10.6 Git Internals - Transfer Protocols

Transfer Protocols

Git can transfer data between two repositories in two major ways: the “dumb” protocol and the “smart” protocol. This section will quickly cover how these two main protocols operate.

The Dumb Protocol

If you’re setting up a repository to be served read-only over HTTP, the dumb protocol is likely what will be used. This protocol is called “dumb” because it requires no Git-specific code on the server side during the transport process; the fetch process is a series of HTTP `GET` requests, where the client can assume the layout of the Git repository on the server.

Note: The dumb protocol is fairly rarely used these days. It’s difficult to secure or make private, so most Git hosts (both cloud-based and on-premises) will refuse to use it. It’s generally advised to use the smart protocol, which we describe a bit further on.

Let’s follow the `http-fetch` process for the simplegit library:

```
$ git clone http://server/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 SHA-1s. 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

Change version 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 SHA-1s 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 Smart Protocol

The dumb protocol is simple but a bit inefficient, and it can't handle writing of data from the client to the server. The smart protocol is a more common method of transferring data, but it requires a process on the remote end that is intelligent about Git – it can read local data, figure out what the client has and needs, and generate a custom packfile 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.

SSH

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@server "git-receive-pack 'simplegit-progit.git'"
00a5ca82a6dff817ec66f4437202690a93763949 refs/heads/master□report-status \
```

```
        delete-refs side-band-64k quiet ofs-delta \
        agent=git/2:2.1.1+github-607-gfba4028 delete-refs
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-1. The first line also has a list of the server's capabilities (here, `report-status`, `delete-refs`, and some others, including the client identifier).

The data is transmitted in chunks. Each chunk starts with a 4-character hex value specifying how long the chunk is (including the 4 bytes of the length itself). Chunks usually contain a single line of data and a trailing linefeed. Your first chunk starts with 00a5, which is hexadecimal for 165, meaning the chunk is 165 bytes long. The next chunk 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:

```
0076ca82a6dff817ec66f44342007202690a93763949 15027957951b64cf874c3557a0f3547bd83b3ff6 \
        refs/heads/master report-status
006c0000000000000000000000000000000000000000 cdfdb42577e2506715f8cfeacdbabc092bf63e8d \
        refs/heads/experiment
0000
```

Git sends a line for each reference you're updating with the line's length, the old SHA-1, the new SHA-1, and the reference that is being updated. The first line also has the client's capabilities. 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.

Next, the client sends a packfile of all the objects the server doesn't have yet. Finally, the server responds with a success (or failure) indication:

```
000eunpack ok
```

HTTP(S)

This process is mostly the same over HTTP, though the handshaking is a bit different. The connection is initiated with this request:

```
=> GET http://server/simplegit-progit.git/info/refs?service=git-receive-pack
001f# service=git-receive-pack
00ab6c5f0e45abd7832bf23074a333f739977c9e8188 refs/heads/master□report-status \
        delete-refs side-band-64k quiet ofs-delta \
        agent=git/2:2.1.1~vmg-bitmaps-bugaloo-608-g116744e
0000
```

That's the end of the first client-server exchange. The client then makes another request, this time a `POST`, with the data that `send-pack` provides.

```
=> POST http://server/simplegit-progit.git/git-receive-pack
```

The `POST` request includes the `send-pack` output and the packfile as its payload. The server then indicates success or failure with its HTTP response.

Keep in mind the HTTP protocol may further wrap this data inside a chunked transfer encoding.

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.

SSH

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

```
$ ssh -x git@server "git-upload-pack 'simplegit-progit.git'"
```

After `fetch-pack` connects, `upload-pack` sends back something like this:

```
00dfca82a6dff817ec66f44342007202690a93763949 HEAD□multi_ack thin-pack \
        side-band side-band-64k ofs-delta shallow no-progress include-tag \
        multi_ack_detailed symref=HEAD:refs/heads/master \
        agent=git/2:2.1.1+github-607-gfba4028
003fe2409a098dc3e53539a9028a94b6224db9d6a6b6 refs/heads/master
0000
```

This is very similar to what `receive-pack` responds with, but the capabilities are different. In addition, it sends back what HEAD points to (`symref=HEAD:refs/heads/master`) 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-1 it wants. It sends all the objects it already has with "have" and then the SHA-1. 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:

```
003cwant ca82a6dff817ec66f44342007202690a93763949 ofs-delta
0032have 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
0009done
0000
```

HTTP(S)

The handshake for a fetch operation takes two HTTP requests. The first is a `GET` to the same endpoint used in the dumb protocol:

```
=> GET $GIT_URL/info/refs?service=git-upload-pack
001e# service=git-upload-pack
00e7ca82a6dff817ec66f44342007202690a93763949 HEAD□multi_ack thin-pack \
        side-band side-band-64k ofs-delta shallow no-progress include-tag \
        multi_ack_detailed no-done symref=HEAD:refs/heads/master \
        agent=git/2:2.1.1+github-607-gfba4028
003fca82a6dff817ec66f44342007202690a93763949 refs/heads/master
0000
```

This is very similar to invoking `git-upload-pack` over an SSH connection, but the second exchange is performed as a separate request:

```
=> POST $GIT_URL/git-upload-pack HTTP/1.0
0032want 0a53e9ddeaddad63ad106860237bbf53411d11a7
0032have 441b40d833fdfa93eb2908e52742248faf0ee993
0000
```

Again, this is the same format as above. The response to this request indicates success or failure, and includes the packfile.

Protocols Summary

This section contains a very basic overview of the transfer protocols. The protocol includes many other features, such as `multi_ack` or `side-band` capabilities, but covering them is outside the scope of this book. We've tried to give you a sense of the general back-and-forth between client and server; if you need more knowledge than this, you'll probably want to take a look at the Git source code.