NAME

gitprotocol-v2 - Git Wire Protocol, Version 2

SYNOPSIS

<over-the-wire-protocol>

DESCRIPTION

This document presents a specification for a version 2 of Git's wire protocol. Protocol v2 will improve upon v1 in the following ways:

- of multiple service names, multiple commands will be supported by a single service
- extendable as capabilities are moved into their own section of the protocol, no longer being hidden behind a NUL byte and limited by the size of a pkt-line
- out other information hidden behind NUL bytes (e.g. agent string as a capability and symrefs can be requested using *ls-refs*)
- advertisement will be omitted unless explicitly requested
- command to explicitly request some refs
- with http and stateless-rpc in mind. With clear flush semantics the http remote helper can simply act as a proxy

In protocol v2 communication is command oriented. When first contacting a server a list of capabilities will advertised. Some of these capabilities will be commands which a client can request be executed. Once a command has completed, a client can reuse the connection and request that other commands be executed.

PACKET-LINE FRAMING

All communication is done using packet-line framing, just as in v1. See **gitprotocol-pack**(5) and **gitprotocol-common**(5) for more information.

In protocol v2 these special packets will have the following semantics:

- Flush Packet (flush-pkt) - indicates the end of a message

- Delimiter Packet (delim-pkt) - separates sections of a message

- Response End Packet (response-end-pkt) - indicates the end of a response for stateless connections

INITIAL CLIENT REQUEST

In general a client can request to speak protocol v2 by sending **version=2** through the respective side-channel for the transport being used which inevitably sets **GIT_PROTOCOL**. More information can be found in **gitprotocol-pack**(5) and **gitprotocol-http**(5), as well as the **GIT_PROTOCOL** definition in **git.txt**. In all cases the response from the server is the capability advertisement.

Git Transport

When using the git:// transport, you can request to use protocol v2 by sending "version=2" as an extra parameter:

```
003egit-upload-pack /project.git\0host=myserver.com\0\0version=2\0
```

SSH and File Transport

When using either the ssh:// or file:// transport, the GIT_PROTOCOL environment variable must be set explicitly to include "version=2". The server may need to be configured to allow this environment variable to pass.

HTTP Transport

When using the http:// or https:// transport a client makes a "smart" info/refs request as described in **gitprotocol-http**(5) and requests that v2 be used by supplying "version=2" in the **Git-Protocol** header.

```
C: GET $GIT_URL/info/refs?service=git-upload-pack HTTP/1.0
C: Git-Protocol: version=2
```

A v2 server would reply:

```
S: 200 OK
S: <Some headers>
```

S: ...
S:
S: 000eversion 2\n
S: <capability-advertisement>

Subsequent requests are then made directly to the service **$GIT_URL/git-upload-pack**. (This works the same for git-receive-pack).

Uses the **--http-backend-info-refs** option to **git-upload-pack**(1).

The server may need to be configured to pass this header's contents via the **GIT_PROTOCOL** variable. See the discussion in **git-http-backend.txt**.

CAPABILITY ADVERTISEMENT

A server which decides to communicate (based on a request from a client) using protocol version 2, notifies the client by sending a version string in its initial response followed by an advertisement of its capabilities. Each capability is a key with an optional value. Clients must ignore all unknown keys. Semantics of unknown values are left to the definition of each key. Some capabilities will describe commands which can be requested to be executed by the client.

```
capability-advertisement = protocol-version
                           capability-list
                           flush-pkt

protocol-version = PKT-LINE("version 2" LF)
capability-list = *capability
capability = PKT-LINE(key[=value] LF)

key = 1*(ALPHA | DIGIT | "-_")
value = 1*(ALPHA | DIGIT | " -_.,?\/{}[]()<>!@#$%^&*+=:;")
```

COMMAND REQUEST

After receiving the capability advertisement, a client can then issue a request to select the command it wants with any particular capabilities or arguments. There is then an optional section where the client can provide any command specific parameters or queries. Only a single command can be requested at a time.

```
request = empty-request | command-request
empty-request = flush-pkt
command-request = command
```

```
                capability-list
                delim-pkt
                command-args
                flush-pkt
        command = PKT-LINE("command=" key LF)
        command-args = *command-specific-arg

        command-specific-args are packet line framed arguments defined by
        each individual command.
```

The server will then check to ensure that the client's request is comprised of a valid command as well as valid capabilities which were advertised. If the request is valid the server will then execute the command. A server MUST wait till it has received the client's entire request before issuing a response. The format of the response is determined by the command being executed, but in all cases a flush-pkt indicates the end of the response.

When a command has finished, and the client has received the entire response from the server, a client can either request that another command be executed or can terminate the connection. A client may optionally send an empty request consisting of just a flush-pkt to indicate that no more requests will be made.

CAPABILITIES

There are two different types of capabilities: normal capabilities, which can be used to convey information or alter the behavior of a request, and commands, which are the core actions that a client wants to perform (fetch, push, etc).

Protocol version 2 is stateless by default. This means that all commands must only last a single round and be stateless from the perspective of the server side, unless the client has requested a capability indicating that state should be maintained by the server. Clients MUST NOT require state management on the server side in order to function correctly. This permits simple round-robin load-balancing on the server side, without needing to worry about state management.

agent

The server can advertise the **agent** capability with a value **X** (in the form **agent=X**) to notify the client that the server is running version **X**. The client may optionally send its own agent string by including the **agent** capability with a value **Y** (in the form **agent=Y**) in its request to the server (but it MUST NOT do so if the server did not advertise the agent capability). The **X** and **Y** strings may contain any printable ASCII characters except space (i.e., the byte range 32 < x < 127), and are typically of the form "package/version" (e.g., "git/1.8.3.1"). The agent strings are purely informative for statistics and debugging purposes, and MUST NOT be used to programmatically assume the presence or absence of

particular features.

ls-refs

ls-refs is the command used to request a reference advertisement in v2. Unlike the current reference advertisement, ls-refs takes in arguments which can be used to limit the refs sent from the server.

Additional features not supported in the base command will be advertised as the value of the command in the capability advertisement in the form of a space separated list of features: "<command>=<feature 1> <feature 2>"

ls-refs takes in the following arguments:

symrefs
: In addition to the object pointed by it, show the underlying ref pointed by it when showing a symbolic ref.

peel
: Show peeled tags.

ref-prefix <prefix>
: When specified, only references having a prefix matching one of the provided prefixes are displayed. Multiple instances may be given, in which case references matching any prefix will be shown. Note that this is purely for optimization; a server MAY show refs not matching the prefix if it chooses, and clients should filter the result themselves.

If the *unborn* feature is advertised the following argument can be included in the client's request.

unborn
: The server will send information about HEAD even if it is a symref pointing to an unborn branch in the form "unborn HEAD symref-target:<target>".

The output of ls-refs is as follows:

```
output = *ref
       flush-pkt
obj-id-or-unborn = (obj-id | "unborn")
ref = PKT-LINE(obj-id-or-unborn SP refname *(SP ref-attribute) LF)
ref-attribute = (symref | peeled)
symref = "symref-target:" symref-target
```

peeled = "peeled:" obj-id

fetch

fetch is the command used to fetch a packfile in v2. It can be looked at as a modified version of the v1 fetch where the ref-advertisement is stripped out (since the **ls-refs** command fills that role) and the message format is tweaked to eliminate redundancies and permit easy addition of future extensions.

Additional features not supported in the base command will be advertised as the value of the command in the capability advertisement in the form of a space separated list of features: "<command>=<feature 1> <feature 2>"

A **fetch** request can take the following arguments:

want <oid>
Indicates to the server an object which the client wants to retrieve. Wants can be anything and are not limited to advertised objects.

have <oid>
Indicates to the server an object which the client has locally. This allows the server to make a packfile which only contains the objects that the client needs. Multiple ’have’ lines can be supplied.

done
Indicates to the server that negotiation should terminate (or not even begin if performing a clone) and that the server should use the information supplied in the request to construct the packfile.

thin-pack
Request that a thin pack be sent, which is a pack with deltas which reference base objects not contained within the pack (but are known to exist at the receiving end). This can reduce the network traffic significantly, but it requires the receiving end to know how to "thicken" these packs by adding the missing bases to the pack.

no-progress
Request that progress information that would normally be sent on

side-band channel 2, during the packfile transfer, should not be sent. However, the side-band channel 3 is still used for error responses.

include-tag
: Request that annotated tags should be sent if the objects they point to are being sent.

ofs-delta
: Indicate that the client understands PACKv2 with delta referring to its base by position in pack rather than by an oid. That is, they can read OBJ_OFS_DELTA (aka type 6) in a packfile.

If the *shallow* feature is advertised the following arguments can be included in the clients request as well as the potential addition of the *shallow-info* section in the server's response as explained below.

shallow <oid>
: A client must notify the server of all commits for which it only has shallow copies (meaning that it doesn't have the parents of a commit) by supplying a 'shallow <oid>' line for each such object so that the server is aware of the limitations of the client's history. This is so that the server is aware that the client may not have all objects reachable from such commits.

deepen <depth>
: Requests that the fetch/clone should be shallow having a commit depth of <depth> relative to the remote side.

deepen-relative
: Requests that the semantics of the "deepen" command be changed to indicate that the depth requested is relative to the client's current shallow boundary, instead of relative to the requested commits.

deepen-since <timestamp>
: Requests that the shallow clone/fetch should be cut at a specific time, instead of depth. Internally it's equivalent to doing "git rev-list --max-age=<timestamp>". Cannot be used with "deepen".

deepen-not <rev>
: Requests that the shallow clone/fetch should be cut at a specific revision specified by '<rev>', instead of a depth. Internally it's equivalent of doing "git rev-list --not <rev>". Cannot be used with "deepen", but can be used with "deepen-since".

If the *filter* feature is advertised, the following argument can be included in the client's request:

filter <filter-spec>
: Request that various objects from the packfile be omitted using one of several filtering techniques. These are intended for use with partial clone and partial fetch operations. See 'rev-list' for possible "filter-spec" values. When communicating with other processes, senders SHOULD translate scaled integers (e.g. "1k") into a fully-expanded form (e.g. "1024") to aid interoperability with older receivers that may not understand newly-invented scaling suffixes. However, receivers SHOULD accept the following suffixes: 'k', 'm', and 'g' for 1024, 1048576, and 1073741824, respectively.

If the *ref-in-want* feature is advertised, the following argument can be included in the client's request as well as the potential addition of the *wanted-refs* section in the server's response as explained below.

want-ref <ref>
: Indicates to the server that the client wants to retrieve a particular ref, where <ref> is the full name of a ref on the server.

If the *sideband-all* feature is advertised, the following argument can be included in the client's request:

sideband-all
: Instruct the server to send the whole response multiplexed, not just the packfile section. All non-flush and non-delim PKT-LINE in the response (not only in the packfile section) will then start with a byte indicating its sideband (1, 2, or 3), and the server may send "0005\2" (a PKT-LINE of sideband 2 with no payload) as a keepalive packet.

If the *packfile-uris* feature is advertised, the following argument can be included in the client's request as well as the potential addition of the *packfile-uris* section in the server's response as explained below.

packfile-uris <comma-separated list of protocols>
: Indicates to the server that the client is willing to receive URIs of any of the given protocols in place of objects in the sent packfile. Before performing the connectivity check, the client should download from all given URIs. Currently, the protocols supported are "http" and "https".

If the *wait-for-done* feature is advertised, the following argument can be included in the client's request.

wait-for-done
: Indicates to the server that it should never send "ready", but should wait for the client to say "done" before sending the packfile.

The response of **fetch** is broken into a number of sections separated by delimiter packets (0001), with each section beginning with its section header. Most sections are sent only when the packfile is sent.

```
output = acknowledgements flush-pkt |
        [acknowledgments delim-pkt] [shallow-info delim-pkt]
        [wanted-refs delim-pkt] [packfile-uris delim-pkt]
        packfile flush-pkt

acknowledgments = PKT-LINE("acknowledgments" LF)
                  (nak | *ack)
                  (ready)
ready = PKT-LINE("ready" LF)
nak = PKT-LINE("NAK" LF)
ack = PKT-LINE("ACK" SP obj-id LF)

shallow-info = PKT-LINE("shallow-info" LF)
               *PKT-LINE((shallow | unshallow) LF)
shallow = "shallow" SP obj-id
unshallow = "unshallow" SP obj-id

wanted-refs = PKT-LINE("wanted-refs" LF)
              *PKT-LINE(wanted-ref LF)
wanted-ref = obj-id SP refname

packfile-uris = PKT-LINE("packfile-uris" LF) *packfile-uri
```

```
packfile-uri = PKT-LINE(40*(HEXDIGIT) SP *%x20-ff LF)

packfile = PKT-LINE("packfile" LF)
        *PKT-LINE(%x01-03 *%x00-ff)

acknowledgments section
   * If the client determines that it is finished with negotiations by
     sending a "done" line (thus requiring the server to send a packfile),
     the acknowledgments sections MUST be omitted from the server's
     response.
```

- begins with the section header "acknowledgments"

- server will respond with "NAK" if none of the object ids sent as have lines were common.

- server will respond with "ACK obj-id" for all of the object ids sent as have lines which are common.

- response cannot have both "ACK" lines as well as a "NAK" line.

- server will respond with a "ready" line indicating that the server has found an acceptable common base and is ready to make and send a packfile (which will be found in the packfile section of the same response)

- the server has found a suitable cut point and has decided to send a "ready" line, then the server can decide to (as an optimization) omit any "ACK" lines it would have sent during its response. This is because the server will have already determined the objects it plans to send to the client and no further negotiation is needed.

```
        shallow-info section
           * If the client has requested a shallow fetch/clone, a shallow
             client requests a fetch or the server is shallow then the
             server's response may include a shallow-info section.  The
             shallow-info section will be included if (due to one of the
             above conditions) the server needs to inform the client of any
```

shallow boundaries or adjustments to the clients already existing shallow boundaries.

- begins with the section header "shallow-info"

- a positive depth is requested, the server will compute the set of commits which are no deeper than the desired depth.

- server sends a "shallow obj-id" line for each commit whose parents will not be sent in the following packfile.

- server sends an "unshallow obj-id" line for each commit which the client has indicated is shallow, but is no longer shallow as a result of the fetch (due to its parents being sent in the following packfile).

- server MUST NOT send any "unshallow" lines for anything which the client has not indicated was shallow as a part of its request.

wanted-refs section
* This section is only included if the client has requested a ref using a ’want-ref’ line and if a packfile section is also included in the response.

- begins with the section header "wanted-refs".

- server will send a ref listing ("<oid> <refname>") for each reference requested using *want-ref* lines.

- server MUST NOT send any refs which were not requested using *want-ref* lines.

packfile-uris section
* This section is only included if the client sent ’packfile-uris’ and the server has at least one such URI to send.

- begins with the section header "packfile-uris".

- each URI the server sends, it sends a hash of the pack’s contents (as output by git index-pack) followed by the URI.

- hashes are 40 hex characters long. When Git upgrades to a new hash algorithm, this might need to be updated. (It should match whatever index-pack outputs after "pack\t" or "keep\t".

  packfile section
  * This section is only included if the client has sent ’want’ lines in its request and either requested that no more negotiation be done by sending ’done’ or if the server has decided it has found a sufficient cut point to produce a packfile.

- begins with the section header "packfile"

- transmission of the packfile begins immediately after the section header

- data transfer of the packfile is always multiplexed, using the same semantics of the *side-band-64k* capability from protocol version 1. This means that each packet, during the packfile data stream, is made up of a leading 4-byte pkt-line length (typical of the pkt-line format), followed by a 1-byte stream code, followed by the actual data.

  The stream code can be one of:
  1 - pack data
  2 - progress messages
  3 - fatal error message just before stream aborts

server-option
If advertised, indicates that any number of server specific options can be included in a request. This is done by sending each option as a "server-option=<option>" capability line in the capability-list section of a request.

The provided options must not contain a NUL or LF character.

object-format

The server can advertise the **object-format** capability with a value **X** (in the form **object-format=X**) to notify the client that the server is able to deal with objects using hash algorithm X. If not specified, the server is assumed to only handle SHA-1. If the client would like to use a hash algorithm other than SHA-1, it should specify its object-format string.

session-id=<session id>

The server may advertise a session ID that can be used to identify this process across multiple requests. The client may advertise its own session ID back to the server as well.

Session IDs should be unique to a given process. They must fit within a packet-line, and must not contain non-printable or whitespace characters. The current implementation uses trace2 session IDs (see **api-trace2**[1] for details), but this may change and users of the session ID should not rely on this fact.

object-info

object-info is the command to retrieve information about one or more objects. Its main purpose is to allow a client to make decisions based on this information without having to fully fetch objects. Object size is the only information that is currently supported.

An **object-info** request takes the following arguments:

size
Requests size information to be returned for each listed object id.

oid <oid>
Indicates to the server an object which the client wants to obtain information for.

The response of **object-info** is a list of the requested object ids and associated requested information, each separated by a single space.

```
output = info flush-pkt

info = PKT-LINE(attrs) LF)
       *PKT-LINE(obj-info LF)
```

attrs = attr | attrs SP attrs

attr = "size"

obj-info = obj-id SP obj-size

bundle-uri

If the *bundle-uri* capability is advertised, the server supports the ‘bundle-uri’ command.

The capability is currently advertised with no value (i.e. not "bundle-uri=somevalue"), a value may be added in the future for supporting command-wide extensions. Clients MUST ignore any unknown capability values and proceed with the ’bundle-uri‘ dialog they support.

The *bundle-uri* command is intended to be issued before **fetch** to get URIs to bundle files (see **git-bundle**(1)) to "seed" and inform the subsequent **fetch** command.

The client CAN issue **bundle-uri** before or after any other valid command. To be useful to clients it’s expected that it’ll be issued after an **ls-refs** and before **fetch**, but CAN be issued at any time in the dialog.

DISCUSSION of bundle-uri

The intent of the feature is optimize for server resource consumption in the common case by changing the common case of fetching a very large PACK during **git-clone**(1) into a smaller incremental fetch.

It also allows servers to achieve better caching in combination with an **uploadpack.packObjectsHook** (see **git-config**(1)).

By having new clones or fetches be a more predictable and common negotiation against the tips of recently produces *.bundle file(s). Servers might even pre-generate the results of such negotiations for the **uploadpack.packObjectsHook** as new pushes come in.

One way that servers could take advantage of these bundles is that the server would anticipate that fresh clones will download a known bundle, followed by catching up to the current state of the repository using ref tips found in that bundle (or bundles).

PROTOCOL for bundle-uri

A **bundle-uri** request takes no arguments, and as noted above does not currently advertise a

capability value. Both may be added in the future.

When the client issues a **command=bundle-uri** request, the response is a list of key-value pairs provided as packet lines with value **<key>=<value>**. Each **<key>** should be interpreted as a config key from the **bundle.*** namespace to construct a list of bundles. These keys are grouped by a **bundle.<id>.** subsection, where each key corresponding to a given **<id>** contributes attributes to the bundle defined by that **<id>**. See **git-config**(1) for the specific details of these keys and how the Git client will interpret their values.

Clients MUST parse the line according to the above format, lines that do not conform to the format SHOULD be discarded. The user MAY be warned in such a case.

bundle-uri CLIENT AND SERVER EXPECTATIONS

URI CONTENTS

The content at the advertised URIs MUST be one of two types.

The advertised URI may contain a bundle file that **git bundle verify** would accept. I.e. they MUST contain one or more reference tips for use by the client, MUST indicate prerequisites (in any) with standard "-" prefixes, and MUST indicate their "object-format", if applicable.

The advertised URI may alternatively contain a plaintext file that **git config --list** would accept (with the **--file** option). The key-value pairs in this list are in the **bundle.*** namespace (see **git-config**(1)).

bundle-uri CLIENT ERROR RECOVERY

A client MUST above all gracefully degrade on errors, whether that error is because of bad missing/data in the bundle URI(s), because that client is too dumb to e.g. understand and fully parse out bundle headers and their prerequisite relationships, or something else.

Server operators should feel confident in turning on "bundle-uri" and not worry if e.g. their CDN goes down that clones or fetches will run into hard failures. Even if the server bundle(s) are incomplete, or bad in some way the client should still end up with a functioning repository, just as if it had chosen not to use this protocol extension.

All subsequent discussion on client and server interaction MUST keep this in mind.

bundle-uri SERVER TO CLIENT

The ordering of the returned bundle uris is not significant. Clients MUST parse their headers to discover their contained OIDS and prerequisites. A client MUST consider the content of

the bundle(s) themselves and their header as the ultimate source of truth.

A server MAY even return bundle(s) that don’t have any direct relationship to the repository being cloned (either through accident, or intentional "clever" configuration), and expect a client to sort out what data they’d like from the bundle(s), if any.

bundle-uri CLIENT TO SERVER

The client SHOULD provide reference tips found in the bundle header(s) as *have* lines in any subsequent **fetch** request. A client MAY also ignore the bundle(s) entirely if doing so is deemed worse for some reason, e.g. if the bundles can’t be downloaded, it doesn’t like the tips it finds etc.

WHEN ADVERTISED BUNDLE(S) REQUIRE NO FURTHER NEGOTIATION

If after issuing **bundle-uri** and **ls-refs**, and getting the header(s) of the bundle(s) the client finds that the ref tips it wants can be retrieved entirely from advertised bundle(s), the client MAY disconnect from the Git server. The results of such a *clone* or *fetch* should be indistinguishable from the state attained without using bundle-uri.

EARLY CLIENT DISCONNECTIONS AND ERROR RECOVERY

A client MAY perform an early disconnect while still downloading the bundle(s) (having streamed and parsed their headers). In such a case the client MUST gracefully recover from any errors related to finishing the download and validation of the bundle(s).

I.e. a client might need to re-connect and issue a *fetch* command, and possibly fall back to not making use of *bundle-uri* at all.

This "MAY" behavior is specified as such (and not a "SHOULD") on the assumption that a server advertising bundle uris is more likely than not to be serving up a relatively large repository, and to be pointing to URIs that have a good chance of being in working order. A client MAY e.g. look at the payload size of the bundles as a heuristic to see if an early disconnect is worth it, should falling back on a full "fetch" dialog be necessary.

WHEN ADVERTISED BUNDLE(S) REQUIRE FURTHER NEGOTIATION

A client SHOULD commence a negotiation of a PACK from the server via the "fetch" command using the OID tips found in advertised bundles, even if’s still in the process of downloading those bundle(s).

This allows for aggressive early disconnects from any interactive server dialog. The client blindly trusts that the advertised OID tips are relevant, and issues them as *have* lines, it then requests any tips it would like (usually from the "ls-refs" advertisement) via *want* lines. The

server will then compute a (hopefully small) PACK with the expected difference between the tips from the bundle(s) and the data requested.

The only connection the client then needs to keep active is to the concurrently downloading static bundle(s), when those and the incremental PACK are retrieved they should be inflated and validated. Any errors at this point should be gracefully recovered from, see above.

bundle-uri PROTOCOL FEATURES

The client constructs a bundle list from the **<key>=<value>** pairs provided by the server. These pairs are part of the **bundle.*** namespace as documented in **git-config**(1). In this section, we discuss some of these keys and describe the actions the client will do in response to this information.

In particular, the **bundle.version** key specifies an integer value. The only accepted value at the moment is **1**, but if the client sees an unexpected value here then the client MUST ignore the bundle list.

As long as **bundle.version** is understood, all other unknown keys MAY be ignored by the client. The server will guarantee compatibility with older clients, though newer clients may be better able to use the extra keys to minimize downloads.

Any backwards-incompatible addition of pre-URI key-value will be guarded by a new **bundle.version** value or values in *bundle-uri* capability advertisement itself, and/or by new future **bundle-uri** request arguments.

Some example key-value pairs that are not currently implemented but could be implemented in the future include:

- a "hash=<val>" or "size=<bytes>" advertise the expected hash or size of the bundle file.

- that one or more bundle files are the same (to e.g. have clients round-robin or otherwise choose one of N possible files).

- "oid=<OID>" shortcut and "prerequisite=<OID>" shortcut. For expressing the common case of a bundle with one tip and no prerequisites, or one tip and one prerequisite.

This would allow for optimizing the common case of servers who'd like to provide one "big bundle" containing only their "main" branch, and/or incremental updates thereof.

A client receiving such a a response MAY assume that they can skip retrieving the header from a bundle at the indicated URI, and thus save themselves and the server(s) the request(s) needed to inspect the headers of that bundle or bundles.

GIT

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

NOTES

1. api-trace2
   git-htmldocs/technical/api-trace2.html