## NAME

gitprotocol-pack - How packs are transferred over-the-wire

## SYNOPSIS

<over-the-wire-protocol>

## DESCRIPTION

Git supports transferring data in packfiles over the ssh://, git://, http:// and file:// transports. There exist two sets of protocols, one for pushing data from a client to a server and another for fetching data from a server to a client. The three transports (ssh, git, file) use the same protocol to transfer data. http is documented in **gitprotocol-http**(5).

The processes invoked in the canonical Git implementation are *upload-pack* on the server side and *fetch-pack* on the client side for fetching data; then *receive-pack* on the server and *send-pack* on the client for pushing data. The protocol functions to have a server tell a client what is currently on the server, then for the two to negotiate the smallest amount of data to send in order to fully update one or the other.

## PKT-LINE FORMAT

The descriptions below build on the pkt-line format described in **gitprotocol-common**(5). When the grammar indicate **PKT-LINE(...)**, unless otherwise noted the usual pkt-line LF rules apply: the sender SHOULD include a LF, but the receiver MUST NOT complain if it is not present.

An error packet is a special pkt-line that contains an error string.

```
  error-line     =  PKT-LINE("ERR" SP explanation-text)
```

Throughout the protocol, where **PKT-LINE(...)** is expected, an error packet MAY be sent. Once this packet is sent by a client or a server, the data transfer process defined in this protocol is terminated.

## TRANSPORTS

There are three transports over which the packfile protocol is initiated. The Git transport is a simple, unauthenticated server that takes the command (almost always *upload-pack*, though Git servers can be configured to be globally writable, in which *receive- pack* initiation is also allowed) with which the client wishes to communicate and executes it and connects it to the requesting process.

In the SSH transport, the client just runs the *upload-pack* or *receive-pack* process on the server over the SSH protocol and then communicates with that invoked process over the SSH connection.

The file:// transport runs the *upload-pack* or *receive-pack* process locally and communicates with it over a pipe.

## EXTRA PARAMETERS

The protocol provides a mechanism in which clients can send additional information in its first message to the server. These are called "Extra Parameters", and are supported by the Git, SSH, and HTTP protocols.

Each Extra Parameter takes the form of **<key>=<value>** or **<key>**.

Servers that receive any such Extra Parameters MUST ignore all unrecognized keys. Currently, the only Extra Parameter recognized is "version" with a value of *1* or *2*. See **gitprotocol-v2**(5) for more information on protocol version 2.

## GIT TRANSPORT

The Git transport starts off by sending the command and repository on the wire using the pkt-line format, followed by a NUL byte and a hostname parameter, terminated by a NUL byte.

```
0033git-upload-pack /project.git\0host=myserver.com\0
```

The transport may send Extra Parameters by adding an additional NUL byte, and then adding one or more NUL-terminated strings:

```
003egit-upload-pack /project.git\0host=myserver.com\0\0version=1\0
```

```
git-proto-request = request-command SP pathname NUL
                    [ host-parameter NUL ] [ NUL extra-parameters ]
request-command   = "git-upload-pack" / "git-receive-pack" /
                    "git-upload-archive"   ; case sensitive
pathname          = *( %x01-ff ) ; exclude NUL
host-parameter    = "host=" hostname [ ":" port ]
extra-parameters  = 1*extra-parameter
extra-parameter   = 1*( %x01-ff ) NUL
```

host-parameter is used for the git-daemon name based virtual hosting. See --interpolated-path option to git daemon, with the %H/%CH format characters.

Basically what the Git client is doing to connect to an *upload-pack* process on the server side over the Git protocol is this:

```
$ echo -e -n \
  "003agit-upload-pack /schacon/gitbook.git\0host=example.com\0" |
  nc -v example.com 9418
```

## SSH TRANSPORT

Initiating the upload-pack or receive-pack processes over SSH is executing the binary on the server via SSH remote execution. It is basically equivalent to running this:

```
$ ssh git.example.com "git-upload-pack '/project.git'"
```

For a server to support Git pushing and pulling for a given user over SSH, that user needs to be able to execute one or both of those commands via the SSH shell that they are provided on login. On some systems, that shell access is limited to only being able to run those two commands, or even just one of them.

In an ssh:// format URI, it's absolute in the URI, so the */* after the host name (or port number) is sent as an argument, which is then read by the remote git-upload-pack exactly as is, so it's effectively an absolute path in the remote filesystem.

```
   git clone ssh://user@example.com/project.git
                |
                v
ssh user@example.com "git-upload-pack '/project.git'"
```

In a "user@host:path" format URI, its relative to the user's home directory, because the Git client will run:

```
   git clone user@example.com:project.git
                  |
                  v
ssh user@example.com "git-upload-pack 'project.git'"
```

The exception is if a ~ is used, in which case we execute it without the leading */*.

```
   ssh://user@example.com/~alice/project.git,
                  |
                  v
ssh user@example.com "git-upload-pack '~alice/project.git'"
```

Depending on the value of the **protocol.version** configuration variable, Git may attempt to send Extra

Parameters as a colon-separated string in the GIT_PROTOCOL environment variable. This is done only if the **ssh.variant** configuration variable indicates that the ssh command supports passing environment variables as an argument.

A few things to remember here:

- "command name" is spelled with dash (e.g. git-upload-pack), but this can be overridden by the client;

- repository path is always quoted with single quotes.

## FETCHING DATA FROM A SERVER

When one Git repository wants to get data that a second repository has, the first can *fetch* from the second. This operation determines what data the server has that the client does not then streams that data down to the client in packfile format.

## REFERENCE DISCOVERY

When the client initially connects the server will immediately respond with a version number (if "version=1" is sent as an Extra Parameter), and a listing of each reference it has (all branches and tags) along with the object name that each reference currently points to.

```
$ echo -e -n "0045git-upload-pack /schacon/gitbook.git\0host=example.com\0\0version=1\0" |
   nc -v example.com 9418
000eversion 1
00887217a7c7e582c46cec22a130adf4b9d7d950fba0 HEAD\0multi_ack thin-pack
             side-band side-band-64k ofs-delta shallow no-progress include-tag
00441d3fcd5ced445d1abc402225c0b8a1299641f497 refs/heads/integration
003f7217a7c7e582c46cec22a130adf4b9d7d950fba0 refs/heads/master
003cb88d2441cac0977faf98efc80305012112238d9d refs/tags/v0.9
003c525128480b96c89e6418b1e40909bf6c5b2d580f refs/tags/v1.0
003fe92df48743b7bc7d26bcaabfddde0a1e20cae47c refs/tags/v1.0^{}
0000
```

The returned response is a pkt-line stream describing each ref and its current value. The stream MUST be sorted by name according to the C locale ordering.

If HEAD is a valid ref, HEAD MUST appear as the first advertised ref. If HEAD is not a valid ref, HEAD MUST NOT appear in the advertisement list at all, but other refs may still appear.

The stream MUST include capability declarations behind a NUL on the first ref. The peeled value of a ref (that is "ref^{}") MUST be immediately after the ref itself, if presented. A conforming server MUST peel the ref if it’s an annotated tag.

```
advertised-refs  =  *1("version 1")
                    (no-refs / list-of-refs)
                    *shallow
                    flush-pkt

no-refs          =  PKT-LINE(zero-id SP "capabilities^{}"
                    NUL capability-list)

list-of-refs     =  first-ref *other-ref
first-ref        =  PKT-LINE(obj-id SP refname
                    NUL capability-list)

other-ref        =  PKT-LINE(other-tip / other-peeled)
other-tip        =  obj-id SP refname
other-peeled     =  obj-id SP refname "^{}"

shallow          =  PKT-LINE("shallow" SP obj-id)

capability-list  =  capability *(SP capability)
capability       =  1*(LC_ALPHA / DIGIT / "-" / "_")
LC_ALPHA         =  %x61-7A
```

Server and client MUST use lowercase for obj-id, both MUST treat obj-id as case-insensitive.

See protocol-capabilities.txt for a list of allowed server capabilities and descriptions.

## PACKFILE NEGOTIATION

After reference and capabilities discovery, the client can decide to terminate the connection by sending a flush-pkt, telling the server it can now gracefully terminate, and disconnect, when it does not need any pack data. This can happen with the ls-remote command, and also can happen when the client already is up to date.

Otherwise, it enters the negotiation phase, where the client and server determine what the minimal packfile necessary for transport is, by telling the server what objects it wants, its shallow objects (if any), and the maximum commit depth it wants (if any). The client will also send a list of the capabilities it wants to be in effect, out of what the server said it could do with the first *want* line.

```
upload-request    =  want-list
                     *shallow-line
                     *1depth-request
                     [filter-request]
                     flush-pkt

want-list         =  first-want
                     *additional-want

shallow-line      =  PKT-LINE("shallow" SP obj-id)

depth-request     =  PKT-LINE("deepen" SP depth) /
                     PKT-LINE("deepen-since" SP timestamp) /
                     PKT-LINE("deepen-not" SP ref)

first-want        =  PKT-LINE("want" SP obj-id SP capability-list)
additional-want   =  PKT-LINE("want" SP obj-id)

depth             =  1*DIGIT

filter-request    =  PKT-LINE("filter" SP filter-spec)
```

Clients MUST send all the obj-ids it wants from the reference discovery phase as *want* lines. Clients MUST send at least one *want* command in the request body. Clients MUST NOT mention an obj-id in a *want* command which did not appear in the response obtained through ref discovery.

The client MUST write all obj-ids which it only has shallow copies of (meaning that it does not have the parents of a commit) as *shallow* lines so that the server is aware of the limitations of the client's history.

The client now sends the maximum commit history depth it wants for this transaction, which is the number of commits it wants from the tip of the history, if any, as a *deepen* line. A depth of 0 is the same as not making a depth request. The client does not want to receive any commits beyond this depth, nor does it want objects needed only to complete those commits. Commits whose parents are not received as a result are defined as shallow and marked as such in the server. This information is sent back to the client in the next step.

The client can optionally request that pack-objects omit various objects from the packfile using one of several filtering techniques. These are intended for use with partial clone and partial fetch operations. An object that does not meet a filter-spec value is omitted unless explicitly requested in a *want* line. See **rev-list** for possible filter-spec values.

Once all the *want's and 'shallow's (and optional 'deepen*) are transferred, clients MUST send a flush-pkt, to tell the server side that it is done sending the list.

Otherwise, if the client sent a positive depth request, the server will determine which commits will and will not be shallow and send this information to the client. If the client did not request a positive depth, this step is skipped.

```
shallow-update   =  *shallow-line
                    *unshallow-line
                    flush-pkt

shallow-line     =  PKT-LINE("shallow" SP obj-id)

unshallow-line   =  PKT-LINE("unshallow" SP obj-id)
```

If the client has requested a positive depth, the server will compute the set of commits which are no deeper than the desired depth. The set of commits start at the client's wants.

The server writes *shallow* lines for each commit whose parents will not be sent as a result. The server writes an *unshallow* line for each commit which the client has indicated is shallow, but is no longer shallow at the currently requested depth (that is, its parents will now be sent). The server MUST NOT mark as unshallow anything which the client has not indicated was shallow.

Now the client will send a list of the obj-ids it has using *have* lines, so the server can make a packfile that only contains the objects that the client needs. In multi_ack mode, the canonical implementation will send up to 32 of these at a time, then will send a flush-pkt. The canonical implementation will skip ahead and send the next 32 immediately, so that there is always a block of 32 "in-flight on the wire" at a time.

```
upload-haves      =  have-list
                     compute-end

have-list         =  *have-line
have-line         =  PKT-LINE("have" SP obj-id)
```

```
compute-end     =  flush-pkt / PKT-LINE("done")
```

If the server reads *have* lines, it then will respond by ACKing any of the obj-ids the client said it had that the server also has. The server will ACK obj-ids differently depending on which ack mode is chosen by the client.

In multi_ack mode:

- server will respond with *ACK obj-id continue* for any common commits.

- the server has found an acceptable common base commit and is ready to make a packfile, it will blindly ACK all *have* obj-ids back to the client.

- server will then send a *NAK* and then wait for another response from the client - either a *done* or another list of *have* lines.

In multi_ack_detailed mode:

- server will differentiate the ACKs where it is signaling that it is ready to send data with *ACK obj-id ready* lines, and signals the identified common commits with *ACK obj-id common* lines.

Without either multi_ack or multi_ack_detailed:

- sends "ACK obj-id" on the first common object it finds. After that it says nothing until the client gives it a "done".

- sends "NAK" on a flush-pkt if no common object has been found yet. If one has been found, and thus an ACK was already sent, it’s silent on the flush-pkt.

After the client has gotten enough ACK responses that it can determine that the server has enough information to send an efficient packfile (in the canonical implementation, this is determined when it has received enough ACKs that it can color everything left in the --date-order queue as common with the server, or the --date-order queue is empty), or the client determines that it wants to give up (in the canonical implementation, this is determined when the client sends 256 *have* lines without getting any of them ACKed by the server - meaning there is nothing in common and the server should just send all of its objects), then the client will send a *done* command. The *done* command signals to the server that the client is ready to receive its packfile data.

However, the 256 limit **only** turns on in the canonical client implementation if we have received at least one "ACK %s continue" during a prior round. This helps to ensure that at least one common ancestor is found before we give up entirely.

Once the *done* line is read from the client, the server will either send a final *ACK obj-id* or it will send a *NAK*. *obj-id* is the object name of the last commit determined to be common. The server only sends ACK after *done* if there is at least one common base and multi_ack or multi_ack_detailed is enabled. The server always sends NAK after *done* if there is no common base found.

Instead of *ACK* or *NAK*, the server may send an error message (for example, if it does not recognize an object in a *want* line received from the client).

Then the server will start sending its packfile data.

```
server-response = *ack_multi ack / nak
ack_multi       = PKT-LINE("ACK" SP obj-id ack_status)
ack_status      = "continue" / "common" / "ready"
ack             = PKT-LINE("ACK" SP obj-id)
nak             = PKT-LINE("NAK")
```

A simple clone may look like this (with no *have* lines):

```
C: 0054want 74730d410fcb6603ace96f1dc55ea6196122532d multi_ack \
  side-band-64k ofs-delta\n
C: 0032want 7d1665144a3a975c05f1f43902ddaf084e784dbe\n
```

```
C: 0032want 5a3f6be755bbb7deae50065988cbfa1ffa9ab68a\n
C: 0032want 7e47fe2bd8d01d481f44d7af0531bd93d3b21c01\n
C: 0032want 74730d410fcb6603ace96f1dc55ea6196122532d\n
C: 0000
C: 0009done\n

S: 0008NAK\n
S: [PACKFILE]
```

An incremental update (fetch) response might look like this:

```
C: 0054want 74730d410fcb6603ace96f1dc55ea6196122532d multi_ack \
  side-band-64k ofs-delta\n
C: 0032want 7d1665144a3a975c05f1f43902ddaf084e784dbe\n
C: 0032want 5a3f6be755bbb7deae50065988cbfa1ffa9ab68a\n
C: 0000
C: 0032have 7e47fe2bd8d01d481f44d7af0531bd93d3b21c01\n
C: [30 more have lines]
C: 0032have 74730d410fcb6603ace96f1dc55ea6196122532d\n
C: 0000

S: 003aACK 7e47fe2bd8d01d481f44d7af0531bd93d3b21c01 continue\n
S: 003aACK 74730d410fcb6603ace96f1dc55ea6196122532d continue\n
S: 0008NAK\n

C: 0009done\n

S: 0031ACK 74730d410fcb6603ace96f1dc55ea6196122532d\n
S: [PACKFILE]
```

## PACKFILE DATA

Now that the client and server have finished negotiation about what the minimal amount of data that needs to be sent to the client is, the server will construct and send the required data in packfile format.

See **gitformat-pack**(5) for what the packfile itself actually looks like.

If *side-band* or *side-band-64k* capabilities have been specified by the client, the server will send the packfile data multiplexed.

Each packet starting with the packet-line length of the amount of data that follows, followed by a single byte specifying the sideband the following data is coming in on.

In *side-band* mode, it will send up to 999 data bytes plus 1 control code, for a total of up to 1000 bytes in a pkt-line. In *side-band-64k* mode it will send up to 65519 data bytes plus 1 control code, for a total of up to 65520 bytes in a pkt-line.

The sideband byte will be a *1*, *2* or a *3*. Sideband *1* will contain packfile data, sideband *2* will be used for progress information that the client will generally print to stderr and sideband *3* is used for error information.

If no *side-band* capability was specified, the server will stream the entire packfile without multiplexing.

## PUSHING DATA TO A SERVER

Pushing data to a server will invoke the *receive-pack* process on the server, which will allow the client to tell it which references it should update and then send all the data the server will need for those new references to be complete. Once all the data is received and validated, the server will then update its references to what the client specified.

## AUTHENTICATION

The protocol itself contains no authentication mechanisms. That is to be handled by the transport, such as SSH, before the *receive-pack* process is invoked. If *receive-pack* is configured over the Git transport, those repositories will be writable by anyone who can access that port (9418) as that transport is unauthenticated.

## REFERENCE DISCOVERY

The reference discovery phase is done nearly the same way as it is in the fetching protocol. Each reference obj-id and name on the server is sent in packet-line format to the client, followed by a flush-pkt. The only real difference is that the capability listing is different - the only possible values are *report-status*, *report-status-v2*, *delete-refs*, *ofs-delta*, *atomic* and *push-options*.

## REFERENCE UPDATE REQUEST AND PACKFILE TRANSFER

Once the client knows what references the server is at, it can send a list of reference update requests. For each reference on the server that it wants to update, it sends a line listing the obj-id currently on the server, the obj-id the client would like to update it to and the name of the reference.

This list is followed by a flush-pkt.

```
update-requests   =  *shallow ( command-list | push-cert )
```

```
shallow         =  PKT-LINE("shallow" SP obj-id)

command-list      =  PKT-LINE(command NUL capability-list)
                   *PKT-LINE(command)
                   flush-pkt

command          =  create / delete / update
create          =  zero-id SP new-id  SP name
delete          =  old-id  SP zero-id SP name
update           =  old-id  SP new-id  SP name

old-id          =  obj-id
new-id           =  obj-id

push-cert        = PKT-LINE("push-cert" NUL capability-list LF)
                  PKT-LINE("certificate version 0.1" LF)
                  PKT-LINE("pusher" SP ident LF)
                  PKT-LINE("pushee" SP url LF)
                  PKT-LINE("nonce" SP nonce LF)
                  *PKT-LINE("push-option" SP push-option LF)
                  PKT-LINE(LF)
                  *PKT-LINE(command LF)
                  *PKT-LINE(gpg-signature-lines LF)
                  PKT-LINE("push-cert-end" LF)

push-option       =  1*( VCHAR | SP )
```

If the server has advertised the *push-options* capability and the client has specified *push-options* as part of the capability list above, the client then sends its push options followed by a flush-pkt.

```
push-options      =  *PKT-LINE(push-option) flush-pkt
```

For backwards compatibility with older Git servers, if the client sends a push cert and push options, it MUST send its push options both embedded within the push cert and after the push cert. (Note that the push options within the cert are prefixed, but the push options after the cert are not.) Both these lists MUST be the same, modulo the prefix.

After that the packfile that should contain all the objects that the server will need to complete the new references will be sent.

```
      packfile          =  "PACK" 28*(OCTET)
```

If the receiving end does not support delete-refs, the sending end MUST NOT ask for delete command.

If the receiving end does not support push-cert, the sending end MUST NOT send a push-cert command. When a push-cert command is sent, command-list MUST NOT be sent; the commands recorded in the push certificate is used instead.

The packfile MUST NOT be sent if the only command used is *delete*.

A packfile MUST be sent if either create or update command is used, even if the server already has all the necessary objects. In this case the client MUST send an empty packfile. The only time this is likely to happen is if the client is creating a new branch or a tag that points to an existing obj-id.

The server will receive the packfile, unpack it, then validate each reference that is being updated that it hasn’t changed while the request was being processed (the obj-id is still the same as the old-id), and it will run any update hooks to make sure that the update is acceptable. If all of that is fine, the server will then update the references.

## PUSH CERTIFICATE

A push certificate begins with a set of header lines. After the header and an empty line, the protocol commands follow, one per line. Note that the trailing LF in push-cert PKT-LINEs is *not* optional; it must be present.

Currently, the following header fields are defined:

**pusher** ident
: Identify the GPG key in "Human Readable Name <**email@address**[1]>" format.

**pushee** url
: The repository URL (anonymized, if the URL contains authentication material) the user who ran **git push** intended to push into.

**nonce** nonce
: The *nonce* string the receiving repository asked the pushing user to include in the certificate, to prevent replay attacks.

The GPG signature lines are a detached signature for the contents recorded in the push certificate before the signature block begins. The detached signature is used to certify that the commands were given by the pusher, who must be the signer.

## REPORT STATUS

After receiving the pack data from the sender, the receiver sends a report if *report-status* or *report-status-v2* capability is in effect. It is a short listing of what happened in that update. It will first list the status of the packfile unpacking as either *unpack ok* or *unpack [error]*. Then it will list the status for each of the references that it tried to update. Each line is either *ok [refname]* if the update was successful, or *ng [refname] [error]* if the update was not.

```
report-status     = unpack-status
                    1*(command-status)
                    flush-pkt

unpack-status     = PKT-LINE("unpack" SP unpack-result)
unpack-result     = "ok" / error-msg

command-status    = command-ok / command-fail
command-ok        = PKT-LINE("ok" SP refname)
command-fail      = PKT-LINE("ng" SP refname SP error-msg)

error-msg         = 1*(OCTET) ; where not "ok"
```

The *report-status-v2* capability extends the protocol by adding new option lines in order to support reporting of reference rewritten by the *proc-receive* hook. The *proc-receive* hook may handle a command for a pseudo-reference which may create or update one or more references, and each reference may have different name, different new-oid, and different old-oid.

```
report-status-v2  = unpack-status
                    1*(command-status-v2)
                    flush-pkt

unpack-status     = PKT-LINE("unpack" SP unpack-result)
unpack-result     = "ok" / error-msg

command-status-v2 = command-ok-v2 / command-fail
command-ok-v2     = command-ok
                    *option-line
```

```
command-ok       = PKT-LINE("ok" SP refname)
command-fail     = PKT-LINE("ng" SP refname SP error-msg)

error-msg        = 1*(OCTET) ; where not "ok"

option-line      = *1(option-refname)
                   *1(option-old-oid)
                   *1(option-new-oid)
                   *1(option-forced-update)

option-refname   = PKT-LINE("option" SP "refname" SP refname)
option-old-oid   = PKT-LINE("option" SP "old-oid" SP obj-id)
option-new-oid   = PKT-LINE("option" SP "new-oid" SP obj-id)
option-force     = PKT-LINE("option" SP "forced-update")
```

Updates can be unsuccessful for a number of reasons. The reference can have changed since the reference discovery phase was originally sent, meaning someone pushed in the meantime. The reference being pushed could be a non-fast-forward reference and the update hooks or configuration could be set to not allow that, etc. Also, some references can be updated while others can be rejected.

An example client/server communication might look like this:

```
S: 006274730d410fcb6603ace96f1dc55ea6196122532d refs/heads/local\0report-status delete-refs ofs-delta\n
S: 003e7d1665144a3a975c05f1f43902ddaf084e784dbe refs/heads/debug\n
S: 003f74730d410fcb6603ace96f1dc55ea6196122532d refs/heads/master\n
S: 003d74730d410fcb6603ace96f1dc55ea6196122532d refs/heads/team\n
S: 0000

C: 00677d1665144a3a975c05f1f43902ddaf084e784dbe 74730d410fcb6603ace96f1dc55ea6196122532d refs/hea
C: 006874730d410fcb6603ace96f1dc55ea6196122532d 5a3f6be755bbb7deae50065988cbfa1ffa9ab68a refs/hea
C: 0000
C: [PACKDATA]

S: 000eunpack ok\n
S: 0018ok refs/heads/debug\n
S: 002ang refs/heads/master non-fast-forward\n
```

## GIT

Part of the **git**(1) suite

## NOTES

1. email@address
   mailto:email@address