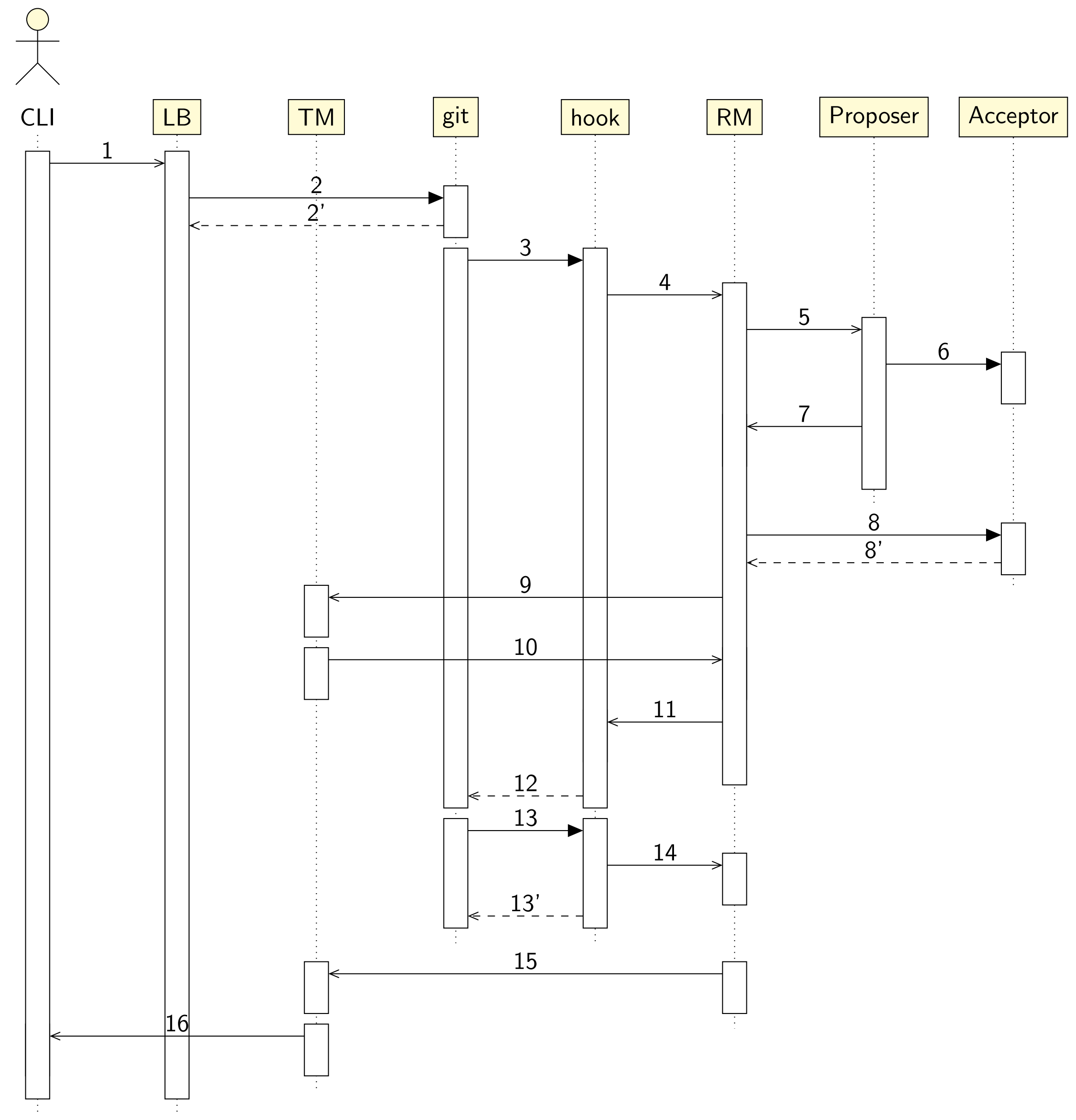


Fig. 7. Transaction commit sequence diagram

4 SPECIAL CASES

4.1 How to create new repository

The diagram 8 explains how administrator creates new repository with replication on predefined storage nodes by specifying its locator IDs:

1 Administrator sends request to the dashboard to create new repository `R` at nodes with locators `L`
2 Dashboard service performs a lookup of `R` in metadata storage
3 Metadata storage returns locator IDs `L'` of `R` repository if any
4, 5 If `L'` locators are not emmpty, dashboard returns error to administrator
6 If no node locator is associated with repository `R`, then dashboard asynchronously sends requests to each node with locator ID `l` from list `L`
7 Dashboard asks node `l` to start manage repository `R`, and specify other node locator IDs; The node `l` try to organize consensus between nodes `L`, and leader asks all other members to manage repository `R`
8 On success, node `l` updates metadata with mapping of repository `R` hash to node locator `l`
9 Meanwhile, the dashboard is performing query requests to the metadata storage to get locator IDs of repository `R`; Metadata storage returns IDs `L'`.
10 In case if `L'` is a subset of `L` and the size of `L'` is greater or equal to half size of `L` plus 1 (`size(L') >= size(L)/2 + 1`)
11 Dashboard finishes with success status
12 If waiting timeout is reached
13 Dashboard finishes with error status
14 Dashboard returns with success or error status to administrator

4.2 How the node finds the replica with git repository

Each node (both front-end and back-end nodes) has metadata exchange and discovery protocol components, both components rely on network modules and locator systems.

Stakeholders:

Node A node which performs repository lookup operation
RDB Regional database (caches repository hash table in local region specific cache table)
DHT Distributed hash table nodes, can perform global query lookup for node address by repository hash

The legend: `r` — repository hash to lookup; `(l, a)` — repository coordinates, pair of replica locator IDs and network address, see figure 9 for details.

Workflow:

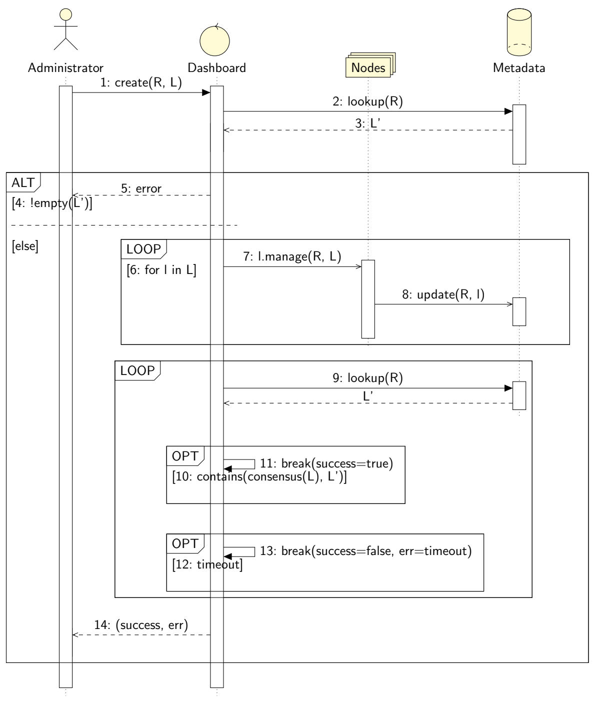


Fig. 8. Create repository workflow diagram

(1) Node checks local cache populated by local peer discovery broadcasts and updated annually after successful query operation
(2) If coordinates were not found in the cache, the node goes to the next lookup layer
(3) Node queries the regional database to find repository coordinates
(4) If coordinates were not found in the database, it goes to the next lookup layer
(5) Node queries DHT nodes using lookup algorithms, e.g., Kademlia.
(6) On success, the node updates the regional database with actual information
(7) Node stores actual information in the local cache

4.3 How GitLab pushes to DeGitX via Gitaly front-end

GitLab communicates with DeGitX storage via Gitaly front-end. The front-end exposes Gitaly gRPC API. GitLab performs push with two RPC calls:

InfoRefsReceivePack — to fetch latest repository git references before pushing
PostReceivePack — for add new git objects to the repository

At figure 10, the front-end updates repository lookup routing table to request git references from one of the replicas and send new git objects to back-end nodes.

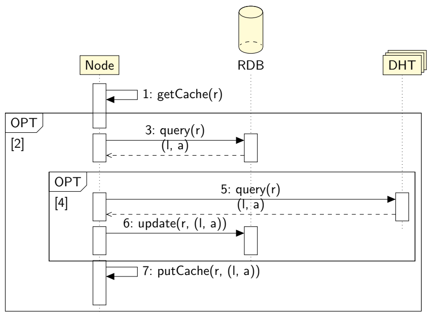


Fig. 9. Node discovery and metadata exchange components

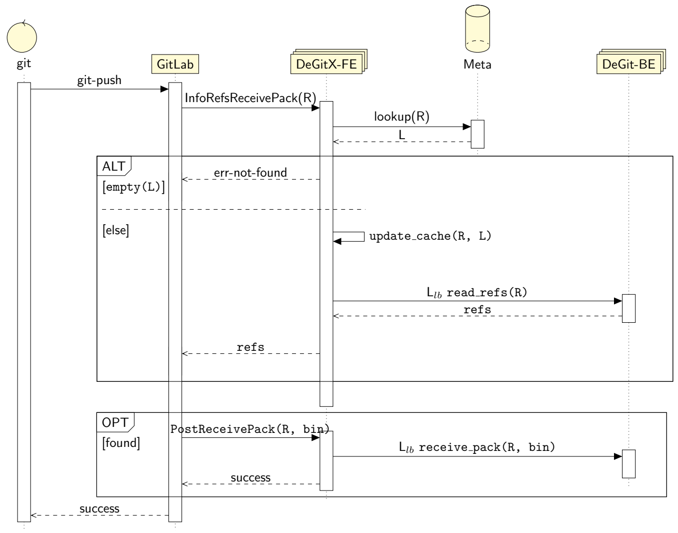


Fig. 10. Communication between GitLab and DeGitX Gitaly front-end

User pushes git data to repository `R` via Gitlab using `git-push` command; Gitlab calls `InfoRefsReceivePack` of DeGitX front-end to get latest information about repository `R`; front-end performs lookup in metadata storage, if repository was not found, front-end returns error for gRPC call; otherwise, it updates local cache with repository locators `L`, and reads git references from any load-balanced replica $\mathtt{L}_{lb}$; the front-end returns git data to GitLan server; on success, GitLab sends `PostReceivePack` gRPC call to DeGitX front-end with data to push `bin`; front-end write data via `receive_pack` git method to any replica $\mathtt{L}_{lb}$; (see other examples to understand how back-end nodes updates repository data on push).

This diagram explains only communication between GitLab and DeGitX Gitaly front-end. To understand how the front-end can read and write git data from/to back-ends, see other diagrams provided below; To become familiar with git internals see 7.

4.4 How the front-end pushes data to the storage

TODO: part of 76 ticket

4.5 How the client fetches git data from repository

4.6 How new replica node connected to the cluster

TODO: explain all questions

5 REQUIREMENTS

5.1 Features

The most critical Non-functional requirements are:

Read scalability The solution should scale-out the read capacity of a system; each region should be able to access the repository using the most available replica node.

Strong consistency All? (TODO: discuss, maybe not all but the majority of replicas) active replica repositories should be synchronized on updates in any node with immediate consistency.

Durability The system must have enough replicas to recover itself in case of corruption. The corrupted repository could be responsible for recovering itself using replica nodes.

Self-management (rename?) Each node performs cleanup when needed (`git gc`) and may remove replica from storage on reading inactivity. A node should be able to find and synchronize new repository on reading. After that, it should be up to date on new updates.

Maintainability Node administrator can change the storage, and perform data migration from one storage to another. Repository administrators can add or delete a node for the new region and get status of all nodes in the repository.

Auditability Node doesn't perform access control operations but logs all requests with identity and performed operation.

Analytics Node collects statistics for each repository and usage metrics, such as Push and Pull operations, etc. The system keeps the whole statistics about nodes, e.g. how many nodes are contained in each repository, the state of nodes, etc.

5.2 Load balancing

The front-end decides how to redirect incoming fetch requests to replica nodes. It knows network addresses and current availability of repository replicas, so it can decide how to route next fetch request to git repository back-end storage node. On read access to the repository, it sends a health check message to replica nodes to verify which are available. It stores the result in local cache with configurable time to live properties. When the front-end cannot access some of the back-end nodes, it marks it as unavailable and forgets about it for some time. Then in check availability periodically. It uses "Round robin" load balancing for fetch requests since it provides a uniformly distributed load for repository nodes on high load peaks. TODO: REF find references and proves about round-robin load-balancing hor high load network traffic and correlation with uniform distribution of storage access. Due to the strong consistency of back-end nodes provided by the data-exchange protocol, it has linearizability property and always provides the latest git data (blobs) and metadata (references). If any node crashes or becomes down, the front-end won't be able to update the repository. However, it still can read it from live replicas, which means that all repositories located on this node become read-only on node failure.

6 COMPARE TO OTHER SOLUTIONS

These products are similar to DeGit by some aspects:

Spokes GitHub announced DGit in 2016 (renamed to Spokes) where they pay attention to the consistency:

> Spokes puts the highest priority on consistency and partition tolerance. In worst-case failure scenarios, it will refuse to accept writes that it cannot commit, synchronously, to at least two replicas.

It is proprietary software that can't be used freely, and the source code is closed. Spokes papers claim that it pays attention to consistency, but on the conference talk they mentioned that it's rarely possible to break the consistency which requires manual intervention. Therefore the approach of distributed system design used by Spokes is not suitable for open source project, where the maintenance team doesn't exist.

Gitaly Gitlab has Gitaly service which provides `gRPC` API for Gitlab website and git-ssh proxy to perform all git operations via API. It's open source component. Gitaly proposed a new design for service which claims to provid strong concistency but in fact it doesn't provide linearizability of commands in system TODO: arguments and proves. Furthermore, GitLab can change HA licensing TODO: find cases, or restrict HA support based on country residence.

JGit Jgit is a Java git server created by Eclipse. Google contributed to this project with Ketch module:

> Git Ketch is a multi-master Git repository management system. Writes (such as git push) can be started on any server, at any time. Writes are successful only if a majority of participant servers agree. Asked writes are durable against server failure because a majority of the participants are storing all required objects.

But this is the only place where Ketch is mentioned. TODO: Analyze source code of Ketch module.

IPFS IPFS is not exactly distributed git repository project but has similar ideas and could be helpful for us. TODO: analyze IPFS project.

brig TODO: analyze the project brig.

6.1 Functional Requirements

The most important functional requirements are:

Front end The system potencially may have different kinds of front-ends, but it's required to support gRPC of GitLab to integrate the system into GitLab service and replace Gitaly.

Back end Each node may be connected to different types of storage for git repos, but it's required to support file-system storage.

6.2 Expected Metrics

In a large enterprise it is expected to have the following numbers, in terms of load, size, and speed:

Repositories	2M
Active users	100K/day
Merges	100K/day
Fetches	15M/day, 15K/m – peak
Push	200K/day
Traffic – download	200Tb/day
Traffic – update	250Gb/day

7 APPENDIX A

7.1 Git Internals

`Git` - is a simple key-value data store that stores 3 main types of Objects: `blob`, `tree`, `commit`, and one additional: `tag`. Git stores content in a manner similar to a UNIX filesystem, but a bit simplified. All the content is stored as tree and blob objects, with trees corresponding to UNIX directory entries and blobs corresponding more or less to inodes or file contents.

Conceptually, the data that Git is storing looks something like this::

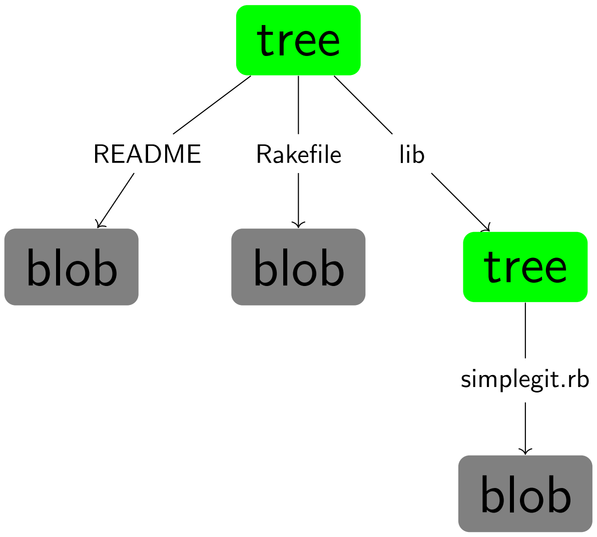


Fig. 11. Simple version of the Git data model

Blobs store content without file names and trees store filenames and also allows to store a group of files together. Top level trees represent the different snapshots of a project that you want to track.

Commit objects point to that top level trees(snapshots) and store information about who saved the snapshots, when they were saved, or why they were saved.

To order snapshots Commit object also points to parent(previous) commit if any.

So snapshots could be reached and tracked via commit objects.

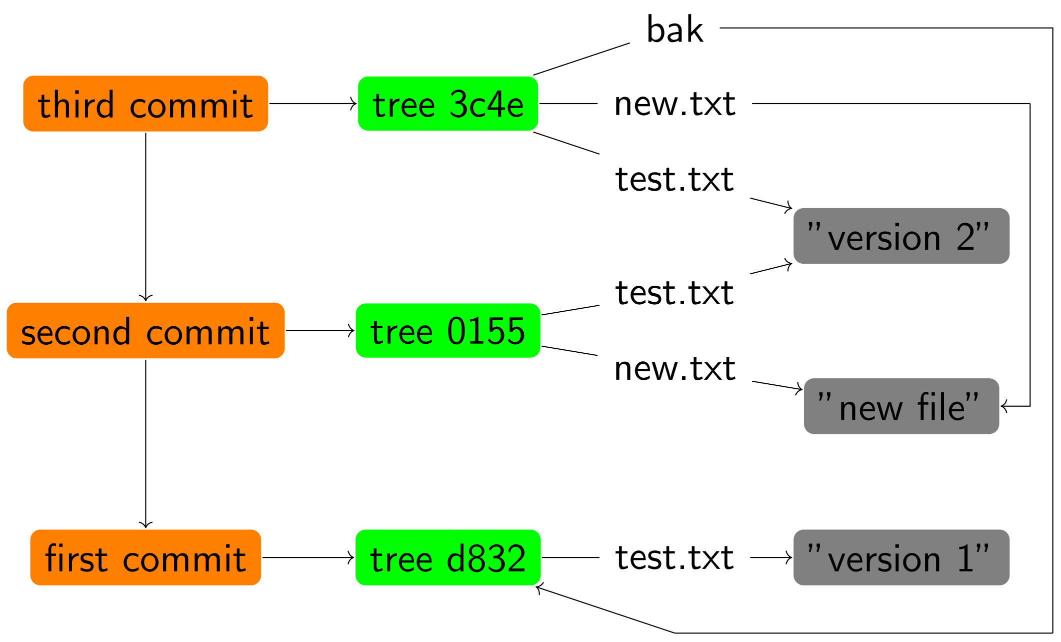


Fig. 12. All the reachable objects in Git directory

Things, listed below, serve to make our interaction with these objects easier.

Reference – A file in which you could store SHA-1 value of a commit under a simple name, so you could use that simple name rather than the raw SHA-1 value.

Branch – is a simple pointer or reference to the head of a line of work(the latest commit) in .git/refs/heads. Branch name is mapped to the SHA of the commit that is latest for this branch.

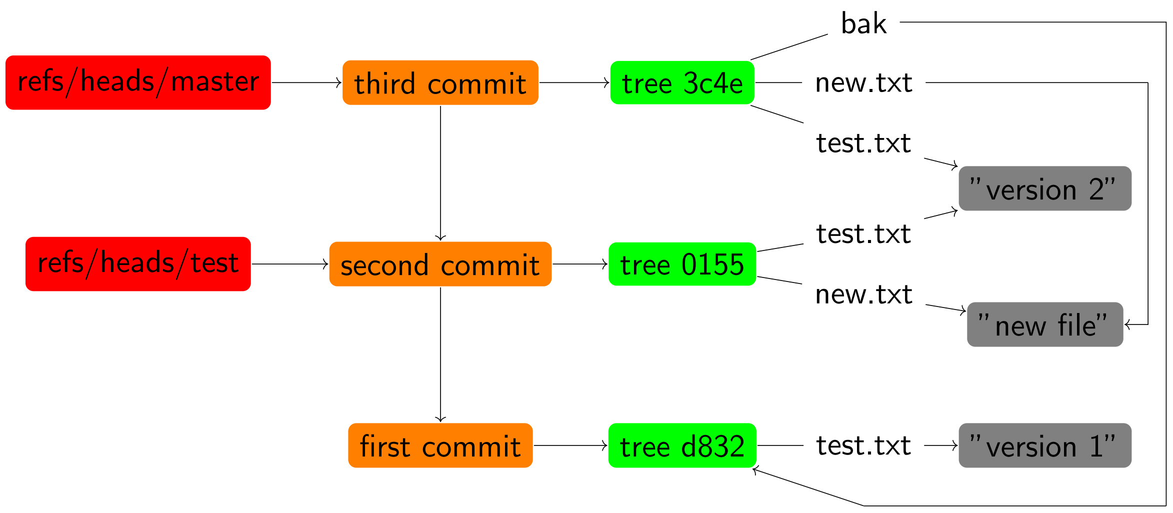


Fig. 13. Git directory objects with branch head references included

When you run commands like `git branch <branch>`, Git basically runs update-ref command to add the SHA-1 of the last commit of the branch you're on into whatever new reference you want to create. So create branch is just create a reference to some commit. And delete branch is only delete a reference. After branch deletion, all its objects will be available via their SHAs. However, unreachable objects could be deleted via other git operations.

HEAD – Usually the HEAD file is a symbolic reference to the branch you're currently on. By symbolic reference, we mean that unlike a normal reference, it contains a pointer to another reference.
If you look at the file, you'll normally see something like this:

```
$ cat .git/HEAD
ref: refs/heads/master
```

If you run `git checkout branch`, Git updates the file to look like this:

```
$ cat .git/HEAD
ref: refs/heads/test
```

When you run `git commit`, it creates the commit object, specifying the parent of that commit object to be whatever SHA-1 value the reference in HEAD points to. This commit also becomes a new head for a current branch. refs/heads/¡branch-name¿ will be updated and map to the SHA of this new commit. When you do `git reset HEAD 1`, it also only update refs/heads/¡branch-name¿. It takes SHA of parent commit of current head commit.

Tag (lightweight) – It's like a branch reference, but it never moves - it always points to the same commit but gives it a friendlier name.
You can create, update and delete such a tag, and no objects would be affected. It's only a reference.

Tag (annotated) – tag object, that points to commit or any other object amd contains metadata, and a reference to this tag object.

7.1.1 Git file storage internals. • **loose object storage**

As we mentioned, a git repository take cares of several types of objects: `blob`, `tree`, `commit`, `tag`. Let's take a look at the details of object storage:

git by default writes the objects to directory `GIT_DIR`[10]`/objects`. Each object is named by a sha1 value, for example *b98191cf971f2418e42877410a6c40fc112a0a93*
it's storage path will be

GIT_DIR/objects/b9/8191cf971f2418e42877410a6c40fc112a0a93

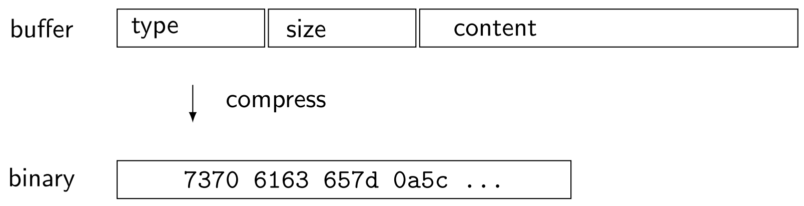
the directories under `GIT_DIR/objects` is named by `SHA[0,2]` of objects' sha1 value, this can avoid too much files in one directory because some filesystem has the max link number of files, such as ext3 has the definition:

```
include/linux/ext3_fs.h:#define EXT3_LINK_MAX 32000
```

One object file is consist of three portions:

type can be one of 'blob', 'tree', 'commit', 'tag'
size the number of bytes of the content
content the content of object

and the object file will be compressed using zlib:



the compression level can be set by '`core.compression`' or '`core.loosecompression`'.

• **pack file storage**

[10]the .git directory

When we do *git gc* or *git repack*, pack files with the suffix `.pack` maybe be created under directory *GIT_DIR/objects/pack*. A pack is a collection of objects, individually compressed, with delta compression applied, stored in a single file, with an associated index file. Packs are used to reduce the load on mirror systems, backup engines, disk storage, etc.

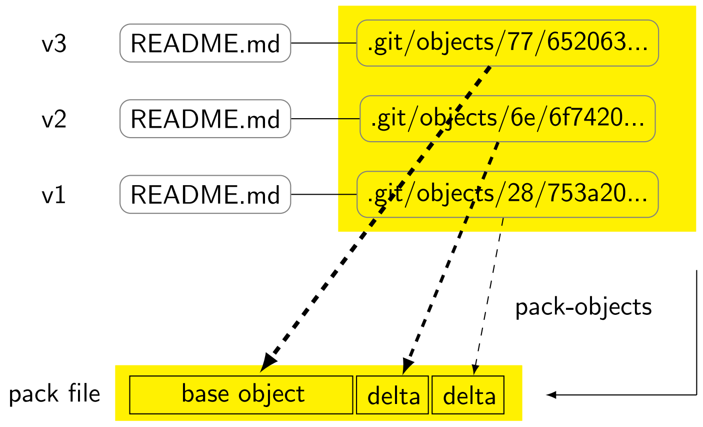
```
$ tree .git/objects/pack
.git/objects/pack
|-- pack-24fcb83682e5a2848ef7bd00a9eda0bff1a372fc.idx
|-- pack-24fcb83682e5a2848ef7bd00a9eda0bff1a372fc.pack
|-- pack-915d8b4ae03b03179912c589cee932e5a990b7f0.idx
|-- pack-915d8b4ae03b03179912c589cee932e5a990b7f0.pack
|-- ...
```

Conceptually there are only four object types in a pack file: commit, tree, tag and blob. However to save space, an object could be stored as a "delta" of another "base" object. These representations are assigned new types `ofs-delta` and `ref-delta`.

delta object Both ofs-delta and ref-delta store the "delta" to be applied to another object (called 'base object') to reconstruct the object. The difference between them is, ref-delta directly encodes 20-byte base object name. If the base object is in the same pack, ofs-delta encodes the offset of the base object in the pack instead.

base object The base object could also be deltified if it's in the same pack. Ref-delta can also refer to an object outside the pack. When stored on disk however, the pack should be self contained to avoid cyclic dependency.

And usualy we have lots of versions of a single file in a repository, a new pack file will store the last version's content as the base object, and computes the earlier versioins to to be delta objects and store them referring to the base object:


v3
README.md
.git/objects/77/652063...
v2
README.md
.git/objects/6e/6f7420...
v1
README.md
.git/objects/28/753a20...
pack-objects
pack file
base object
delta
delta

Usually objects contained in the pack are compressed using zlib, the compression level can be set by '`core.compression`' or '`pack.compression`'.

When objects contained in the pack are stored using delta compression. The objects are first internally sorted by type, size and optionally names and compared against the other objects to see if using delta compression saves space. '`pack.depth`' limits the maximum delta depth; Making it too deep affects the performance on the unpacker side, because delta data needs to be applied that many times to get to the necessary object.

To quickly find objects in the pack file, an associated index file is created with the *format*, And '*`git index-pack`*' can be used to regenerate the *.idx file on the *.pack file.

- **when pack file is created**

When we talk about the pack files, we said that packs are used to reduce the load on mirror systems, backup engines, disk storage, etc. So pack files will created in several scenes:

git gc '*`git gc`*' runs a number of housekeeping tasks within the repository, such as compressing file revisions (to reduce disk space and increase performance), removing unreachable objects which may have been created from prior git commands. We can run it manually, and it may create a new pack file.

git repack '*`git repack`*' is used to combine all objects that do not currently reside in a pack file into a pack. It can also be used to re-organize existing packs into a single, more efficient pack.

git push '*`git push`*' will run `send-pack` to connects to the remote side, it gets one file descriptor which is either a socket (over the network) or a pipe (local). What's written to this file descriptor goes to '*`git-receive-pack`*' to be unpacked. We can get the following pipeline flow from *Documentation/technical/send-pack-pipeline.txt*:

```
send-pack
   |
   pack_objects() --> fd --> receive-pack
      | ^ (pipe)
      v |
   (child)
```

so we can see that '*`git push`*' will create a pack file and send it to the remote side.

git-receive-pack As menthioned above, '*`git-receive-pack`*' will receive a pack file from '*`git push`*'. And '*`git receive-pack`*' will also forked a child '*`git gc --auto --quiet`*' to check if there are too many loose objects to pack. Ususaly the threshold is 6700 loose files, we can set it by '`gc.auto`'.

git-upload-pack When clients runs '*`git clone`*' or '*`git fetch`*', the client connects to the remote side and invokes '*`git-upload-pack`*'. '*`git-upload-pack`*' will communicate with client and compute how many objects the client want then fork and exec child '*git-pack-objects*' to create a pack file for all the needed objects and pipe the pack file to client.

```
upload-pack
   |
   pack_objects() --> fd --> fetch-pack
      | ^ (pipe)
      v |
   (child)
```

We can also use the '`repack.writeBitmaps`' to let git write a bitmap index when packing all objects to disk (e.g., when git repack -a is run). This index can speed up the "counting objects" phase of subsequent packs created for clones and fetches, at the cost of some disk space and extra time spent on the initial repack.

REFERENCES

[1] J. Gray and L. Lamport, “Consensus on transaction commit,” *ACM Trans. Database Syst.*, vol. 31, p. 133–160, Mar. 2006.