bundle-uri

Bundle URIs ==========

Git bundles are files that store a pack-file along with some extra metadata, including a set of refs and a (possibly empty) set of necessary commits. See linkgit:git-bundle[1] and linkgit:gitformat-bundle[5] for more information.

Bundle URIs are locations where Git can download one or more bundles in order to bootstrap the object database in advance of fetching the remaining objects from a remote.

One goal is to speed up clones and fetches for users with poor network connectivity to the origin server. Another benefit is to allow heavy users, such as CI build farms, to use local resources for the majority of Git data and thereby reducing the load on the origin server.

To enable the bundle URI feature, users can specify a bundle URI using command-line options or the origin server can advertise one or more URIs via a protocol v2 capability.

Design Goals ----------

The bundle URI standard aims to be flexible enough to satisfy multiple workloads. The bundle provider and the Git client have several choices in how they create and consume bundle URIs.

- Bundles can have whatever name the server desires. This name could refer to immutable data by using a hash of the bundle contents. However, this means that a new URI will be needed after every update of the content. This might be acceptable if the server is advertising the URI (and the server is aware of new bundles being generated) but would not be ergonomic for users using the command line option.

- The bundles could be organized specifically for bootstrapping full clones, but could also be organized with the intention of bootstrapping incremental fetches. The bundle provider must decide on one of several organization schemes to minimize client downloads during incremental fetches, but the Git client can also choose whether to use bundles for either of these operations.

- The bundle provider can choose to support full clones, partial clones, or both. The client can detect which bundles are appropriate for the repository’s partial clone filter, if any.

- The bundle provider can use a single bundle (for clones only), or a list of bundles. When using a list of bundles, the provider can specify whether or not the client needs *all* of the bundle URIs for a full clone, or if *any* one of the bundle URIs is sufficient. This allows the bundle provider to use different URIs for different geographies.

- The bundle provider can organize the bundles using heuristics, such as creation tokens, to help the client prevent downloading bundles it does not need. When the bundle provider does not provide these heuristics, the client can use optimizations to minimize how much of the data is downloaded.

- The bundle provider does not need to be associated with the Git server. The client can choose to use the bundle provider without it being advertised by the Git server.

- The client can choose to discover bundle providers that are advertised by the Git server. This could happen during `git clone`, during `git fetch`, both, or neither. The user can choose which combination works best for them.

- The client can choose to configure a bundle provider manually at any time. The client can also choose to specify a bundle provider manually as a command-line option to `git clone`.

Each repository is different and every Git server has different needs. Hopefully the bundle URI feature is flexible enough to satisfy all needs. If not, then the feature can be extended through its versioning mechanism.

Server requirements ----------

To provide a server-side implementation of bundle servers, no other parts of the Git protocol are required. This allows server maintainers to use static content solutions such as CDNs in order to serve the bundle files.

At the current scope of the bundle URI feature, all URIs are expected to be HTTP(S) URLs where content is downloaded to a local file using a `GET` request to that URL. The server could include authentication requirements to those requests with the aim of triggering the configured credential helper for secure access. (Future extensions could use "file://" URIs or SSH URIs.)

Assuming a `200 OK` response from the server, the content at the URL is inspected. First, Git attempts to parse the file as a bundle file of version 2 or higher. If the file is not a bundle, then the file is parsed as a plain-text file using Git’s config parser. The key-value pairs in that config file are expected to describe a list of bundle URIs. If neither of these parse attempts succeed, then Git will report an error to the user that the bundle URI provided erroneous data.

Any other data provided by the server is considered erroneous.

Bundle Lists ------------

The Git server can advertise bundle URIs using a set of `key=value` pairs. A bundle URI can also serve a plain-text file in the Git config format containing these same `key=value` pairs. In both cases, we consider this to be a *bundle list*. The pairs specify information about the bundles that the client can use to make decisions for which bundles to download and which to ignore.

A few keys focus on properties of the list itself.

bundle.version
(Required) This value provides a version number for the bundle list. If a future Git change enables a feature that needs the Git client to react to a new key in the bundle list file, then this version will increment. The only current version number is 1, and if any other value is specified then Git will fail to use this file.

bundle.mode
(Required) This value has one of two values: `all` and `any`. When `all` is specified, then the client should expect to need all of the listed bundle URIs that match their repository's requirements. When `any` is specified, then the client should expect that any one of the bundle URIs that match their repository's requirements will suffice. Typically, the `any` option is used to list a number of different bundle servers located in different geographies.

bundle.heuristic
If this string-valued key exists, then the bundle list is designed to work well with incremental `git fetch` commands. The heuristic signals that there are additional keys available for each bundle that help determine which subset of bundles the client should download. The only heuristic currently planned is `creationToken`.

The remaining keys include an `<id>` segment which is a server-designated name for each available bundle. The `<id>` must contain only alphanumeric and `-` characters.

bundle.<id>.uri
(Required) This string value is the URI for downloading bundle `<id>`. If the URI begins with a protocol (`http://` or `https://`) then the URI is absolute. Otherwise, the URI is interpreted as relative to the URI used for the bundle list. If the URI begins with `/`, then that relative path is relative to the domain name used for the bundle list. (This use of relative paths is intended to make it easier to distribute a set of bundles across a large number of servers or CDNs with different domain names.)

bundle.<id>.filter
This string value represents an object filter that should also appear in the header of this bundle. The server uses this value to differentiate different kinds of bundles from which the client can choose those that match their object filters.

bundle.<id>.creationToken
This value is a nonnegative 64-bit integer used for sorting the bundles list. This is used to download a subset of bundles during a fetch when `bundle.heuristic=creationToken`.

bundle.<id>.location
This string value advertises a real-world location from where the bundle URI is served. This can be used to present the user with an option for which bundle URI to use or simply as an informative indicator of which bundle URI was selected by Git. This is only valuable when `bundle.mode` is `any`.

Here is an example bundle list using the Git config format:

```
[bundle]
    version = 1
    mode = all
    heuristic = creationToken

[bundle "2022-02-09-1644442601-daily"]
    uri = https://bundles.example.com/git/git/2022-02-09-1644442601-daily.bundle
    creationToken = 1644442601

[bundle "2022-02-02-1643842562"]
    uri = https://bundles.example.com/git/git/2022-02-02-1643842562.bundle
    creationToken = 1643842562

[bundle "2022-02-09-1644442631-daily-blobless"]
    uri = 2022-02-09-1644442631-daily-blobless.bundle
    creationToken = 1644442631
    filter = blob:none

[bundle "2022-02-02-1643842568-blobless"]
    uri = /git/git/2022-02-02-1643842568-blobless.bundle
    creationToken = 1643842568
    filter = blob:none
```

This example uses `bundle.mode=all` as well as the `bundle.<id>.creationToken` heuristic. It also uses the `bundle.<id>.filter` options to present two parallel sets of bundles: one for full clones and another for blobless partial clones.

Suppose that this bundle list was found at the URI `https://bundles.example.com/git/git/` and so the two blobless bundles have the following fully-expanded URIs:

- `https://bundles.example.com/git/git/2022-02-09-1644442631-daily-blobless.bundle`
- `https://bundles.example.com/git/git/2022-02-02-1643842568-blobless.bundle`

Advertising Bundle URIs ----------

If a user knows a bundle URI for the repository they are cloning, then they can specify that URI manually through a command-line option. However, a Git host may want to advertise bundle URIs during the clone operation, helping users unaware of the feature.

The only thing required for this feature is that the server can advertise one or more bundle URIs. This advertisement takes the form of a new protocol v2 capability specifically for discovering bundle URIs.

The client could choose an arbitrary bundle URI as an option *or* select the URI with best performance by some exploratory checks. It is up to the bundle provider to decide if having multiple URIs is preferable to a single URI that is geodistributed through server-side infrastructure.

Cloning with Bundle URIs ----------

The primary need for bundle URIs is to speed up clones. The Git client will interact with bundle URIs according to the following flow:

1. The user specifies a bundle URI with the `--bundle-uri` command-line option *or* the client discovers a bundle list advertised by the Git server.

2. If the downloaded data from a bundle URI is a bundle, then the client inspects the bundle headers to check that the prerequisite commit OIDs are present in the client repository. If some are missing, then the client delays unbundling until other bundles have been unbundled, making those OIDs present. When all required OIDs are present, the client unbundles that data using a refspec. The refspec used is `+refs/:refs/bundles/`. These refs are stored so that later `git fetch` negotiations can communicate each bundled ref as a `have`, reducing the size of the fetch over the Git protocol. To allow pruning refs from this ref namespace, Git may introduce a numbered namespace (such as `refs/bundles/<i>/*`) such that stale bundle refs can be deleted.

3. If the file is instead a bundle list, then the client inspects the `bundle.mode` to see if the list is of the `all` or `any` form.

```
   a. If `bundle.mode=all`, then the client considers all bundle
      URIs. The list is reduced based on the `bundle.<id>.filter` options
      matching the client repository's partial clone filter. Then, all
      bundle URIs are requested. If the `bundle.<id>.creationToken`
      heuristic is provided, then the bundles are downloaded in decreasing
      order by the creation token, stopping when a bundle has all required
      OIDs. The bundles can then be unbundled in increasing creation token
      order. The client stores the latest creation token as a heuristic
      for avoiding future downloads if the bundle list does not advertise
      bundles with larger creation tokens.

   b. If `bundle.mode=any`, then the client can choose any one of the
      bundle URIs to inspect. The client can use a variety of ways to
      choose among these URIs. The client can also fallback to another URI
      if the initial choice fails to return a result.
```

Note that during a clone we expect that all bundles will be required, and heuristics such as `bundle.<uri>.creationToken` can be used to download bundles in chronological order or in parallel.

If a given bundle URI is a bundle list with a `bundle.heuristic` value, then the client can choose to store that URI as its chosen bundle URI. The client can then navigate directly to that URI during later `git` `fetch` calls.

When downloading bundle URIs, the client can choose to inspect the initial content before committing to downloading the entire content. This may provide enough information to determine if the URI is a bundle list or a bundle. In the case of a bundle, the client may inspect the bundle header to determine that all advertised tips are already in the client repository and cancel the remaining download.

Fetching with Bundle URIs ------------------------

When the client fetches new data, it can decide to fetch from bundle servers before fetching from the origin remote. This could be done via a command-line option, but it is more likely useful to use a config value such as the one specified during the clone.

The fetch operation follows the same procedure to download bundles from a bundle list (although we do *not* want to use parallel downloads here). We expect that the process will end when all prerequisite commit OIDs in a thin bundle are already in the object database.

When using the `creationToken` heuristic, the client can avoid downloading any bundles if their creation tokens are not larger than the stored creation token. After fetching new bundles, Git updates this local creation token.

If the bundle provider does not provide a heuristic, then the client should attempt to inspect the bundle headers before downloading the full bundle data in case the bundle tips already exist in the client repository.

Error Conditions ----------------

If the Git client discovers something unexpected while downloading information according to a bundle URI or the bundle list found at that location, then Git can ignore that data and continue as if it was not given a bundle URI. The remote Git server is the ultimate source of truth, not the bundle URI.

Here are a few example error conditions:

- The client fails to connect with a server at the given URI or a connection is lost without any chance to recover.

- The client receives a 400-level response (such as `404` `Not` `Found` or `401` `Not` `Authorized`). The client should use the credential helper to find and provide a credential for the URI, but match the semantics of Git's other HTTP protocols in terms of handling specific 400-level errors.

- The server reports any other failure response.

- The client receives data that is not parsable as a bundle or bundle list.
- A bundle includes a filter that does not match expectations.
- The client cannot unbundle the bundles because the prerequisite commit OIDs are not in the object database and there are no more bundles to download.

There are also situations that could be seen as wasteful, but are not error conditions:

- The downloaded bundles contain more information than is requested by the clone or fetch request. A primary example is if the user requests a clone with `--single-branch` but downloads bundles that store every reachable commit from all `refs/heads/*` references. This might be initially wasteful, but perhaps these objects will become reachable by a later ref update that the client cares about.
- A bundle download during a `git fetch` contains objects already in the object database. This is probably unavoidable if we are using bundles for fetches, since the client will almost always be slightly ahead of the bundle servers after performing its "catch-up" fetch to the remote server. This extra work is most wasteful when the client is fetching much more frequently than the server is computing bundles, such as if the client is using hourly prefetches with background maintenance, but the server is computing bundles weekly. For this reason, the client should not use bundle URIs for fetch unless the server has explicitly recommended it through a `bundle.heuristic` value.

Example Bundle Provider organization ----------

The bundle URI feature is intentionally designed to be flexible to different ways a bundle provider wants to organize the object data. However, it can be helpful to have a complete organization model described here so providers can start from that base.

This example organization is a simplified model of what is used by the GVFS Cache Servers (see section near the end of this document) which have been beneficial in speeding up clones and fetches for very large repositories, although using extra software outside of Git.

The bundle provider deploys servers across multiple geographies. Each server manages its own bundle set. The server can track a number of Git repositories, but provides a bundle list for each based on a pattern. For example, when mirroring a repository at `https://<domain>/<org>/<repo>` the bundle server could have its bundle list available at `https://<server-url>/<domain>/<org>/<repo>`. The origin Git server can list all of these servers under the "any" mode:

```
[bundle]
    version = 1
    mode = any

[bundle "eastus"]
    uri = https://eastus.example.com/<domain>/<org>/<repo>
```

```
[bundle "europe"]
    uri = https://europe.example.com/<domain>/<org>/<repo>

[bundle "apac"]
    uri = https://apac.example.com/<domain>/<org>/<repo>
```

This "list of lists" is static and only changes if a bundle server is added or removed.

Each bundle server manages its own set of bundles. The initial bundle list contains only a single bundle, containing all of the objects received from cloning the repository from the origin server. The list uses the `creationToken` heuristic and a `creationToken` is made for the bundle based on the server's timestamp.

The bundle server runs regularly-scheduled updates for the bundle list, such as once a day. During this task, the server fetches the latest contents from the origin server and generates a bundle containing the objects reachable from the latest origin refs, but not contained in a previously-computed bundle. This bundle is added to the list, with care that the `creationToken` is strictly greater than the previous maximum `creationToken`.

When the bundle list grows too large, say more than 30 bundles, then the oldest "*N* minus 30" bundles are combined into a single bundle. This bundle's `creationToken` is equal to the maximum `creationToken` among the merged bundles.

An example bundle list is provided here, although it only has two daily bundles and not a full list of 30:

```
[bundle]
    version = 1
    mode = all
    heuristic = creationToken

[bundle "2022-02-13-1644770820-daily"]
    uri = https://eastus.example.com/<domain>/<org>/<repo>/2022-02-09-1644770820-daily.bundle
    creationToken = 1644770820

[bundle "2022-02-09-1644442601-daily"]
    uri = https://eastus.example.com/<domain>/<org>/<repo>/2022-02-09-1644442601-daily.bundle
    creationToken = 1644442601

[bundle "2022-02-02-1643842562"]
    uri = https://eastus.example.com/<domain>/<org>/<repo>/2022-02-02-1643842562.bundle
    creationToken = 1643842562
```

To avoid storing and serving object data in perpetuity despite becoming unreachable in the origin server, this bundle merge can be more careful. Instead of taking an absolute union of the old bundles, instead the bundle can be created by looking at the newer bundles and ensuring that their necessary commits are all available in this merged bundle (or in another one of the newer bundles). This allows "expiring" object data that is not being used by new commits in this window of time. That data could be reintroduced by a later push.

The intention of this data organization has two main goals. First, initial clones of the repository become faster by downloading precomputed object data from a closer source. Second, `git fetch` commands can be faster, especially if the client has not fetched for a few days. However, if a client does not fetch for 30 days, then the bundle list organization would cause redownloading a large amount of object data.

One way to make this organization more useful to users who fetch frequently is to have more frequent bundle creation. For example, bundles could be created every hour, and then once a day those "hourly" bundles could be merged into a "daily" bundle. The daily bundles are merged into the oldest bundle after 30 days.

It is recommended that this bundle strategy is repeated with the `blob:none` filter if clients of this repository are expecting to use blobless partial clones. This list of blobless bundles stays in the same list as the full bundles, but uses the `bundle.<id>.filter` key to separate the two groups. For very large repositories, the bundle provider may want to *only* provide blobless bundles.

Implementation Plan ----------

This design document is being submitted on its own as an aspirational document, with the goal of implementing all of the mentioned client features over the course of several patch series. Here is a potential outline for submitting these features:

1. Integrate bundle URIs into `git clone` with a `--bundle-uri` option. This will include a new `git fetch --bundle-uri` mode for use as the implementation underneath `git clone`. The initial version here will expect a single bundle at the given URI.

2. Implement the ability to parse a bundle list from a bundle URI and update the `git fetch --bundle-uri` logic to properly distinguish between `bundle.mode` options. Specifically design the feature so that the config format parsing feeds a list of key-value pairs into the bundle list logic.

3. Create the `bundle-uri` protocol v2 command so Git servers can advertise bundle URIs using the key-value pairs. Plug into the existing key-value input to the bundle list logic. Allow `git clone` to discover these bundle URIs and bootstrap the client repository from the bundle data. (This choice is an opt-in via a config option and a command-line option.)

4. Allow the client to understand the `bundle.heuristic` configuration key and the `bundle.<id>.creationToken` heuristic. When `git clone` discovers a bundle URI with `bundle.heuristic`, it configures the client repository to check that bundle URI during later `git fetch <remote>` commands.

5. Allow clients to discover bundle URIs during `git` `fetch` and configure a bundle URI for later fetches if `bundle.heuristic` is set.

6. Implement the "inspect headers" heuristic to reduce data downloads when the `bundle.<id>.creationToken` heuristic is not available.

As these features are reviewed, this plan might be updated. We also expect that new designs will be discovered and implemented as this feature matures and becomes used in real-world scenarios.

Related Work: Packfile URIs ----------

The Git protocol already has a capability where the Git server can list a set of URLs along with the packfile response when serving a client request. The client is then expected to download the packfiles at those locations in order to have a complete understanding of the response.

This mechanism is used by the Gerrit server (implemented with JGit) and has been effective at reducing CPU load and improving user performance for clones.

A major downside to this mechanism is that the origin server needs to know *exactly* what is in those packfiles, and the packfiles need to be available to the user for some time after the server has responded. This coupling between the origin and the packfile data is difficult to manage.

Further, this implementation is extremely hard to make work with fetches.

Related Work: GVFS Cache Servers ----------

The GVFS Protocol [2] is a set of HTTP endpoints designed independently of the Git project before Git's partial clone was created. One feature of this protocol is the idea of a "cache server" which can be colocated with build machines or developer offices to transfer Git data without overloading the central server.

The endpoint that VFS for Git is famous for is the `GET` `/gvfs/objects/{oid}` endpoint, which allows downloading an object on-demand. This is a critical piece of the filesystem virtualization of that product.

However, a more subtle need is the `GET` `/gvfs/prefetch?lastPackTimestamp=<t>` endpoint. Given an optional timestamp, the cache server responds with a list of precomputed packfiles containing the commits and trees that were introduced in those time intervals.

The cache server computes these "prefetch" packfiles using the following strategy:

1. Every hour, an "hourly" pack is generated with a given timestamp.
2. Nightly, the previous 24 hourly packs are rolled up into a "daily" pack.
3. Nightly, all prefetch packs more than 30 days old are rolled up into one pack.

When a user runs `gvfs` `clone` or `scalar` `clone` against a repo with cache servers, the client requests all prefetch packfiles, which is at most `24` `+` `30` `+` `1` packfiles downloading only

commits and trees. The client then follows with a request to the origin server for the references, and attempts to checkout that tip reference. (There is an extra endpoint that helps get all reachable trees from a given commit, in case that commit was not already in a prefetch packfile.)

During a `git fetch`, a hook requests the prefetch endpoint using the most-recent timestamp from a previously-downloaded prefetch packfile. Only the list of packfiles with later timestamps are downloaded. Most users fetch hourly, so they get at most one hourly prefetch pack. Users whose machines have been off or otherwise have not fetched in over 30 days might redownload all prefetch packfiles. This is rare.

It is important to note that the clients always contact the origin server for the refs advertisement, so the refs are frequently "ahead" of the prefetched pack data. The missing objects are downloaded on-demand using the `GET` `gvfs/objects/{oid}` requests, when needed by a command such as `git` `checkout` or `git` `log`. Some Git optimizations disable checks that would cause these on-demand downloads to be too aggressive.

See Also --------

[1] https://lore.kernel.org/git/RFC-cover-00.13-0000000000-20210805T150534Z-avarab@gmail.com/ An earlier RFC for a bundle URI feature.

[2] https://github.com/microsoft/VFSForGit/blob/master/Protocol.md The GVFS Protocol

---

□□□ #2
□ □□□ □□□ 2026-10-31 20:16:22 UTC
□ □□□ □□□ 2026-10-31 20:16:23 UTC