



TECHNICAL REPORT FCDD-AMX-XX-XX

SOFTWARE COLLABORATION BETWEEN DoD ORGANIZATIONS: AN OFFLINE TRANSFER METHOD FOR GIT REPOSITORIES

Kelsey D Brown

Technology Development Directorate
Combat Capabilities Development Command
Aviation & Missile Center

And

Colin Stagner

Dynetics

March 2021

Distribution Statement A: Approved for public release: distribution unlimited.

Approved for Public Release

DESTRUCTION NOTICE

FOR CLASSIFIED DOCUMENTS, FOLLOW THE PROCEDURES IN DoD 5200.22-M, INDUSTRIAL SECURITY MANUAL, SECTION II-19 OR DoD 5200.1-R, INFORMATION SECURITY PROGRAM REGULATION, CHAPTER IX. FOR UNCLASSIFIED, LIMITED DOCUMENTS, DESTROY BY ANY METHOD THAT WILL PREVENT DISCLOSURE OF CONTENTS OR RECONSTRUCTION OF THE DOCUMENT.

DISCLAIMER

THE FINDINGS IN THIS REPORT ARE NOT TO BE CONSTRUED AS AN OFFICIAL DEPARTMENT OF THE ARMY POSITION UNLESS SO DESIGNATED BY OTHER AUTHORIZED DOCUMENTS.

TRADE NAMES

USE OF TRADE NAMES OR MANUFACTURERS IN THIS REPORT DOES NOT CONSTITUTE AN OFFICIAL ENDORSEMENT OR APPROVAL OF THE USE OF SUCH COMMERCIAL HARDWARE OR SOFTWARE.

Approved for Public Release

Approved for Public Release

REPORT DOCUMENTATION PAGE				<i>Form Approved OMB No. 0704-0188</i>	
The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing the burden, to the Department of Defense, Executive Service Directorate (0704-0188). Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ORGANIZATION.					
1. REPORT DATE (DD-MM-YYYY) March 2021		2. REPORT TYPE Final		3. DATES COVERED (From - To)	
4. TITLE AND SUBTITLE SOFTWARE COLLABORATION BETWEEN DOD ORGANIZATIONS: AN OFFLINE TRANSFER METHOD FOR GIT REPOSITORIES				5a. CONTRACT NUMBER	
				5b. GRANT NUMBER	
				5c. PROGRAM ELEMENT NUMBER	
6. AUTHOR(S) Kelsey D Brown Colin Stagner				5d. PROJECT NUMBER	
				5e. TASK NUMBER	
				5f. WORK UNIT NUMBER	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) DEVCOM Aviation and Missile Center (AvMC) ATTN: FCDD-AMT-MGR Redstone Arsenal, AL 35898-5000				8. PERFORMING ORGANIZATION REPORT NUMBER TR-FCDD-XXX-20-XX	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)				10. SPONSOR/MONITOR'S ACRONYM(S)	
				11. SPONSOR/MONITOR'S REPORT NUMBER(S)	
12. DISTRIBUTION/AVAILABILITY STATEMENT					
13. SUPPLEMENTARY NOTES					
14. ABSTRACT (U) The Git version control system is essential for many software engineering workflows. In an effort to reduce costs and deliver relevant, high-quality software to the warfighter, organizations within the U.S. Department of Defense (DOD) are embracing "agile" software development processes. This report builds on the Git bundling process by automating the generation and ingestion of bundle files. Automation reduces the potential for human error and improves the reliability of transfers.					
15. SUBJECT TERMS (U)					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT SAR or UU	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			19b. TELEPHONE NUMBER (Include area code)
(U)	(U)	(U)			

TABLE OF CONTENTS

I.	Introduction	4
II.	Background	5
III.	Methods.....	7
IV.	The Git-Teleport tool	10
V.	Discussion	11
a)	Maintenance versus Ownership	11
b)	The Contribution Process	12
c)	Git Branching Workflow	13
d)	Git Forking Workflow.....	14
e)	Submitting Patches to the Upstream.....	16
VI.	Conclusion	18
A.1.	File Listing “README.md”	23
A.2.	File Listing “bin/git-teleport”	37
A.3.	File Listing “share/template/receive/teleport-mirror”	84
A.4.	File Listing “share/template/receive/hooks/post-receive.mirror.sample” 84	
A.5.	File Listing “share/template/receive/hooks/pre-receive”	86
A.6.	File Listing “share/template/send/.gitignore”	88
A.7.	File Listing “share/template/send/Beamdownfile”	88
A.8.	File Listing “share/template/share/template/send/teleportw”	88
A.9.	File Listing “share/template/send/Beamdownstate/.gitignore”	90
A.10.	File Listing “test/run.sh”	90
A.11.	File Listing “test/test beamdownfile/Beamdownfile”	104
A.12.	File Listing “test/test beamdownfile/Beamdownfile2”	104
A.13.	File Listing “test/test beamdownfile/Beamupfile”	105

LIST OF ILLUSTRATIONS

<u>Figure</u>	<u>Title</u>	<u>Page</u>
Figure 1.	Example Git history traversal for the <code>master</code> branch.	6
Figure 2.	Comparing commits with a remote repository.	7
Figure 3.	A Memory for Bundles.....	8
Figure 4.	A Three-Tiered Branching Workflow.	13
Figure 5.	Forking Workflow	15
Figure 6.	Two-Way Git-Teleport Scenario.....	16
Figure 7.	Git-Teleport with Patch File Submission	17

I. Introduction

The Git version control system is essential for many software engineering workflows. In software engineering, multiple developers collaborate to produce a functioning software product. The process which governs this collaboration can substantially impact both cost and schedule. In an effort to reduce costs and deliver relevant, high-quality software to the warfighter, organizations within the U.S. Department of Defense (DOD) are embracing “agile” software development processes [1]. Agile and similar workflows use iterated development cycles to increase the pace of development.

During each Agile cycle, new features are implemented, tested, and integrated completely—all within the span of a few weeks, at most [2, p. 8]. Rapid cycles ensure that new functionality is released as soon as possible. Since there is no dedicated integration or testing phase, the product must always be in a deliverable, working state. To facilitate this, it is necessary to reduce the burden of collaboration between developers as much as possible.

Rapid iteration demands highly concurrent development. Each developer must be able to work their assigned task(s) without waiting on others. All source code changes must sum together neatly, without a great deal of human intervention. Changes must be peer-reviewed and tested before being included in a production release. To support these activities, developers need a *version control system* to collect and manage their changes.

Git is a Distributed Version Control System (DVCS). Instead of synchronizing individual files, a DVCS synchronizes the complete edit *history* of all files [3]. Since the history is decentralized, a developer can append their changes anywhere in the history—without concern for which revision is “most current.” This enables new features to evolve independently of one another in separate, isolated branches. When a feature is ready, it can be “merged” into the mainline of development. Git includes novel tools and algorithms for automating this process.

The DOD’s software projects can be distributed across many different physical locations. Projects may involve multiple government and contractor facilities, which may be located in different parts of the country. These facilities may share very little computing infrastructure—if any. Because Git is distributed, it is a natural choice for these types of software projects. With Git, each site has its own copy of the history and can make changes independently. Git was designed for this use-case: it was developed to version the linux kernel, which has no central repository or build infrastructure [4].

Even a distributed version control system, like Git, must synchronize its histories occasionally. In the “open source” world, this is typically accomplished via a shared repository host, such as those provided by Github, Gitlab, and others. Private industry can use either in-house versions of these tools or simpler mechanisms, such as shared folders. On DOD contracts, cybersecurity practices such as [5] require that access to these shares be heavily restricted. These restrictions often make few allowances for users from other organizations. As a result, Git repositories which span DOD enclaves or contractor organizations often cannot be shared in the usual way.

Approved for Public Release

Git includes a mechanism for offline transfer of repository content. This mechanism is known as “Git bundles.” Updating a repository with bundles requires only the ability to transfer a file. A direct network connection is unnecessary [6]. This simplifies cybersecurity compliance and enables cross-organization collaboration.

The bundling process included with Git involves many different shell commands [7]. When performed “by hand,” the process is error-prone and brittle. An erroneous or omitted command can render the bundle file unusable. If an error occurs, it can only be mitigated with an additional file transfer. The problem intensifies when multiple, interdependent repositories must be transferred: an error in any bundle will result in an incomplete update.

This report builds on the Git bundling process by automating the generation and ingestion of bundle files. Automation reduces the potential for human error and improves the reliability of transfers. Developer workflows are proposed for use in cross-enclave or disconnected environments. This report includes a reference implementation of the proposed methods. The reference software, git-teleport, should be portable to most environments in which Git can be found.

II. Background

Each Git repository consists of a history database and zero or more “working trees.” The working tree contains checked-out files that the developer can edit. Although the developer typically interacts with git via the working tree, only the *database* must be sent from site-to-site. In working copies, the database is stored in a directory named “.git.” “Bare,” server-style repositories contain only the database.

The database directory contains all information from the Git repository. This makes it possible to mirror a complete Git repository, from one location to another, just by copying the database directory. The directory can be compressed to save space. On the remote side, the directory is extracted. The extracted directory is immediately available as a complete Git repository.

The main shortcoming with this procedure is file size. For long-running projects, the Git database files may become quite large. Many transfer methods limit the amount of data (i.e., file size) that can be sent at once, and large files take more time to process. A method for reducing the size of the data transfer is desired. The git bundle process is a high-level, user-facing method for sending a Git database [7]. When invoked like

```
$ git bundle create my.bundle --all
```

Git will write a bundle file (*my.bundle*) which contains the complete history for a repository. The resulting file is very similar to a compressed Git database directory. Bundle files can be added and used like regular remotes, with the exception that they are read-only.

It is also possible to generate *incremental* bundles. Incremental bundles contain only a portion of the history: i.e., a range of commits. By limiting the amount of history to bundle, the resulting file size can be significantly reduced. The mechanism for specifying these limits requires a brief discussion of Git’s internals.

Approved for Public Release

Each Git commit contains the following information [8, p. 362]:

- An identifying Secure Hash Algorithm 1(SHA-1) *hash*, computed from its contents
- Metadata, such as author information and a message
- A snapshot of the versioned files: i.e., the contents of the version-controlled files, as of the commit
- Zero or more—but typically one—parent commits, identified by hash. Commits with more than one parent are a merge.

The last item relates each commit to the commit(s) which immediately precede it in the history. The complete history can be discovered by recursively following the parent commit pointer(s). Below depicted in Figure 1, the history traversal starts at commit **12aa**. It continues with **34bb**, following the arrows to discover each parent commit. The traversal ends when the last commit, **g00d**, is discovered. The **coolfeature** branch has not been merged into master. As a result, it is not discovered as a part of the history traversal. Git performs such a traversal each time it searches or displays the history, as is done in `git log` or `gitk`.

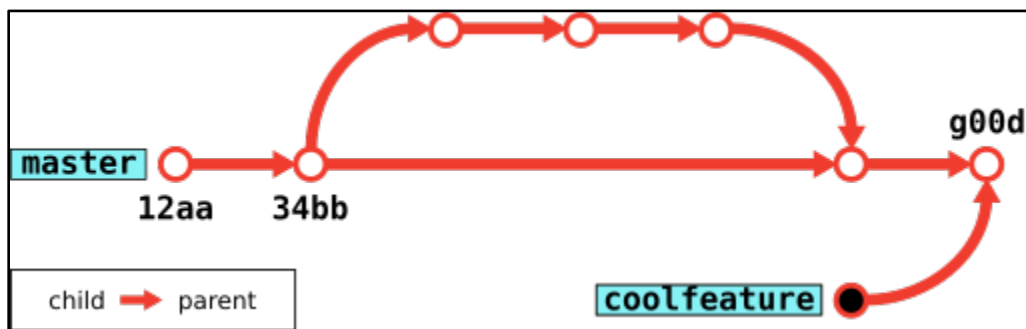


Figure 1. Example Git history traversal for the *master* branch.

For ease-of-use, commits which are important to the user are assigned symbolic names. These “references to commits” are known as refs [8, pp. 365–366]. Branches and tags are both represented as refs. For branches like **master**, the ref always points at the “head” (i.e., the tip) of the branch. Branch refs may change over time. To determine the history of a branch, Git starts from the head commit, which is pointed to by the ref, and then traverses backwards through the parents of each commit. Tag refs, such as **v1.0.1**, operate similarly.

This structure gives rise to two important requirements. These requirements are enforced by the Git executables.

1. All refs must point to commits which are known to Git
2. The parents of each commit must also be commits which are known to Git

In order for the chain of history to be complete and unbroken, both of these requirements must be met. These requirements are useful for incremental history transfers. As shown in Figure 2, if the remote side is known to have Commit **b33f**, then it is also known to have all commits which are parents of **b33f**. Because the remote must also have all of **b33f**’s ancestors,

it is known to have all the light-colored commits. The remote does not necessarily have the dark-colored commits. By omitting commits that the receiving side already has, it is possible to reduce the amount of data to transfer.

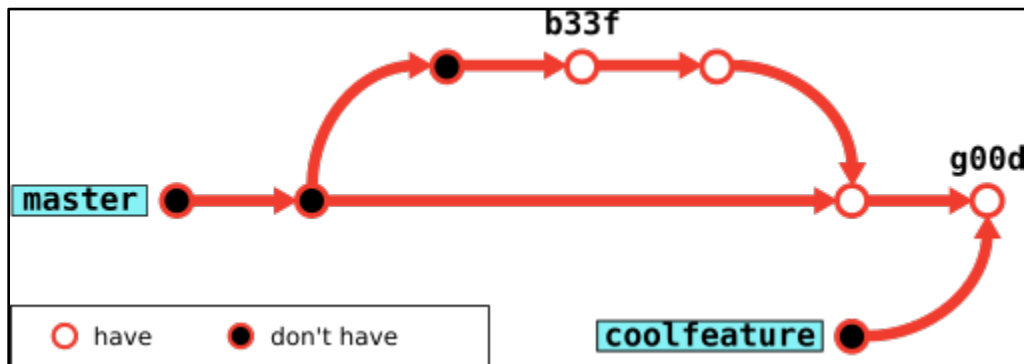


Figure 2. Comparing commits with a remote repository.

These rules are employed by the network/on-line download program, `git fetch`. In order to minimize traffic, the fetch program performs the following operations [8, p. 377]:

1. The uploading/sending side sends all refs which are available. Refs are represented as a list of SHA-1 hashes and symbolic names.
2. The downloading/receiving side sends:
 - the commit objects that it already *has*;
 - the commit objects that it is *missing*
3. The sending side then prepares and sends a data pack which contains the missing commits.

The key to this process is that the receiver must advertise the objects that it already has. As long as the sending side knows—or can make a reasonable guess—about the contents of the receiving repository, it can send a minimal amount of information. Although the fetch process is online, a similar process can be used offline.

III. Methods

The `git bundle` command can accept a list of refs to bundle. For example,

```
$ git bundle create my.bundle master
```

will output a bundle which contains the ref for the master branch and all commits that are on it. Bundles can also contain arbitrary ranges of commits, such as

```
$ git bundle create my.bundle master~3..master g00d~1..g00d
```

Commit ranges, such as the example above, will generate *incremental* bundles. Incremental bundles contain only a portion of the history. The example command outputs a bundle which contains:

Approved for Public Release

1. The last two commits on the master branch
2. The ref `master`
3. The commit `g00d`

When fetching from this incremental bundle, the rules in sec. 2 apply. The receiving side must have the *ancestor commits* of all the commits in the bundle. In this example, the receiver must already have `g00d~1`. If it does not, then it will be unable to import the bundle. It is also possible to specify refs by subtraction [9]. In Git, prefixing a ref with a caret (“^”) negates it. Executing

```
$ git bundle create my.bundle --all ^g00d ^b33f
```

will package all commits which are reachable from any ref (i.e., branches and tags), *except* for the commits `g00d`, `b33f`, and all of the commits which are reachable from either. The bundled commits are the same ones which are marked “don’t have” in Figure 2.

If it is known that the destination repository already contains `g00d` and `b33f`, then that part of the history can be excluded. This often results in a substantial reduction in bundle file size. With `git fetch`, this is accomplished via the “backchannel” from the receiving side to the sending side. For bundles, no such backchannel exists. It is necessary to make one.

Consider the scenario in Figure 3. Here, the repositories `source` and `mirror` are separated by a firewall and cannot communicate via any of the Git networking protocols. The goal is to send all refs from `source` to `mirror`. It is safe to limit the transfer to only ref’d commits, as these are the only parts of the history which are user-facing. The scope of the problem is also limited to *one-way* mirroring. The only source of commits and refs is `source`. The mirror is only updated during the mirroring process—and never at any other time.

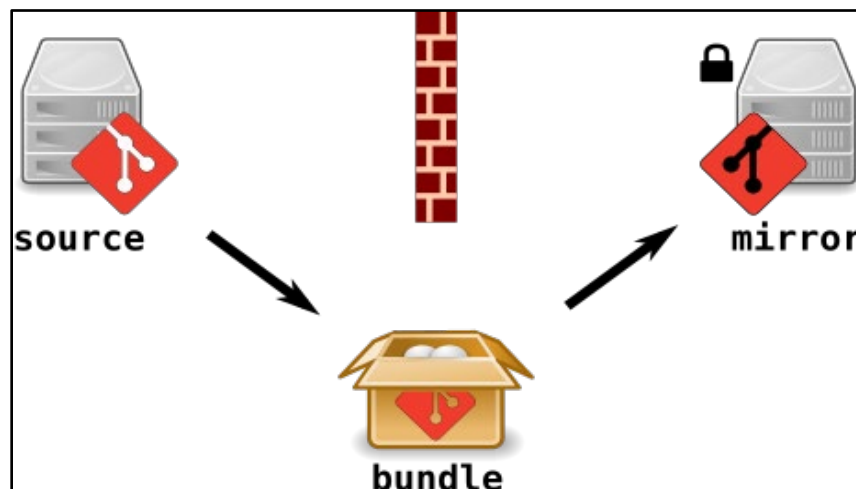


Figure 3. The Git-Teleport Scenario.

Approved for Public Release

Listing 1: Example incremental update process. The output of `show-ref` from the previous update is used to exclude commits from the current update. Hashes are truncated to four characters for display.

```
# initial bundle is generated with complete history
git show-ref
g00d... refs/heads/develop
b33f... refs/heads/master
f00d... refs/tags/1.0.1
git bundle create full.bundle --all

# hashes saved, full.bundle sent, and time passes ...
git bundle create incremental.bundle --all \
^g00d... \
^b33f... \
^f00d...
```

This scenario ensures that the only commits in `mirror` are those which are placed there, by the mirroring process, from `source`. The state of `mirror` will always be identical to the state of `source`, as of the last incremental update. This relationship allows us to easily generate bundles which satisfy the rules in sec. 2. The sender need only send commits which were not present during the previous update. The receiving side always has the commits from previous bundles. The sending side knows the commits that the receiving side has—because it put them there.

To generate minimal bundles, the sending side records the SHA-1 digest of all refs which were sent during the last update. The recorded digests are then provided as “negated” refs during the next update. An example of this process is given in listing 1.

This process is robust against branch creation, deletion, appending, and rebasing.

- When the source branch is unchanged, no new commits need to be bundled.
- When the source branch is deleted, it is no longer important and will not be bundled.
- When new commits are appended to the source branch, then all existing commits on the mirror branch are known and can be excluded. The incremental bundle will contain only the new, novel commits, `mirror/[branch]..source/[branch]`.
- If the source branch has changed completely, then the previous refs will exclude nothing. The entire branch will be bundled.

Although this process can be performed by hand, an automated tool can ensure that the operations are carried out quickly and accurately. The `git-teleport` tool implements the required storage and handles many other details.

IV. The Git-Teleport tool

The process described above requires persistent storage. The commits emitted in the previous bundle must be retained in between updates. The bundle files themselves do not need to be stored—only the commit hashes. Although many storage engines are available, the most straightforward and readily-available option is files on disk.

The git-teleport tool organizes transfers on a per-destination basis. The same repository may be sent to multiple destinations, and each destination requires its own storage. Modern software suites may be spread across several different repositories, and it is important to send updates for each simultaneously. git-teleport thus adopts a many-to-one approach: multiple repositories are bundled for one destination.

The state for each destination site is stored in a directory tree. This directory is referred to as a “Mirror Set.” Each Mirror Set contains:

- A catalog file, named `Beamdownfile`, which lists the repositories that are to be transferred
- A database, which provides the required persistence for refs
- A wrapper script, which finds and invokes the main git-teleport executable.

The Mirror Set is stored separately from the mirrored content—i.e., the repositories which are mirrored. Mirroring can be performed without modifying the source repositories. Any developer with read-only access to a Git repository can mirror it.

In large workgroups, several developers might need to prepare bundles. To ensure that these bundles are always consistent, git-teleport stores the Mirror Set information in its own git repository. The state of the mirroring process can easily be shared among an entire workgroup, if required. Any developer with sufficient access permissions can generate bundles. The Mirror Set git repository is only used to generate the bundles. It is never sent to the destination site.

To generate fresh bundles, a developer pulls the Mirror Set repository, runs git-teleport, and sends the generated files to the destination system. git-teleport commits the SHA-1 digest of all current refs to its database. The digests are obtained from `git show-ref`. These saved commits are automatically excluded from future bundles.

No special software is required on the receiving end. The output of git-teleport is a normal bundle file which can be fetched like any other bundle. git-teleport includes several commands which are useful for automating the receiving process, however—especially if many mirrors need to be updated.

During the mirroring process, it is essential to protect both the source and the mirror repositories from accidental corruption. To generate a bundle, Git must have filesystem-level access to the source repository. git-teleport protects source repositories by bundling them indirectly. Before bundling, the source repository is `--mirror` cloned locally. Bundles are made from this clone. This eliminates the need to use the repository server, or its filesystem, directly. Bundles can also be generated from sources which are only accessible over HTTP or SSH.

On the receiving end, it is important to protect mirrors from accidental modification. When the mirror is updated, any commits which are not present in the source may be summarily and irretrievably deleted. If a developer inadvertently pushes their work to the mirror, then that work could be lost. When creating receive mirrors, git-teleport uses hooks to deny these inadvertent pushes. git-teleport also ensures that the destination repository is actually a mirror—and not some other repository—before writing to it. These checks greatly improve the safety of the mirroring process.

V. Discussion

The git-teleport tool can automate the process of mirroring a repository from one DOD enclave to another. The mirroring process is only one part of an effective strategy for cross-organizational software development. In any large software project, team dynamics and human factors are just as important as—and are sometimes more important than—the technical aspects of the work. In addition to creating the software itself, software engineers must also create the machinery and processes by which the software is modified and released. The following sections discuss practical considerations for cross-organizational software development, with a focus on Git and git-teleport.

a) Maintenance versus Ownership

Software projects of all sizes tend to have two essential roles: maintainers and contributors. Maintainers can be management, such as Agile’s “product owner” [2], or they have a more technical background. Contributors are generally technical, but they need not be “coders:” software projects might employ technical writers, graphic artists, hardware engineers, and other domain-specific experts. In small projects, a single person might fill both or all of these roles.

Maintainers are responsible for setting the overall direction of the project. Direction is usually established by prioritizing new features and triaging “bug” or “problem” reports. A maintainer might review change requests to ensure that they are effective, well-written, and relevant. Maintainers are also responsible for publishing new *releases*, which are versions that are deemed suitable for production use.

Contributors, meanwhile, perform more of the technical labor of software development. Contributors propose changes to the source code or other project assets, either to introduce new functionality or to correct an existing problem. Users can also make contributions: most often in the form of feature requests or bug reports. The maintainers, however, can exert considerable influence over any contributions.

In open-source communities, projects often have no formal, legal owner—the code is, after all, typically free for anyone to take and use. The DOD, as well as industry, often takes a more formal view of ownership. Organizations assert ownership rights either by license, by Memorandum of Agreement (MOA/MOU), or both. In these situations, it is common for the owning organization to perform all maintainer activities. This can be counterproductive, however.

Projects which accept outside contributions will, over time, inevitably grow small ideas into larger subsystems. In this case, the developers who are the most familiar with these subsystems

are also the best equipped to maintain them. The organization which owns the software as a whole may lack the expertise or manpower to effectively manage these subsystems.

Open-source projects typically organize themselves around a “benevolent dictator.” The benevolent dictator has the final word on all disputes, but they are also subject to replacement or obsolescence if they do not act in the community’s best interest [10, pp. 42–44]. While the phrase suggests a centralized authority, a key trait of the benevolent dictator is the ability to delegate decisions to others [11] [12]. Many projects are simply too large for one person to micro-manage, regardless of ability.

For large projects, it may be prudent to cede the maintenance duties of a subsystem to those who are most familiar with it... even if they are from a different home organization. A top-level integrator, like the “benevolent dictator,” is still required to oversee the system as a whole. Without top-level integration, a project is merely a collection of systems which may or may not inter-operate.

b) The Contribution Process

Accepting contributions from external organizations is a challenging proposition. Development processes that depend on teams being co-located simply do not function. Friction between internal and external developers is likely to occur—especially in projects which have historically been closed to outsiders [13].

On the opposite side of the process, software projects often have high barriers of entry for newcomers. Incomplete documentation, unusual or unknown conventions, and unfamiliarity with subject matter can make it difficult for outside organizations to contribute. Adding a cumbersome or sluggish process for submitting changes tends to compound these issues.

Projects should have a well-defined, simple process for submitting and managing changes. This process is often called the “contributor guidelines” or the “change control process.” It might be documented in a MOA or in a less formal CONTRIBUTING guide. Ideally, this process should be followed for all changes—including changes from trusted internal contributors.

For the submitter, the contributor guidelines should define where and how code should be submitted, state any required formatting, enumerate testing and documentation requirements, and provide a clear pathway to acceptance. For the receiver, the contributor guidelines outline the reviews which will be performed and how decisions are reached. All changes, regardless of their source, should be peer-reviewed before being merged [13]. This is particularly critical in security-aware environments [14, pp. 12–13]. Once changes are submitted, they should be reviewed promptly.

Automated tests, such as test-driven development unit tests, can complement the manual review process [15, pp. 115–118]. Automated tools can also enforce a coding style, perform static analysis, and check for memory leaks or undefined behavior. Large projects, both open and closed, rely heavily on these tools to manage a constant deluge of change requests. If all contributors have access to these tools, then the number of iterations needed to obtain a usable change set is reduced.

After a change-control process is established, a project is ready to accept contributions. By sharing Git repositories, all contributors have ready access to the same decentralized code submission and review mechanisms that are used in linux kernel development. The following sections detail the process for submitting change requests, which are often called “patches,” via Git and git-teleport.

c) Git Branching Workflow

No discussion of Git is complete without an overview of *branching workflows*. The purpose of branching workflows is to represent the software release cycle in Git. Most projects, regardless of scale, publish released versions at regular intervals. A “release” is a version which is deemed suitable for general users— i.e., non-developers. Versions are typically assigned meaningful numbers [16] [17] for identification purposes. In Agile workflows, releases can occur once every few weeks. This makes it important to have a streamlined release process.

In branching workflows, explicit merges are used to accept commits into the main line of development. The branch(es) which contain a commit reflect both its review status and its acceptance into current or future releases. One of the more popular models, “git-flow,” uses three tiers of acceptance [18] [19]:

1. **Feature branches:** All new development begins as a branch off the main line of development.
2. **Integration branch:** After the feature has stabilized and has been accepted, it is merged into an integration branch in preparation for the next release. The integration branch is typically named “develop.”
3. The **master branch** is reserved for releases. Releases are made by merging develop into master. Every commit on master is a release. Version numbers are tagged for easy lookups.

Below depicted in Figure 4 is an example Git branching workflow. In this workflow, commits begin life on feature branches. They then progress “up” the ladder of stability over time. Commits which are merged into develop are staged for inclusion in the next release. Commits which are merged into master are part of a release: in this case, version 1.2.0. Every commit on master is a release, and annotated tags are version numbers.

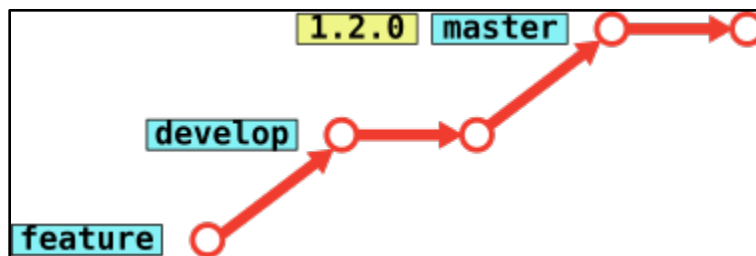


Figure 4. A Three-Tiered Branching Workflow.

Approved for Public Release

This workflow offers several key advantages over flat or unorganized repositories:

- The original commits from feature branches are retained. Each commit, and therefore each source line of code, can be traced back to its original author.
- Since merges are explicit, the approver's identity is also recorded.
- It is straightforward to determine the changes from release-to-release. This can aid in the generation of documentation, such as a “changelog.”
- Normal development can continue during the release process. Neither process has a major impact on the other.

The git-teleport tool is workflow-agnostic and can mirror any repository, regardless of workflow. Adopting a formal workflow greatly benefits remote contributors, however. Branch workflows allow remote contributors to see their progress towards inclusion in a release. When the maintainer merges the commit into `master`, the remote collaborator will know exactly when and what was merged. This level of visibility is very important when issue trackers or other documentation systems cannot be shared with remote users. Full instructions for establishing and using a branching workflow can be found in [18], [19], or elsewhere.

In situations where git-teleport is required, remote enclaves cannot push new features directly to the source repository. This is because they lack network access to the source repository. In these situations, remote enclaves must submit their changes via a *forking workflow*.

d) Git Forking Workflow

The git-teleport tool mirrors are necessarily read-only. They cannot be modified at the remote enclave. If the remote enclave has changes to contribute, then they must do so via a *forked repository*. In the forking workflow, a second read/write repository is created to contain changes for a read-only original. The forking workflow is commonly used in open-source communities [3, pp. 113–115].

A forked project starts with a clone of the “upstream” repository—i.e., the original source of the project. In Figure 5, each developer has two repositories available to them. Developers pull in upstream changes from `upstream`, which is a git-teleport mirror. New feature branches are stored in a local repository, named `fork`, where they can be tested and discussed. All new work contributed by this enclave is stored in `fork`. The `fork` repository allows multiple contributors to work any or all features at once. Feature branches are kept up-to-date by merging `upstream/develop` into them. Example commands for this are given in `lst. 2`.

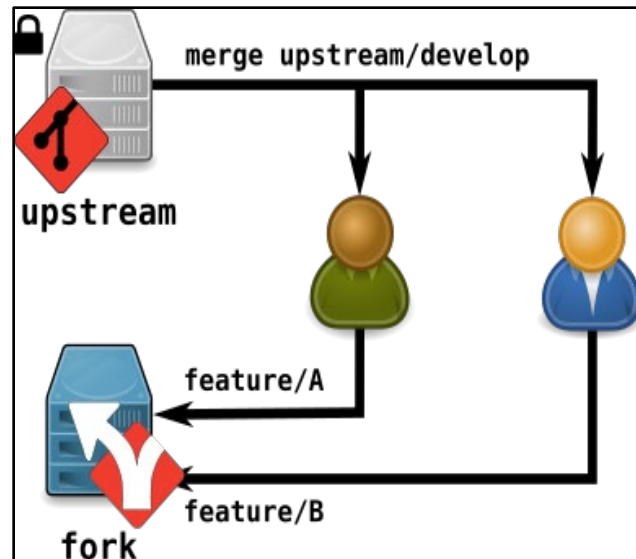


Figure 5. Forking Workflow

Listing 2: Example feature branch in a simple forking workflow. Here, there are two repositories: a read-only view of the upstream project and a read-write repository for local changes.

beginning a new feature branch

```
git checkout upstream/develop  
git checkout -b feature/A
```

share with other developers at the local enclave

```
git commit -m "My cool feature"  
git push fork feature/A
```

feature branch is kept up-to-date with updated upstream

```
git fetch upstream fork  
git checkout feature/A  
git merge upstream/develop  
git push fork feature/A
```

The fork described in Figure 5 lacks an explicit local integration branch: i.e., a “develop” or “master” branch. The expectation in this workflow is that all changes will be contributed, individually, back to the maintainer organization. The fork might require its own integration branch for testing multiple features at once. It is the feature branches, rather than this local integration branch, which should be submitted to the upstream project. It is easier to review smaller patches than larger ones, and most maintainers prefer to review a single, atomic feature at a time.

e) Submitting Patches to the Upstream

Over time, forks tend to diverge from their parent project. In the absence of regular communication, software engineers on either side of the project may elect to take radically different directions. The fork may diverge far enough that merging in a new version from the upstream may be challenging. If the merge is a complicated one, it will rarely—if ever—be performed. The end result is not one project... but two.

To keep both enclaves in sync, all changes made in the fork should be submitted back to the upstream project for review and eventual merging. While the git-teleport process is one-way, a one-way process in *each direction* allows for bidirectional synchronization. Figure 6 demonstrates a two-way process. In addition to the outgoing git-teleport, the forked repositories are returned back to the project's maintainer via a second git-teleport. The remote enclave, on the right side of the firewall, uses git-teleport to send their *forked* repository back to the project's maintainers. The maintainers can then checkout, examine, and comment on the changes. This process is very similar to making a “pull request” on Github.

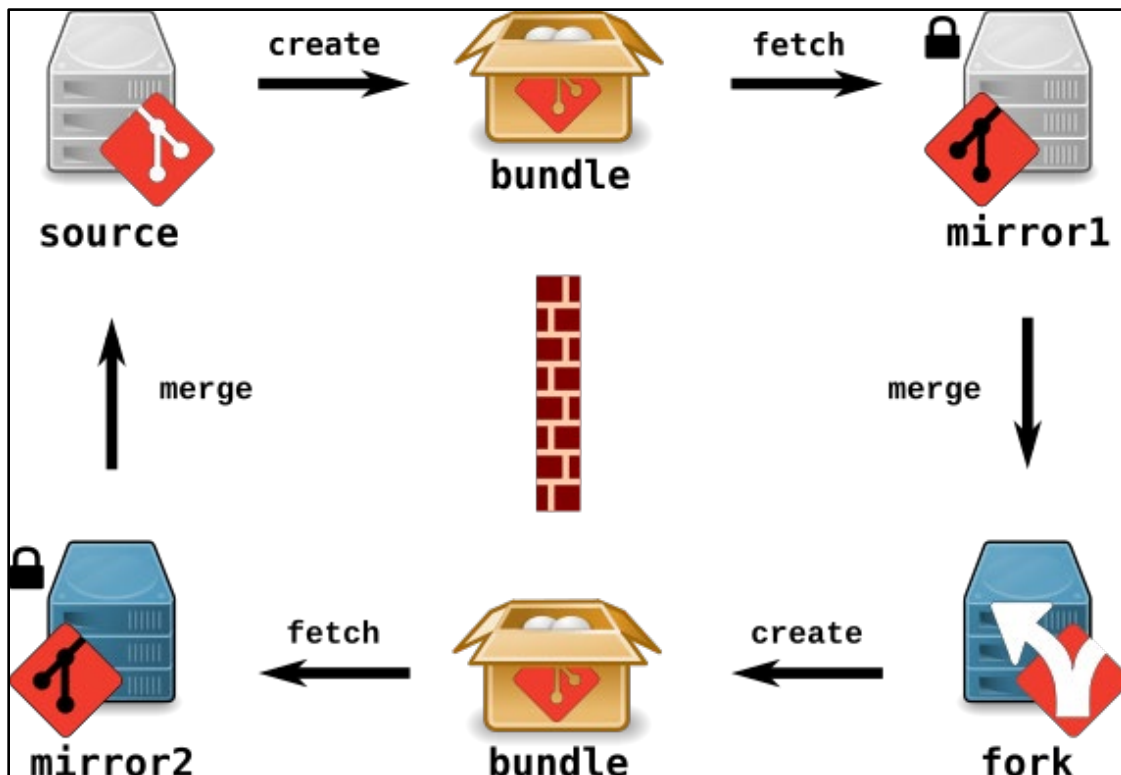


Figure 6. Two-Way Git-Teleport Scenario.

Alternatively, the remote enclave can send *patch files* to the maintainer. Patches, which are sometimes referred to as “diffs,” are a human-readable version of the changes. Read verbally, a patch file might say, “Go to line 102 in *main.c*. Change the line which reads ‘YYYY’ to ‘ZZZZ.’” Patches use a shorthand notation to express these instructions. Git contains two tools, *format-patch* and *am* (“apply mailbox”), for exporting and importing patches.

An example of the patching workflow can be found in Figure 7 and lst. 3. In this workflow, developers send patch files—and not bundles—back to the maintainers.

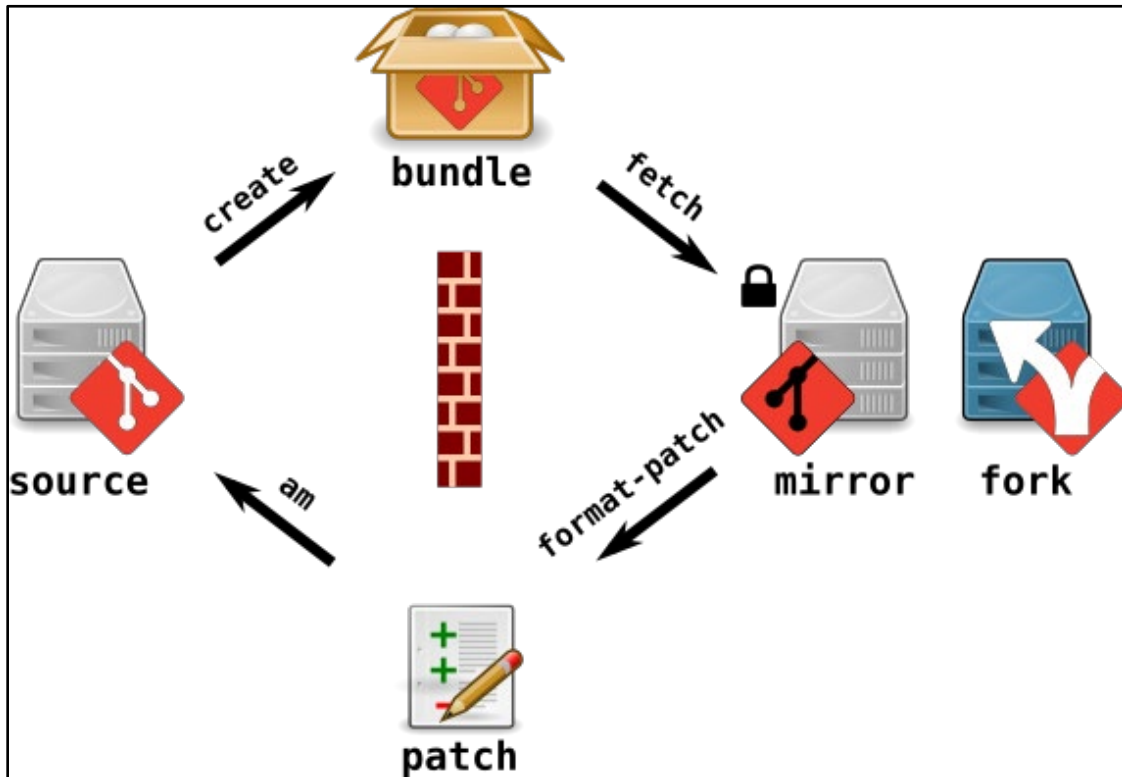


Figure 7. Git-Teleport with Patch File Submission

Listing 3: Example process for generating and applying a patch file. Patches of text files, which are commonly used for source code, are human-readable.

```
# generating a series of patches, as one file
git format-patch --stdout upstream/develop..feature/A > changes.patch

# on the receiving end, create a new branch from "develop"
# and apply the changes
git checkout develop
git checkout -b feature/A
git am changes.patch
```

Of these two methods, a two-way git-teleport is more straightforward. The main advantage of patches is that the output is a text-based format, which is completely human-readable. Patches can even be applied “by hand” if need be. Patches are also common for “many-to-many” communication flows. While git-teleport can only send a repository to a single destination, patches can be emailed to many recipients simultaneously. As long as all participants have the base commit, the patch will apply. Further reading on patch workflows can be found in [3, pp. 116–118].

VI. Conclusion

The decentralized design of the Git DVCS opens the doorway for collaborative software efforts. Software development can span DOD enclaves, departments, and even services. No longer is the version control history for a project tied inexorably to a single server in a single, physical location. With the novel git-teleport process, it is simple to send and share Git history between disconnected networks. The only requirement is the ability to send binary data in one direction, from sender to receiver.

The git-teleport tool can be used within an organization to update computers which lack an internet connection, such as test equipment in the field. It can also be used to link customer sites with limited or no VPN connectivity. git-teleport enables collaboration between DoD enclaves, and private industry, by enabling software developers to share source code without sharing infrastructure. The complete git-teleport example program is attached to this report in Appendix A.

As software projects grow in size and complexity, they can also gain additional stakeholders and users. This gain in mindshare is often accompanied by the need for increased communications, as bug reports and feature requests arrive at an accelerated rate. An open development model, properly implemented, can help stakeholders become full-fledged contributors to a project. Sharing code repositories, however, is only the first step.

An open development model is not as simple as opening a code repository and declaring that “patches are welcome.” To be successful, an open project must *empower* its users to become contributors. New contributors must eventually, though time and education, become familiar enough with the project to make bold and meaningful changes. Administrative and technical processes must align to integrate developers from other organizations. While the recommendations in this report are a good starting point, those who are considering open development should look to the many thriving open communities for guidance and advice on how to build their own.

Approved for Public Release

REFERENCES

1. (U) C. Norton, K. Mayfield, and R. Benito, “Handbook for Implementing Agile in Department of Defense Information Technology Acquisition,” DTIC, ADA546756.
2. K. Schwaber and J. Sutherland, “The Scrum Guide” [Online]. Available: <https://www.scrumguides.org/docs/scrumguide/v2017/2017-Scrum-Guide-US.pdf>
3. S. Chacon and B. Straub, “About Version Control,” in *Pro Git*, 2nd ed., Apress, pp. 1–4 [Online]. Available: <https://doi.org/10.1007/978-1-4842-0076-6>
4. Z. Brown, “A Git Origin Story,” *Linux Journal* [Online]. Available: <https://www.linuxjournal.com/content/git-origin-story>
5. R. Ross, P. Viscuso, G. Guissanie, K. Dempsey, and M. Riddle, “Protecting Controlled Unclassified Information in Nonfederal Systems and Organizations,” National Institute of Standards and Technology, Standard 800-171 [Online]. Available: <https://doi.org/10.6028/NIST.SP.800-171r1>
6. S. Chacon and B. Straub, “Bundling,” in *Pro Git*, 2nd ed., Apress, pp. 264–267 [Online]. Available: <https://doi.org/10.1007/978-1-4842-0076-6>
7. L. Torvalds, J. Hamano, and et. al., “git-bundle(1) Manual Page,” Man Page [Online]. Available: <https://git.github.io/htmldocs/git-bundle.html>
8. S. Chacon and B. Straub, “Git Internals,” in *Pro Git*, 2nd ed., Apress, pp. 357–388 [Online]. Available: <https://doi.org/10.1007/978-1-4842-0076-6>
9. L. Torvalds, J. Hamano, and et. al., “git-rev-list(1) Manual Page,” Man Page [Online]. Available: <https://git.github.io/htmldocs/git-rev-list.html>
10. H. Ingo, *Open Life: The Philosophy of Open Source*. Ingram.
11. E. Raymond, “Homesteading the Noosphere,” in *The Cathedral & the Bazaar: Musings On Linux And Open Source By An Accidental Revolutionary*, O’Reilly.
12. K. Fogel, “Social and Political Infrastructure,” in *Producing Open Source Software: How to Run a Successful Free Software Project*, O’Reilly.
13. K. Fogel, “Getting Started,” in *Producing Open Source Software: How to Run a Successful Free Software Project*, O’Reilly.
14. L. Conklin and G. Robinson, Eds., “OWASP Code Review Guide” [Online]. Available: https://www.owasp.org/images/5/53/OWASP_Code_Review_Guide_v2.pdf
15. R. Martin, “Testing Strategies,” in *The Clean Coder: A Code of Conduct for Professional Programmers*, Pearson.

Approved for Public Release

16. T. Preston-Werner, “Semantic Versioning 2.0.0” [Online]. Available: <https://semver.org/spec/v2.0.0.html>
17. M. Hashemi, S. LaPorte, M. Williams, G. Lefkowitz, A. Brown, and H. Schlawack, “Calendar Versioning” [Online]. Available: <https://calver.org/>
18. V. Driessen, “A successful Git branching model” [Online]. Available: <https://nvie.com/posts/a-successful-git-branching-model/>
19. S. Chacon and B. Straub, “Branching Workflows,” in *Pro Git*, 2nd ed., Apress, pp. 58–61 [Online]. Available: <https://doi.org/10.1007/978-1-4842-0076-6>

LIST OF ABBREVIATIONS, ACRONYMS, AND SYMBOLS

Department of Defense (DOD)

Distributed Version Control System (DVCS)

Memorandum of Agreement (MOA/MOU)

Secure Hash Algorithm 1(SHA-1)

Approved for Public Release

APPENDIX A
GIT TELEPORT TOOL SOURCE CODE

Approved for Public Release

Approved for Public Release

A.1. File Listing “README.md”

```
# GIT-TELEPORT
```

```
> `git push --mirror`... without networking
>
> Send your data to the Mirror Universe!
```

`git-teleport` mirrors Git repositories between sites which cannot connect directly via SSH, HTTPS, or any other Git networking protocol. If

- * you have a Git repository in one place,
- * you need it to be in another, and
- * you can't use the normal Git transfer protocols to get it there,

then git-teleport is for you. Git-teleport allows you to **mirror** one or more local repositories to a remote system that is not accessible over the network. Updates can be transmitted via any binary-safe medium, including "sneakernets" if need-be.

Git-teleport builds on and simplifies the existing "bundle" functionality that is built into Git. Multiple repositories can be mirrored en masse—reliably—with a single command. Git-teleport can update test equipment and computers in the field, without a working internet connection. It can also be used to collaborate across restrictive firewall environments.

THEORY OF OPERATION

Git allows you to make `bundle` files, which are binary archives of commits. Mirrors can `fetch` from these bundles, just as if they were regular remotes.

Complete bundles of some repositories are very large. This makes them inconvenient to transfer. Fortunately, bundles can be **incremental**: i.e., they can contain only a range of commits. As long as your bundle has commits in common with the mirror, then the mirror can `fetch` it and be updated.

Creating bundles which are minimal—yet still contain enough commits to do the job—is a non-trivial process. It requires a careful pruning of the history, which is difficult to do without knowing what the destination already has.

git-teleport solves this problem by keeping a record of all the "refs" it has already sent. Refs are named objects, like branches and tags. You can list the refs in any repository with

```
```bash
git show-ref
```
```

When selecting commits (and tags) to bundle, one can use an old list of refs to tell git: *"*Bundle everything EXCEPT commits reachable from X, Y, and Z.*"* If the destination repository already contains commits `g00d`, `b33f`, and a tag `1.0.0`, then you can exclude these with something like

```
```bash
git bundle create repo.bundle --all ^g00d ^b33f ^1.0.0 ...
```
```

See `git help rev-list` for more information about how commits are specified.

Git-teleport saves the output of `show-ref` to keep a record of what has already been transferred. It provides this list to future calls to `bundle`. This makes future updates much, much smaller. Git-teleport also provides automation for mirroring many repositories at once.

Approved for Public Release

BEFORE YOU BEGIN

You may want to run some variation on the below Git commands

```
```bash
basic configuration
git config --global user.name 'Iam Chroot'
git config --global user.email 'Iam.Chroot@mycorp.example'

if you connect via http(s), and you have no credential helper enabled
On linux, see the /usr/share/doc/git/contrib/credential directory
for additional credential stores. Git for Windows uses the Windows
Credential Store by default, so there is no need to set this.
git config --global credential.helper cache

if you connect via http(s), and you need to specify a username
git config --global \
 credential."https://internal.mycorp.example".username iamchroot
```
```

You will probably want the last two lines if you connect over HTTP(S). Turning on a credential cache will greatly reduce the number of times that you have to type your password. Since git-teleport URLs are shared, they do not have usernames in them. If your repositories are stored on an HTTP(S) server, you may need to specify your login username manually.

Git-teleport aims to be compatible with both bash on linux and "Git Bash" on Windows, and it should run in both environments.

INSTALLATION

Ensure that the git-teleport Git repository—i.e., **this** repository—is checked out to one of the following locations:

```
* `${HOME}/git-teleport`
* `${HOME}/.git-teleport`
```

Then add the following line to your `~/.profile` (preferred) or `~/.bashrc` (alternate) file to put `git-teleport` on your `\$PATH`:

```
```bash
PATH="${HOME}/git-teleport/bin:${PATH}"
```
```

These changes will take effect the next time you log in. Run `source ~/.profile` to set them immediately for the current shell.

You can execute the unit tests by running

```
```bash
test/run.sh
```
```

from the top-level `git-teleport` directory.

GETTING STARTED

The following sections will walk you through creating your first Mirror Set and generating your first set of bundles. If you already have a Mirror Set that you want to use, skip to "[Packing the Bundles](#)".

Bootstrapping

Approved for Public Release

In order to provide fast, incremental updates, git-teleport must keep a history of what has already been sent. git-teleport stores its database files in (where else?) a Git repository. This repository is largely managed by git-teleport, and it is referred to as the Mirror Set. The Mirror Set repository is **NOT** transferred to the destination system.

To bootstrap git-teleport, pick a non-existing directory `BDIR` on your workstation, and run:

```
```bash
git teleport bootstrap BDIR
```
```

A new Git repository will be created in that location.

Create a unique Mirror Set for each of your destinations. **Never** re-use a Mirror Set for different destinations; make a new one. This is because git-teleport needs to remember what the destination side has received, and it can't do that if you mix-and-match destinations.

Defining What to Mirror

The newly-bootstrapped repository has a text file, `Beamdownfile`, which is used to define the list of repositories to bundle. Let's assume that you want to send three repositories, and they reside on two different servers:

1. `https://extranet.mycorp.example/git/ProjectAlpha.git`
2. `ssh://git@internal.mycorp.example/ProjectBravo.git`
3. `ssh://git@internal.mycorp.example/mysubfolder/ProjectCharlie.git`

The first lines in your `Beamdownfile` should be:

```
```txt
BEAMDOWNFILE
...
Comments get ignored
...

root definitions
! https://extranet.mycorp.example/git
! ssh://git@internal.mycorp.example
```
```

Lines with a leading exclamation point (`"!`) are **repository roots**. These root definitions list the servers and path prefixes where repositories are hosted. List all the servers you intend to use in this `Beamdownfile`. Exclude subdirectories which are important to your project structure, such as "mysubfolder" in the last URL. "mysubfolder" is **not** part of the root definition.

Next, list the full absolute URLs of all the repositories you want to mirror.

```
```txt
repository list
https://extranet.mycorp.example/git/ProjectAlpha.git
ssh://git@internal.mycorp.example/ProjectBravo.git
ssh://git@internal.mycorp.example/mysubfolder/ProjectCharlie.git
```
```

When git-teleport generates bundles, it assumes that the destination system will have mirrors for them at

Approved for Public Release

```
```txt
${SOMEWHERE}/ProjectAlpha.git
${SOMEWHERE}/ProjectBravo.git
${SOMEWHERE}/mysubfolder/ProjectCharlie.git
```
```

where `\${SOMEWHERE}` is an arbitrary location on the destination system.
The `\${SOMEWHERE}` will be selected later, when the bundles are unpacked.

The repository root definitions ("!") exist to separate *physical* storage—i.e., where the repositories are kept—from their *logical* storage. Logical storage is the sub-folder hierarchy that is important to your project. As shown above, git-teleport will preserve subfolders which are not matched by the root definitions.

The repository roots must be a prefix match on the URL. The match must be byte-for-byte, including case. All entries must have a matching repository root definition.

You may use any URL scheme supported by git, including `file://`. If a unique username is required, always set it in your `git config` as described above. Only include usernames if they are common to all clients, like "ssh://git@mygitlab/repo.git". Common usernames are typical of hosting software like Gitlab. You can use "SCP syntax" too: "`git@mygitlab:repo.git`".

Git-teleport is not a package manager and is not aware of submodules. If your repository depends on other repositories, it is your responsibility to include all of them in your Beamdownfile.

When you are finished, save this file and commit your changes.

The Git-Teleport Wrapper

Each Mirror Set has a `teleportw` script. This is the git-teleport wrapper. It is a lightweight script which runs the main shell script wherever you have installed it: see "`INSTALLATION`" above. Its purpose is to help keep your various Mirror Sets separate from one another.

You can run the wrapper script from any directory. It will always update the Mirror Set in which it lives.

Packing the Bundles

Simply `cd` to the Mirror Set repository and run

```
```bash
./teleportw pack
```
```

and new bundles will be output to `bundles/`. A new sub-directory of `bundles/` is created for each and every update, but the current bundles are always available in `bundles/latest`. This directory is a symbolic link on linux and a copy on Windows.

Only repositories which have updated will generate new bundle files.

Collect the latest bundle files together for transfer to the destination system. Use any binary-safe file transfer method, such as email, shared folders, USB drives, optical media, or carrier pigeons. You need *all* the files in `bundles/latest`.

If the bundles are too large for your chosen transport mechanism, consider splitting them like this:

Approved for Public Release

```
```bash
cd bundles && \
 tar --dereference --create 'latest' | split -b1024M - allbundles.tar.
```
```

which splits the bundles into 1024 MiB files. Split files can be reconstituted later with

```
```bash
cat allbundles.tar.* | tar -xv
```
```

Once you have finished transferring the bundles, it is safe to delete them. You can remove the entire `bundles/` directory if you want.

If you are sharing your Mirror Set, you should push up your changes so that others will see what the latest bundle should contain.

The `pack` command is the combination of three other commands:

```
* `fetch`
* `bundle`
* `commit`
```

You can read more about these commands on their respective help pages.

If you examine the output `bundles/latest` directory, you will notice that git-teleport has removed all special characters and converted all path-names to lower-case. `Alpha.git` becomes `alpha.bundle`. This is to support deficient filesystems which do not adequately support these characters.

In addition to the `.bundle` files, you will find a matching `.bundle.refs` file for each. This is the `\*refs bundle\*`. It contains only commits which are refs: i.e., tags, branch heads, and other commit references. This additional file is used to prune the destination repository of refs which no longer exist, such as merged topic branches.

Install Git-teleport on the Destination System

Ensure that the destination system has `git-teleport` installed. You can do without it if necessary, but it will make things much easier.

Creating New Receive Mirrors

The content in the Git bundles needs to be stored "somewhere" on the destination system. That "somewhere" is a new Git repository that is dedicated strictly to mirroring the repositories from the sending side. These repositories, which are called **Receive Mirrors**, are ***read-only*** mirrors. They are intended to represent the remote repositories exactly, ref-for-ref, just as if you had been able to `clone` them down.

You can keep your Receive Mirrors anywhere that Git can access, including your local disk, a shared filesystem, or behind a git remote accessible via SSH or HTTP(S). Wherever you keep them, you will need to set aside space in your hierarchy for these mirrors. If you usually store your repositories on NFS in `/net/git`, and the mirrored repositories originated from your corporate intranet, then a path of

```
...
/net/git/mirrors/intranet
...
```

Approved for Public Release

might be appropriate.

Here, the `mirrors` directory is dedicated to Receive Mirrors *only*. This helps keep them separate from your normal, writable repositories that include original commits and content. The `intranet` subdirectory further segregates these mirrors from repositories that originate from other places, such as your corporate `partner`s.

Once you have selected a storage location, you will need to create the Receive Mirror repositories.

Examine the directory of bundle files you created during the previous step. (If you compressed them, extract them.) In the root directory of the bundles, you will find a `Beamupfile`. This text file is automatically generated by `git-teleport`. Continuing with our example from above, you will find the following entries:

```
```.txt
GIT-TELEPORT BEAMUPFILE
...
This file was automatically generated by git-teleport.

ProjectAlpha.git
ProjectBravo.git
mysubfolder/ProjectCharlie.git
```.
```

This file lists the repository names you must create. The names for these repositories on your system must match these strings *exactly*, including case. Continuing with the above example, you will need to create:

```
* `/net/git/mirrors/intranet/ProjectAlpha.git`
* `/net/git/mirrors/intranet/ProjectBravo.git`
* `/net/git/mirrors/intranet/mysubfolder/ProjectCharlie.git`
```

Your repositories may or may not have the ".git" extension—this extension is optional. Use the same convention as the sender.

The creation instructions are different for repositories stored on your local filesystem versus on SSH/HTTP(S) remotes. Follow the instructions in the applicable sub-section.

Receive Mirrors on Local Filesystem

This includes shared filesystems, like NFS or SMB.

```
```.bash
git teleport skeleton \
 /net/git/mirrors/intranet/ProjectAlpha.git \
 /net/git/mirrors/intranet/ProjectBravo.git \
 /net/git/mirrors/intranet/mysubfolder/ProjectCharlie.git
```.
```

The `skeleton` command will create the above repositories, if they don't already exist, and configure them for use with git-teleport. This script takes the following actions:

1. Creates the repository in `--bare` mode
2. `git config core.sharedRepository true`, which helps with linux permission issues on shared folders
3. Adds a `pre-receive` hook which denies all pushes. Git-teleport always updates local repositories by fetching into them, rather than pushing to them. This hook prevents accidental pushes.

Approved for Public Release

4. Adds an ``teleport-mirror`` file to the root of the repository directory. Before updating a local repository, `git-teleport` will check for the presence of this file. This safety check prevents `git-teleport` from overwriting original content.

If this command succeeds, you are ready to go.

Receive Mirrors on Remote Server

You can store your repositories on a "smart" server, like Gitlab, or on a "dumb" directory over SSH. The important difference from "local filesystem" mode is:

> ****THE NAMING CONVENTION IS MANDATORY****, and `git-teleport` will enforce it.

- * ✓ ``git@mygitlab:mirrors/intranet/ProjectAlpha.git``
- * ✓ ``https://mygitlab/git/mirror/intranet/ProjectAlpha.git``
- * ✗ ``git@mygitlab:mystuff/ProjectAlpha.git`` (invalid)

Your Receive Mirrors must be stored in a subdirectory named either `"`mirror`"` or `"`mirrors`"` (plural), exactly. It does not matter which path component is named `"`mirrors`"`. Regardless of which variation you choose, it is **vital** that you keep **only** Receive Mirrors in the ``mirrors/`` directory.

Beyond this requirement, it is up to you to create these repositories. Everyone who updates bundles must have write permission.

We also recommend that you use a subdirectory like ``mirrors/intranet``, in case other sites also start sending you teleport bundles.

Smart repository hosts might restrict push access to certain branches, like ``master`` or ``develop``. This will prevent you from receiving updates. To avoid these restrictions on Gitlab, ensure that everyone who updates bundles has **maintainer access** or higher on ``mirrors/``. Ensure that branch write protection is turned off.

Unpacking the Bundles

Now that your Receive Mirrors exist, it is time to put them to use.

```
```bash
git teleport unpack path/to/bundles /net/git/mirrors/intranet
```
```

The first argument is the path to the directory of bundle files. This directory must contain a ``Beamupfile``.

The second argument is the URL or path prefix where your Receive Mirrors are stored.

When this command completes successfully, the repositories in ``mirrors/intranet`` will be exact duplicates of the sending side. They will have all the tags, all the branches, and all of the other refs, too—like Gitlab merge requests.

THINGS WHICH CAN GO WRONG

- * If the sending side added a new repository, the ``unpack`` will fail. Create the missing repository and run the ``unpack`` step again. The ``unpack`` command is safe to re-run.
- * If you have multiple updates to apply at once—perhaps because you let them accumulate without applying them—be sure to apply them in the order in which

Approved for Public Release

they were generated. Otherwise, the `unpack` will fail.

* If a bundle won't apply, run `git bundle verify` on the bundle file. Determine the commits you are missing. You will probably need the sender to generate new bundles. You can:

- Force full bundles to be generated with `./teleportw fetch`, followed by `./teleportw bundle-all`; OR
- Revert the Mirror Set (on the sending side) to a previous version and try normal incremental bundles again.
- Have the sender check and be sure that the bundle contains the required commit(s)

* Whenever git-teleport needs to `fetch` a remote repository, it caches it locally. This allows git-teleport to be used with network-based remotes. On linux, caches are stored in `~/cache/git-teleport/`. Cache files are transient, and you may delete them to save space.

* Git-teleport generates two different bundle files: the incremental update (`.bundle`) and the refs bundle (`.bundle.refs`). The first bundle is used to update changed refs. The second is used to delete (or "prune," as git calls it) refs which are no longer in the source repository. If git-teleport fails to generate a refs bundle, it will print a warning and continue. If you see this warning, please report this as a bug.

OTHER CONSIDERATIONS

The source and destination systems will almost certainly use different URLs and paths to refer to the same content. To be portable, your software must use **relative paths** or determine paths from the environment. If your project uses `git submodules`, then your submodules should use **relative paths**.

On the sending side, you can switch submodules to relative urls by editing your `.gitmodules` file and replacing absolute URLs with relative ones. A `git submodule sync` will bring your checkouts up-to-date.

If you have received submodules which use absolute paths, you can use Git's internal URL rewriting functionality to fix them:

```
```bash
you might want to add --global
git config \
 url.'file:///net/git/'.insteadOf \
 'https://some.bad.absolute.url/'
```
```

With the above configuration directive in place, URL prefixes which match `https://some.bad.absolute.url/` will be automatically replaced with `file:///net/git/` instead. This works for git submodules or any other attempt to `git clone` the given URL. It can also be useful for package managers which use `libgit` or the `git` executable to clone repositories. Rust's `cargo`, for example, can use dependencies cloned from `git` repositories.

When mirroring software repositories, it is also necessary to mirror any of your software's **dependencies**. If your software depends on package servers, like apt, yum, PyPI, or Cargo registries, then you may also need to mirror this content. If your software depends on database servers, CI/CD servers, license servers, shared folders full of files, or other network **infrastructure**, then you may need to mirror these services as well. Mirroring other types of services is beyond the scope of this tutorial.

Approved for Public Release

Be especially careful of forcing dependencies on IDEs, such as Eclipse or Visual Studio, on your downstream systems. When possible, use build systems which are not tied to a particular IDE. You can use `cmake` instead of Eclipse projects or `gradle` instead of IntelliJ IDEA projects.

FORKS OF GIT-TELEPORT MIRRORS

As mentioned above, git-teleport Receive Mirrors are ****READ ONLY****. You ****MUST NOT**** make any commits to git-teleport mirrors.

If you want to make changes to a mirrored project, you need to **fork** it. "Forking" a repository creates a new one with the same commits as the upstream project. This process is identical to that used by Github et. al.: A copy of the original repository is made. This copy, called the "fork," is a place where you can make your changes. Forks are kept up-to-date by manually merging in changes from the original repository, which is called the "upstream."

Since the conflict resolution process needs a man in the loop, git-teleport cannot assist you in managing your forks. The following guide will help you get started.

Creating the Forked Repository

Make a new server-style repository for your forked changes. It will start empty.

```
```bash
cd /net/git
```

```
--shared ensures your repository is always group read-write
git init SomeRepoFork.git --shared --bare
```
```

Always create the fork by hand or via your repository hosting software. Never create forks with git-teleport.

Working Copy: Where Merges Happen

Git merges require a working copy, so let's make one of those.

```
```bash
cd "$SOMEWORKDIR"
git clone file:///net/git/SomeRepoFork.git
git remote add upstream file:///net/git/mirror/intranet/SomeRepo.git
git remote set-url --push upstream /dev/null
```
```

The working copy now has two remotes:

- * ``origin``, which refers to the forked repository. You will use this remote on a day-to-date basis.
- * ``upstream``, which points to the git-teleport mirror from the remote system. This is a read-only mirror. Like a Github project which belongs to someone else, you can't change it. The ``set-url`` command above will ensure that you cannot accidentally push to ``upstream``.

Alternatively, you can configure git-teleport to automatically push branches of the upstream repository into your mirror. See ``Automatic Updates to Forks`` below.

Branching Workflow

These instructions assume that your repository uses the ``git-flow`` branching

Approved for Public Release

workflow. In the ``git-flow`` workflow,

- * The ``master`` branch is only for releases. Every commit on ``master`` is a release.
- * The ``develop`` branch contains all changes which are planned for the next release. This is an integration branch. Most developers use the ``develop`` branch for day-to-day tasks.
- * Individual topic branches are named ``feature/something`` or ``bugfix/something``. These topic branches contain a single, well-defined, feature or idea.

These instructions are written around the `git-flow` branching workflow. If your projects are using a different branching convention, it is best to ask the maintainer how to proceed.

Forking Workflows

There are two main forking workflows to consider. The choice of workflow depends on your application.

Remote Contributor Workflow

In the remote contributor workflow, you want to make a change to the project and then **submit** it back to the upstream repository's maintainer. These types of forks are often temporary and are only used to test and review the changes. The ultimate goal is to have these changes merged into the upstream project.

Your forked repository might be temporary—maybe even local to your ``$HOME`` directory. It might also be shared among other users at your site.

Start by creating a new **feature branch** on top of ``upstream/develop``.

```
```bash
cd "$SOMEWORKDIR"
git fetch upstream
git checkout upstream/develop
git checkout -b feature/mycoolthing
... do work ...
git commit
git push --set-upstream origin feature/mycoolthing
```
```

When your feature branch is ready, you may push it to ``origin``. This allows you to share the branch among others at your site. From there, you can send the branch back to the upstream repository's maintainer for review and eventual incorporation. Instructions for doing this can be found in the `"`PATCH SUBMISSION`"` section.

For purposes of review and integration, it is best to limit your feature branch to a single, well-defined feature or fix. If you need to test multiple feature branches at once, you can create a local integration branch which combines them. You should always submit individual feature branches—rather than combinations of many features—back for review. Smaller patches are easier to review—and are correspondingly much more likely to be accepted—than large patches.

Sometimes, the remote repository will update before you have finished with your feature. To keep your feature branch up-to-date, merge ``upstream/develop`` into it.

```
```bash
git fetch --all
git checkout feature/mycoolthing
```

## Approved for Public Release

```
git merge upstream/develop
git push
```
```

Diverging Project Workflow

The Remote Contributor Workflow is much simpler, but it assumes that all changes will be submitted back to the upstream project. Sometimes, it is desirable to integrate and release your own version of the mirrored software. Perhaps you need to make changes which are very specific to your site, such as adding a plugin or changing a configuration file. In these cases, it can be appropriate to create a permanent fork for your site.

In this workflow, your site will maintain and release its own, distinct version of the project. Not all of the changes from your fork will be submitted to the upstream project. This will increase your maintenance burden, but Git can allow such forks to exist if you are careful.

If you are using the ``git-flow`` tool, then run

```
```bash
git flow init -d
```
```

to set up the ``master`` and ``develop`` branches. This repository deliberately starts empty. We will populate it with content during the next step.

In this type of fork, you will probably take **released versions** from the upstream repository and use them as the basis for your releases. The merging strategy of choice may be to merge ``upstream/master`` into ``origin/develop``.

```
```bash
git fetch --all
git checkout develop
git merge upstream/master
```
```

If you have just started a new fork, you will probably need to add the ``--allow-unrelated-histories`` option to the ``merge`` command.

The merge step might require you to fix merge conflicts. If you need to fix conflicts, do so. Then run ``git commit`` to finish up.

When you are satisfied with the results, run

```
```bash
git push --set-upstream --follow-tags origin develop
```
```

This merging strategy will ensure

- * That your fork contains all **commits** and **tags** from upstream, commit-for-commit.
- * That the time at which you updated your fork is always represented in your history
- * That merge conflicts between your project and the upstream are immediately visible and apparent.

Your fork will have its own master branch, releases, and tags. Be sure to pick a tagging convention which does not conflict with the upstream. We recommend the use of the `"`+maintainer`"` version suffix, such as ``1.0.3+mysite``. You can also suppress the tags from the upstream repository and create your own.

Approved for Public Release

To update your fork later, obtain a new version of the upstream project with git-teleport. Then merge its master branch into your develop branch, as before:

```
```bash
git fetch --all
git checkout develop
git merge upstream/master
git push
```
```

Your project may use different branches than this example, but the process is largely the same.

Automatic Updates to Forks

This step is optional, but it greatly simplifies the process of keeping your forks up-to-date. It is also very useful for submodules because both mirrored and forked commits can be sourced from one repository.

git-teleport ships with a ``post-receive`` [githook] (<https://git-scm.com/docs/githooks>) script which can perform automatic updates. If you

- * ✓ are using the standard directory layout for mirrors and forks; AND
- * ✓ have direct filesystem access to your git-teleport Receive Mirror and Fork

then you can enable our hook to have branches from the upstream automatically brought into your fork repository.

This procedure only works if you have direct filesystem access to your repositories. If you are using a server solution like Gitlab, you will need to arrange for your fork to be updated some other way. Most versions of Gitlab support automatic mirroring.

With our hook installed,

- * if the mirror is located in ``/git/mirror/intranet/SomeRepo.git`` and
- * if a repository exists in ``/git/SomeRepo.git``, then it is assumed to be a fork; and
- * git-teleport will automatically push upstream branches to branches named ``intranet/...``.

If the upstream contains branches named ``master`` and ``develop``, then the fork will contain branches named ``intranet/master`` and ``intranet/develop``. Tags are not transferred by default.

****WARNING****: Our ``post-receive`` hook will only work "out of the box" if you are using our standard layout for mirrors and forks. The ``post-receive`` hook will **forcibly push** to the destination repository. If your Receive Mirrors are called `"`intranet`"`, then any branches under ``intranet/`` will be unceremoniously and permanently updated to match the upstream. Branches which do not exist on the upstream will get removed. Ensure that your developers understand how this works.

To install the ``post-receive`` hook, run something like this:

```
```bash
in the Mirror Repository...
cd /git/mirror/intranet/SomeRepo.git

examine to make sure this does what you want
less hooks/post-receive.mirror.sample
```

## Approved for Public Release

```
install it
mv hooks/post-receive.mirror.sample hooks/post-receive
chmod g+rx,a+rx hooks/post-receive
```
```

If you are missing the `post-receive.mirror.sample` file, run

```
```bash
git-teleport skeleton /git/mirror/intranet/SomeRepo.git
```
```

and it will be installed. Then try again. Avoid confusing this file with `post-receive.sample`, which is a sample provided by Git and not git-teleport.

The stock script doesn't work on CentOS 6, and it may not support all use-cases. If you want it to do something else, modify it or write your own! If you want tags, you can change the `git push` invocation to have `--tags` instead of `--no-tags`.

To test the `post-receive` script, perform a dry run of it like so:

```
```bash
note: the working directory matters!
cd /git/mirror/intranet/SomeRepo.git
DRY_RUN_POSTRECEIVE=y ./hooks/post-receive
```
```

If this will do what you want, run it for real:

```
```bash
./hooks/post-receive
```
```

From now on, `git-teleport` will invoke this script every time it updates the mirror.

PATCH SUBMISSION

Everyone hates merge conflicts. If there are many merge conflicts during an update, it indicates that the fork has diverged radically from its upstream. The conflicts will only worsen as time progresses. The problem: the outside world is completely unaware of your changes.

To reduce merge conflicts, it is necessary to have your changes sent back to the upstream project for inclusion. This is a *mandatory* part of the Remote Contributor Workflow, and it is a good idea to use in the Diverging Project Workflow, if you can.

In disconnected environments, the best way to send changes is to use Git's patch submission mechanism. Patch files are a set of instructions for making changes to a document: i.e.,

> go to line 107, strike the word "is," and replace it with the word "was."

These instructions are both human- and machine-readable, which makes them an ideal way to review patches. Patch files are a text-only format, so they can even be printed on paper.

Git can create patches with the `format-patch` command and apply them with the dubiously-named `am` command. "am" stands for "apply mailbox." (Patch files are frequently sent over email.)

Preparing a Patch

Approved for Public Release

Let's assume you have a feature branch named `feature/mycoolthing`, and you want to submit this branch back to the upstream project.

First, ensure that your branch is rebased or merged to be "on top" of the latest `upstream/develop`. It is important to use the upstream project's latest develop branch, as this will result in fewer changes and conflicts for the project's maintainers.

```
```bash
git format-patch upstream/develop..feature/mycoolthing
```
```

This will write one or more `.patch` files to your working directory. You can concatenate these into one file, if you want. The patch files contain instructions for reproducing `feature/mycoolthing` from the present state of `upstream/develop`.

When emailing patches, be advised that many email clients and servers will "helpfully" alter (read: destroy) the content of your messages. Exchange, for example, is particularly bad about altering line endings—even in attachments. Thunderbird will re-wrap lines in your message body. `git send-email` and `git imap-send` can both prepare email messages for you if you have SMTP or IMAP server access.

Printing Patch Files

Sometimes, you may want to print a patch file for purposes of review. You might also need to apply the patch by hand. When printing, use an editor which supports syntax highlighting for patch files, and print in color. Use a readable, monospaced font.

To reduce the size of the printed document, try generating a patch with `--ignore-all-space` and/or tweaking the number of context lines with `--unified=N`.

Applying a Patch File

After you have transferred the patch file to the destination system, you can apply it with the `am` command. First, create a branch to hold your patch.

```
```bash
git fetch
git checkout the_commit_your_patch_is_based_on # usu listed in the patch
git checkout -b feature/mycoolthing
```
```

Then run

```
```bash
git am /path/to/patch/files/*.patch
```
```

to import and apply the patch. You can then make further changes. Follow your project's normal review process for code submissions.

When you complete this process, your forked changes will no longer be necessary. Simply send an update with git-teleport and start using the new version from the upstream project. Your feature branch (`feature/mycoolthing`) is no longer necessary, and you can delete it.

TWO-WAY FLOWS

Approved for Public Release

Git-teleport only works in one direction at a time. With that said, you can book a round-trip flight for your data by purchasing two separate one-way tickets. In *two-way* development, each site has

1. their own read/write "working" repositories; and
2. mirrors of the remote side's repositories.

In this scenario, each site

1. sends out its working repositories with `git teleport pack`; and
2. receives the remote updates with `git teleport unpack`.

If you have a fork that you would like to send back to the maintainer, simply add it to an git-teleport Mirror Set and send it on its way.

In this type of deployment, it is necessary to designate one site as the maintainer and the other site as the fork. This designation can be different for each repository. This mainly determines which site is responsible for performing the final integration work and making (and tagging) *releases*.

TIPS FOR DISTRIBUTED DEVELOPMENT

You can help avoid merge conflicts and nasty, nasty bugs by adhering to the following recommendations:

1. **Submit** all relevant changes to the upstream project.
2. Keep patches **small**. Massive formatting changes will quickly wreck a project's merge-ability. Review your changes with `git diff --patch ...`. If you don't absolutely need to touch that line of code, leave it alone.
3. Keep topics **isolated**. If your changes do two separate things that do not depend on one another, consider splitting them into two separate feature branches. This allows the maintainer to merge what is ready and to iterate on the rest.
4. Start your topic branches from the **right place**. If your site is the maintainer, topic branches should probably start from your `origin/develop` branch. If your site is the fork, topic branches should probably start from `upstream/develop`.
5. Exchange bundles as **frequently** as possible. More frequent integration is much easier to do: there is less to merge at once, and everyone remembers what they've been doing.
6. **Talk** about the changes you plan to make, especially if they are big or breaking.
7. Test **ferociously** until there is nothing left to test. Write simple test-cases and automate their execution.

A.2. File Listing "bin/git-teleport"

```
#!/bin/bash
#
# git-teleport: Make minimal Git bundles for sneakernet updates
#
# You can source this script to use it as a library of functions,
# or you can run it to get its executable behavior.
```

```
GTELEPORT_VERSION="1.0.0"
GTELEPORT_EXEC_NAME="${GTELEPORT_EXEC_NAME:-git teleport}"
```

Approved for Public Release

```
# Regular expression which all Receive Mirror URLs used by
# `git teleport unpack` must match. The repository's URL
# *must* have a non-terminal directory component which
# matches this expression.
GTELEPORT_MIRROR_URL_REGEX=" ${GTELEPORT_MIRROR_URL_REGEX:-^mirrors?\$}"

# If empty, support for "--ignore-missing" and "--stdin" in
# git rev-list will be autodetected. If "yes", support for
# these features is assumed. If "no", these features will
# not be used. See gteleport_git_has_new_rev_list().
GTELEPORT_GIT_HAS_NEW_REVLIST=" ${GTELEPORT_GIT_HAS_NEW_REVLIST}"

function gteleport_rund() {
    # Usage: gteleport_rund DIR ARGS...
    #
    # Runs the command specified by ARGS in the given DIR.
    # Relative DIRs are interpreted relative to the current
    # working directory.
    #
    # All additional inputs are treated as arguments to the
    # command.
    (cd "$1" && shift && "$@" );
}

function gteleport_die() {
    # Usage: gteleport_die STRING
    #
    # Terminate execution with the given message.
    echo -e "$@" 1>&2
    exit 1
}

function gteleport_path_to_running_script() {
    # Usage: gteleport_path_to_running_script
    #
    # Gets the path to the running bash script, assuming that
    # this function was invoked by a script. If the path was
    # obtained, it is echoed to stdout, and true is returned.
    # Otherwise, false is returned.

    if [[ -n "${GTELEPORT_EXE_PATH}" && -f "${GTELEPORT_EXE_PATH}" ]]; then
        echo -n "${GTELEPORT_EXE_PATH}"
        return 0
    fi

    local prg="$0"
    local dir

    if [ ! -e "$prg" ]; then
        case "$prg" in
            (*) return 1;;
            (*) prg=$(command -v -- "$prg") || return 1;;
        esac
    fi
    dir=$(
        cd -P -- "$(dirname -- "$prg")" && pwd -P
    ) || return 1
    prg="$dir"/$(basename -- "$prg") || return 1

    echo -n "$prg"
    return 0
}
```

Approved for Public Release

```
function gteleport_path_to_self() {
    # Usage: gteleport_path_to_self
    #
    # Try to find git-teleport. This method is like
    # gteleport_path_to_running_script, but it also looks in the current
    # directory. If found, echoes path to git-teleport to stdout.

    local self

    self=$(gteleport_path_to_running_script)
    if [[ -z "$self" ]]; then
        self="$PWD/git-teleport"
    fi
    if [[ ! -f "$self" ]]; then
        return 1
    fi

    echo -n "$self"
    return 0
}

function gteleport_path_to_templates() {
    # Usage: gteleport_path_to_templates SUBDIR CHECKFILE
    #
    # Finds the SUBDIR directory within the git-teleport Git repository.
    # If CHECKFILE is defined, we double-check that a file of that
    # name exists in SUBDIR.
    #
    # If found, echoes and returns true.

    local SUBDIR="$1"
    local CHECKFILE="$2"

    local self
    local selfdir

    self=$(gteleport_path_to_running_script)
    selfdir=$(dirname "$self") # defaults to "." if prev command fails
    local templates="$selfdir/../../share/template/$SUBDIR"

    [ -d "$templates" ] || return 1
    [ -z "$CHECKFILE" ] || [ -e "$templates/$CHECKFILE" ] || return 1

    echo "$templates"
}

function gteleport_cachedir() {
    # Usage: gteleport_cachedir
    #
    # Echoes a suitable cache directory for git-teleport. It is created if it
    # does not already exist. The cache directory is determined from:
    #
    # - environment variable $GTELEPORT_CACHE
    # - environment variables $XDG_CACHE_HOME or $APPDATA
    # - ${HOME}/.cache/git-teleport

    local default_cach="${XDG_CACHE_HOME:-${APPDATA:-${HOME}/.cache}}"
    local cach="${GTELEPORT_CACHE}"

    if [[ -z "$cach" ]]; then
        cach="$default_cach/git-teleport"
    fi
}
```

Approved for Public Release

```
mkdir -p "$cach" ||
    gteleport_die "Could not create cache directory '${cach}'"
echo -n "$cach"
}

function gteleport_slugify() {
    # Usage: gteleport_slugify STRING
    #
    # Echoes a "slugified" version of STRING. Slugification
    # converts a string into a set of lowercase letters, digits,
    # and hyphens for use in file names, URLs, and the like.

    echo -n "$1" | iconv -t ascii//TRANSLIT 2>/dev/null | \
    sed -r \
        -e 's/(\w+):\\/(\\w+)?(.*)/\\3/g' \
        -e 's/[^a-zA-Z0-9]+/-/g' \
        -e 's/^(+)/g' \
        -e 's/(-)$/g' | \
    tr '[:upper:]' '[:lower:]' | \
    tail --bytes=63
}

function gteleport_strip_prefixes() {
    # Usage: gteleport_strip_prefixes STRING PRE1 PRE2 ... PREN
    #
    # Strip string prefixes PRE1...PREN from the beginning of STRING.
    # If STRING starts with any of the prefixes PRE1...PREN, then the
    # leading prefix is trimmed. At most one of the prefixes is
    # matched. Matches must be byte-for-byte.
    #
    # The trimmed string is output. This method always succeeds.

    local str="$1"
    shift
    local out="$str"
    local i
    for i in "$@"; do
        [[ -n "$i" ]] || continue;
        out="${str##${i}}"
        [ "$out" = "$str" ] || break;
    done
    printf '%s' "$out"
}

function gteleport_urldecode() {
    # Usage: gteleport_urldecode URL
    #
    # Performs primitive URL decoding of the given URL. Escaped
    # characters such as %20 are converted to their equivalents,
    # such as space. The URL decoding is echoed.

    local url_encoded="${1//+/ }"
    printf '%b' "${url_encoded//%/\\x}" | head -n 1
}

function gteleport_url_to_cache_repo() {
    # Usage: gteleport_url_to_cache_repo URL
    #
    # Converts URL to a cached repository directory on disk.
    # The storage directory is echoed.
    local cache
    local second
}
```

Approved for Public Release

```
cache=$(gteleport_cachedir)
second=$(gteleport_slugify "$1")

echo -n "${cache}/${second}"
}

function gteleport_url_to_projectname() {
    # Usage: gteleport_url_to_projectname URL
    #
    # Attempts to guess a reasonable project name from an URL.
    # The project name is echoed.

    local urldec
    local projname

    urldec=$(gteleport_urldecode "$1")
    projname=$(basename "${urldec}")

    # strip any extension off the end
    projname=$(echo -n "$projname" | sed -r 's/\.[^\.]*/$//')
    gteleport_slugify "$projname"
}

function gteleport_url_is_relative() {
    # Usage: gteleport_url_is_relative URL
    #
    # Check that URL is actually a relative URL or relative
    # path. Relative URLs must not specify a protocol and must
    # not depart from the enclosing tree—for example, via '..'.
    #
    # This method supports UNIX paths, SCP-style SSH URLs,
    # HTTP-style URLs, Windows drive-letter paths, and Windows
    # UNC paths.

    # regexp map:
    # |
    # |--
    #   |- test for protocol://, leading /, or Windows drive
    #   |- test for SCP-style URL
    #   |- test for path departure via '..'

    local url="$1"
    echo "$url" | \
        grep --ignore-case --invert-match -q -E \
            ' (^([a-zA-Z]+:)?[/\\]|(\w+@)?[^\./]{2,}(:.))|((^|/|\\)\.{2,}($|/|\\)) '
}

function gteleport_url_to_relative() {
    # Usage: gteleport_url_to_relative URL ROOT1 ROOT2 ...
    #
    # Attempts to convert the given absolute URL to a URL which is
    # relative to one of the given ROOTs. The URL must match one
    # of the ROOTs exactly, byte-for-byte. Query strings, anchors,
    # trailing slashes, and any trailing ".git" extensions are
    # removed.
    #
    # This method will return true if the URL was made relative.
    # Relative URLs may not contain protocol specifiers, usernames,
    # leading slashes, leading "../" or "./", Windows drive letters,
    # and the like. If the calculated URL contains any of these
    # things, it is not not relative and this method will error.

    # strip roots
```

Approved for Public Release

```
local input_path
local pathpart

input_path=$(gteleport_strip_prefixes "$@")

# strip any number of leading slashes, trailing slashes, and query strings
pathpart=$(echo -n "$input_path" | \
    sed -r 's|^(\.*/+)|; s/^(\[^\?#\]+).*/\1/g; s|/+$||g')

gteleport_url_is_relative "$pathpart" || return 1

echo -n "$pathpart"
}

function gteleport_relative_url_to_local_file() {
    # Usage: gteleport_relative_url_to_local_file RELATIVE_URL
    #
    # Attempts to convert the given RELATIVE_URL, such as one produced
    # by gteleport_url_to_relative(), to a file on the local filesystem.
    # The output file path is relative.
    #
    # To generate the local file name, all path components are normalized
    # to lowercase and have all non-alphanumeric ASCII characters removed.
    # File names are protected against unintentional departures
    # from the tree: for example, via "../.../somewhere_else" or
    # "~elsewhere/etc." The output file name will not have leading or
    # trailing slashes.
    #
    # The local file name is echoed. The output should be safe even for
    # crippled file systems, like NTFS.
    #
    # This method will fail if the given RELATIVE_URL is not relative.

    local RELATIVE_URL="$1"
    local pathpart

    gteleport_url_is_relative "$RELATIVE_URL" || return 1

    # urldecode special characters
    pathpart=$(gteleport_urldecode "$RELATIVE_URL")

    # strip any .git off the end
    pathpart="${pathpart%.git}"
    pathpart="${pathpart%.GIT}"

    # split by slashes, and slugify each part
    declare -a pathparts
    local part
    local relpath=""
    local slug
    IFS="/" read -ra pathparts <<< "$(echo -n -E "$pathpart")"
    for part in "${pathparts[@]}"; do
        slug=$(gteleport_slugify "$part")
        [[ -n "$slug" ]] || continue;
        printf -v relpath "%s%s/" "$relpath" "$slug"
    done

    # strip any slash off the end
    relpath="${relpath%}/"

    # if we are empty, we fail
    [[ -n "$relpath" ]] || return 1
    echo -n "$relpath"
}
```

Approved for Public Release

```
}

function gteleport_url_allowed_as_receiver() {
# Usage: gteleport_url_allowed_as_receiver URL IS_PREFIX
#
# Checks if the given URL is allowed as a destination for
# `git teleport unpack.` The URL may be absolute or relative.
#
# Repositories which are used for unpacking/receiving must
# have a non-terminal path component which matches
# GTELEPORT_MIRROR_URL_REGEX. This allows URLs like
#
#   http://myhost/mirror/project.git
#
# and -not-
#
#   http://myhost/mirror.git
#
# This is a safety check to prevent the user from overwriting
# repos which are not mirrors.
#
# If IS_PREFIX is non-empty, assume the input is a path prefix
# and not a full URL. For example,
#
#   http://myhost/mirror
#
# is a prefix and not a full path to a repository, like project.git.
#
# Returns zero if the URL is allowed as a receiving repository.

local URL="$1"
local IS_PREFIX="$2"

if [ -n "$IS_PREFIX" ]; then
    URL="${URL%*/}" # strip trailing slash
else
    URL=$(dirname "$URL") # strip trailing component
fi

echo "$URL" | \
    tr ':/' "\n" | \
    grep -q -E -i "${GTELEPORT_MIRROR_URL_REGEX}"
}

function gteleport_beamdownfile_find() {
# Usage: gteleport_beamdownfile_find FILE
#
# Searches for a Beamdownfile in the following locations:
# - the FILE argument, if given
# - the $GTELEPORT_BEAMDOWNFILE environment variable, if given
# - the location of the currently executing script, if available
# - the current working directory
#
# If found, the file is echoed and true is returned. If not found,
# then false is returned.

local manualfile="${1:-${GTELEPORT_BEAMDOWNFILE}}"
if [[ -n "$manualfile" && -d "$manualfile" ]]; then
    manualfile="$manualfile/Beamdownfile"
fi

if [[ -f "$manualfile" ]]; then
    echo -n "$manualfile"

```

Approved for Public Release

```
    return 0
fi

local self
local autofile

self=$(gteleport_path_to_running_script)
if [[ -z "$self" ]]; then
    self="$PWD/git-teleport"
fi
autofile=$(dirname "$self")/"Beamdownfile"
if [[ -f "$autofile" ]]; then
    echo -n "$autofile"
    return 0
fi
if [[ -f "$PWD/Beamdownfile" ]]; then
    echo -n "$PWD/Beamdownfile"
    return 0
fi
return 1
}

function gteleport_beamupfile_find() {
    # Usage: gteleport_beamupfile_find FILE
    #
    # Searches for a Beamupfile in the following locations:
    #   - the FILE argument, if given
    #   - the current working directory
    #
    # If found, the file is echoed and 0 is returned. If not found,
    # then non-zero is returned.

    local FILE="$1"

    if [ -f "$FILE" ]; then
        echo -n "$FILE"
        return 0
    fi

    if [[ -n "$FILE" && -d "$FILE" && -f "$FILE/Beamupfile" ]]; then
        echo -n "$FILE/Beamupfile"
        return 0
    fi

    if [ -f "$(pwd)/Beamupfile" ]; then
        echo -n "$(pwd)/Beamupfile"
        return 0
    fi

    return 1
}

function gteleport_beamfile_preparse() {
    # Usage: gteleport_beamfile_preparse
    #
    # Pre-parse the Beamdownfile or Beamupfile, read from stdin.
    #
    # The following operations are performed:
    #
    # 1. leading and trailing whitespaces are stripped
    # 2. trailing slashes are stripped
    # 3. root definitions have spaces fixed: "! root" → "!root"
    #
```

Approved for Public Release

```
# Comments are *NOT* removed, as the caller may want accurate
# line counts.

sed -r -e 's/^[ \t]*//; s/[ \t]*\/*$//; s/^[ \t]*!/[ \t]*!/'
}

function gteleport_beamfile_read() {
# Usage: gteleport_beamfile_read \
#       TYPE OUT_ABS_URLS OUT_MAP_REL_URL OUT_MAP_FILE_PATH
#
# Loads a BeamTYPEfile from stdin. If TYPE is "down," a Beamdown file
# is read. If TYPE is "up," a Beamupfile is read.
#
# This method prints a block of bash commands that, when eval'd, will
# set the following variables whose names are defined by arguments:
#
# - OUT_ABS_URLS      (declare -a ...): List of all absolute URLs in the
#                               Beamdownfile
#
# - OUT_MAP_REL_URL   (declare -A ...): Maps from entries in OUT_ABS_URLS
#                               to relative URLs.
#
# - OUT_MAP_FILE_PATH (declare -A ...): Maps from entries in OUT_ABS_URLS
#                               to local file paths.
#
# This method returns zero if the Beamdownfile was successfully
# parsed and non-zero for error conditions. This method will error
# if the Beamdownfile defines no !roots and contains any absolute URLs.
# It will also error if there are any duplicate absolute URLs, relative
# URLs, or local file paths.
#
# When reading Beamupfiles, the output OUT_ABS_URLS are always
# relative URLs. Absolute URLs are not allowed.

local TYPE="$1"
local OUT_ABS_URLS="${2:-beamfile_abs_urls}"
local OUT_MAP_REL_URL="${3}"
local OUT_MAP_FILE_PATH="${4}"

local is_beamdownfile
case "$TYPE" in
  down | DOWN) is_beamdownfile="yes";;
  up | UP) is_beamdownfile="";;
  *) gteleport_die "Expected TYPE in (up|down)";;
esac

# Clear outputs
if [ -n "$OUT_ABS_URLS" ]; then
  echo "$OUT_ABS_URLS"='()'
fi
if [ -n "$OUT_MAP_REL_URL" ]; then
  echo "unset $OUT_MAP_REL_URL"
  echo "declare -A $OUT_MAP_REL_URL"='()'
fi
if [ -n "$OUT_MAP_FILE_PATH" ]; then
  echo "unset $OUT_MAP_FILE_PATH"
  echo "declare -A $OUT_MAP_FILE_PATH"='()'
fi

local line
local rel_url
local local_file_path
declare -A have_url
```

Approved for Public Release

```
declare -A have_relative_url
declare -A have_local_file_path
declare -a roots
declare -i count=0
gteleport_beamfile_preparse | while read -r line;
do
    ((count = count + 1))
    [ -n "$line" ] || continue;
    [[ "$line" == '#'* ]] && continue;

    local rootdef="${line##!}"
    if [ "$line" != "${rootdef}" ]; then
        # root declarations not allowed here
        [ -n "$is_beamdownfile" ] || {
            gteleport_err_bfr_badroot "$count" "$line"
            return 6
        }

        roots+="${rootdef}"
        continue
    fi

    # prevent duplicate absolute URLs
    [ -z "${have_url[$line]}" ] || {
        gteleport_err_bfr_dup "$count" "$line"
        return 1
    }
    have_url["$line"]="yes"

    # convert to relative URL
    rel_url=$(gteleport_url_to_relative "$line" "${roots[@]}")
    [ -n "$rel_url" ] || {
        gteleport_err_bfr_badrelative "$count" "$line"
        return 2
    }
    if [ -z "$is_beamdownfile" ]; then
        # When reading a Beamupfile, ensure the output URLs
        # are always relative.
        line="$rel_url"
    fi

    # prevent duplicate relative URLs
    [ -z "${have_relative_url[$rel_url]}" ] || {
        gteleport_err_bfr_dup_relative "$count" "$line" "$rel_url"
        return 3
    }
    have_relative_url["$rel_url"]="yes"

    # calculate local file path, and prevent duplicates
    local_file_path=$(gteleport_relative_url_to_local_file "$rel_url")
    [ -n "$local_file_path" ] || {
        gteleport_err_bfr_calclocal "$count" "$line"
        return 4
    }

    [ -z "${have_local_file_path[$local_file_path]}" ] || {
        gteleport_err_bfr_dup_local "$count" "$line" "$local_file_path"
        return 5
    }
    have_local_file_path["$local_file_path"]="yes"

    # Now assign outputs
    if [ -n "$OUT_ABS_URLS" ]; then
```

Approved for Public Release

```
    echo "$OUT_ABS_URLS+=(\"$line\")"
fi
if [ -n "$OUT_MAP_REL_URL" ]; then
    echo "${OUT_MAP_REL_URL} [\"$line\"]=\"$rel_url\""
fi
if [ -n "$OUT_MAP_FILE_PATH" ]; then
    echo "${OUT_MAP_FILE_PATH} [\"$line\"]=\"$local_file_path\""
fi
done
}

function gteleport_beamupfile_gen() {
    # Usage: gteleport_beamupfile_gen REL_URL0 REL_URL1 ... REL_URLN
    #
    # Generates a Beamupfile with the relative URLs given in
    # arguments. The file is printed to stdout.
    # Create Beamupfile, which lists all URL paths from the
    # Beamupfile
    cat <<-EOF
# GIT-TELEPORT BEAMUPFILE
#   Generated By : $(git config --global user.name) <$(git config --global
user.email)>
#   Date          : $(date -u)Z
#   Version       : ${GTELEPORT_VERSION}
#
# This directory contains incremental Git repository update bundles
# which were generated by git-teleport. To use this directory, run
#
#   git teleport unpack path/to/here myserver/mirrors
#
# The second path component must be a directory or URL prefix which
# contains your RECEIVE MIRRORS: i.e., the Git repositories that you
# intend to update. All of the repositories listed in the file below
# MUST EXIST under that path or prefix. If this file contains the
# entries:
#
#   foo.git
#   somedir/BAR.git
#
# and you run `git teleport unpack . ssh://myserver/mirror/`, then
# repositories must exist at:
#
#   ssh://myserver/mirror/foo.git
#   ssh://myserver/mirror/somedir/BAR.git
#
# In other words: to unpack these repositories, you must have a
# filesystem or URL-space which matches these values EXACTLY,
# byte-for-byte. The ".git" extension in the URL is optional.
#
# You must have unlimited write access to all of these repositories.
#
# The `git teleport skeleton` command can help you create compatible
# receive mirror repos on your local filesystem. For network
# servers, like Gitlab, you're on your own.
#
# This file was automatically generated by git-teleport.
EOF

local url
for url in "$@"; do
    gteleport_url_is_relative "$url" || gteleport_die "Expected relative URL"
    echo "$url"
done
```

Approved for Public Release

```
done

}

function gteleport_refdb_path() {
    # Usage: gteleport_refdb_path    BEAMDOWNFILE
    #
    # Calculates the "refs DB" path relative to the given Beamdownfile,
    # which is required and must exist. The refs DB folder will be
    # created if it does not exist.
    #
    # Echoes the refs DB folder to stdout

    [[ -f "$1" ]] || gteleport_die "Expected BEAMDOWNFILE"

    local refdb
    refdb=$(dirname "$1")/"Beamdownstate"
    mkdir -p "${refdb}" || gteleport_die "Could not make Beamdownstate directory"

    echo -n "$refdb"
    return 0
}

function gteleport_refdb_forurl() {
    # Usage: gteleport_refdb_forurl    ROOT    URL
    #
    # Calculates the refs DB file name for the given repository URL. The ROOT
    # directory of the refs DB must be specified. Echoes the file name to
    # stdout.

    [[ -d "$1" ]] || gteleport_die "Expected refs DB ROOT"

    local dbfile
    dbfile="$1"/$(gteleport_slugify "$2")

    # Create file if not exists
    if [[ ! -f "$dbfile" ]]; then
        touch "$dbfile" || gteleport_die "Could not make Beamdownstate file"
    fi
    echo -n "$dbfile"
    return 0
}

function gteleport_output_path() {
    # Usage: gteleport_output_path    BEAMDOWNFILE
    #
    # Calculates the bundle output root relative to BEAMDOWNFILE. Creates the
    # output root if it doesn't exist. Echoes the output root to stdout.

    [[ -f "$1" ]] || gteleport_die "Expected BEAMDOWNFILE"

    local bndls
    bndls=$(dirname "$1")/"bundles"
    mkdir -p "${bndls}" || gteleport_die "Could not make bundles directory"

    echo -n "$bndls"
    return 0
}

function gteleport_output_new() {
    # Usage: gteleport_output_path    OUTPUT_ROOT
    #
    # Creates new output directory for a new bundle run. The name is based
```

Approved for Public Release

```
# on the current UTC time. Creates the directory and echoes the full path
# stdout.

[[ -d "$1" ]] || gteleport_die "Expected OUTPUT_ROOT"

local outname
local outdir

outname="bundles_$(date -u +%Y%m%d_%H%M%S')Z"
outdir="$1/$outname"
mkdir -p "${outdir}" || gteleport_die "Could not make output directory"

echo -n "$outdir"
return 0
}

function gteleport_output_trylink() {
# Usage: gteleport_output_trylink OUTPUT_DIR
#
# Attempts to make or update the symbolic link OUTPUT_DIR/../latest
# point to OUTPUT_DIR. This method is used to update the "latest"
# symlink at the end of a run.
#
# In Git Bash, this operation will be a copy. On file systems which
# don't support symlinks, it will silently fail.
#
# Returns 0 if the link succeeds or nonzero if the link fails.

[[ -d "$1" ]] || gteleport_die "Expected OUTPUT_DIR"

local outname
local outroot

outname=$(basename "$1")
outroot=$(dirname "$1")
local ok=0

pushd "$outroot" >/dev/null || \
    gteleport_die "Could not change directory to '$outroot'"

# Remove existing link
rm -rf "latest" || true

# Make new link
ln -sT ".$outname" "latest" || ok=1

popd > /dev/null

return $ok
}

function gteleport_git_discover() {
# Usage: gteleport_git_discover URL
#
# Attempts to discover a Git repository at the given URL.
# Various forms of the ".git" extension are tried.
#
# If a repo is found, prints its path and returns zero. If no
# repo is found, returns non-zero.

local URL="$1"

# Strip any .git extension off the end, and test that.
```

Approved for Public Release

```
URL="${URL%%.git}"
URL="${URL%%.GIT}"

declare -a test_suffix=('.git' '' '.GIT')

local suffix
for suffix in "${test_suffix[@]}; do
    if git --git-dir=/dev/null ls-remote --heads "${URL}${suffix}" \
        1>/dev/null 2>&1; then
        echo -n "${URL}${suffix}"
        return 0
    else
        continue
    fi
done

return 1
}

function gteleport_git_update() {
    # Usage: gteleport_git_update URL PATH
    #
    # Updates the given Git repository, specified by $URL, at the
    # given $PATH on the file system. If no local copy exists, the
    # repository is --mirror cloned. If a local copy exists, the
    # remotes are fetched.
    #
    # When this method returns, the repo at the given PATH will
    # have an "origin" mirror remote that points to URL.
    #
    # Returns exit status of git

    if [ ! -d "$2" ]; then
        git clone --no-hardlinks --mirror "$1" "$2"
    else
        # work around *stupid* git 1.7.x bug which prevents --prune
        # fetches from repos by URL
        gteleport_rund "$2" git remote set-url origin "$1" 2>/dev/null || \
            gteleport_rund "$2" git remote add origin "$1" || \
                return 1

        gteleport_rund "$2" \
            git config remote.origin.fetch '+refs/*:refs/*' || return 1

        gteleport_rund "$2" git fetch --prune origin '+refs/*:refs/*' || return 1
    fi
    return $?
}

function gteleport_git_show_ref() {
    # Usage: gteleport_git_show_ref PATH
    #
    # Echoes the SHA-1 digest which correspond to all refs
    # (branches, tags, etc.) which currently exist in the
    # repository pointed to by $PATH. The refs are sorted,
    # and only unique refs are kept.
    #
    # Returns exit status of git

    gteleport_rund "$1" git show-ref -d | sort --stable
}

function gteleport_git_bundle_all() {
```

Approved for Public Release

```
# Usage: gteleport_git_bundle_all PATH OUTFILE
#
# Creates a git bundle which contains the complete history
# of all refs in the repository at $PATH. The bundle is written
# to $OUTFILE.
#
# Returns exit status of git.

local outfile
outfile=$(readlink -f "$2")

gteleport_rund "$1" git bundle create "$outfile" --all
return $?
}

function gteleport_git_bundle_all_forced() {
# Usage: echo EXISTING_REFS | gteleport_git_bundle_all_forced PATH OUTFILE
#
# Creates a git bundle which contains the complete history
# of all refs in the repository at $PATH. The bundle is written
# to $OUTFILE.
#
# This method is equivalent to gteleport_git_bundle_all, but it presents
# the same refs-reading interface as gteleport_git_bundle_incr. This
# allows it to be a drop-in replacement for gteleport_git_bundle_incr.
#
# Returns exit status of git.

while IFS='' read -r line || [[ -n "$line" ]]; do
    true;
done < "/dev/stdin"

gteleport_git_bundle_all "$1" "$2"
}

function gteleport_git_has_new_rev_list() {
# Usage: gteleport_git_has_new_rev_list PATH
#
# Using the repository given at PATH, checks to see if
# git rev-list supports both --ignore-missing and
# --stdin. If it does, a better ref filter can be used.
#
# Prints "yes" if rev-list supports both --ignore-missing
# and --stdin. Prints "no" otherwise. An error is returned
# if PATH is not a valid repository.
#
# The answer is cached for the duration of the shell.
# See GTELEPORT_GIT_HAS_NEW_REVLIST.

if [[ -z "$GTELEPORT_GIT_HAS_NEW_REVLIST" ]]; then
# check that the git repo exists
gteleport_rund "$1" git rev-list \
    --max-count=1 \
    --all \
    --quiet >/dev/null </dev/null || return $?

if gteleport_rund "$1" \
    git rev-list \
    --max-count=1 \
    --quiet \
    --ignore-missing \
    --stdin \
    2>/dev/null \
```

Approved for Public Release

```
        </dev/null; then
        GTELEPORT_GIT_HAS_NEW_REVLIST=yes
    else
        GTELEPORT_GIT_HAS_NEW_REVLIST=no
    fi
fi

echo "$GTELEPORT_GIT_HAS_NEW_REVLIST"
[[ -n "$GTELEPORT_GIT_HAS_NEW_REVLIST" ]]
}

function gteleport_git_filter_refs() {
    # Usage: echo SOME_REFS | gteleport_git_filter_refs PATH
    #
    # A ref filtering function which does not require either
    # --ignore-missing or --stdin support in git rev-list.
    #
    # It is slow.
    #
    # Filters refs piped in and limits to only those which are
    # present in the git repository. refs should be separated
    # by a newline. Duplicate refs are not suppressed. This
    # function does not resolve refs, so if you provide a
    # branch or tag name, then that is the output you will
    # receive.
    #
    # This function will succeed as long as the Git repository
    # is valid and readable. If the Git repository is not
    # readable, then this function will die.
    #
    # The output may be empty, however.
    #
    # This method is an alternative to using the "--ignore-missing"
    # option to git rev-list. Ancient gits do not have this option.

    # Check that this repo can have refs listed at all.
    gteleport_rund "$1" git rev-list --all --max-count=1 --quiet </dev/null || \
    gteleport_die \
    "Cannot list any refs from git repository\n"\
    "at $1. Does it exist?"

    # Read the old refs, piped in. Ensure uniqueness. Foreach ref,
    # check with Git to see if it is still there. Eliminate the
    # refs which are gone.
    local ref
    local ign
    # shellcheck disable=SC2034
    while read -r ref ign
    do
        gteleport_rund "$1" \
        git rev-list --max-count=1 --quiet 2>/dev/null "$ref" || continue

        # The ref exists
        echo "$ref"
    done

    return 0
}

function gteleport_git_bundle_incr() {
    # Usage: echo EXISTING_REFS | gteleport_git_bundle_incr PATH OUTFILE
    #
    # Creates a git bundle which contains an incremental update
```

Approved for Public Release

```
# of all refs in the repository at $PATH. The bundle is written
# to $OUTFILE.
#
# The user should pipe some source of SHA-1 digests to exclude to
# this function. For example, if you saved all refs from the
# last time you did this with $(gteleport_git_show_ref), you can pipe these
# back in and have them all excluded. Refs in the input which are
# not presently in the repository are silently ignored.
#
# This method will fail if the given bundle would be empty.

# Path to output bundle
local outfile
outfile=$(readlink -f "$2" </dev/null)

# Limit the old refs to those which are available in the repo now.
# Exclude all of these refs from the bundle by negating them.
if [[ $(gteleport_git_has_new_rev_list "$1") = "no" ]]; then
    # Filter to refs which exist in the repo.
    #
    # Here, we use xargs to create the command line to "git bundle"
    # which lists all the refs.
    gteleport_git_filter_refs "$1" | \
        sed -r -e 's/^(\\w+).*$/^\\1/' | \
        gteleport_rund "$1" xargs git bundle create \
            "$outfile" --all
else
    # Use automatic filtering built-in to git.
    sed -r -e 's/^(\\w+).*$/^\\1/' | \
        gteleport_rund "$1" git bundle create \
            "$outfile" \
            --all \
            --ignore-missing \
            --stdin
fi

return $?
}

function gteleport_git_refs_to_prerequisites() {
    # Usage: gteleport_git_refs_to_prerequisites REPO
    #
    # Calculates a set of bundle prerequisites. The prerequisites
    # require that the repo be absolutely current: i.e., contain
    # all the commits needed by its current refs.
    #
    # This function is used to build prerequisites for refs-only
    # bundles. See gteleport_git_bundle_refonly.
    #
    # The output is in git-rev-list's "boundary" format, with
    # "uninteresting" commits (i.e., all of them) marked with
    # leading minus signs

    local repo="$1"

    local reffhash
    local reftype
    local refname
    local dereffhash

    while IFS=' ' read -r reffhash reftype refname dereffhash; do
        case "$reftype" in
            # print commit as a prerequisite
```

Approved for Public Release

```
commit) printf '%s%s\n' '-' "$refhash $refname (commit)";;

# for tags, print the de-referenced commit hash if present
tag)
    if [ -n "$derefhash" ]; then
        printf '%s%s\n' '-' "$derefhash $refname (annotated tag)"
    else
        printf '%s%s\n' '-' "$refhash $refname"
    fi;;

*) continue;;

esac
done < <(gteleport_rund "$repo" git for-each-ref \
    --format='% (objectname) % (objecttype) % (refname) % (*objectname)')
}

function gteleport_git_bundle_refonly() {
    # Usage: gteleport_git_bundle_refonly REPO OUTFILE
    #
    # Creates a git bundle which contains all refs, and
    # only the refs, from REPO. The bundle is written to
    # OUTFILE.
    #
    # Unlike `gteleport_git_bundle_incr` or
    # `gteleport_git_bundle_all`, bundles generated by this
    # method contain ONLY REFS without any commits or objects
    # at all. These bundles are useful for pruning but are
    # *not* useful for actual updating.
    #
    # To use one of these bundles, first fetch an incremental
    # bundle. Then fetch one of these "refs only" bundles with
    # `--prune`.
    #
    # This method will verify that the refs in the bundle are
    # identical to the refs in the source.

    local repo="$1"
    local bundle="$2"
    local refs_src=""
    local refs_bndl=""

    [ ! -e "$bundle" ] || return 1

    # Git has problems creating the type of bundle we want,
    # which has only refs and no pack. We can create this type
    # of bundle ourselves!

    {
        # a simple header
        echo '# v2 git bundle';

        # prerequisites in git-rev-list format, with required commits
        # marked with a minus sign
        gteleport_git_refs_to_prerequisites "$repo";

        # refs, in git-show-ref format
        gteleport_rund "$repo" git show-ref;

        # normally, a pack file would follow... but we don't need one
    } >"$bundle" || return 1

    # verify the bundle
    gteleport_rund "$repo" git bundle verify "$bundle" >/dev/null 2>&1 || {
```

Approved for Public Release

```
gteleport_err_verify_refsbundle "$repo" "$bundle" "$refs_src" "$refs_bndl"
rm -f "$bundle"
return 1
}

refs_src=$(gteleport_rund "$repo" git ls-remote --refs "." | sort)
refs_bndl=$(gteleport_rund "$repo" git ls-remote --refs "$bundle" | sort)
[ "$refs_src" = "$refs_bndl" ] || {
    gteleport_err_bad_refsbundle "$repo" "$bundle" "$refs_src" "$refs_bndl"
    rm -f "$bundle"
    return 1
}
}

function gteleport_git_fetchbundle() {
    # Usage: gteleport_git_fetchbundle REPO BUNDLE PRUNE
    #
    # Try to update the given REPO with the given BUNDLE file. Returns 0
    # if the update succeeds or 1 if it fails.
    #
    # If PRUNE is non-empty, then the fetch will be with `--prune`.
    # NEVER use PRUNE with an incremental bundle. ONLY USE PRUNE
    # in conjunction with a "refs-only" bundle. See
    # gteleport_git_bundle_refsonly.

    local bundlefile
    local prune="$3"
    bundlefile=$(readlink -f "$2")

    gteleport_rund "$1" \
        git bundle verify "$bundlefile" >/dev/null 2>&1 || return 1

    if [ -z "$prune" ]; then
        gteleport_rund "$1" \
            git fetch "$bundlefile" '+refs/*:refs/*' || return 1
    else
        # work around *stupid* git 1.7.x bug which prevents --prune
        # fetches from bundles by URL
        gteleport_rund "$1" git remote add bndl "$bundlefile" || \
            gteleport_rund "$1" git remote set-url bndl "$bundlefile" || \
                return 1

        gteleport_rund "$1" \
            git config remote.bndl.fetch '+refs/*:refs/*' || return 1

        gteleport_rund "$1" \
            git fetch --prune bndl '+refs/*:refs/*' || {
                gteleport_rund "$1" git remote rm bndl 2>/dev/null || true
                return 1
            }

        gteleport_rund "$1" git remote rm bndl 2>/dev/null
    fi
}

function gteleport_git_pushto() {
    # Usage: Usage: gteleport_git_pushto REPO URL
    #
    # Performs a --mirror push of the given REPO to URL. The
    # destination repository at URL will be IRREVOCABLY
    # OVERWRITTEN!

    gteleport_rund "$1" git push --mirror "$2"
}
```

Approved for Public Release

```
}

function gteleport_git_need_bundle() {
    # Usage: echo EXISTING_REFS | gteleport_git_need_bundle PATH
    #
    # Checks if a new bundle is required for the given git repository
    # at $PATH. This method prints "yes" if any refs have changed.

    cmp --quiet - <(gteleport_git_show_ref "$1") || echo "yes"
}

function gteleport_install_skeleton() {
    # Usage: gteleport_install_skeleton REPO
    #
    # Installs the Receive Mirror template files to REPO. Existing
    # files are overwritten. This method requires that git-teleport
    # have access to the template_skeleton files.

    local REPO="$1"
    [ -d "$REPO" ] || gteleport_die "Expected destination REPO directory"

    local templates
    templates=$(gteleport_path_to_templates \
        "receive" "teleport-mirror") || \
        gteleport_die "Could not find templates directory!"

    (
        umask 002 && \
        cp -R "$templates/." "$REPO" && \
        chmod g+rwX,a+rx "$REPO/hooks/*"
    )
}

function gteleport_git_init_skeleton() {
    # Usage: gteleport_git_init_skeleton PATH
    #
    # Initializes a bare, shared Git repository with all pushes
    # rejected and a "teleport-mirror" file, which marks it as
    # a mirror. This command will fail if the repository
    # already exists, regardless of whether or not it is
    # configured as a mirror. This command will die if it should
    # be able to create the skeleton, but can't.

    [[ -n "$1" ]] || gteleport_die "Expected DIRECTORY"

    local repo="$1"
    if [ -e "$repo" ]; then
        [ -e "$repo/teleport-mirror" ] || {
            echo "The directory \"$repo\" already exists" 1>&2
            echo "and does not appear to be a Receive Mirror." 1>&2
            return 1
        }
        gteleport_install_skeleton "$repo"
        echo "" 1>&2
        echo "The requested repository \"$repo\" already exists!" 1>&2
        echo "The skeleton template files were updated." 1>&2
        echo "" 1>&2
        return 0
    fi

    # make directory and init as git repo
    (
        umask 002 && \
```

Approved for Public Release

```
mkdir -p "$repo" && \
chmod ug+rwX "$repo" && \
gteleport_rund "$repo" git init --bare --shared
) || {
    echo "Could not create repository \"$repo\"!" 1>&2
    echo "Check that the destination is writeable." 1>&2
    return 1
}

# install template files
gteleport_install_skeleton "$repo" || {
    echo "Could not install template files to \"$repo\"." 1>&2
    echo "This repository will not be usable as an gteleport" 1>&2
    echo "receive mirror. Check your git-teleport installation" 1>&2
    echo "and try again." 1>&2
    return 1
}
}

function gteleport_git_run_post_update() {
    # Usage: gteleport_git_run_post_update REPO
    #
    # Run the post-update and post-receive hooks in REPO, if either
    # exist. The post-update hook is executed first.
    #
    # The scripts must be executable by the current user and must define
    # an interpreter. The current environment is preserved when running
    # scripts.
    #
    # Git supplies both of these scripts with arguments (for post-update)
    # or standard input (for post-receive) which informs the script about
    # the refs that were updated. git-teleport does not do this. The input
    # in both cases will be empty!
    #
    # The exit status from the post-update hook is returned, but the
    # caller should probably ignore it.

    local REPO="$1"

    if [ -x "$REPO/hooks/post-update" ] ; then
        (
            cd "$REPO" && hooks/post-update
        ) 0<&- 1>&2 || return $?
    else
        true
    fi

    if [ -x "$REPO/hooks/post-receive" ] ; then
        (
            cd "$REPO" && hooks/post-receive
        ) 0</dev/null 1>&2 || return $?
    else
        true
    fi
}

function gteleport_check_unpack_prefix() {
    # Usage: gteleport_check_unpack_prefix PREFIX
    #
    # Verifies that the given PREFIX is suitable as an `git teleport unpack`
    # target.
    #
    # * Local paths always pass, so long as they exist
}
```

Approved for Public Release

```
#
# * Network paths must contain a *path component* which matches
#   GTELEPORT_MIRROR_URL_REGEX. The default regexp matches
#   "/mirror/" and "/mirrors/" (plural).
#
# If the check succeeds, returns zero and prints either "local" OR
# "network". If the check fails, returns non-zero and writes an
# error message to stderr.

# strip trailing slashes
local PREFIX="${1%/}"

if [ -z "$PREFIX" ]; then
    gteleport_err_missingprefix
    return 1
fi

if [ -e "$PREFIX" ]; then
    # local paths are always OK
    echo "local"
    return 0
fi

if gteleport_url_allowed_as_receiver "$PREFIX" 1; then
    echo "network"
    return 0
fi

# fails: prefix not allowed
gteleport_err_prohibited_network "$PREFIX"
return 3
}

function gteleport_find_receive_mirror() {
    # Usage: gteleport_find_receive_mirror PREFIX REL_URL TYPE BEAMUPFILE
    #
    # Determines if the Git repository located at PREFIX/REL_URL is
    # usable as a Receive Mirror. The URL may be modified. The actual URL
    # to use is printed to stdout.
    #
    # This method tries to resolve the actual repository URL and performs
    # safety checks to see if it is [probably] a git-teleport Receive
    # Mirror.
    #
    # TYPE must be one of the following:
    #
    # * "local" : for files on the local filesystem. This includes network
    #   shares which have been mounted as directories.
    #
    # * "network" : repositories which are accessed using network protocols.
    #
    # If the repository appears usable, its URL is printed, and zero is
    # returned. Otherwise, a descriptive error is printed, and non-zero
    # is returned.
    #
    # The BEAMUPFILE argument is intended for error-reporting purposes
    # only. The BEAMUPFILE is not read.

    local PREFIX="${1%/}"          # strip trailing slashes
    local REL_URL="${2%/}"
    local TYPE="$3"
    local BEAMUPFILE="$4"
```

Approved for Public Release

```
[[ -n "$PREFIX" && -n "$REL_URL" ]] || return 1

local is_local=""
case "$TYPE" in
    local) is_local="yes" ;;
    network) is_local="" ;;
esac

local search_url
local url

if [ -n "$is_local" ]; then
    # Perform URL decoding
    search_url=$(gteleport_urldecode "$REL_URL")

    # Find repository
    url=$(gteleport_git_discover "$PREFIX/$search_url") || {
        gteleport_err_frm_destnotfound "$PREFIX/$search_url" "$TYPE"
        return 2
    }

    # Check that we may use it
    [ -f "$url/teleport-mirror" ] || {
        gteleport_err_frm_prohibited_local "$url"
        return 3
    }
else
    # Find repository
    url=$(gteleport_git_discover "$PREFIX/$REL_URL") || {
        gteleport_err_frm_destnotfound "$PREFIX/$REL_URL" "$TYPE"
        return 2
    }

    # Check that we may use it
    gteleport_url_allowed_as_receiver "$url" || {
        gteleport_err_prohibited_network "$url"
        return 3
    }
fi

# Looks good to go
echo "$url"
}

function gteleport_write_manifest() {
    # Usage: gteleport_write_manifest DIRECTORY
    #
    # Writes a MANIFEST.txt file to the given DIRECTORY which
    # contains a tab-delimited file that consists of
    #
    # 1. The file's *apparent* size, in human-readable powers of
    #    1024; and
    #
    # 2. The file path and name
    #
    # The manifest file is not listed in itself. (That would be silly.)

    local DIRECTORY="$1"
    [ -d "$DIRECTORY" ] || gteleport_die "Expected directory"

    local manifest_file="$1/MANIFEST.txt"

    [ ! -e "$manifest_file" ] || \
```

Approved for Public Release

```
gteleport_die "Cowardly refusing to overwrite existing manifest file"

# shellcheck disable=SC2094
du --apparent-size --all -h \
  --files0-from=<(find "$DIRECTORY" \
    -type f -not -path "$manifest_file" -print0) \
  >"$manifest_file" || {
  echo "Error generating manifest file" 1>&2
  rm -f "$manifest_file"
  return 1
}

[ -e "$manifest_file" ]
}

function gteleport_bootstrap() {
  # Usage: gteleport_bootstrap DIRECTORY
  #
  # Bootstrap git-teleport Mirror Set to DIRECTORY

  [[ -n "$1" ]] || gteleport_die "Expected DIRECTORY"
  [[ ! -a "$1" ]] || gteleport_die \
    "Cannot bootstrap to existing directory."

  local templates
  local parent

  # Find required files
  templates=$(gteleport_path_to_templates send Beamdownfile) || \
    gteleport_die \
      "Cannot find template/ directory.\n"
  "You can only bootstrap git-teleport from its original Git repository. You\n"\
  "cannot bootstrap git-teleport from one of the Mirror Sets you have created."

  parent=$(dirname "$1")

  # Check for existing repository
  if gteleport_rund "$parent" git branch >/dev/null 2>&1; then
    gteleport_die \
      "The requested bootstrap directory is inside a Git repository.\n"
    "Refusing to bootstrap into an existing repository."
  else
    true
  fi

  # Create repository
  git init "$1"

  # Copy templates
  cp -R --preserve=all "$templates/." "$1"

  # Commit. If we are committing on Windows, be sure to chmod +x
  gteleport_rund "$1" git add .
  gteleport_rund "$1" git update-index --chmod=+x "teleportw"
  gteleport_rund "$1" \
    git commit -m "Bootstrapped git-teleport with wrapper script"

  cat <<-EOF

Bootstrapped git-teleport to '${1}'.

Now edit '${1}/Beamdownfile'
and add a list of repositories to mirror. When you are finished,
```

Approved for Public Release

commit the results.

Then run:

```
cd '${1}'
./teleportw update
```

to make your first set of bundles.

EOF

}

```
function gteleport_fetch() {
    # Usage: gteleport_fetch BEAMDOWNFILE
    #
    # Updates the local mirror of all repositories listed in the given
    # BEAMDOWNFILE, which must exist. Returns true if the updates succeeded,
    # or false if any update failed.

    # Find Beamdownfile
    local bdfile
    local bdvars
    local repo_urls
    bdfile=$(gteleport_beamdownfile_find "$1") || \
        gteleport_die "Could not find Beamdownfile '$1'"

    # Peek at it. Do we have anything defined?
    bdvars=$(gteleport_beamfile_read "down" "repo_urls" <"$bdfile") || \
        gteleport_die "Error reading Beamdownfile '$1'"
    eval "$bdvars"
    if [[ -z "${repo_urls[0]}" ]]; then
        echo -e "No repository URLs defined in '$1'. Nothing to do."
        return 1
    fi

    # Foreach repository, fetch.
    local url
    local cur_repo_name
    local cur_repo_dir
    for url in "${repo_urls[@]}"; do
        cur_repo_name=$(gteleport_url_to_projectname "$url")
        cur_repo_dir=$(gteleport_url_to_cache_repo "$url")

        # Create or update local mirror
        echo "[git-teleport][$cur_repo_name]: fetching...";
        if gteleport_git_update "$url" "$cur_repo_dir"; then
            true
        else
            echo "Unable to update local mirror of '$url': '$cur_repo_dir'" 1>&2
            return 1
        fi
    done

    echo "[git-teleport] All local mirrors updated."

    return 0
}
```

```
function gteleport_bundle() {
    # Usage: gteleport_bundle BEAMDOWNFILE FORCE_FULL OUTVAR
    #
    # Creates bundles for the given BEAMDOWNFILE, which must exist.
    # The BEAMDOWNFILE must be stored in a Git repository which has
    # a clean working directory state. If FORCE_FULL is non-empty,
```

Approved for Public Release

```
# then all bundles will be full bundles which record a complete
# history. If OUTVAR is non-empty, it will be set to "0" if at
# least one bundle was updated, or "1" if no bundles were updated.
#
# This command only creates bundles and does not update the local
# mirrors.

# Which method?
local bundlefcn="gteleport_git_bundle_incr"
if [[ -n "$2" ]]; then
    bundlefcn="gteleport_git_bundle_all_forced"
    echo "[git-teleport] Forcing full bundles."
fi

[[ -n "$3" ]] && printf -v "${3}" '%s' "1"

# Find Beamdownfile
local bdfile
local bdfiledir
bdfile=$(gteleport_beamdownfile_find "$1") || \
    gteleport_die "Could not find Beamdownfile '$1'"

# Check that we have a clean working copy
bdfiledir=$(dirname "$bdfile")
if [ -n "$(gteleport_rund "$bdfiledir" git status --porcelain)" ]; then
    cat 1>&2 <<-EOF
The Git repository which contains '$bdfile' is not clean.

This can happen when you run "./teleportw bundle" without committing
the results. Git-teleport needs this data committed to ensure that you
can return to a previous bundling state if a data transfer fails---
i.e., if the bundle is corrupted during transport.

If the previous bundles worked, or if you intend to try and transfer
them, run

    git -C '$bdfiledir' add Beamdownstate
    git -C '$bdfiledir' commit

and optionally push your work. If the previous bundles were never
transferred or failed to transfer, run

    git -C '$bdfiledir' reset --hard HEAD

Aborting.

EOF
    return 1
fi

# Peek at it. Do we have anything defined?
local bdvars
declare -a repo_urls
declare -A repo_rel_urls
declare -A repo_bundle_files
bdvars=$(gteleport_beamfile_read \
    "down" \
    "repo_urls" \
    "repo_rel_urls" \
    "repo_bundle_files" <"$bdfile") || \
    gteleport_die "Error reading Beamdownfile '$1'"
eval "$bdvars"
if [[ -z "${repo_urls[0]}" ]]; then
```

Approved for Public Release

```
    echo -e "No repository URLs defined in '$1'. Nothing to do."
    return 1
fi
declare -i num_repos="${#repo_urls[@]}"

# Ensure that our database exists
local dbpath
dbpath=$(gteleport_refdb_path "$bdfilename")

# Check for local mirrors
local url
local cur_repo_dir
for url in "${repo_urls[@]}"
do
    cur_repo_dir=$(gteleport_url_to_cache_repo "$url")

    # Do we have a local mirror?
    if [[ ! -d "$cur_repo_dir" ]]; then
        echo "No local mirror of '$url' found." \
            "Did you run \`../teleportw fetch\` first?" 1>&2
        return 1
    fi
done

echo "[git-teleport] Preparing to bundle $num_repos repositories..."

# Make a new output directory
local output_root
local output_path
output_root=$(gteleport_output_path "$bdfilename")
output_path=$(gteleport_output_new "$output_root")
echo "[git-teleport] Created output directory '$output_path'"
output_path=$(readlink -f "$output_path")

# Foreach repository, bundle
local did_anything=""
local cur_repo_dir
local cur_repo_outfile
local cur_repo_name
local cur_refs_db
local needbdl
for url in "${repo_urls[@]}"
do
    cur_repo_dir=$(gteleport_url_to_cache_repo "$url")
    cur_repo_name=$(gteleport_url_to_projectname "$url")
    cur_repo_outfile="${repo_bundle_files[$url]}"
    cur_bundle="$output_path/${cur_repo_outfile}.bundle"
    cur_refs_db=$(gteleport_refdb_forurl "$dbpath" "$url")

    if [[ -z "$cur_bundle" ]]; then
        echo "Unable to calculate local file name for '${url}'" 1>&2
        return 1
    fi

    # Do we need a bundle?
    needbdl=$(gteleport_git_need_bundle "$cur_repo_dir" "<$cur_refs_db")
    if [[ -z "$2" && -z "$needbdl" ]]; then
        echo "[git-teleport][$cur_repo_name]: up to date";
        continue
    fi

    # Check for existing bundle file
    if [[ -f "$cur_bundle" ]]; then
```

Approved for Public Release

```
cat 1>&2 <<EOF
Duplicate bundle file

Git-teleport detected an existing bundle file from this run:

${cur_bundle}

that would be overwritten by the output from this repository,

${url}

This might happen if the above URL is too similar to another URL in
your Beamdownfile. The remainder of the bundling process has been
aborted.
EOF
    return 1
fi

# Make parent directories
local newdir
newdir=$(dirname "$cur_bundle")
mkdir -p "$newdir" || {
    echo "Unable to create directory $newdir" 1>&2
    return 1
}

# Write bundle file
if "$bundlefcn" "$cur_repo_dir" "$cur_bundle" < "$cur_refs_db"; then
    true
else
    echo "WARNING: Unable to bundle '$url' to '$cur_bundle'." 1>&2
    echo \
        "This is usually okay: it merely indicates that refs (like" 1>&2
    echo "branches) have changed but there are no new commits to send." 1>&2
    echo "The main bundle will be skipped for this repo." 1>&2
fi

# Write refs file
if gteleport_git_bundle_refsonly "$cur_repo_dir" "${cur_bundle}.refs"; then
    true
else
    echo -n "[git-teleport][$cur_repo_name]: " 1>&2
    echo "WARNING - Unable to create refs bundle for '$url'" 1>&2
    echo -n "[git-teleport][$cur_repo_name]: " 1>&2
    echo "the destination will not be pruned!" 1>&2
fi

# Update refs DB
if gteleport_git_show_ref "$cur_repo_dir" > "$cur_refs_db"; then
    true
else
    echo "Unable to update refs database for '$cur_refs_db'" 1>&2
    return 1
fi

did_anything="yes"
echo "[git-teleport][$cur_repo_name]: BUNDLED"
done

if [[ -n "$did_anything" ]]; then
    # Write Beamupfile
    gteleport_beamupfile_gen "${repo_rel_urls[@]}" >"$output_path/Beamupfile" \
        || return 1
fi
```

Approved for Public Release

```
echo "[git-teleport] Wrote Beamupfile."

# Write manifest
gteleport_rund "$output_path" gteleport_write_manifest . || true

# Re-link latest bundle file
if gteleport_output_trylink "$output_path"; then
    echo "[git-teleport] Created bundles at: '$output_root/latest'"
else
    echo "[git-teleport] Created bundles at: '$output_path'"
fi

[[ -n "$3" ]] && printf -v "${3}" '%s' "0"
else
    rm -rf "$output_path"
    echo "[git-teleport] No updates."
fi

return 0
}

function gteleport_commit() {
    # Usage: gteleport_commit BEAMDOWNFILE OUTVAR
    #
    # Commits any outstanding changes to the Beamdownstate database
    # for the BEAMDOWNFILE specified. This command silently succeeds
    # if no changes need committing. If OUTVAR is non-empty, it will
    # be set to "0" if a commit was performed or "1" if no commit was
    # performed.

    # Find Beamdownfile
    local bdfile
    bdfile=$(gteleport_beamdownfile_find "$1") || \
        gteleport_die "Could not find Beamdownfile '$1'"

    [[ -n "$2" ]] && printf -v "${2}" '%s' "1"

    # Check if we have a clean working copy
    local bdfiledir
    bdfiledir=$(dirname "$bdfile")

    if [ -z "$(gteleport_rund "$bdfiledir" git status --porcelain -- "Beamdownstate")" ]; then
        # Clean.
        echo "[git-teleport] No database changes to commit."
        return 0
    fi

    gteleport_rund "$bdfiledir" git add "Beamdownstate" || \
        gteleport_die "Failure staging database changes"

    gteleport_rund "$bdfiledir" git commit \
        -m "$(whoami) generated fresh bundles" \
        -m "Commit authored automatically by git-teleport-${GTELEPORT_VERSION}."

    local ok=$?
    if [[ "$ok" != "0" ]]; then
        echo "Failed to commit database changes!" 1>&2
        return 1
    fi

    [[ -n "$2" ]] && printf -v "${2}" '%s' "0"
    return 0
}
```

Approved for Public Release

```
}

function gteleport_skeleton() {
  # Usage: gteleport_skeleton REPO_PATH1 REPO_PATH2 ...
  #
  # For each argument, create an empty receiving repository
  # at that location. The new repo will have sane defaults,
  # including:
  #   - Marked as a git-teleport mirror
  #   - Have all pushes denied
  #   - Be bare and shared
  # Relative paths are interpreted relative to the current
  # working directory.

  if [ ${#@} -eq 0 ]; then
    gteleport_die "Expected list of new receiving repositories"
  fi

  for d in "$@"
  do
    if [[ -z "${d}" ]]; then
      continue;
    fi

    echo "[git-teleport]: Making receive skeleton at \"${d}\"" 1>&2
    gteleport_git_init_skeleton "$d" || return 1
  done

  return 0
}

function gteleport_unpack() {
  # Usage: gteleport_unpack SRC_DIR DEST_PREFIX
  #
  # Reads the Beamupfile in SRC_DIR and scans for bundle files
  # in that directory. The repositories in DEST_PREFIX, which
  # may be either a directory or an URL prefix, are updated.
  #
  # This method incorporates safety checks to ensure that all
  # repositories are OK to use as Receive Mirrors.

  local SRC="$1"
  local DEST="$2"

  [[ -n "$SRC" && -n "$DEST" ]] || {
    cat 1>&2 <<-EOF
Usage: ${GTELEPORT_EXEC_NAME} unpack SRC_BUNDLE_DIR DEST_PREFIX

For details, run \"\${GTELEPORT_EXEC_NAME} help unpack\"
EOF
    return 1
  }

  # Do we have a Beamdownfile?
  local beamupfile
  local buvars

  beamupfile=$(gteleport_beamupfile_find "$SRC")
  [ -f "$beamupfile" ] || {
    gteleport_err_missingbeamup "$SRC"
    return 1
  }
  SRC=$(dirname "$beamupfile")
}
```

Approved for Public Release

```
echo "[git-teleport] Reading Beamupfile \"${beamupfile}\"..."

# Read and validate the beamupfile
declare -a beamup_rel_urls
declare -A beamup_files
buvars=$(gteleport_beamfile_read \
    "up" \
    "beamup_rel_urls" \
    "" \
    "beamup_files" \
    <"${beamupfile}") || return $?
eval "$buvars"
if [[ -z "${beamup_rel_urls[0]}" ]]; then
    echo -e "No repository URLs defined in '${beamupfile}'. Nothing to do."
    return 1
fi

# Validate the DEST_PREFIX, and determine the type
local type
type=$(gteleport_check_unpack_prefix "$DEST")
[ -n "$type" ] || return $?

echo "[git-teleport] Beamupfile contains ${#beamup_rel_urls[@]} repositories."
echo "[git-teleport] Searching for $type mirrors in \"${DEST}\"..."

# Find all destination repos. They must exist.
declare -A destination_urls
local relurl
for relurl in "${beamup_rel_urls[@]"; do
    destination_urls["$relurl"]=$(gteleport_find_receive_mirror \
        "$DEST" \
        "$relurl" \
        "$type" \
        "${beamupfile}")
    [ -n "${destination_urls[$relurl]}" ] || return $?
done

echo "[git-teleport] All mirrors found. Attempting $type update..."

# forall repositories
declare -i num_updated=0
local repourl
local bundlefile
local bundlerefsfile
local local_repo_dir
for relurl in "${beamup_rel_urls[@]"; do
    # Do we have a bundle?
    repourl="${destination_urls[$relurl]}"
    bundlefile="$SRC/${beamup_files[$relurl]}.bundle"
    bundlerefsfile="$bundlefile.refs"
    if [[ ! -f "$bundlefile" && ! -f "$bundlerefsfile" ]]; then
        printf "[git-teleport]    * %18s: no update\n" "$relurl"
        continue
    fi

    if [ "$type" = "local" ]; then
        # just fetch the bundle in the local repo
        if [ -e "$bundlefile" ]; then
            gteleport_git_fetchbundle "$repourl" "$bundlefile" || {
                gteleport_err_update_badfetch "$relurl" "$repourl" "$bundlefile"
                return 1
            }
        }
    }
}
```

Approved for Public Release

```
else
    true
fi

# if a .refs bundle exists, use it to prune
if [ -e "${bundlerefsfile}" ]; then
    gteleport_git_fetchbundle "$repourl" "${bundlerefsfile}" "yes" || {
        gteleport_err_update_badfetch \
            "$relurl" "$repourl" "${bundlefile}.refs"
        return 1
    }
else
    true
fi

# manually invoke the post-update hook if it exists
gteleport_git_run_post_update "$repourl" || true
elif [ "$type" = "network" ]; then
    # attempt to make or update the local mirror
    local_repo_dir=$(gteleport_url_to_cache_repo "$repourl") || \
        gteleport_die "Unable to calculate local repo for \"$repourl\""

    gteleport_git_update "$repourl" "$local_repo_dir" || {
        gteleport_err_update_badclone "$relurl" "$repourl" "$local_repo_dir"
        return 1
    }

    # fetch the bundle into it
    if [ -e "${bundlefile}" ]; then
        gteleport_git_fetchbundle "$local_repo_dir" "$bundlefile" || {
            gteleport_err_update_badfetch \
                "$local_repo_dir" "$local_repo_dir" "$bundlefile"
            return 1
        }
    else
        true
    fi

    # if a .refs bundle exists, use it to prune
    if [ -e "${bundlerefsfile}" ]; then
        gteleport_git_fetchbundle \
            "$local_repo_dir" "${bundlefile}.refs" "yes" || {
            gteleport_err_update_badfetch \
                "$local_repo_dir" "$local_repo_dir" "${bundlerefsfile}"
            return 1
        }
    else
        true
    fi

    # push the changes back up
    gteleport_git_pushto "$local_repo_dir" "$repourl" || {
        gteleport_err_update_badpush \
            "$relurl" "$repourl" "$local_repo_dir"
        return 1
    }
else
    gteleport_die "Unknown update type \"$type\""
fi

(( num_updated = num_updated + 1 ))

printf "[git-teleport]    * %18s: UPDATED → %s\n" "$relurl" "$repourl"
```

Approved for Public Release

```
done

echo "[git-teleport] Finished updating $num_updated $type Receive Mirrors."

return 0
}

function gteleport_cmd_fetch() {
# fetch command
local bdfilename
bdfilename=$(gteleport_beamdownfile_find "$2") || true
if [[ ! -f "$bdfilename" ]]; then
    gteleport_print helptext_nobdfilename
    return 1
fi

gteleport_fetch "$bdfilename"
}

function gteleport_cmd_bundle() {
# bundle command
local bdfilename
bdfilename=$(gteleport_beamdownfile_find "$2") || true
if [[ ! -f "$bdfilename" ]]; then
    gteleport_print helptext_nobdfilename
    return 1
fi

local made_any_bundles=1
gteleport_bundle "$bdfilename" '' 'made_any_bundles'
[[ "$made_any_bundles" = "1" ]] || {
    gteleport_print helptext_bundle_complete;
    gteleport_print helptext_commit_reminder
}
}

function gteleport_cmd_bundle_all() {
# bundle-all command
local bdfilename
bdfilename=$(gteleport_beamdownfile_find "$2") || true
if [[ ! -f "$bdfilename" ]]; then
    gteleport_print helptext_nobdfilename
    return 1
fi

local made_any_bundles=1
gteleport_bundle "$bdfilename" 'full' 'made_any_bundles'
[[ "$made_any_bundles" = "1" ]] || {
    gteleport_print helptext_bundle_complete;
    gteleport_print helptext_commit_reminder
}
}

function gteleport_cmd_commit() {
# commit command

local bdfilename
bdfilename=$(gteleport_beamdownfile_find "$2") || true
if [[ ! -f "$bdfilename" ]]; then
    gteleport_print helptext_nobdfilename
    return 1
fi
}
```

Approved for Public Release

```
local didcommit=1
gteleport_commit "$bdfilename" "didcommit"
[[ "$didcommit" = "1" ]] || gteleport_print helptext_committed
}

function gteleport_cmd_pack() {
    # pack command
    #
    # fetch + bundle + commit

    local bdfilename
    bdfilename=$(gteleport_beamdownfile_find "$2") || true
    if [[ ! -f "$bdfilename" ]]; then
        gteleport_print helptext_nobdfilename
        return 1
    fi

    gteleport_fetch "$bdfilename" || gteleport_die "Unable to fetch new updates"

    local made_any_bundles=1
    gteleport_bundle "$bdfilename" '' 'made_any_bundles'
    [[ "$made_any_bundles" = "1" ]] || gteleport_print helptext_bundle_complete

    local didcommit=1
    gteleport_commit "$bdfilename" "didcommit"
    [[ "$didcommit" = "1" ]] || gteleport_print helptext_committed
}

function gteleport_cmd_unpack() {
    # unpack command
    gteleport_unpack "$2" "$3"
}

function gteleport_cmd_skeleton() {
    # skeleton command
    shift
    gteleport_skeleton "$@" || return $?

    cat 1>&2 <<-EOF
Created skeleton repositories

git-teleport has made one or more bare, "server-style" repositories
at the requested locations. These repos are marked as git-teleport
mirrors. You can receive bundles into these repositories with

    ${GTELEPORT_EXEC_NAME} unpack SRC_BUNDLES DEST_REPOS

Pushes to these repositories are blocked via a pre-receive hook.

Although an attempt was made to set broad permissions on the
repositories, you might need to perform some "chmod -R"s to fix
the permissions.
EOF
}

function gteleport_cmd_update() {
    shift
    gteleport_cmd_pack "$@"
}

function gteleport_err_bfr_badroot() {
    local count="$1"
    local line="$2"
```

Approved for Public Release

```
cat 1>&2 <<-EOF
ERROR: line $count: unexpected root declaration
```

We are trying to read a Beamupfile, but we encountered the !root declaration

\$line

```
Root declarations are not valid in Beamupfiles. Did you
specify the correct directory in \`git teleport unpack\`?
EOF
}
```

```
function gteleport_err_bfr_dup() {
  local count="$1"
  local line="$2"
  cat 1>&2 <<-EOF
ERROR: line $count: duplicate URL
```

The URL

\$line

is defined multiple times in this Beamdownfile, which is not allowed. Consider removing duplicate entries from your Beamdownfile.

```
EOF
}
```

```
function gteleport_err_bfr_badrelative() {
  local count="$1"
  local line="$2"

  cat 1>&2 <<-EOF
ERROR: line $count: cannot convert to relative URL
```

The URL

\$line

has no corresponding "!root" entry. Root entries tell git-teleport what the base URLs are for this mirror set. We require relative URLs to make the directory of bundle files and to preserve the tree/directory structure at the remote site. If your URL is

<https://somewhere.org/repos/project.git>

then consider adding the root entry

!https://somewhere.org

to your Beamdownfile. Note the leading exclamation point.

```
EOF
}
```

```
function gteleport_err_bfr_dup_relative() {
  local count="$1"
  local line="$2"
  local rel_url="$3"
  cat 1>&2 <<-EOF
ERROR: line $count: duplicate relative URL
```

Multiple entries in this Beamdownfile shorten to the same

Approved for Public Release

relative URL:

```
$line → $rel_url
```

This can happen if you have a project with the same name and path under more than one !root entry, or if multiple copies of the same project are specified with different URL schemes.

Relative URLs are used by the receiving side to determine the storage location for the mirrors. As such, they must be unique. Consider removing duplicated repositories from your Beamdownfile.

```
EOF  
}
```

```
function gteleport_err_bfr_calclocal() {  
    local count="$1"  
    local line="$2"
```

```
    cat 1>&2 <<-EOF  
ERROR: line $count: unable to generate local file path
```

Git-teleport was unable to calculate a local file path from the URL

```
    $line  
EOF  
}
```

```
function gteleport_err_bfr_dup_local() {  
    local count="$1"  
    local line="$2"  
    local local_file_path="$3"
```

```
    cat 1>&2 <<-EOF  
ERROR: line $count: duplicate local file path
```

Multiple entries in this Beamdownfile shorten to the same relative local file path:

```
$line → $local_file_path
```

This can happen if your project paths differ only by case or non-alphanumeric characters, like underscores or punctuation.

Local file paths are used to write bundle files. Git-teleport uses special URL-rewriting rules to determine the path to the bundle, so it is always stored in a known location. Lower-case names are enforced for filesystems which do not support mixed-case, like NTFS.

Consider making your project names more distinct.
EOF
}

```
function gteleport_err_missingprefix() {  
    cat 1>&2 <<-EOF
```

```
ERROR: missing destination directory or URL prefix
```

You must provide the path or an URL prefix of the Receive Mirrors to update. These repositories will be overwritten with the content

Approved for Public Release

from the bundle files.

```
EOF
}
```

```
function gteleport_err_missingbeamup() {
  cat 1>&2 <<-EOF
  ERROR: missing Beamupfile
```

Your directory of bundle files

\$1

does not contain a Beamupfile. This file should have been generated on the sending side, but it is missing.

Are you using the correct bundle source directory?

This file lists the relative URLs of all the repositories in this Mirror Set-i.e., all the repositories that git-teleport should update. Git-teleport cannot run ``update`` without it.

```
EOF
}
```

```
function gteleport_err_prohibited_network() {
  local PREFIX="$1"

  cat 1>&2 <<-EOF
  ERROR: destination prefix "$PREFIX" is prohibited
```

The networked repository prefix

\$PREFIX

is prohibited for use with Receive Mirrors. git-teleport will **not** update repositories in this location.

Are you SURE you have the correct destination prefix?

```
EOF
```

```
if [[ "$PREFIX" = "file://"* ]]; then
  cat 1>&2 <<-EOF
  If you are attempting to update repositories on your
  local filesystem, use a path (like "/my/path") instead
  of "file:///...".
```

```
EOF
```

```
  return
  elif echo "$PREFIX" | \
    grep -E -v -q '^[a-zA-Z]{3,}:/|([a-zA-Z-_]+@)?[^/]{2,}:)'; then
    cat 1>&2 <<-EOF
    It looks like you might be trying to use a local file
    path as your prefix. git-teleport could not find the
    prefix, however, so it is treating it as a network URL.
    Check that you have the correct prefix.
```

```
EOF
```

```
  return
  else
    if [ "$GTELEPORT_MIRROR_URL_REGEX" = "^mirrors?\$" ]; then
      cat 1>&2 <<-EOF
      In your repo-hosting software, like Gitlab, search for a
```

Approved for Public Release

subdirectory named "/mirrors/". This is the prefix URL you want to use in \`git teleport unpack\`.

EOF

```
else
```

```
cat 1>&2 <<-EOF
```

In your repo-hosting software, like Gitlab, search for a subdirectory which matches GTELEPORT_MIRROR_URL_REGEX, which is currently:

```
$GTELEPORT_MIRROR_URL_REGEX
```

This is the prefix URL you want to use in \`git teleport unpack\`.

EOF

```
fi
```

```
fi
```

```
cat 1>&2 <<-EOF
```

For networked repositories, git-teleport requires that you keep your Receive Mirrors in a separate path from your normal repositories. This is to prevent git-teleport from OBLITERATING your original code or forks.

If for some reason you did not create your Receive Mirror repositories in a matching prefix, you'll need to relocate them. The URL must contain a subdirectory which matches GTELEPORT_MIRROR_URL_REGEX, which is currently

```
$GTELEPORT_MIRROR_URL_REGEX
```

EOF

```
}
```

```
function gteleport_err_frm_destnotfound() {  
    local URL="$1"  
    local TYPE="$2"
```

```
cat 1>&2 <<-EOF
```

ERROR: destination mirror "\$URL" not found

We could not find a Git repository at the following URL:

```
$URL
```

This is one of the repositories listed in the source Beamupfile,

```
$BEAMUPFILE
```

Check that you have provided the correct URL or path prefix to \`git teleport unpack\`!

EOF

```
if [ "$TYPE" = "local" ]; then
```

```
cat 1>&2 <<-EOF
```

You can create this Receive Mirror on your local filesystem with

```
git teleport skeleton "$URL"
```

EOF

```
else
```

```
cat 1>&2 <<-EOF
```

Approved for Public Release

This prefix appears to be a network path or otherwise does not exist on your local filesystem. To create the repository, follow the instructions in your repo-hosting software, like Gitlab. Ensure that:

1. The repository is created in a subdirectory. One of the path components must match the following regexp EXACTLY:

```
$GTELEPORT_MIRROR_URL_REGEX
```

This is a safety measure to ensure that git-teleport won't accidentally overwrite original repos or forks.

2. The repository has unrestricted read/write permissions for everyone in your workgroup. Otherwise, other users won't be able to update it.

```
EOF
fi
}
```

```
function gteleport_err_frm_prohibited_local() {
    cat 1>&2 <<-EOF
    ERROR: destination repository "$1" is prohibited
```

The following repository is not marked as an git-teleport Receive Mirror:

```
$1
```

Are you sure you have the correct destination prefix?

You might be trying to overwrite forked or original repositories with updates from git-teleport. You almost certainly do not want to do this.

If you are ABSOLUTELY CERTAIN that you want this repository to be OBLITERATED and replaced with the contents of the bundle files, then you may place an "teleport-mirror" file in the root directory of the repository. But this is probably not what you want.

You can avoid this error message in the future by creating your repositories with `\`git teleport skeleton\``.

```
EOF
}
```

```
function gteleport_err_update_badclone() {
    local repo_rel_url="$1"
    local repo_url="$2"
    local local_repo_dir="$3"
    cat 1>&2 <<-EOF
    ERROR: Unable to mirror "$repo_rel_url"
```

Git could not create a local mirror of the remote repository

```
$repo_url
```

on your local filesystem in

```
$local_repo_dir
```

Approved for Public Release

See the output from Git above.

git-teleport needs to make a local copy of this repository in order to update it, but this step failed. Check that you have full permissions on the networked repository.

```
EOF
}
```

```
function gteleport_err_update_badfetch() {
    local repo_rel_url="$1"
    local repo_url="$2"
    local bundle_file="$3"
    cat 1>&2 <<-EOF
ERROR: Unable to fetch into "$repo_rel_url"
```

Git rejected our attempt to fetch a bundle into the local Git repository

```
    $repo_url
```

from

```
    $bundle_file
```

See the output from Git above.

For "missing commits," you may have missed one or more previous incremental updates. Incremental updates must be applied in order. If you are missing commits, see if there are previous bundles to apply. You may need to have the sending side transmit full bundles.

Also check for permissions issues on the destination repository.

```
EOF
}
```

```
function gteleport_err_update_badpush() {
    local repo_rel_url="$1"
    local repo_url="$2"
    local local_repo_dir="$3"
    cat 1>&2 <<-EOF
ERROR: Unable to push to "$repo_rel_url"
```

Git rejected our attempt to push mirror changes to

```
    $repo_url
```

from

```
    $local_repo_dir
```

See the output from Git above.

The bundle file was successfully applied locally, but the changes must be pushed to the network repository to complete the update. This push failed.

It is likely that you do not have *full* permissions on this repository. Updates may change protected branches or do "unusual" things like change or delete tags. Smart repository hosting software frequently denies these changes to less privileged users.

Approved for Public Release

```
Fix the permissions issues, then retry this \`git teleport unpack\`
operation.

EOF
}

function gteleport_err_verify_refsbundle() {
    local repo_rel_url="$1"
    local bundle_file="$2"
    local refs_repo="$3"
    local refs_bundle="$4"
    cat 1>&2 <<-EOF
ERROR: Unable to create refs bundle for "$repo_rel_url"

git-teleport attempted to generate a ".bundle.refs" file for

    $repo_rel_url

but the bundle failed \`git bundle verify\`.

A ".bundle.refs" file contains the full listing of refs from your source
repository. The receiving side needs this file to prune / cleanup deleted
refs. We will skip generating the refs bundle for now. Please report this
error as a potential git-teleport bug.

EOF
}

function gteleport_err_bad_refsbundle() {
    local repo_rel_url="$1"
    local bundle_file="$2"
    local refs_repo="$3"
    local refs_bundle="$4"
    cat 1>&2 <<-EOF
ERROR: Unable to create refs bundle for "$repo_rel_url"

git-teleport attempted to generate a ".bundle.refs" file for

    $repo_rel_url

but the bundled refs don't match the repository.

REPO REFS

$refs_repo

BUNDLE REFS

$refs_bundle

A ".bundle.refs" file contains the full listing of refs from your source
repository. The receiving side needs this file to prune / cleanup deleted
refs. We will skip generating the refs bundle for now. Please report this
error as a potential git-teleport bug.

EOF
}

function gteleport_print helptext_nobdfile() {
    cat 1>&2 <<-EOF
```

Approved for Public Release

Cannot find Beamdownfile

git-teleport looks for a Beamdownfile in the same directory as itself, or in the directory or file specified on the command line. Did you run bootstrap? See the help option for more details.

EOF

}

```
function gteleport_print helptext_mirrorset() {
```

```
  cat <<-'EOF'
```

A MIRROR SET is a list of repositories to mirror to a UNIQUE DESTINATION. Each time you run `./teleportw bundle``, new Git Bundles are generated for all repositories that have changed. Git-teleport will remember the contents of previous bundles, and subsequent updates will use minimally-sized incremental bundles.

If you plan to send a repository to more than one location, you'll need a separate Mirror Set for each.

Git-teleport stores all of this state information in a Git repository. You can share this repository among multiple contributors, and they can all run the update process.

EOF

}

```
function gteleport_print helptext_bundle_complete() {
```

```
  cat <<-'EOF'
```

Git-teleport has finished bundling your repositories. If there are updates, collect the `.bundle` files from the above path, and transfer them to your destination system.

EOF

}

```
function gteleport_print helptext_commit_reminder() {
```

```
  cat <<-'EOF'
```

You need to record the state of everything you have mirrored by committing to the Mirror Set repository. Once you have done this, future calls to git-teleport will ignore everything that you have already transferred. To make this commit, run:

```
    ${GTELEPORT_EXEC_NAME} commit
```

If, in the future, you discover that there is something wrong with these bundles, you can revert this commit to try again. If you push this commit, others will know what you have already transferred. This will make their own bundles smaller.

EOF

}

```
function gteleport_print helptext_committed() {
```

```
  cat <<-'EOF'
```

Git-teleport has committed changes to its "Beamdownstate" database files in your name.

If, in the future, you discover that there is something wrong with these bundles, you can revert this commit to try again. If you push this commit, others will know what you have already transferred. This will make their own bundles smaller.

EOF

Approved for Public Release

```
}

function gteleport_cmd_bootstrap() {
    # Bootstrap command

    if [[ -n "$2" ]]; then
        gteleport_bootstrap "$2"
        return 0
    fi

    echo -e "You are about to create a new MIRROR SET.\n"
    gteleport_print helptext_mirrorset
    cat <<-EOF

    If someone else has already made a mirror set for this destination, you
    should use it. Otherwise, enter a new directory for your new mirror set.
    It must be located somewhere outside of all other Git repositories. You
    should name it based on where it's going--not what is in it.
```

EOF

```
    read -r -p "Directory: " "NEW_DIR"
    echo
    if [[ -n "$NEW_DIR" ]]; then
        gteleport_bootstrap "$NEW_DIR"
        return $?
    fi
}

function gteleport_cmd_version() {
    # Print version and exit

    echo "git-teleport v${GTELEPORT_VERSION}"
    return 0
}

function gteleport_cmd_help() {
    # Print help

    if [[ -n "$2" ]]; then
        cmd=$(echo "$2" | tr - _)
        if [[ $(type -t "gteleport_cmd_help_${cmd}") = "function" ]]; then
            "gteleport_cmd_help_${cmd}" "$@"
            return 1
        else
            gteleport_die "No help available for '$2'"
        fi
    fi
}
```

```
cat <<-EOF
Usage: ${GTELEPORT_EXEC_NAME} COMMAND
${gteleport_cmd_version}
```

Where COMMAND is one of:

```
bootstrap - Create a new Mirror Set
help      - Help for a command
pack      - Send bundles from your repos to the downstream site
skeleton  - Make empty repository for receiving teleport bundles
unpack    - Receive bundles from the upstream site into your repos
version   - Print version and exit
```

For details on a command, run `\`${GTELEPORT_EXEC_NAME} help COMMAND\``.
For utility commands, see `\`${GTELEPORT_EXEC_NAME} help more\``.

Approved for Public Release

For a full man-page, run `\`git help teleport\``.

Send your data to the Mirror Universe!

git-teleport is a mirroring tool for Git repositories. To get started, run the bootstrap command.

Environment variables:

```
GTELEPORT_CACHE           : "${GTELEPORT_CACHE}"
GTELEPORT_BEAMDOWNFILE    : "${GTELEPORT_BEAMDOWNFILE}"
GTELEPORT_MIRROR_URL_REGEX : "${GTELEPORT_MIRROR_URL_REGEX}"
```

EOF

```
    return 1
}
```

```
function gteleport_cmd_help_bootstrap() {
    # Help for bootstrap
```

```
    cat <<-EOF
Usage: ${GTELEPORT_EXEC_NAME} bootstrap [DIRECTORY]
```

Creates a new Mirror Set. A new Git repository will be initialized in DIRECTORY and populated with an empty Beamdownfile. Add a list of repositories that you want to mirror to the Beamdownfile, commit them, and then run the git-teleport wrapper in that directory,

```
    ./teleportw pack
```

to create your first set of bundles. Additional instructions can be found in the template Beamdownfile after you bootstrap.

This command creates a new Mirror Set. To use an existing Mirror Set that someone else created, simply obtain the Git repository for it. Then use the git-teleport wrapper, 'teleportw', within.

EOF

```
gteleport_print helptext_mirrorset

echo
}
```

```
function gteleport_cmd_help_fetch() {
    # Help for bootstrap
```

```
    cat <<-EOF
Usage: ${GTELEPORT_EXEC_NAME} fetch [BEAMDOWNFILE]
```

Updates local mirrors listed in your Beamdownfile.

Git-teleport needs to keep a local copy of all repositories in your Mirror Set. Existing working copies will not do---a "---mirror" clone is required. These mirrors are stored in your user profile's cache directory, which can be overridden by the `\${GTELEPORT_CACHE}` environment variable. Currently, git-teleport will look for its cache in

```
    '${gteleport_cachedir}'
```

Repositories are de-duplicated across all of your Mirror Sets.

The fetch command will update all of the repositories specified in your

Approved for Public Release

Beamdownfile. You can specify the path to a Beamdownfile on the command line. Otherwise, git-teleport will look for it in the directory in which it resides.

```
EOF
}
```

```
function gteleport_cmd_help_more() {
    cat <<-EOF
Usage: ${GTELEPORT_EXEC_NAME} COMMAND
```

All COMMANDs are:

```
bootstrap - Create a new Mirror Set
bundle     - Make new incremental bundles for the downstream site
bundle-all - Make new full-history bundles for the downstream site
commit     - Record bundled state of all mirrored repositories
fetch      - Update local mirrors in a Mirror Set
help       - Help for a command
pack       - Alias for fetch + bundle + commit
skeleton   - Make empty repository for receiving teleport bundles
unpack     - Receive bundles from the upstream site into your repos
update     - Deprecated alias for pack (fetch + bundle + commit)
version    - Print version and exit
```

For details on a command, run `\`${GTELEPORT_EXEC_NAME} help COMMAND\``.

```
EOF
}
```

```
function gteleport_cmd_help_bundle() {
    # Help for bootstrap

    cat <<-EOF
Usage: ${GTELEPORT_EXEC_NAME} bundle [BEAMDOWNFILE]
```

Generates incremental bundles for a Beamdownfile.

This command outputs a new git-bundle file for all changed repositories in your Beamdownfile. If a repository has no changes, then no bundle is output. Git-teleport keeps a record of all the "refs---" i.e., named things like branches and tags---in the mirrored repositories. When bundles are generated, this record is updated. Git-teleport uses this record to avoid bundling things that it knows are already present on the remote side.

Output files are written to the directory where your Beamdownfile resides. The latest bundles will always be available in

`\${BEAMDOWNFILE}/bundles/latest`

Previous bundles are kept until manually deleted. It is perfectly safe to delete the `\${BEAMDOWNFILE}/bundles` directory, if you're finished with those bundles.

When this command completes, you will likely need to use the "commit" command to save the refs database.

```
EOF
}
```

```
function gteleport_cmd_help_bundle_all() {
    cat <<-EOF
Usage: ${GTELEPORT_EXEC_NAME} bundle-all [BEAMDOWNFILE]
```

Approved for Public Release

Generates full bundles for a Beamdownfile.

This command behaves similarly to "bundle," but it makes full-history bundles. Use it when something has gone wrong with the incremental bundles and they are failing to "fetch" properly. Output files are written to the directory where your Beamdownfile resides. The latest bundles will always be available in

```
\$BEAMDOWNFILE/bundles/latest
```

Previous bundles are kept until manually deleted. It is perfectly safe to delete the \\$BEAMDOWNFILE/bundles directory, if you're finished with those bundles.

When this command completes, you will likely need to use the "commit" command to save the refs database.

```
EOF  
}
```

```
function gteleport_cmd_help_pack() {  
  cat <<-EOF  
Usage: ${GTELEPORT_EXEC_NAME} pack [BEAMDOWNFILE]
```

"One touch" express packing command. This command combines the fetch, bundle (incremental), and commit commands into one.

The fastest way to run the packer is via the git-teleport wrapper script in the Mirror Set. Just \`cd\` to the Mirror Set repo, and then run \`../teleportw pack\`.

When this command concludes, freshly-packed bundle files will be available in

```
\$BEAMDOWNFILE/bundles/latest
```

You can also specify a path to a Beamdownfile on the command line or the \\$GTELEPORT_BEAMDOWNFILE environment variable. If unspecified, Git-teleport will look for a Beamdownfile in the directory in which it resides.

Previous bundles are kept until manually deleted. It is perfectly safe to delete the \\$BEAMDOWNFILE/bundles directory, if you're finished with those bundles.

You can also run all the steps of the packing process individually. See help for the following commands:

```
fetch  
bundle  
bundle-all  
commit
```

```
EOF  
}
```

```
function gteleport_cmd_help_unpack() {  
  cat <<-EOF  
Usage: ${GTELEPORT_EXEC_NAME} unpack SRC_BUNDLE_DIR DEST_PREFIX
```

Receive bundle files from SRC_BUNDLE_DIR, and place their commits on corresponding Receive Mirrors located beneath DEST_PREFIX. This action updates *your* repositories, updating them with the *remote* content from the bundle files.

If you just unzipped an archive full of bundle files at

Approved for Public Release

~/Downloads/latest, and the root directory on your server for these repositories is /share/repositories/mirrors, then run

```
{GTELEPORT_EXEC_NAME} unpack \\  
~/Downloads/latest \\  
/share/repositories/mirrors
```

You can also update local mirrors via SSH or other supported protocols

```
{GTELEPORT_EXEC_NAME} unpack \\  
~/Downloads/latest \\  
git@somewhere.example:mirrors/
```

If the destination repository is on your local filesystem, git-teleport will only update it if it was created with ``git teleport skeleton`` or it is configured as described in the README.

If the destination is a network URL prefix, then one of the path components must be "mirror" or "mirrors". PLACE ONLY NON-ORIGINAL MIRRORS under this directory! git-teleport will freely overwrite these repos without confirmation.

A Mirror Set is not required in order to receive bundles---only to generate them.

```
EOF  
}
```

```
function gteleport_cmd_help_skeleton() {  
    cat <<-EOF  
Usage: {GTELEPORT_EXEC_NAME} skeleton REPO1 REPO2 .. REPO_N
```

Prepares your local filesystem to receive updates from a remote git-teleport. Bundles can only be unpacked into existing repositories. This command creates repositories which are compatible with the "unpack" command. New repos will be empty and ready to receive bundles.

This command is only necessary the FIRST TIME you begin RECEIVING a bundle. After that, you just need to run the "unpack" command.

Each argument should be a full path and name of the repository to create, such as /net/share/repositories/incoming/mycoolproject.git. Relative paths are interpreted relative to the current working directory.

Repositories are created bare (server-style), shared, and with pushes prevented.

```
EOF  
}
```

```
function gteleport_cmd_help_help() {  
    echo "May your data live long and prosper."  
}
```

```
function gteleport_cmd_help_update() {  
    shift  
    gteleport_cmd_help_pack "$@"  
}
```

```
if [[ "$0" == "${BASH_SOURCE[0]}" ]]; then  
    if [[ -z "$1" ]]; then  
        cmd="help"  
    else  
        cmd=$(echo "$1" | tr - _)  
    fi  
fi
```

Approved for Public Release

```
fi

if [[ $(type -t "gteleport_cmd_${cmd}") = "function" ]]; then
    set -e
    set -o pipefail
    "gteleport_cmd_${cmd}" "$@"
    exit $?
fi

gteleport_die "Unknown command '$1'. Try running: '$0' help"
fi
```

A.3. File Listing “share/template/receive/teleport-mirror”

This file marks this repository as a git-teleport receive mirror.

A.4. File Listing “share/template/receive/hooks/post-receive.mirror.sample”

```
#!/bin/bash
# Automatically update a fork
#
# This is a githook and must be installed as hooks/post-receive
# in your MIRROR REPOSITORY. Remember to make it executable
# by all.
#
# If you use our standard layout convention for repositories,
# this script will attempt to locate your fork and update it.
# The system running this script must have direct filesystem
# access to your fork.
#
# If this repo is located at
#
#     /whatever/mirror/intranet/some/path.git
#
# Then this script will search for your mirror at
#
#     /whatever/some/path.git
#
# By default, a branch named `develop` from the upstream will be
# available in your fork as `intranet/develop`. Tags will not be
# pushed. You can customize this behavior by changing the
# `git push` line below.
#
# Due to a lack of a missing --prune option, this script doesn't
# work on CentOS 6. For CentOS 6, you may omit --prune or configure
# the fork as a named remote.
#
# TEST THIS FIRST by running the script with the DRY_RUN_POSTRECEIVE
# environment variable set, like this:
#
# ```bash
# cd /git/mirror/intranet/SomeRepo.git
# DRY_RUN_POSTRECEIVE=y ./hooks/post-receive
# ```

gteleport_push_to_fork() {
    # Usage: gteleport_push_to_fork REPO NAMESPACE ARG0 ARG1 ... ARGN
```

Approved for Public Release

```
#
# !!! EDIT ME!!!
#
# Performs a git push to update a fork repository located at REPO.
# A branch named "develop" in the upstream will be pushed to
# NAMESPACE/develop. You can edit the push line to make this
# method do what you want. The remaining ARGS are passed on to
# `git`: this is used only to pass --dry-run if requested

local REPO="$1"
local NAMESPACE="${2:-mirror}"
shift; shift

"$GIT" push \
  -f \
  --no-tags \
  --prune \
  "${@}" \
  "$REPO" \
  '+refs/heads/*:refs/heads/'"$NAMESPACE"'/*'
}

gteleport_find_fork() {
# Usage: find_fork PWD
# Find where the fork might be.
#
# PWD should be set to the current working directory,
# which is assumed to be the ".git" directory for this repo.
# This is always the case when git runs this script.
#
# This method performs a few sanity checks but does not
# actually check that the path exists.
#
# The first word of output is the mirror set source name.
# The remaining words are the path to the fork.

local workdir="$1"

local source_name=""
local fork_path=""
declare -a mirror_parts

repo_base="${workdir%%/mirror*}"      # everything before /mirror
mirror_path="${workdir##${repo_base}/}" # mirror/...

# nibble off the first two components: mirror/whatever/keep_this
IFS='/' read -r -a mirror_parts <<<"$mirror_path"

# check that we start with /mirror
[[ "${mirror_parts[0]}" =~ ^mirrors?$ ]] || return 1

# calculate path to the fork
printf -v fork_path '%s' "${mirror_parts[@]:2}"
[ "${fork_path}" != '/' ] || return 1
fork_path="${repo_base}${fork_path}"

# sanity check: path not empty and not the input path
[ -n "$fork_path" ] || return 1
[ "$fork_path" != "z${workdir}" ] || return 1

# ensure we have a space-free name for the mirror source
source_name="${mirror_parts[*]:1:1}"
source_name="${source_name//[[:blank:]]/}"
}
```

Approved for Public Release

```
source_name="${source_name:-mirror}"

echo "$source_name $fork_path"
}

gteleport_dry_run_replace_git() {
    # Usage: gteleport_dry_run_replace_git ARGS...
    #
    # Prints ARGS. Useful for dry runs.

    echo git "$@"
}

# If executing
if [[ "$0" == "${BASH_SOURCE[0]}" ]]; then
    set -e

    # Ignore stdin (prevents SIGPIPE on Git<2.2)
    exec </dev/null

    GIT="git"

    # don't propagate a push override (for our pre-receive)
    unset -v CONFIRM_MIRROR_UPDATE
    [ "z${GIT_PUSH_OPTION_0}" != zCONFIRM_MIRROR_UPDATE ] || \
        unset -v GIT_PUSH_OPTION_0

    # Try to locate your forked repository.
    mirror_source=""           # mirror set name (i.e., "intranet")
    autodetected_fork=""       # repo we think is a fork of this

    # Try to find a fork repository
    read -r mirror_source autodetected_fork \
        <<<"$(gteleport_find_fork "$(pwd)")" || exit 0

    # If the fork doesn't exist, exit silently
    [ -e "$autodetected_fork/objects" ] || exit 0

    # If environment or push args dictate, do a dry run
    if [ -n "$DRY_RUN_POSTRECEIVE" ] || \
        [ "z${GIT_PUSH_OPTION_0}" = "zDRY_RUN_POSTRECEIVE" ]; then
        # dry run
        ( GIT="gteleport_dry_run_replace_git" && \
            gteleport_push_to_fork "$autodetected_fork" "$mirror_source" --dry-run )
        gteleport_push_to_fork "$autodetected_fork" "$mirror_source" --dry-run
    else
        gteleport_push_to_fork "$autodetected_fork" "$mirror_source"
    fi

    printf "Updated fork \"%s\"\n" "$autodetected_fork"
fi
```

A.5. File Listing “share/template/receive/hooks/pre-receive”

```
#!/bin/bash
#
# Prevents pushes unless CONFIRM_MIRROR_UPDATE is provided as either
# an environment variable or a git push option (where supported).
# If the standard layout is in use, a helpful error message is
# printed.
```

Approved for Public Release

```
gteleport_find_fork() {
# Usage: find_fork PWD
# Find where the fork might be.
#
# PWD should be set to the current working directory,
# which is assumed to be the ".git" directory for this repo.
# This is always the case when git runs this script.
#
# This method performs a few sanity checks but does not
# actually check that the path exists.

declare -a mirror_parts
local workdir="$1"
local fork_path=""

repo_base="${workdir%/mirror*}"          # everything before /mirror
mirror_path="${workdir##${repo_base}/}"  # mirror/...

# nibble off the first two components: mirror/whatever/keep_this
IFS='/' read -r -a mirror_parts <<<"$mirror_path"

# check that we start with /mirror
[[ "${mirror_parts[0]}" =~ ^mirrors?$ ]] || return 1
printf -v fork_path '/%s' "${mirror_parts[@]:2}"
[ "${fork_path}" != '/' ] || return 1
fork_path="${repo_base}${fork_path}"

# sanity check: path not empty and not the input path
[ -n "$fork_path" ] || return 1
[ "$fork_path" != "z${workdir}" ] || return 1
echo "$fork_path"
}

[ -z "${CONFIRM_MIRROR_UPDATE}" ] || exit 0
[ "z${GIT_PUSH_OPTION_0}" != zCONFIRM_MIRROR_UPDATE ] || exit 0

echo "Prevented push to mirror repository!"
echo
echo "This mirror repository should only be updated via git-teleport."
echo "You almost certainly do not want to push here. You probably"
echo "meant to push to a fork of this repo instead."
echo ""

potential_fork=$(gteleport_find_fork "$(pwd)") || true

if [ -n "$potential_fork" ] && [ -e "$potential_fork/objects" ]; then
echo "We searched your filesystem and found a potential fork,"
echo ""
echo "    $potential_fork"
echo ""
echo "Perhaps you meant this remote instead?"
else
echo "Please use a different remote and try your push again."
echo "For further information, search the git-teleport README.md"
echo "for \"FORKS OF GIT-TELEPORT MIRRORS\""
fi

exit 1
```

A.6. File Listing “share/template/send/.gitignore”

/bundles

A.7. File Listing “share/template/send/Beamdownfile”

```
# BEAMDOWNFILE
#
# This file is a list of repositories to mirror with git-teleport. Enter
# one fully-qualified URL per line. You can use any URL scheme supported
# by git. An example Beamdownfile is
#
# # BEAMDOWNFILE
# ! https://extranet.mycorp.example/git
# ! ssh://git@internal.mycorp.example
# https://extranet.mycorp.example/git/ProjectAlpha.git
# ssh://git@internal.mycorp.example/ProjectBravo.git
# ssh://git@internal.mycorp.example/mysubfolder/ProjectCharlie.git
#
# Lines which begin with an exclamation point ("!") are interpreted as
# repository roots. The output bundles will be made *relative* to these
# locations. The above Beamdownfile will update the following repositories
# on the destination system.
#
# ProjectAlpha.git
# ProjectBravo.git
# mysubfolder/ProjectCharlie.git
#
# The server or directory structure which is matched by the !roots will be
# stripped away. All repository URLs must match one root, byte-for-byte,
# including the protocol.
#
# This file may be shared across multiple users. As such, DO NOT include your
# unique username in any of the URLs. If you need to supply your username, do
# something like
# $ git config --global credential."https://rome".username sparticus
#
```

A.8. File Listing “share/template/share/template/send/teleportw”

```
#!/bin/bash
#
# Git-Teleport Wrapper
#
# This script locates the main git-teleport installation in your user
# folder and passes all commands through to it.

GTELEPORT_SEARCH_PATH=(
    "${HOME}/git-teleport/bin"
    "${HOME}/.git-teleport/bin"
)

function gteleport_die() {
    # Usage: gteleport_die STRING
    #
    # Terminate execution with the given message.
    echo -e "$@" 1>&2
    exit 1
}
```

Approved for Public Release

```
function gteleport_strjoin() {
    # Usage: gteleport_strjoin SEP ARRAY
    #
    # Joins all elements of ARRAY, separated by SEP.
    local IFS="$1"; shift; echo "$*";
}

function gteleport_exe_find() {
    # Usage: gteleport_exe_find
    #
    # Finds git-teleport executable. Returns 0 and prints path if found.
    # Otherwise, returns non-zero.

    local joinpath
    joinpath=$(gteleport_strjoin ":" "${GTELEPORT_SEARCH_PATH[@]}")

    PATH="${PATH}:${joinpath}" command -v "git-teleport"
}

function gteleport_path_to_running_script() {
    # Usage: gteleport_path_to_running_script
    #
    # Gets the path to the running bash script, assuming that
    # this function was invoked by a script. If the path was
    # obtained, it is echoed to stdout, and true is returned.
    # Otherwise, false is returned.

    if [[ -n "${GTELEPORT_EXE_PATH}" && -f "${GTELEPORT_EXE_PATH}" ]]; then
        echo -n "${GTELEPORT_EXE_PATH}"
        return 0
    fi

    local prg="$0"
    local dir

    if [ ! -e "$prg" ]; then
        case "$prg" in
            (*/) return 1;;
            (*) prg=$(command -v -- "$prg") || return 1;;
        esac
    fi
    dir=$(
        cd -P -- "$(dirname -- "$prg)" && pwd -P
    ) || return 1
    prg="$dir"/$(basename -- "$prg") || return 1

    echo -n "$prg"
    return 0
}
```

```
function gteleport_die_cantfind() {
    cat > /dev/stderr <<-EOF
```

Cannot find the git-teleport executable shell script.

We looked for git-teleport on your `\$PATH`, with some standard storage locations appended for good measure:

```
$PATH=$(gteleport_strjoin ":" "${GTELEPORT_SEARCH_PATH[@]}")
```

If you do not have the main git-teleport repository, you should clone it to your `\$HOME` directory. We also recommend adding

Approved for Public Release

\\${HOME}/git-teleport/bin to your \\${PATH}.

EOF

```
    gteleport_die ""
}

if [[ "$0" == "${BASH_SOURCE[0]}" ]]; then
    # Find the main git-teleport script. We'll call it in a moment.
    GTELEPORT_EXE=$(gteleport_exe_find) || gteleport_die_cantfind

    # Find this wrapper script, and set GTELEPORT_BEAMDOWNFILE if unset
    self=$(dirname "$(gteleport_path_to_running_script)")

    GTELEPORT_BEAMDOWNFILE="${GTELEPORT_BEAMDOWNFILE:-${self}}" \
    GTELEPORT_EXEC_NAME="$0" \
    "${GTELEPORT_EXE}" "$@"

    exit $?
fi
```

A.9. File Listing

“share/template/send/Beamdownstate/.gitignore”

empty file

A.10. File Listing “test/run.sh”

```
#!/usr/bin/env test/bats/libexec/bats
#
# Invoke this file from the top-level git-teleport directory as
#   test/run.sh
#
#
# Load git-teleport and save path to it
export GTELEPORT_EXE_PATH="${BATS_TEST_DIRNAME}/../bin/git-teleport"
load "${GTELEPORT_EXE_PATH}"
EVILSPOCK_DIR=$(dirname "$GTELEPORT_EXE_PATH")

# Load post-receive sample script for gteleport_find_fork()
load "${EVILSPOCK_DIR}/../share/template/receive/hooks/post-receive.mirror.sample"

# Setup script: before each test, makes temporary directory
setup() {
    GTELEPORT_TESTDIR=$(mktemp -d 'gteleport_unit.XXXXXXXXXX' --tmpdir)
    mkdir -p "${GTELEPORT_TESTDIR}"
    touch "${GTELEPORT_TESTDIR}/.safedelelete"
}

# Teardown script: after each test, removes temporary directory
teardown() {
    if [[ -d "${GTELEPORT_TESTDIR}" ]] && \
        -f "${GTELEPORT_TESTDIR}/.safedelelete" ]]; then
        rm -rf "${GTELEPORT_TESTDIR}"
    fi
}
```

Approved for Public Release

```
function gteleport_test_mkbeamdown {
    # gteleport_test_mkbeamdown MIRRORSET SRC_REPO_PATH
    echo "!"$3" > "$1/Beamdownfile"
    echo "$2" >> "$1/Beamdownfile"
    gteleport_rund "$1" git commit -a -m "Add test beamdownfile"
}

function gteleport_test_fakecommit_make {
    # gteleport_test_fakecommit_make REPO_PATH SOME_TEXT
    gteleport_rund "$1" git commit --allow-empty -m "$2"
}

function gteleport_test_fakecommit_check {
    # gteleport_test_fakecommit_check REPO_PATH CHECK_TEXT BRANCH
    local branch="${3:-master}"
    pushd "$1"
    local chktxt=$(git log "$branch" --pretty=format:"%s" -n 1 | \
        head --lines 1)
    popd
    [[ "$chktxt" = "$2" ]]
}

function gteleport_test_fakecommit_hash {
    # gteleport_test_fakecommit_hash REPO_PATH
    gteleport_rund "$1" git log --pretty=format:"%H" -n 1
}

function gteleport_test_fakecommit_taghash {
    # gteleport_test_fakecommit_taghash REPO_PATH TAG
    #
    # Obtain the hash for the tag object TAG... and NOT the commit to which
    # it points.
    gteleport_rund "$1" git log -n 1 --pretty=format:"%H" "$2"
}

function gteleport_test_tageqcommit {
    # gteleport_test_fakecommit_make REPO_PATH TAG COMMIT
    local tag=$(gteleport_rund "$1" git rev-parse "$2"^{})
    local commit=$(gteleport_rund "$1" git rev-parse "$3")
    [[ -n "$tag" && -n "$commit" && "$tag" = "$commit" ]]
}

@test "temp directory exists" {
    [[ -d "$GTELEPORT_TESTDIR" ]]
}

@test "gteleport_die" {
    run gteleport_die "sorry, we had a fail"
    [ "$status" -eq 1 ]
    [ "${lines[0]}" = "sorry, we had a fail" ]
}

@test "gteleport_path_to_running_script" {
    # Just check that we succeed
    gteleport_path_to_running_script
}

@test "gteleport_cachedir" {
    # Check that we can obtain a cache directory on this system
    run gteleport_cachedir
    [ "$status" -eq 0 ]
    [[ -d "${lines[0]}" ]]
}
```

Approved for Public Release

```
@test "gteleport_slugify" {
  run gteleport_slugify \
    "this string is much much much too too too long for us to tolerate"
  [[ "${lines[0]}" = \
    "ring-is-much-much-much-much-too-too-too-long-for-us-to-tolerate" ]]
  run gteleport_slugify \
    "ssh://nohost.org/somefiles"
  [[ "${lines[0]}" = "nohost-org-somefiles" ]]
  run gteleport_slugify \
    "http://nobody@nohost.org/somefiles"
  [[ "${lines[0]}" = "nohost-org-somefiles" ]]
  run gteleport_slugify "€¥@≤≥βπ"
  echo "${lines[0]}" | grep -E '^[A-Za-z0-9\~]+${}'
  run gteleport_slugify "MiXEdCAsE"
  [[ "${lines[0]}" = "mixedcase" ]]
  run gteleport_slugify "in-dash-es"
  [[ "${lines[0]}" = "in-dash-es" ]]
  run gteleport_slugify "un_der_score"
  [[ "${lines[0]}" = "un-der-score" ]]
  run gteleport_slugify "not\\allowed/in/filenames"
  [[ "${lines[0]}" = "not-allowed-in-filenames" ]]
  run gteleport_slugify "///some/stuff//"
  [[ "${lines[0]}" = "some-stuff" ]]
}

@test "gteleport_strip_prefixes" {
  run gteleport_strip_prefixes "/this/is" "/not/matched"
  [[ "${lines[0]}" = "/this/is" ]]

  run gteleport_strip_prefixes "/this/is/matched" "/this" "/is"
  [[ "${lines[0]}" = "/is/matched" ]]

  run gteleport_strip_prefixes "/this/is/matched" "/is" "/this"
  [[ "${lines[0]}" = "/is/matched" ]]

  run gteleport_strip_prefixes "noop"
  [[ "${lines[0]}" = "noop" ]]

  run gteleport_strip_prefixes
  [[ "${lines[0]}" = "" ]]
}

@test "gteleport_urldecode" {
  run gteleport_urldecode \
    "%E2%82%AC%C2%A5%C2%A9%E2%89%A4%E2%89%A5%CE%B2%CF%80%20me+me+me"
  [[ "${lines[0]}" = "€¥@≤≥βπ me me me" ]]
}

@test "gteleport_url_to_cache_repo" {
  run gteleport_url_to_cache_repo "https://nowhere.org/somerepo.git"
  [[ $(basename "${lines[0]}") = "nowhere-org-somerepo-git" ]]
}

@test "gteleport_url_to_projectname" {
  run gteleport_url_to_projectname \
    "https://code.org/stuff/git/%E2%82%AC%C2%A5ProJEct.gIT"
  echo "${lines[0]}" | grep -E '^[A-Za-z0-9\~]+${}'
  echo "${lines[0]}" | grep -E 'project${}'
}

@test "gteleport_url_is_relative" {
  run gteleport_url_is_relative "/root/path"
```

Approved for Public Release

```
[ "$status" -ne 0 ]
run gteleport_url_is_relative "//unproto/path"
[ "$status" -ne 0 ]
run gteleport_url_is_relative "c:/windows"
[ "$status" -ne 0 ]
run gteleport_url_is_relative "c:\\windows"
[ "$status" -ne 0 ]
run gteleport_url_is_relative "ssh://somewhere"
[ "$status" -ne 0 ]
run gteleport_url_is_relative "HTTps://somewhere"
[ "$status" -ne 0 ]
run gteleport_url_is_relative "../yes/but/not/wanted"
[ "$status" -ne 0 ]
run gteleport_url_is_relative "bad../escape"
[ "$status" -ne 0 ]
run gteleport_url_is_relative "bad/.."
[ "$status" -ne 0 ]
run gteleport_url_is_relative "../bad"
[ "$status" -ne 0 ]
run gteleport_url_is_relative "windows\\..\\bad"
[ "$status" -ne 0 ]
run gteleport_url_is_relative "this/is/..strange/butok"
[ "$status" -eq 0 ]
run gteleport_url_is_relative "git@somewhere:blah"
[ "$status" -ne 0 ]
run gteleport_url_is_relative "git@somewhere:/blah"
[ "$status" -ne 0 ]
run gteleport_url_is_relative "ok/this/is"
[ "$status" -eq 0 ]
run gteleport_url_is_relative ".ok"
[ "$status" -eq 0 ]
}

@test "gteleport_url_to_relative" {
  run gteleport_url_to_relative "blah"
  [[ "${lines[0]}" = "blah" ]]

  run gteleport_url_to_relative "blah.GIT"
  [[ "${lines[0]}" = "blah.GIT" ]]

  run gteleport_url_to_relative "ssh://hello/mirror/blah.GIT" "ssh://hello"
  [[ "${lines[0]}" = "mirror/blah.GIT" ]]

  run gteleport_url_to_relative "ssh://hello/mirror/blah?what=hax" "ssh://hello/"
  [[ "${lines[0]}" = "mirror/blah" ]]

  run gteleport_url_to_relative "ssh://hello/mirror/blah.git/" "ssh://hello/"
  [[ "${lines[0]}" = "mirror/blah.git" ]]
}

@test "gteleport_url_allowed_as_receiver" {
  run gteleport_url_allowed_as_receiver "http://myhost/mirrors/something.git"
  [ "$status" -eq 0 ]

  run gteleport_url_allowed_as_receiver "MIRRORS/something"
  [ "$status" -eq 0 ]

  run gteleport_url_allowed_as_receiver "git@somewhere:mirrors/something"
  [ "$status" -eq 0 ]

  run gteleport_url_allowed_as_receiver "mirrors"
  [ "$status" -ne 0 ]
}
```

Approved for Public Release

```
run gteleport_url_allowed_as_receiver "mirrors/"
[ "$status" -ne 0 ]

run gteleport_url_allowed_as_receiver "//mirrors/"
[ "$status" -ne 0 ]

run gteleport_url_allowed_as_receiver "git@somewhere:mirrors"
[ "$status" -ne 0 ]

GTELEPORT_MIRROR_URL_REGEX='.*' \
    run gteleport_url_allowed_as_receiver "whatever"
[ "$status" -eq 0 ]

GTELEPORT_MIRROR_URL_REGEX='^mirror$' \
    run gteleport_url_allowed_as_receiver "MIRRORs/something"
[ "$status" -ne 0 ]

# prefix mode
run gteleport_url_allowed_as_receiver "git@somewhere:mirrors" "1"
[ "$status" -eq 0 ]

run gteleport_url_allowed_as_receiver "git@somewhere:mirrors/" "1"
[ "$status" -eq 0 ]

run gteleport_url_allowed_as_receiver "git@somewhere:nope" "1"
[ "$status" -ne 0 ]

run gteleport_url_allowed_as_receiver \
    "http://myhost/mirrors/something.git" "1"
[ "$status" -eq 0 ]
}

@test "gteleport_relative_url_to_local_file" {
    run gteleport_relative_url_to_local_file \
        "delicious.data/PROject.git"
    [[ "${lines[0]}" = "delicious-data/project" ]]

    run gteleport_relative_url_to_local_file ""
    [ "$status" -eq 1 ]

    run gteleport_relative_url_to_local_file "/"
    [ "$status" -eq 1 ]

    run gteleport_relative_url_to_local_file "files/BE/OK"
    [[ "${lines[0]}" = "files/be/ok" ]]

    run gteleport_relative_url_to_local_file "../..../.."
    [ "$status" -eq 1 ]

    run gteleport_relative_url_to_local_file "../..../nope"
    [ "$status" -eq 1 ]

    run gteleport_relative_url_to_local_file "~nope"
    [[ "${lines[0]}" = "nope" ]]

    run gteleport_relative_url_to_local_file "${HOME}/somewhere/else>nope"
    [[ "${lines[0]}" = "home/somewhere/else-nope" ]]

    printf -v badinput "malicious\nls"
    run gteleport_relative_url_to_local_file "$badinput"
    [[ "${lines[0]}" = "malicious" ]]

    run gteleport_relative_url_to_local_file '-e also\nmalicious'
```

Approved for Public Release

```
[[ "${lines[0]}" = "e-also" ]]

run gteleport_relative_url_to_local_file "omg%57%54%46"
[[ "${lines[0]}" = "omgwtf" ]]

run gteleport_relative_url_to_local_file "ok/ok//../"
[ "$status" -eq 1 ]

run gteleport_relative_url_to_local_file "ok/////"
[[ "${lines[0]}" = "ok" ]]

run gteleport_relative_url_to_local_file "spaces are/generally ok"
[[ "${lines[0]}" = "spaces-are/generally-ok" ]]

run gteleport_relative_url_to_local_file \
    "more.normal/scm/Project.GIT"
[[ "${lines[0]}" = "more-normal/scm/project" ]]

run gteleport_relative_url_to_local_file \
    "foo/bar/baz"
[[ "${lines[0]}" = "foo/bar/baz" ]]

run gteleport_relative_url_to_local_file \
    "c:/foo/bar/baz"
[ "$status" -eq 1 ]
}

@test "gteleport_beamdownfile_find" {
    local searchdir="${BATS_TEST_DIRNAME}/test beamdownfile"
    GTELEPORT_BEAMDOWNFILE="$searchdir" run gteleport_beamdownfile_find
    [ "$status" -eq 0 ]
    [[ $(basename "${lines[0]}") = "Beamdownfile" ]]
    GTELEPORT_BEAMDOWNFILE="$searchdir/Beamdownfile" \
    run gteleport_beamdownfile_find
    [ "$status" -eq 0 ]
    [[ $(basename "${lines[0]}") = "Beamdownfile" ]]
    GTELEPORT_BEAMDOWNFILE="$searchdir/Beamdownfile" \
    run gteleport_beamdownfile_find "$searchdir/Beamdownfile2"
    [ "$status" -eq 0 ]
    [[ $(basename "${lines[0]}") = "Beamdownfile2" ]]
    PWD="$searchdir" run gteleport_beamdownfile_find
    [ "$status" -eq 0 ]
    [[ $(basename "${lines[0]}") = "Beamdownfile" ]]
}

@test "gteleport_beamupfile_find" {
    local searchdir="${BATS_TEST_DIRNAME}/test beamdownfile"
    run gteleport_beamupfile_find "$searchdir"
    [ "$status" -eq 0 ]
    [[ $(basename "${lines[0]}") = "Beamupfile" ]]

    run gteleport_beamupfile_find "$searchdir/Beamupfile"
    [ "$status" -eq 0 ]
    [[ $(basename "${lines[0]}") = "Beamupfile" ]]
}

@test "gteleport_beamfile_read" {
    # Simple file

    local bdfilename="${BATS_TEST_DIRNAME}/test beamdownfile/Beamdownfile"
    local bdout=$(gteleport_beamfile_read "down" "abs" "rel" <"$bdfilename")
    eval "$bdout"
    [[ "${abs[0]}" = "https://some.domain.test/projects/cool.git" ]]
```

Approved for Public Release

```
[[ "${rel[${abs[0]}]}" = "cool.git" ]]

# More complicated file
local bdfilename="${BATS_TEST_DIRNAME}/test beamdownfile/Beamdownfile2"
local bdout=$(gteleport_beamfile_read "down" "abs" "rel" <"$bdfilename")
eval "$bdout"
[[ "${abs[0]}" = "http://space.striped/foo.git" ]]
[[ "${rel[${abs[0]}]}" = "foo.git" ]]
[[ "${abs[1]}" = "http://trailing.slash/striped" ]]
[[ "${rel[${abs[1]}]}" = "striped" ]]

# Good Beamupfile
local bdfilename="${BATS_TEST_DIRNAME}/test beamdownfile/Beamupfile"
local bdout=$(gteleport_beamfile_read "up" "abs" "rel" "fil" <"$bdfilename")
eval "$bdout"
[[ "${abs[0]}" = "source_repo" ]]
[[ "${rel[${abs[0]}]}" = "source_repo" ]]
[[ "${fil[${abs[0]}]}" = "source-repo" ]]

# Okay file (query string stripped from Beamupfile)
local bdout=$(gteleport_beamfile_read "up" "abs" "rel" \
    <<< $(echo "my%20project/something.git?what=do"))
eval "$bdout"
[[ "${abs[0]}" = "my%20project/something.git" ]]

# Bad files
# duplicate URLs
echo -e "one\none" | gteleport_beamfile_read "down" "abs" "rel" || status=$?
[[ "$status" -eq 1 ]]

# duplicate relative URLs
echo -e '!/a\n!/b\n/a/one\n/b/one' | \
    gteleport_beamfile_read "down" "abs" "rel" || status=$?
[[ "$status" -eq 3 ]]

# duplicate local filenames
echo -e 'one\nONE' | \
    gteleport_beamfile_read "down" "abs" "rel" || status=$?
[[ "$status" -eq 5 ]]

# root statements in a Beamupfile
echo -e '!/a\n!/b\n/a/one\n/b/one' | \
    gteleport_beamfile_read "up" "abs" "rel" || status=$?
[[ "$status" -eq 6 ]]
}

@test "gteleport_refdb_path" {
    touch "${GTELEPORT_TESTDIR}/Beamdownfile"
    run gteleport_refdb_path "${GTELEPORT_TESTDIR}/Beamdownfile"
    [ "$status" -eq 0 ]
    [[ -d "${lines[0]}" ]]
}

@test "gteleport_refdb_forurl" {
    mkdir -p "${GTELEPORT_TESTDIR}/Beamdownstate"
    run gteleport_refdb_forurl "${GTELEPORT_TESTDIR}/Beamdownstate" \
        "https://code.plex/foo/blah.git"
    [ "$status" -eq 0 ]
    [[ -f "${lines[0]}" ]]
}

@test "gteleport_beamupfile_gen" {
    run gteleport_beamupfile_gen "one" "TWO"
}
```

Approved for Public Release

```
[ "$status" -eq 0 ]
(( num_lines = "${#lines[@]}" - 1 ))
[[ "${lines[$num_lines]}" = "TWO" ]]
}

@test "gteleport_output_new" {
  touch "${GTELEPORT_TESTDIR}/Beamdownfile"
  run gteleport_output_path "${GTELEPORT_TESTDIR}/Beamdownfile"
  [ "$status" -eq 0 ]
  [[ -d "${lines[0]}" ]]
  run gteleport_output_new "${lines[0]}"
  [ "$status" -eq 0 ]
  [[ -d "${lines[0]}" ]]
}

@test "gteleport_output_trylink" {
  mkdir -p "${GTELEPORT_TESTDIR}/bundles/myout"
  touch "${GTELEPORT_TESTDIR}/bundles/myout/myfile"
  run gteleport_output_trylink "${GTELEPORT_TESTDIR}/bundles/myout"
  [[ -e "${GTELEPORT_TESTDIR}/bundles/latest" ]]
  [[ -f "${GTELEPORT_TESTDIR}/bundles/latest/myfile" ]]
}

@test "gteleport_path_to_templates" {
  GTELEPORT_EXE_PATH="${GTELEPORT_EXE_PATH}" run gteleport_path_to_templates
  [ "$status" -eq 0 ]
  [[ -d "${lines[0]}" ]]
}

@test "gteleport_bootstrap" {
  GTELEPORT_EXE_PATH="${GTELEPORT_EXE_PATH}" \
  run "gteleport_bootstrap" "${GTELEPORT_TESTDIR}/test_bootstrap"
  [ "$status" -eq 0 ]
  [[ -d "${GTELEPORT_TESTDIR}/test_bootstrap" ]]
  gteleport_rund "${GTELEPORT_TESTDIR}/test_bootstrap" git status --porcelain

  # the following line will fail if you have not installed git-teleport
  # correctly. ensure that the git-teleport repository is
  # ~/git-teleport or ~/.git-teleport/... that the git-teleport script
  # is on your $PATH
  "${GTELEPORT_TESTDIR}/test_bootstrap/teleportw" version
}

@test "gteleport_git_discover" {
  gteleport_rund "${GTELEPORT_TESTDIR}" git init --bare "findme"

  run gteleport_git_discover "${GTELEPORT_TESTDIR}/findme"
  [ "$status" -eq 0 ]
  [ "${lines[0]}" = "${GTELEPORT_TESTDIR}/findme" ]

  # We prefer repos ending in .git when they exist
  gteleport_rund "${GTELEPORT_TESTDIR}" git init --bare "findme.git"
  run gteleport_git_discover "${GTELEPORT_TESTDIR}/findme"
  [ "$status" -eq 0 ]
  [ "${lines[0]}" = "${GTELEPORT_TESTDIR}/findme.git" ]
}

@test "gteleport_git_filter_refs" {
  # This test relies on commits which are in the
  # main git-teleport repository.

  local myinput=$(cat <<'EOF'
623fcbae73ec1bfedfe587718535d7d590f817e5
```

Approved for Public Release

```
383a60cb6614dc6e8f3f86b49acb76d67f3c55a7
HEAD
0000000000000000000000000000000000000000
totallyinvalid
EOF
)

    local expect_out=$(cat <<'EOF'
623fcbae73ec1bfedfe587718535d7d590f817e5
383a60cb6614dc6e8f3f86b49acb76d67f3c55a7
HEAD
EOF
)

    local gotout=$(echo "$myinput" | gteleport_git_filter_refs "${GTELEPORT_DIR}")
    [[ "$gotout" = "$expect_out" ]]
}

@test "gteleport_git_need_bundle" {
    # This is really just a comparison under the hood
    local out
    local my_refs
    my_refs=$(gteleport_git_show_ref "${GTELEPORT_DIR}")

    # we expect a repo's own refs to be identical to itself
    out=$(echo "$my_refs" |
        gteleport_git_need_bundle "${GTELEPORT_DIR}")
    [ -z "$out" ]

    # append some refs
    printf -v my_refs "%s%s\n" "$my_refs" \
        "beb21265e3969c9282bf16bfa5e41c693f311bd4 refs/sir_not_appearing"

    out=$(echo "$my_refs" |
        gteleport_git_need_bundle "${GTELEPORT_DIR}")
    [ "$out" = "yes" ]
}

@test "gteleport_find_receive_mirror" {
    gteleport_rund "${GTELEPORT_TESTDIR}" git init --bare "my dest.git"

    # an ok prefix, but we didn't actually search for anything
    run gteleport_find_receive_mirror \
        "${GTELEPORT_TESTDIR}" \
        '' \
        'local' \
        'placeholder'
    [ "$status" -eq 1 ]

    # the prefix is ok (local file)
    run gteleport_check_unpack_prefix "${GTELEPORT_TESTDIR}"
    [ "$status" -eq 0 ]
    [[ "${lines[0]}" = "local" ]]

    # prohibited: local repo doesn't contain teleport-mirror
    run gteleport_find_receive_mirror \
        "${GTELEPORT_TESTDIR}" \
        'my dest' \
        'local' \
        'placeholder'
    [ "$status" -eq 3 ]

    # ok!
}
```

Approved for Public Release

```
touch "${GTELEPORT_TESTDIR}/my dest.git/teleport-mirror"
run gteleport_find_receive_mirror \
  "${GTELEPORT_TESTDIR}" \
  'my dest' \
  'local' \
  'placeholder'
[ "$status" -eq 0 ]
[[ "${lines[0]}" = "${GTELEPORT_TESTDIR}/my dest.git" ]]

# this will fail prefix validation
run gteleport_check_unpack_prefix "file://${GTELEPORT_TESTDIR}"
[ "$status" -ne 0 ]

# prohibited: using a network-style path w/o allowed components
# NOTE: url decoding of file:/// URLs is not implemented in Git 1.7.1,
# but it is supported in 2.17.1.
run gteleport_find_receive_mirror \
  "file://${GTELEPORT_TESTDIR}" \
  'my dest' \
  'network' \
  'placeholder'
[ "$status" -eq 3 ]

mkdir -p "${GTELEPORT_TESTDIR}/mirrors"
mv "${GTELEPORT_TESTDIR}/my dest.git" \
  "${GTELEPORT_TESTDIR}/mirrors/my dest.git"

# now we pass
run gteleport_check_unpack_prefix "file://${GTELEPORT_TESTDIR}/mirrors"
[ "$status" -eq 0 ]
[[ "${lines[0]}" = "network" ]]

# ok!
run gteleport_find_receive_mirror \
  "file://${GTELEPORT_TESTDIR}/mirrors" \
  'my dest' \
  'network' \
  'placeholder'
[ "$status" -eq 0 ]
[[ "${lines[0]}" = "file://${GTELEPORT_TESTDIR}/mirrors/my dest.git" ]]
}

@test "gteleport_find_fork" {
  # This method is in our post-receive.mirror.sample file

  run gteleport_find_fork ""
  [ "$status" -eq 1 ]

  run gteleport_find_fork "/"
  [ "$status" -eq 1 ]

  run gteleport_find_fork "."
  [ "$status" -eq 1 ]

  run gteleport_find_fork "/git/somewhere"
  [ "$status" -eq 1 ]

  run gteleport_find_fork "/git/somewhere/mirrors"
  [ "$status" -eq 1 ]

  run gteleport_find_fork "/git/somewhere/mirrors-are-cool"
  [ "$status" -eq 1 ]
}
```

Approved for Public Release

```
run gteleport_find_fork "/git/somewhere/mirrors/intranet"
[ "$status" -eq 1 ]

run gteleport_find_fork "/git/somewhere/mirrors/intranet/a.git"
[ "$status" -eq 0 ]
[[ "${lines[0]}" = "intranet /git/somewhere/a.git" ]]

run gteleport_find_fork "/git/somewhere/mirrors/intranet/deep/b"
[ "$status" -eq 0 ]
[[ "${lines[0]}" = "intranet /git/somewhere/deep/b" ]]

run gteleport_find_fork "/git/somewhere/mirror/singular/c.git"
[ "$status" -eq 0 ]
[[ "${lines[0]}" = "singular /git/somewhere/c.git" ]]

run gteleport_find_fork "/mirror/singular/c.git"
[ "$status" -eq 0 ]
[[ "${lines[0]}" = "singular /c.git" ]]
}

@test "gteleport_git_bundle_refsonly" {
    # Verify that we can use --all --no-walk to make a refs-only bundle
    #
    # The --no-walk option to git rev-list does what it says on the tin:
    # it reports *only* the commits given on the command-line... and
    # nothing else. We can use this to make a separate "refs" bundle
    # for fetch --prune.

    local repo="${GTELEPORT_TESTDIR}/repo"
    local bndl="${GTELEPORT_TESTDIR}/repo/test.bundle"
    local cmp
    local cmp2

    # Create a source repository and make some fake commits
    git init "${repo}"
    gteleport_test_fakecommit_make "${repo}" "A"
    gteleport_rund "${repo}" git tag -a "0" -m "A Tag Message"
    gteleport_rund "${repo}" git tag "0-light"
    gteleport_rund "${repo}" git checkout -b "somebranch"
    gteleport_test_fakecommit_make "${repo}" "B"
    gteleport_rund "${repo}" git checkout "master"
    gteleport_rund "${repo}" git merge --no-ff somebranch -m "C"
    gteleport_rund "${repo}" git tag -a "1" -m "C Tag Message"
    gteleport_rund "${repo}" git tag "1-light"
    gteleport_test_fakecommit_make "${repo}" "D"
    gteleport_test_fakecommit_make "${repo}" "E"

    # Make refs-only bundle
    gteleport_git_bundle_refsonly "${repo}" "${bndl}"

    # Remember the state
    cmp=$(gteleport_rund "${repo}" git show-ref -d)

    # Make some refs that will go away
    gteleport_rund "${repo}" git checkout "0-light"
    gteleport_rund "${repo}" git checkout -b "this_ref_will_go_away"
    gteleport_rund "${repo}" git checkout "master"
    gteleport_test_fakecommit_make "${repo}" "F-will-go-away"

    # Fetch with prune into the repo
    gteleport_rund "${repo}" git remote add bndl "${bndl}"
    gteleport_rund "${repo}" git config remote.bndl.fetch '+refs/*:refs/*'
    gteleport_rund "${repo}" git fetch --update-head-ok --prune bndl '+refs/*:refs/*'
```

Approved for Public Release

```
# Do we still have everything, just as it was?
cmp2=$(gteleport_rund "${repo}" git show-ref -d)

[ "$cmp" = "$cmp2" ]

# Is our bundle minimal?
gteleport_rund "${repo}" git bundle verify "${bndl}" | \
    grep -q -E 'bundle requires (this|these)'
}

@test "gteleport_write_manifest" {
    gteleport_rund "${GTELEPORT_TESTDIR}" touch blah
    gteleport_rund "${GTELEPORT_TESTDIR}" touch blah2

    run gteleport_rund "${GTELEPORT_TESTDIR}" gteleport_write_manifest .
    [ "$status" -eq 0 ]

    [ -f "${GTELEPORT_TESTDIR}/MANIFEST.txt" ]
    grep -E -q '^0\w*\s+([a-z]+)?blah$' "${GTELEPORT_TESTDIR}/MANIFEST.txt"
    grep -E -q '^0\w*\s+([a-z]+)?blah2$' "${GTELEPORT_TESTDIR}/MANIFEST.txt"
    grep -E -q --invert-match 'MANIFEST\.txt' "${GTELEPORT_TESTDIR}/MANIFEST.txt"
}

@test "end_to_end" {
    # This is a large test case which checks the complete full and
    # incremental bundling process.

    local mirrorset="${GTELEPORT_TESTDIR}/mirrorset"
    local rsrc="${GTELEPORT_TESTDIR}/source_repo"
    local destdir="${GTELEPORT_TESTDIR}/dest"
    local rdst="${destdir}/source_repo"
    local cach="${GTELEPORT_TESTDIR}/cache"
    local latest="${mirrorset}/bundles/latest"
    local latest_bundle="${latest}/source-repo.bundle"

    # Create a source repository and make four fake commits
    git init "${rsrc}"
    gteleport_test_fakecommit_make "${rsrc}" "A"
    gteleport_test_fakecommit_make "${rsrc}" "B"
    gteleport_rund "${rsrc}" git tag -a "temptag" -m "will delete later"
    gteleport_rund "${rsrc}" git tag -a "temptag2" -m "will delete later too"
    gteleport_rund "${rsrc}" git checkout -b "somebranch"
    gteleport_test_fakecommit_make "${rsrc}" "C"
    gteleport_rund "${rsrc}" git checkout "master"
    gteleport_rund "${rsrc}" git merge --no-ff somebranch -m "D"
    gteleport_test_fakecommit_check "${rsrc}" "D"

    # Bootstrap and add source repository
    GTELEPORT_EXE_PATH="${GTELEPORT_EXE_PATH}" \
    run "gteleport_bootstrap" "$mirrorset"
    [ "$status" -eq 0 ]
    gteleport_test_mkbeamdown "${mirrorset}" "${rsrc}" "${GTELEPORT_TESTDIR}"

    # Make a cache directory, so it gets cleaned up at exit
    mkdir -p "$cach"

    # fetch
    GTELEPORT_CACHE="$cach" run gteleport_fetch "${mirrorset}" 1>&2
    [ "$status" -eq 0 ]

    # bundle
    GTELEPORT_CACHE="$cach" run gteleport_bundle "${mirrorset}" 1>&2
}
```

Approved for Public Release

```
[ "$status" -eq 0 ]
[[ -d "${latest}" ]]
[[ -f "${latest}/Beamupfile" ]]
[[ -f "${latest_bundle}" ]]

# commit
GTELEPORT_CACHE="$cach" run gteleport_commit "${mirrorset}" 1>&2
[ "$status" -eq 0 ]

# Create a destination repository. Add the bundle, and fetch it.
run gteleport_skeleton "${rdst}"
run gteleport_unpack "${latest}" "${destdir}"

# Check for head commit
gteleport_test_fakecommit_check "${rdst}" "D"

# Add some new commits to the source. Also remove a tag.
gteleport_test_fakecommit_make "${rsrc}" "E"
gteleport_rund "${rsrc}" git checkout -b "topicbranch"
gteleport_test_fakecommit_make "${rsrc}" "F1"
gteleport_rund "${rsrc}" git tag -a "tag0" -m "F1 Tag Message"
gteleport_rund "${rsrc}" git checkout "master"
gteleport_rund "${rsrc}" git tag -d "temptag2"

# delete the previous set of bundles
#   on linux, tests run so quickly that bundle directories
#   can duplicate
rm -rf "${mirrorset}/bundles"

# update the bundles
GTELEPORT_CACHE="$cach" run gteleport_fetch "${mirrorset}" 1>&2
[ "$status" -eq 0 ]
GTELEPORT_CACHE="$cach" run gteleport_bundle "${mirrorset}" 1>&2
[ "$status" -eq 0 ]
GTELEPORT_CACHE="$cach" run gteleport_commit "${mirrorset}" 1>&2
[ "$status" -eq 0 ]

# in the destination, fetch and verify
run gteleport_unpack "${latest}" "${destdir}"
gteleport_test_fakecommit_check "${rdst}" "E" "master"
gteleport_test_fakecommit_check "${rdst}" "F1" "topicbranch"
gteleport_test_tageqcommit "${rdst}" "tag0" "topicbranch"
gteleport_rund "${rdst}" git show "temptag"
gteleport_rund "${rdst}" git show "somebranch"

if gteleport_rund "${rdst}" git show "temptag2"; then
    false
else
    true
fi

# introduce some changes:
# - tag a commit in the past
# - remove a topic branch
#
# also do major, breaking changes that might require --force
# - in the source repository, alter the topicbranch!
# - change a tag!
# - remove a tag!
#
gteleport_rund "${rsrc}" git checkout "topicbranch"
gteleport_rund "${rsrc}" git reset --hard HEAD~
gteleport_rund "${rsrc}" git tag -d "tag0"
```

Approved for Public Release

```
gteleport_test_fakecommit_make "${rsrc}" "F2"
gteleport_rund "${rsrc}" git tag -a "tag0" -m "F2 Tag Message"
gteleport_rund "${rsrc}" git checkout "master"
gteleport_rund "${rsrc}" git tag -a "tag1" -m "Master Tag Message"
gteleport_rund "${rsrc}" git tag -d "temptag"
gteleport_rund "${rsrc}" git branch -d "somebranch"

# delete the previous set of bundles
#   on linux, tests run so quickly that bundle directories
#   can duplicate
rm -rf "${mirrorset}/bundles"

# update the bundles, using the old method for git bundling
GTELEPORT_CACHE="$cach" run gteleport_fetch "${mirrorset}" 1>&2
[ "$status" -eq 0 ]
GTELEPORT_GIT_HAS_NEW_REVLIST="no"
GTELEPORT_CACHE="$cach" run gteleport_bundle "${mirrorset}" 1>&2
GTELEPORT_GIT_HAS_NEW_REVLIST=""
[ "$status" -eq 0 ]
GTELEPORT_CACHE="$cach" run gteleport_commit "${mirrorset}" 1>&2
[ "$status" -eq 0 ]

# drop the pre-receive hook and fast-forward limitations
rm -f "${rdst}/hooks/pre-receive"
gteleport_rund "${rdst}" git config receive.denynonfastforwards false

# in the destination, fetch. use the network protocol. fails because of
# the mirror URL regexp
GTELEPORT_CACHE="$cach" GTELEPORT_MIRROR_URL_REGEXP='^nomatch$' \
  run gteleport_unpack "${mirrorset}/bundles/latest" "file://${destdir}"
[ "$status" -ne 0 ]

# now try again, with a more permissive regexp
GTELEPORT_CACHE="$cach" GTELEPORT_MIRROR_URL_REGEXP='dest' \
  run gteleport_unpack "${mirrorset}/bundles/latest" "file://${destdir}"
[ "$status" -eq 0 ]
echo "${lines[${#lines[@]}-2]}" | grep -E -q "UPDATED"

# verify updated refs
gteleport_test_fakecommit_check "${rdst}" "E" "master"
gteleport_test_fakecommit_check "${rdst}" "F2" "topicbranch"
gteleport_test_tageqcommit "${rdst}" "tag0" "topicbranch"
gteleport_test_tageqcommit "${rdst}" "tag1" "master"

# verify removed refs
if gteleport_rund "${rdst}" git show "temptag"; then
  false
else
  true
fi

if gteleport_rund "${rdst}" git show "somebranch"; then
  false
else
  true
fi

# ensure that we made a cache repository for the *destination*
# (otherwise, we might have used the local unpacker and not the
# network unpacker)
local check_cacherepo=$(GTELEPORT_CACHE="$cach" \
  gteleport_url_to_cache_repo \
  "file://${destdir}/source_repo")
```

Approved for Public Release

```
[[ -d "$check_cacherepo" ]]

rm -rf "${mirrorset}/bundles"

# try a complete bundle
GTELEPORT_CACHE="$cach" run gteleport_bundle "${mirrorset}" "force" 1>&2
[ "$status" -eq 0 ]
gteleport_rund "${rsrc}" git bundle verify "${latest_bundle}" | \
grep -E "(records a complete history|these 0 ref)"

# Fetch and verify that we are already up-to-date
gteleport_rund "${rdst}" git remote add --mirror origin "${latest_bundle}"
run gteleport_rund "${rdst}" git fetch --verbose origin
for line in "${lines[@]}"
do
    echo "$line" | grep -E "(^From|up to date)"
done

# remove all tags and send new bundles
# this will cause a refs-only update
gteleport_rund "${rsrc}" git tag -d tag0
gteleport_rund "${rsrc}" git tag -d tag1

rm -rf "${mirrorset}/bundles"

# update the bundles
GTELEPORT_CACHE="$cach" run gteleport_fetch "${mirrorset}" 1>&2
[ "$status" -eq 0 ]
GTELEPORT_CACHE="$cach" run gteleport_bundle "${mirrorset}" 1>&2
[ "$status" -eq 0 ]
GTELEPORT_CACHE="$cach" run gteleport_commit "${mirrorset}" 1>&2
[ "$status" -eq 0 ]

GTELEPORT_CACHE="$cach" GTELEPORT_MIRROR_URL_REGEX='dest' \
    run gteleport_unpack "${mirrorset}/bundles/latest" "file://${destdir}"
[ "$status" -eq 0 ]
echo "${lines[${#lines[@]}-2]}" | grep -E -q "UPDATED"

if gteleport_rund "${rdst}" git show "tag0"; then
    false
else
    true
fi
}
```

A.11.File Listing “test/test beamdownfile/Beamdownfile”

This file is needed for the unit tests. Don't move it.

```
!https://some.domain.test/projects
https://some.domain.test/projects/cool.git
```

A.12.File Listing “test/test beamdownfile/Beamdownfile2”

This file is needed for the unit tests. Don't move it.

```
# I am ignored
#Me too
```

```
!http://trailing.slash/
! http://space.stripped
```

Approved for Public Release

```
http://space.stripped/foo.git  
http://trailing.slash/stripped/  
# comment
```

A.13.File Listing “test/test beamdownfile/Beamupfile”

```
# This file is needed for the unit tests. Don't move it.
```

```
source_repo
```

(U) INITIAL DISTRIBUTION LIST

| | | <u>Copies</u> |
|---|--|---------------|
| Defense Systems Information
Analysis Center
SURVICE Engineering Company
4695 Millennium Drive
Belcamp, MD 21017 | Ms. Christi Rothrock
christi.rothrock@dsiac.org | Electronic |
| Defense Technical Information Center
8725 John J. Kingman Rd., Suite 0944
Fort Belvoir, VA 22060-6218 | Mr. Jack L. Rike | Electronic |
| FCDD-AM | | Electronic |
| FCDD-AMG-CSI | | Electronic |
| FCDD-AMT-MGR | Ms. Kelsey Brown | Electronic |
| Dynetics
1002 Explorer Blvd NW
Huntsville, AL 35806 | Dr. Colin Stagner
colin.stagner@dynetics.com | Electronic |