large-object-promisors

Large Object Promisors ======================

Since Git has been created, users have been complaining about issues with storing large files in Git. Some solutions have been created to help, but they haven’t helped much with some issues.

Git currently supports multiple promisor remotes, which could help with some of these remaining issues, but it’s very hard to use them to help, because a number of important features are missing.

The goal of the effort described in this document is to add these important features.

We will call a "Large Object Promisor", or "LOP" in short, a promisor remote which is used to store only large blobs and which is separate from the main remote that should store the other Git objects and the rest of the repos.

By extension, we will also call "Large Object Promisor", or LOP, the effort described in this document to add a set of features to make it easier to handle large blobs/files in Git by using LOPs.

This effort aims to especially improve things on the server side, and especially for large blobs that are already compressed in a binary format.

This effort aims to provide an alternative to Git LFS (https://git-lfs.com/) and similar tools like git-annex (https://git-annex.branchable.com/) for handling large files, even though a complete alternative would very likely require other efforts especially on the client side, where it would likely help to implement a new object representation for large blobs as discussed in:

https://lore.kernel.org/git/xmqqbkdometi.fsf@gitster.g/

Non goals ---------

- We will not discuss those client side improvements here, as they would require changes in different parts of Git than this effort.
  So we don’t pretend to fully replace Git LFS with only this effort, but we nevertheless believe that it can significantly improve the current situation on the server side, and that other separate efforts could also improve the situation on the client side.

- In the same way, we are not going to discuss all the possible ways to implement a LOP or their underlying object storage, or to optimize how LOP works.
  Our opinion is that the simplest solution for now is for LOPs to use object storage through a remote helper (see section II.2 below for more details) to store their objects. So we consider that this is the default implementation. If there are improvements on top of this,

that’s great, but our opinion is that such improvements are not necessary for LOPs to already be useful. Such improvements are likely a different technical topic, and can be taken care of separately anyway.
So in particular we are not going to discuss pluggable ODBs or other object database backends that could chunk large blobs, dedup the chunks and store them efficiently. Sure, that would be a nice improvement to store large blobs on the server side, but we believe it can just be a separate effort as it’s also not technically very related to this effort.
We are also not going to discuss data transfer improvements between LOPs and clients or servers. Sure, there might be some easy and very effective optimizations there (as we know that objects on LOPs are very likely incompressible and not deltifying well), but this can be dealt with separately in a separate effort.

In other words, the goal of this document is not to talk about all the possible ways to optimize how Git could handle large blobs, but to describe how a LOP based solution can already work well and alleviate a number of current issues in the context of Git clients and servers sharing Git objects.

Even if LOPs are used not very efficiently, they can still be useful and worth using in some cases, as we will see in more details later in this document:

- they can make it simpler for clients to use promisor remotes and therefore avoid fetching a lot of large blobs they might not need locally,

- they can make it significantly cheaper or easier for servers to host a significant part of the current repository content, and even more to host content with larger blobs or more large blobs than currently.

I Issues with the current situation ----------

- Some statistics made on GitLab repos have shown that more than 75% of the disk space is used by blobs that are larger than 1MB and often in a binary format.

- So even if users could use Git LFS or similar tools to store a lot of large blobs out of their repos, it’s a fact that in practice they don’t do it as much as they probably should.

- On the server side ideally, the server should be able to decide for itself how it stores things. It should not depend on users deciding to use tools like Git LFS on some blobs or not.

- It’s much more expensive to store large blobs that don’t delta compress well on regular fast seeking drives (like SSDs) than on object storage (like Amazon S3 or GCP Buckets). Using fast drives for regular Git repos makes sense though, as serving regular Git content (blobs containing text or code) needs drives where seeking is fast, but the content is relatively small. On the other hand, object storage for Git LFS blobs makes sense as seeking speed is not as important when dealing with large files, while costs are more important. So the fact that users don’t use Git LFS or similar tools for a significant number of large blobs has likely some bad consequences on the cost of repo storage for most Git hosting platforms.

- Having large blobs handled in the same way as other blobs and Git objects in Git repos instead of on object storage also has a cost in increased memory and CPU usage, and therefore decreased performance, when creating packfiles. (This is because Git tries to use delta compression or zlib compression which is unlikely to work well on already compressed binary content.) So it’s not just a storage cost increase.

- When a large blob has been committed into a repo, it might not be possible to remove this blob from the repo without rewriting history, even if the user then decides to use Git LFS or a similar tool to handle it.

- In fact Git LFS and similar tools are not very flexible in letting users change their minds about the blobs they should handle or not.

- Even when users are using Git LFS or similar tools, they are often complaining that these tools require significant effort to set up, learn and use correctly.

II Main features of the "Large Object Promisors" solution ----------

The main features below should give a rough overview of how the solution may work. Details about needed elements can be found in following sections.

Even if each feature below is very useful for the full solution, it is very likely to be also useful on its own in some cases where the full solution is not required. However, we’ll focus primarily on the big picture here.

Also each feature doesn’t need to be implemented entirely in Git itself. Some could be scripts, hooks or helpers that are not part of the Git repo. It would be helpful if those could be shared and improved on collaboratively though. So we want to encourage sharing them.

1) Large blobs are stored on LOPs ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Large blobs should be stored on special promisor remotes that we will call "Large Object Promisors" or LOPs. These LOPs should be additional remotes dedicated to contain large blobs especially those in binary format. They should be used along with main remotes that contain the other objects.

Note 1 ^^^^^^

To clarify, a LOP is a normal promisor remote, except that:

- it should store only large blobs,

- it should be separate from the main remote, so that the main remote can focus on serving other objects and the rest of the repos (see feature 4) below) and can use the LOP as a promisor remote for itself.

Note 2 ^^^^^^

Git already makes it possible for a main remote to also be a promisor remote storing both regular objects and large blobs for a client that clones from it with a filter on blob size. But here we explicitly want to avoid that.

Rationale ^^^^^^^^^^

LOPs aim to be good at handling large blobs while main remotes are already good at handling other objects.

Implementation ^^^^^^^^^^^^^^^

Git already has support for multiple promisor remotes, see the partial clone documentation.

Also, Git already has support for partial clone using a filter on the size of the blobs (with `git clone --filter=blob:limit=<size>`). Most of the other main features below are based on these existing features and are about making them easy and efficient to use for the purpose of better handling large blobs.

2) LOPs can use object storage ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

LOPs can be implemented using object storage, like an Amazon S3 or GCP Bucket or MinIO (which is open source under the GNU AGPLv3 license) to actually store the large blobs, and can be accessed through a Git remote helper (see linkgit:gitremote-helpers[7]) which makes the underlying object storage appear like a remote to Git.

Note ^^^^

A LOP can be a promisor remote accessed using a remote helper by both some clients and the main remote.

Rationale ^^^^^^^^^^

This looks like the simplest way to create LOPs that can cheaply handle many large blobs.

Implementation ^^^^^^^^^^^^^^^

Remote helpers are quite easy to write as shell scripts, but it might be more efficient and maintainable to write them using other languages like Go.

Some already exist under open source licenses, for example:

- https://github.com/awslabs/git-remote-s3
- https://gitlab.com/eric.p.ju/git-remote-gs

Other ways to implement LOPs are certainly possible, but the goal of this document is not to discuss how to best implement a LOP or its underlying object storage (see the "0) Non goals"

section above).

3) LOP object storage can be Git LFS storage
~~~~~~

The underlying object storage that a LOP uses could also serve as storage for large files handled by Git LFS.

Rationale ^^^^^^^^^

This would simplify the server side if it wants to both use a LOP and act as a Git LFS server.

4) A main remote can offload to a LOP with a configurable threshold
~~~~~~
~~~~~~~~

On the server side, a main remote should have a way to offload to a LOP all its blobs with a size over a configurable threshold.

Rationale ^^^^^^^^^

This makes it easy to set things up and to clean things up. For example, an admin could use this to manually convert a repo not using LOPs to a repo using a LOP. On a repo already using a LOP but where some users would sometimes push large blobs, a cron job could use this to regularly make sure the large blobs are moved to the LOP.

Implementation ^^^^^^^^^^^^^^^

Using something based on `git` `repack` `--filter=…` to separate the blobs we want to offload from the other Git objects could be a good idea. The missing part is to connect to the LOP, check if the blobs we want to offload are already there and if not send them.

5) A main remote should try to remain clean from large blobs
~~~~~~
~

A main remote should try to avoid containing a lot of oversize blobs. For that purpose, it should offload as needed to a LOP and it should have ways to prevent oversize blobs to be fetched, and also perhaps pushed, into it.

Rationale ^^^^^^^^^

A main remote containing many oversize blobs would defeat the purpose of LOPs.

Implementation ^^^^^^^^^^^^^^^

The way to offload to a LOP discussed in 4) above can be used to regularly offload oversize blobs. About preventing oversize blobs from being fetched into the repo see 6) below. About preventing oversize blob pushes, a pre-receive hook could be used.

Also there are different scenarios in which large blobs could get fetched into the main remote, for example:

- A client that doesn’t implement the "promisor-remote" protocol (described in 6) below) clones from the main remote.

- The main remote gets a request for information about a large blob and is not able to get that information without fetching the blob from the LOP.

It might not be possible to completely prevent all these scenarios from happening. So the goal here should be to implement features that make the fetching of large blobs less likely. For example adding a `remote-object-info` command in the `git cat-file --batch` protocol and its variants might make it possible for a main repo to respond to some requests about large blobs without fetching them.

6) A protocol negotiation should happen when a client clones
~~~~~~
~

When a client clones from a main repo, there should be a protocol negotiation so that the server can advertise one or more LOPs and so that the client and the server can discuss if the client could directly use a LOP the server is advertising. If the client and the server can agree on that, then the client would be able to get the large blobs directly from the LOP and the server would not need to fetch those blobs from the LOP to be able to serve the client.

Note ^^^^

For fetches instead of clones, a protocol negotiation might not always happen, see the "What about fetches?" FAQ entry below for details.

Rationale ^^^^^^^^^^

Security, configurability and efficiency of setting things up.

Implementation ^^^^^^^^^^^^^^^^

A "promisor-remote" protocol v2 capability looks like a good way to implement this. The way the client and server use this capability could be controlled by configuration variables.

Information that the server could send to the client through that protocol could be things like: LOP name, LOP URL, filter-spec (for example `blob:limit=<size>`) or just size limit that should be used as a filter when cloning, token to be used with the LOP, etc.
~~~~~~

7) A client can offload to a LOP ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

When a client is using a LOP that is also a LOP of its main remote, the client should be able to offload some large blobs it has fetched, but might not need anymore, to the LOP.

Note ^^^^

It might depend on the context if it should be OK or not for clients to offload large blobs they have created, instead of fetched, directly to the LOP without the main remote checking them in some ways (possibly using hooks or other tools).

This should be discussed and refined when we get closer to implementing this feature.

Rationale ^^^^^^^^^^

On the client, the easiest way to deal with unneeded large blobs is to offload them.

Implementation ^^^^^^^^^^^^^^^

This is very similar to what 4) above is about, except on the client side instead of the server side. So a good solution to 4) could likely be adapted to work on the client side too.

There might be some security issues here, as there is no negotiation, but they might be mitigated if the client can reuse a token it got when cloning (see 6) above). Also if the large blobs were fetched from a LOP, it is likely, and can easily be confirmed, that the LOP still has them, so that they can just be removed from the client.

III Benefits of using LOPs --------------------------

Many benefits are related to the issues discussed in "I) Issues with the current situation" above:

- No need to rewrite history when deciding which blobs are worth handling separately than other objects, or when moving or removing the threshold.
- If the protocol between client and server is developed and secured enough, then many details might be setup on the server side only and all the clients could then easily get all the configuration information and use it to set themselves up mostly automatically.
- Storage costs benefits on the server side.
- Reduced memory and CPU needs on main remotes on the server side.
- Reduced storage needs on the client side.

IV FAQ ------

What about using multiple LOPs on the server and client side?
~~~~~~
~~~~~~

~~

That could perhaps be useful in some cases, but for now it’s more likely that in most cases a single LOP will be advertised by the server and should be used by the client.

A case where it could be useful for a server to advertise multiple LOPs is if a LOP is better for some users while a different LOP is better for other users. For example some clients might have a better connection to a LOP than others.

In those cases it’s the responsibility of the server to have some documentation to help clients. It could say for example something like "Users in this part of the world might want to pick only LOP A as it is likely to be better connected to them, while users in other parts of the world should pick only LOP B for the same reason."

When should we trust or not trust the LOPs advertised by the server?
~~~~~~
~~~~~~~~~~

In some contexts, like in corporate setup where the server and all the clients are parts of an internal network in a company where admins have all the rights on every system, it’s OK, and perhaps even a good thing, if the clients fully trust the server, as it can help ensure that all the clients are on the same page.

There are also contexts in which clients trust a code hosting platform serving them some repos, but might not fully trust other users managing or contributing to some of these repos. For example, the code hosting platform could have hooks in place to check that any object it receives doesn’t contain malware or otherwise bad content. In this case it might be OK for the client to use a main remote and its LOP if they are both hosted by the code hosting platform, but not if the LOP is hosted elsewhere (where the content is not checked).

In other contexts, a client should just not trust a server.

So there should be different ways to configure how the client should behave when a server advertises a LOP to it at clone time.

As the basic elements that a server can advertise about a LOP are a LOP name and a LOP URL, the client should base its decision about accepting a LOP on these elements.

One simple way to be very strict in the LOP it accepts is for example for the client to check that the LOP is already configured on the client with the same name and URL as what the server advertises.

In general default and "safe" settings should require that the LOP are configured on the client separately from the "promisor-remote" protocol and that the client accepts a LOP only when information about it from the protocol matches what has been already configured separately.

What about LOP names? ~~~~~~~~~~~~~~~~~~~~~

In some contexts, for example if the clients sometimes fetch from each other, it can be a good idea for all the clients to use the same names for all the remotes they use, including LOPs.

In other contexts, each client might want to be able to give the name it wants to each remote, including each LOP, it interacts with.

So there should be different ways to configure how the client accepts or not the LOP name the server advertises.

If a default or "safe" setting is used, then as such a setting should require that the LOP be configured separately, then the name would be configured separately and there is no risk that the server could dictate a name to a client.

Could the main remote be bogged down by old or paranoid clients? ~~~~~~

Yes, it could happen if there are too many clients that are either unwilling to trust the main remote or that just don’t implement the "promisor-remote" protocol because they are too old or not fully compatible with the ’git’ client.

When serving such a client, the main remote has no other choice than to first fetch from its LOP, to then be able to provide to the client everything it requested. So the main remote, even if it has cleanup mechanisms (see section II.4 above), would be burdened at least temporarily with the large blobs it had to fetch from its LOP.

Not behaving like this would be breaking backward compatibility, and could be seen as segregating clients. For example, it might be possible to implement a special mode that allows the server to just reject clients that don’t implement the "promisor-remote" protocol or aren’t willing to trust the main remote. This mode might be useful in a special context like a corporate environment. There is no plan to implement such a mode though, and this should be discussed separately later anyway.

A better way to proceed is probably for the main remote to show a message telling clients that don’t implement the protocol or are unwilling to accept the advertised LOP(s) that they would get faster clone and fetches by upgrading client software or properly setting them up to accept LOP(s).

Waiting for clients to upgrade, monitoring these upgrades and limiting the use of LOPs to repos that are not very frequently accessed might be other good ways to make sure that some benefits are still reaped from LOPs. Over time, as more and more clients upgrade and benefit from LOPs, using them in more and more frequently accessed repos will become worth it.

Corporate environments, where it might be easier to make sure that all the clients are up-to-date and properly configured, could hopefully benefit more and earlier from using LOPs.

What about fetches? ~~~~~~~~~~~~~~~~~~~

There are different kinds of fetches. A regular fetch happens when some refs have been updated on the server and the client wants the ref updates and possibly the new objects added with them. A "backfill" or "lazy" fetch, on the contrary, happens when the client needs to use some objects it already knows about but doesn't have because they are on a promisor remote.

Regular fetch ^^^^^^^^^^^^^^

In a regular fetch, the client will contact the main remote and a protocol negotiation will happen between them. It's a good thing that a protocol negotiation happens every time, as the configuration on the client or the main remote could have changed since the previous protocol negotiation. In this case, the new protocol negotiation should ensure that the new fetch will happen in a way that satisfies the new configuration of both the client and the server.

In most cases though, the configurations on the client and the main remote will not have changed between 2 fetches or between the initial clone and a subsequent fetch. This means that the result of a new protocol negotiation will be the same as the previous result, so the new fetch will happen in the same way as the previous clone or fetch, using, or not using, the same LOP(s) as last time.

"Backfill" or "lazy" fetch ^^^^^^^^^^^^^^^^^^^^^^^^^^^^

When there is a backfill fetch, the client doesn't necessarily contact the main remote first. It will try to fetch from its promisor remotes in the order they appear in the config file, except that a remote configured using the `extensions.partialClone` config variable will be tried last. See the partial clone documentation.

This is not new with this effort. In fact this is how multiple remotes have already been working for around 5 years.

When using LOPs, having the main remote configured using `extensions.partialClone`, so it's tried last, makes sense, as missing objects should only be large blobs that are on LOPs.

This means that a protocol negotiation will likely not happen as the missing objects will be fetched from the LOPs, and then there will be nothing left to fetch from the main remote.

To secure that, it could be a good idea for LOPs to require a token from the client when it fetches from them. The client could get the token when performing a protocol negotiation with the main remote (see section II.6 above).

V Future improvements ----------

It is expected that at the beginning using LOPs will be mostly worth it either in a corporate context where the Git version that clients use can easily be controlled, or on repos that are infrequently accessed. (See the "Could the main remote be bogged down by old or paranoid clients?" section in the FAQ above.)

Over time, as more and more clients upgrade to a version that implements the "promisor-remote" protocol v2 capability described above in section II.6), it will be worth it to use LOPs more widely.

A lot of improvements may also help using LOPs more widely. Some of these improvements are part of the scope of this document like the following:

- Implementing a "remote-object-info" command in the `git cat-file --batch` protocol and its variants to allow main remotes to respond to requests about large blobs without fetching them. (Eric Ju has started working on this based on previous work by Calvin Wan.)

- Creating better cleanup and offload mechanisms for main remotes and clients to prevent accumulation of large blobs.

- Developing more sophisticated protocol negotiation capabilities between clients and servers for handling LOPs, for example adding a filter-spec (e.g., blob:limit=<size>) or size limit for filtering when cloning, or adding a token for LOP authentication.

- Improving security measures for LOP access, particularly around token handling and authentication.

- Developing standardized ways to configure and manage multiple LOPs across different environments. Especially in the case where different LOPs serve the same content to clients in different geographical locations, there is a need for replication or synchronization between LOPs.

Some improvements, including some that have been mentioned in the "0) Non Goals" section of this document, are out of the scope of this document:

- Implementing a new object representation for large blobs on the client side.

- Developing pluggable ODBs or other object database backends that could chunk large blobs, dedup the chunks and store them efficiently.

- Optimizing data transfer between LOPs and clients/servers, particularly for incompressible and non-deltifying content.

- Creating improved client side tools for managing large objects more effectively, for example tools for migrating from Git LFS or git-annex, or tools to find which objects could be offloaded and how much disk space could be reclaimed by offloading them.

Some improvements could be seen as part of the scope of this document, but might already have their own separate projects from the Git project, like:

- Improving existing remote helpers to access object storage or developing new ones.

- Improving existing object storage solutions or developing new ones.

Even though all the above improvements may help, this document and the LOP effort should try to focus, at least first, on a relatively small number of improvements mostly those that are in its current scope.

For example introducing pluggable ODBs and a new object database backend is likely a multi-year effort on its own that can happen separately in parallel. It has different technical requirements, touches other part of the Git code base and should have its own design document(s).

□□□ #2
□ □□□ □□□ 2026-10-31 20:16:33 UTC
□ □□□ □□□ 2026-10-31 20:16:34 UTC