9.6 Transfer Protocols

Git can transfer data between two repositories in two major ways: over HTTP and via the so-called smart protocols used in the `file://`, `ssh://`, and `git://` transports. This section will quickly cover how these two main protocols operate.

9.6.1 The Dumb Protocol

Git transport over HTTP is often referred to as the dumb protocol because it requires no Git-specific code on the server side during the transport process. The fetch process is a series of GET requests, where the client can assume the layout of the Git repository on the server. Let's follow the `http-fetch` process for the simplegit library:

```
$ git clone http://github.com/schacon/simplegit-progit.git
```

The first thing this command does is pull down the `info/refs` file. This file is written by the `update-server-info` command, which is why you need to enable that as a `post-receive` hook in order for the HTTP transport to work properly:

```
=> GET info/refs
ca82a6dff817ec66f44342007202690a93763949     refs/heads/master
```

Now you have a list of the remote references and SHAs. Next, you look for what the HEAD reference is so you know what to check out when you're finished:

```
=> GET HEAD
ref: refs/heads/master
```

You need to check out the `master` branch when you've completed the process. At this point, you're ready to start the walking process. Because your starting point is the `ca82a6` commit object you saw in the `info/refs` file, you start by fetching that:

```
=> GET objects/ca/82a6dff817ec66f44342007202690a93763949
(179 bytes of binary data)
```

You get an object back — that object is in loose format on the server, and you fetched it over a static HTTP GET request. You can zlib-uncompress it, strip off the header, and look at the commit content:

```
$ git cat-file -p ca82a6dff817ec66f44342007202690a93763949
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
author Scott Chacon <schacon@gmail.com> 1205815931 -0700
committer Scott Chacon <schacon@gmail.com> 1240030591 -0700

changed the verison number
```

Next, you have two more objects to retrieve — `cfda3b`, which is the tree of content that the commit we just retrieved points to; and `085bb3`, which is the parent commit:

```
=> GET objects/08/5bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
(179 bytes of data)
```

That gives you your next commit object. Grab the tree object:

```
=> GET objects/cf/da3bf379e4f8dba8717dee55aab78aef7f4daf
(404 - Not Found)
```

Oops — it looks like that tree object isn't in loose format on the server, so you get a 404 response back. There are a couple of reasons for this — the object could be in an alternate repository, or it could be in a packfile in this repository. Git checks for any listed alternates first:

```
=> GET objects/info/http-alternates
(empty file)
```

If this comes back with a list of alternate URLs, Git checks for loose files and packfiles there — this is a nice mechanism for projects that are forks of one another to share objects on disk. However, because no alternates are listed in this case, your object must be in a packfile. To see what packfiles are available on this server, you need to get the `objects/info/packs` file, which contains a listing of them (also generated by `update-server-info`):

```
=> GET objects/info/packs
P pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
```

There is only one packfile on the server, so your object is obviously in there, but you'll check the index file to make sure. This is also useful if you have multiple packfiles on the server, so you can see which packfile contains the object you need:

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.idx
(4k of binary data)
```

Now that you have the packfile index, you can see if your object is in it — because the index lists the SHAs of the objects contained in the packfile and the offsets to those objects. Your object is there, so go ahead and get the whole packfile:

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
(13k of binary data)
```

You have your tree object, so you continue walking your commits. They're all also within the packfile you just downloaded, so you don't have to do any more requests to your server. Git checks out a working copy of the `master` branch that was pointed to by the HEAD reference you downloaded at the beginning.

The entire output of this process looks like this:

```
$ git clone http://github.com/schacon/simplegit-progit.git
Initialized empty Git repository in /private/tmp/simplegit-progit/.git/
got ca82a6dff817ec66f44342007202690a93763949
walk ca82a6dff817ec66f44342007202690a93763949
got 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Getting alternates list for http://github.com/schacon/simplegit-progit.git
Getting pack list for http://github.com/schacon/simplegit-progit.git
Getting index for pack 816a9b2334da9953e530f27bcac22082a9f5b835
Getting pack 816a9b2334da9953e530f27bcac22082a9f5b835
 which contains cfda3bf379e4f8dba8717dee55aab78aef7f4daf
walk 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
walk a11bef06a3f659402fe7563abf99ad00de2209e6
```

9.6.2 The Smart Protocol

The HTTP method is simple but a bit inefficient. Using smart protocols is a more common method of transferring data. These protocols have a process on the remote end that is intelligent about Git — it can read local data and figure out what the client has or needs and generate custom data for it. There are two sets of processes for transferring data: a pair for uploading data and a pair for downloading data.

Uploading Data

To upload data to a remote process, Git uses the `send-pack` and `receive-pack` processes. The `send-pack` process runs on the client and connects to a `receive-pack` process on the remote side.

For example, say you run `git push origin master` in your project, and `origin` is defined as a URL that uses the SSH protocol. Git fires up the `send-pack` process, which initiates a connection over SSH to your server. It tries to run a command on the remote server via an SSH call that looks something like this:

```
$ ssh -x git@github.com "git-receive-pack 'schacon/simplegit-progit.git'"
005bca82a6dff817ec66f4437202690a93763949 refs/heads/master report-status delete-refs
003e085bb3bcb608e1e84b2432f8ecbe6306e7e7 refs/heads/topic
0000
```

The `git-receive-pack` command immediately responds with one line for each reference it currently has — in this case, just the `master` branch and its SHA. The first line also has a list of the server's capabilities (here, `report-status` and `delete-refs`).

Each line starts with a 4-byte hex value specifying how long the rest of the line is. Your first line starts with 005b, which is 91 in hex, meaning that 91 bytes remain on that line. The next line starts with 003e, which is 62, so you read the remaining 62 bytes. The next line is 0000, meaning the server is done with its references listing.

Now that it knows the server's state, your `send-pack` process determines what commits it has that the server doesn't. For each reference that this push will update, the `send-pack` process tells the `receive-pack` process that information. For instance, if you're updating the `master` branch and adding an `experiment` branch, the `send-pack` response may look something like this:

```
0085ca82a6dff817ec66f44342007202690a93763949  15027957951b64cf874c3557a0f3547bd83b3ff6 refs/heads/
00670000000000000000000000000000000000000000 cdfdb42577e2506715f8cfeacdbabc092bf63e8d refs/heads/e
0000
```

The SHA–1 value of all '0's means that nothing was there before — because you're adding the experiment reference. If you were deleting a reference, you would see the opposite: all '0's on the right side.

Git sends a line for each reference you're updating with the old SHA, the new SHA, and the reference that is being updated. The first line also has the client's capabilities. Next, the client uploads a packfile of all the objects the server doesn't have yet. Finally, the server responds with a success (or failure) indication:

```
000Aunpack ok
```

Downloading Data

When you download data, the `fetch-pack` and `upload-pack` processes are involved. The client initiates a `fetch-pack` process that connects to an `upload-pack` process on the remote side to negotiate what data will be transferred down.

There are different ways to initiate the `upload-pack` process on the remote repository. You can run via SSH in the same manner as the `receive-pack` process. You can also initiate the process via the Git daemon, which listens on a server on port 9418 by default. The `fetch-pack` process sends data that looks like this to the daemon after connecting:

```
003fgit-upload-pack schacon/simplegit-progit.git\0host=myserver.com\0
```

It starts with the 4 bytes specifying how much data is following, then the command to run followed by a null byte, and then the server's hostname followed by a final null byte. The Git daemon checks that the command can be run and that the repository exists and has public permissions. If everything is cool, it fires up the `upload-pack` process and hands off the request to it.

If you're doing the fetch over SSH, `fetch-pack` instead runs something like this:

```
$ ssh -x git@github.com "git-upload-pack 'schacon/simplegit-progit.git'"
```

In either case, after `fetch-pack` connects, `upload-pack` sends back something like this:

```
0088ca82a6dff817ec66f44342007202690a93763949 HEAD\0multi_ack thin-pack \
  side-band side-band-64k ofs-delta shallow no-progress include-tag
003fca82a6dff817ec66f44342007202690a93763949 refs/heads/master
003e085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 refs/heads/topic
0000
```

This is very similar to what `receive-pack` responds with, but the capabilities are different. In addition, it sends back the HEAD reference so the client knows what to check out if this is a clone.

At this point, the `fetch-pack` process looks at what objects it has and responds with the objects that it needs by sending "want" and then the SHA it wants. It sends all the objects it already has with "have" and then the SHA. At the end of this list, it writes "done" to initiate the `upload-pack` process to begin sending the packfile of the data it needs:

```
0054want ca82a6dff817ec66f44342007202690a93763949 ofs-delta
0032have 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
0000
0009done
```

That is a very basic case of the transfer protocols. In more complex cases, the client supports `multi_ack` or `side-band` capabilities; but this example shows you the basic back and forth used by the smart protocol processes.

9.7 Maintenance and Data Recovery

Occasionally, you may have to do some cleanup — make a repository more compact, clean up an imported repository, or recover lost work. This section will cover some of these scenarios.