format

The NBD protocol

Introduction

The Network Block Device is a Linux-originated lightweight block access protocol that allows one to export a block device to a client. While the name of the protocol specifically references the concept of block devices, there is nothing inherent in the *protocol* which requires that exports are, in fact, block devices; the protocol only concerns itself with a range of bytes, and several operations of particular lengths at particular offsets within that range of bytes.

For matters of clarity, in this document we will refer to an export from a server as a block device, even though the actual backing on the server need not be an actual block device; it may be a block device, a regular file, or a more complex configuration involving several files. That is an implementation detail of the server.

Conventions

In the below protocol descriptions, the label 'C:' is used for messages sent by the client, whereas 'S:' is used for messages sent by the server). `monotype text` is for literal character data or (when used in comments) constant names, `0xdeadbeef` is used for literal hex numbers (which are always sent in network byte order), and (brackets) are used for comments. Anything else is a description of the data that is sent.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119. The same words in lower case carry their natural meaning.

Where this document refers to a string, then unless otherwise stated, that string is a sequence of UTF-8 code points, which is not `NUL` terminated, MUST NOT contain `NUL` characters, SHOULD be no longer than 256 bytes and MUST be no longer than 4096 bytes. This applies to export names and error messages (amongst others). The length of a string is always available through information sent earlier in the same message, although it may require some

computation based on the size of other data also present in the same message.

Protocol phases

The NBD protocol has two phases: the handshake and the transmission. During the handshake, a connection is established and an exported NBD device along other protocol parameters are negotiated between the client and the server. After a successful handshake, the client and the server proceed to the transmission phase in which the export is read from and written to.

On the client side under Linux, the handshake is implemented in userspace, while the transmission phase is implemented in kernel space. To get from the handshake to the transmission phase, the client performs

```
ioctl(nbd, NBD_SET_SOCK, sock)
ioctl(nbd, NBD_DO_IT)
```

with `nbd` in the above being a file descriptor for an open `/dev/nbdX` device node, and `sock` being the socket to the server. The second of the above two calls does not return until the client disconnects.

Note that there are other `ioctl` calls available, that are used by the client to communicate the options to the kernel which were negotiated with the server during the handshake. This document does not describe those.

When handling the client-side transmission phase with the Linux kernel, the socket between the client and server can use either Unix or TCP sockets. For other implementations, the client and server can use any agreeable communication channel (a socket is typical, but it is also possible to implement the NBD protocol over a pair of uni-directional pipes). If TCP sockets are used, both the client and server SHOULD disable Nagle's algorithm (that is, use `setsockopt` to set the `TCP_NODELAY` option to non-zero), to eliminate artificial delays caused by waiting for an ACK response when a large message payload spans multiple network packets.

Handshake

The handshake is the first phase of the protocol. Its main purpose is to provide means for both the client and the server to negotiate which export they are going to use and how.

There are three versions of the negotiation. They are referred to as “oldstyle”, “newstyle”, and “fixed newstyle” negotiation. Oldstyle was the only version of the negotiation until nbd 2.9.16; newstyle was introduced for nbd 2.9.17. A short while later, it was discovered that newstyle was insufficiently structured to allow protocol options to be added while retaining backwards compatibility. The minor changes introduced to fix this problem are, where necessary, referred to as “fixed newstyle” to differentiate from the original version of the newstyle negotiation.

Oldstyle negotiation

S: 64 bits, `0x4e42444d41474943` (ASCII ‘`NBDMAGIC`’) (also known as the `INIT_PASSWD`)
S: 64 bits, `0x00420281861253` (`cliserv_magic`, a magic number)
S: 64 bits, size of the export in bytes (unsigned)
S: 32 bits, flags
S: 124 bytes, zeroes (reserved).

As can be seen, this isn’t exactly a negotiation; it’s just a matter of the server sending a bunch of data to the client. If the client is unhappy with what he receives, he should disconnect and not look back.

The fact that the size of the export was specified before the flags were sent, made it impossible for the protocol to be changed in a backwards-compatible manner to allow for named exports without ugliness. As a result, the old style negotiation is now no longer developed; starting with version 3.10 of the reference implementation, it is also no longer supported.

Newstyle negotiation

A client who wants to use the new style negotiation SHOULD connect on the IANA-reserved port for NBD, 10809. The server MAY listen on other ports as well, but it SHOULD use the old style handshake on those. The server SHOULD refuse to allow oldstyle negotiations on the newstyle port. For debugging purposes, the server MAY change the port on which to listen for newstyle negotiation, but this SHOULD NOT happen for production purposes.

The initial few exchanges in newstyle negotiation look as follows:

S: 64 bits, `0x4e42444d41474943` (ASCII ‘`NBDMAGIC`’) (as in the old style handshake)
S: 64 bits, `0x49484156454F5054` (ASCII ‘`IHAVEOPT`’) (note different magic number)
S: 16 bits, handshake flags
C: 32 bits, client flags

This completes the initial phase of negotiation; the client and server now both know they understand the first version of the newstyle handshake, with no options. The client SHOULD ignore any handshake flags it does not recognize, while the server MUST close the TCP connection if it does not recognize the client's flags. What follows is a repeating group of options. In non-fixed newstyle only one option can be set (`NBD_OPT_EXPORT_NAME`), and it is not optional.

At this point, we move on to option haggling, during which point the client can send one or (in fixed newstyle) more options to the server. The generic format of setting an option is as follows:

C: 64 bits, `0x49484156454F5054` (ASCII '`IHAVEOPT`') (note same newstyle handshake's magic number)
C: 32 bits, option
C: 32 bits, length of option data (unsigned)
C: any data needed for the chosen option, of length as specified above.

The presence of the option length in every option allows the server to skip any options presented by the client that it does not understand.

If the value of the option field is `NBD_OPT_EXPORT_NAME` and the server is willing to allow the export, the server replies with information about the used export:

S: 64 bits, size of the export in bytes (unsigned)
S: 16 bits, transmission flags
S: 124 bytes, zeroes (reserved) (unless `NBD_FLAG_C_NO_ZEROES` was negotiated by the client)

If the server is unwilling to allow the export, it MUST terminate the session.

The reason that the flags field is 16 bits large and not 32 as in the oldstyle negotiation is that there are now 16 bits of transmission flags, and 16 bits of handshake flags. Concatenated together, this results in 32 bits, which allows for using a common set of macros for both. If we ever run out of flags, the server will set the most significant flag bit, signalling that an extra flag field will follow, to which the client will have to reply with a flag field of its own before the extra flags are sent. This is not yet implemented.

Fixed newstyle negotiation

Unfortunately, due to a mistake, the server would immediately close the connection when it saw an option it did not understand, rather than signalling this fact to the client, which would've allowed it to retry; and replies from the

server were not structured either, which meant that if the server were to send something the client did not understand, it would have to abort negotiation as well.

To fix these two issues, the following changes were implemented:

- The server will set the handshake flag `NBD_FLAG_FIXED_NEWSTYLE`, to signal that it supports fixed newstyle negotiation.
- The client SHOULD reply with `NBD_FLAG_C_FIXED_NEWSTYLE` set in its flags field too, though its side of the protocol does not change incompatibly.
- The client MAY now send other options to the server as appropriate, in the generic format for sending an option as described above.
- The server will reply to any option apart from `NBD_OPT_EXPORT_NAME` with reply packets in the following format:

S: 64 bits, `0x3e889045565a9` (magic number for replies)
S: 32 bits, the option as sent by the client to which this is a reply
S: 32 bits, reply type (e.g., `NBD_REP_ACK` for successful completion, or `NBD_REP_ERR_UNSUP` to mark use of an option not known by this server
S: 32 bits, length of the reply. This MAY be zero for some replies, in which case the next field is not sent
S: any data as required by the reply (e.g., an export name in the case of `NBD_REP_SERVER`)

The client MUST NOT send any option until it has received a final reply to any option it has sent (note that some options e.g. `NBD_OPT_LIST` have multiple replies, and the final reply is the last of those).

Some messages the client sends instruct the server to change some of its internal state. The client SHOULD NOT send such messages more than once; if it does, the server MAY fail the repeated message with `NBD_REP_ERR_INVALID`.

Termination of the session during option haggling

There are three possible mechanisms to end option haggling:

- Transmission mode can be entered (by the client sending `NBD_OPT_EXPORT_NAME` or by the server responding to an `NBD_OPT_GO` with `NBD_REP_ACK`). This is documented elsewhere.
- The client can send (and the server can reply to) an `NBD_OPT_ABORT`. This MUST be followed by the client shutting down TLS (if it is running), and

the client dropping the connection. This is referred to as 'initiating a soft disconnect'; soft disconnects can only be initiated by the client.

- The client or the server can disconnect the TCP session without activity at the NBD protocol level. If TLS is negotiated, the party initiating the transaction SHOULD shutdown TLS first if it is running. This is referred to as 'initiating a hard disconnect'.

This section concerns the second and third of these, together called 'terminating the session', and under which circumstances they are valid.

If either the client or the server detects a violation of a mandatory condition ('MUST' etc.) by the other party, it MAY initiate a hard disconnect.

A client MAY use a soft disconnect to terminate the session whenever it wishes.

A party that is mandated by this document to terminate the session MUST initiate a hard disconnect if it is not possible to use a soft disconnect. Such circumstances include: where that party is the server and it cannot return an error (e.g. after an `NBD_OPT_EXPORT_NAME` it cannot satisfy), and where that party is the client following a failed TLS negotiation.

A party MUST NOT initiate a hard disconnect save where set out in this section. Therefore, unless a client's situation falls within the provisions of the previous paragraph or the client detects a breach of a mandatory condition, it MUST NOT use a hard disconnect, and hence its only option to terminate the session is via a soft disconnect.

There is no requirement for the client or server to complete a negotiation if it does not wish to do so. Either end MAY simply terminate the session. In the client's case, if it wishes to do so it MUST use soft disconnect.

In the server's case it MUST (save where set out above) simply error inbound options until the client gets the hint that it is unwelcome, except that if a server believes a client's behaviour constitutes a denial of service, it MAY initiate a hard disconnect. If the server is in the process of being shut down it MAY error any inflight option and SHOULD error further options received (other than an `NBD_OPT_ABORT`) with `NBD_REP_ERR_SHUTDOWN`.

If the client receives `NBD_REP_ERR_SHUTDOWN` it MUST initiate a soft disconnect.

Transmission

There are three message types in the transmission phase: the request, the simple reply, and the structured reply chunk. The transmission phase consists of a series of transactions, where the client submits requests and the server sends corresponding replies with either a single simple reply or a series of one or more structured reply chunks per request. The phase continues until either side terminates transmission; this can be performed cleanly only by the client.

Note that without client negotiation, the server MUST use only simple replies, and that it is impossible to tell by reading the server traffic in isolation whether a data field will be present; the simple reply is also problematic for error handling of the `NBD_CMD_READ` request. Therefore, structured replies can be used to create a a context-free server stream; see below.

Replies need not be sent in the same order as requests (i.e., requests may be handled by the server asynchronously), and structured reply chunks from one request may be interleaved with reply messages from other requests; however, there may be constraints that prevent arbitrary reordering of structured reply chunks within a given reply. Clients SHOULD use a handle that is distinct from all other currently pending transactions, but MAY reuse handles that are no longer in flight; handles need not be consecutive. In each reply message (whether simple or structured), the server MUST use the same value for handle as was sent by the client in the corresponding request. In this way, the client can correlate which request is receiving a response.

Ordering of messages and writes

The server MAY process commands out of order, and MAY reply out of order, except that:

- All write commands (that includes `NBD_CMD_WRITE`, `NBD_CMD_WRITE_ZEROES` and `NBD_CMD_TRIM`) that the server completes (i.e. replies to) prior to processing a `NBD_CMD_FLUSH` MUST be written to non-volatile storage prior to replying to that `NBD_CMD_FLUSH`. This paragraph only applies if `NBD_FLAG_SEND_FLUSH` is set within the transmission flags, as otherwise `NBD_CMD_FLUSH` will never be sent by the client to the server.
- A client which uses multiple connections to a server to parallelize commands MUST NOT issue an `NBD_CMD_FLUSH` request until it has received the reply for all write commands which it expects to be covered by the flush.

- A server MUST NOT reply to a command that has `NBD_CMD_FLAG_FUA` set in its command flags until the data (if any) written by that command is persisted to non-volatile storage. This only applies if `NBD_FLAG_SEND_FUA` is set within the transmission flags, as otherwise `NBD_CMD_FLAG_FUA` will not be set on any commands sent to the server by the client.

`NBD_CMD_FLUSH` is modelled on the Linux kernel empty bio with `REQ_PREFLUSH` set. `NBD_CMD_FLAG_FUA` is modelled on the Linux kernel bio with `REQ_FUA` set. In case of ambiguity in this specification, the kernel documentation may be useful.

Request message

The request message, sent by the client, looks as follows:

C: 32 bits, 0×25609513, magic (`NBD_REQUEST_MAGIC`)
C: 16 bits, command flags
C: 16 bits, type
C: 64 bits, handle
C: 64 bits, offset (unsigned)
C: 32 bits, length (unsigned)
C: (*length* bytes of data if the request is of type `NBD_CMD_WRITE`)

Simple reply message

The simple reply message MUST be sent by the server in response to all requests if structured replies have not been negotiated using `NBD_OPT_STRUCTURED_REPLY`. If structured replies have been negotiated, a simple reply MAY be used as a reply to any request other than `NBD_CMD_READ`, but only if the reply has no data payload. The message looks as follows:

S: 32 bits, 0×67446698, magic (`NBD_SIMPLE_REPLY_MAGIC`; used to be `NBD_REPLY_MAGIC`)
S: 32 bits, error (MAY be zero)
S: 64 bits, handle
S: (*length* bytes of data if the request is of type `NBD_CMD_READ` and *error* is zero)

Structured reply chunk message

Some of the major downsides of the default simple reply to `NBD_CMD_READ` are as follows. First, it is not possible to support partial reads or early errors (the command must succeed or fail as a whole, and either *length* bytes of data must be sent or a hard disconnect must be initiated, even if the failure is `NBD_EINVAL` due to bad flags). Second, there is no way to efficiently skip over

portions of a sparse file that are known to contain all zeroes. Finally, it is not possible to reliably decode the server traffic without also having context of what pending read requests were sent by the client. Therefore structured replies are also permitted if negotiated.

A structured reply in the transmission phase consists of one or more structured reply chunk messages. The server MUST NOT send this reply type unless the client has successfully negotiated structured replies via `NBD_OPT_STRUCTURED_REPLY`. Conversely, if structured replies are negotiated, the server MUST use a structured reply for any response with a payload, and MUST NOT use a simple reply for `NBD_CMD_READ` (even for the case of an early `NBD_EINVAL` due to bad flags), but MAY use either a simple reply or a structured reply to all other requests. The server SHOULD prefer sending errors via a structured reply, as the error can then be accompanied by a string payload to present to a human user.

A structured reply MAY occupy multiple structured chunk messages (all with the same value for "handle"), and the `NBD_REPLY_FLAG_DONE` reply flag is used to identify the final chunk. Unless further documented by individual requests below, the chunks MAY be sent in any order, except that the chunk with the flag `NBD_REPLY_FLAG_DONE` MUST be sent last. Even when a command documents further constraints between chunks of one reply, it is always safe to interleave chunks of that reply with messages related to other requests. A server SHOULD try to minimize the number of chunks sent in a reply, but MUST NOT mark a chunk as final if there is still a possibility of detecting an error before transmission of that chunk completes. A structured reply is considered successful only if it did not contain any error chunks, although the client MAY be able to determine partial success based on the chunks received.

A structured reply chunk message looks as follows:

S: 32 bits, 0×668e33ef, magic (`NBD_STRUCTURED_REPLY_MAGIC`)
S: 16 bits, flags
S: 16 bits, type
S: 64 bits, handle
S: 32 bits, length of payload (unsigned)
S: *length* bytes of payload data (if *length* is nonzero)

The use of *length* in the reply allows context-free division of the overall server traffic into individual reply messages; the *type* field describes how to further interpret the payload.

Terminating the transmission phase

There are two methods of terminating the transmission phase:

- The client sends `NBD_CMD_DISC` whereupon the server MUST close down the TLS session (if one is running) and then close the TCP connection. This is referred to as 'initiating a soft disconnect'. Soft disconnects can only be initiated by the client.
- The client or the server drops the TCP session (in which case it SHOULD shut down the TLS session first). This is referred to as 'initiating a hard disconnect'.

Together these are referred to as 'terminating transmission'.

Either side MAY initiate a hard disconnect if it detects a violation by the other party of a mandatory condition within this document.

On a server shutdown, the server SHOULD wait for inflight requests to be serviced prior to initiating a hard disconnect. A server MAY speed this process up by issuing error replies. The error value issued in respect of these requests and any subsequently received requests SHOULD be `NBD_ESHUTDOWN`.

If the client receives an `NBD_ESHUTDOWN` error it MUST initiate a soft disconnect.

The client MAY issue a soft disconnect at any time, but SHOULD wait until there are no inflight requests first.

The client and the server MUST NOT initiate any form of disconnect other than in one of the above circumstances.

TLS support

The NBD protocol supports Transport Layer Security (TLS) (see RFC5246 as updated by RFC6176).

TLS is negotiated with the `NBD_OPT_STARTTLS` option. This is performed as an in-session upgrade. Below the term 'negotiation' is used to refer to the sending and receiving of NBD options and option replies, and the term 'initiation' of TLS is used to refer to the actual upgrade to TLS.

Certificates, authentication and authorisation

This standard does not specify what encryption, certification and signature algorithms are used. This standard does not specify authentication and

authorisation (for instance whether client and/or server certificates are required and what they should contain); this is implementation dependent.

TLS requires fixed newstyle negotiation to have completed.

Server-side requirements

There are three modes of operation for a server. The server MUST support one of these modes.

- The server operates entirely without TLS ('NOTLS'); OR
- The server insists upon TLS, and forces the client to upgrade by erroring any NBD options other than `NBD_OPT_STARTTLS` or `NBD_OPT_ABORT` with `NBD_REP_ERR_TLS_REQD` ('FORCEDTLS'); this in practice means that all option negotiation (apart from the `NBD_OPT_STARTTLS` itself) is carried out with TLS; OR
- The server provides TLS, and it is mandatory on zero or more exports, and is available at the client's option on all other exports ('SELECTIVETLS'). The server does not force the client to upgrade to TLS during option haggling (as if the client ultimately were to choose a non-TLS-only export, stopping TLS is not possible). Instead it permits the client to upgrade as and when it chooses, but unless an upgrade to TLS has already taken place, the server errors attempts to enter transmission mode on TLS-only exports, MAY refuse to provide information about TLS-only exports via `NBD_OPT_INFO`, MAY refuse to provide information about non-existent exports via `NBD_OPT_INFO`, and MAY omit exports that are TLS-only from `NBD_OPT_LIST`.

The server MAY determine the mode in which it operates dependent upon the session (for instance it might be more liberal with TCP connections made over the loopback interface) but it MUST be consistent in its mode of operation across the lifespan of a single TCP connection to the server. A client MUST NOT assume indications from a prior TCP session to a given server will be relevant to a subsequent session.

The server MUST operate in NOTLS mode unless the server set flag `NBD_FLAG_FIXED_NEWSTYLE` and the client replied with `NBD_FLAG_C_FIXED_NEWSTYLE` in the fixed newstyle negotiation.

These modes of operations are described in detail below.

NOTLS mode

If the server receives `NBD_OPT_STARTTLS` it MUST respond with `NBD_REP_ERR_POLICY` (if it does not support TLS for policy reasons), `NBD_REP_ERR_UNSUP` (if it does not support the `NBD_OPT_STARTTLS` option at all) or another error explicitly permitted by this document. The server MUST NOT respond to any option request with `NBD_REP_ERR_TLS_REQD`.

FORCEDTLS mode

If the server receives `NBD_OPT_STARTTLS` prior to negotiating TLS, it MUST reply with `NBD_REP_ACK`. If the server receives `NBD_OPT_STARTTLS` when TLS has already been negotiated, it it MUST reply with `NBD_REP_ERR_INVALID`.

After an `NBD_REP_ACK` reply has been sent, the server MUST be prepared for a TLS handshake, and all further data MUST be sent and received over TLS. There is no downgrade to a non-TLS session.

As per the TLS standard, the handshake MAY be initiated either by the server (having sent the `NBD_REP_ACK`) or by the client. If the handshake is unsuccessful (for instance the client's certificate does not match) the server MUST terminate the session as by this stage it is too late to continue without TLS as the acknowledgement has been sent.

If the server receives any other option, including `NBD_OPT_INFO` and unsupported options, it MUST reply with `NBD_REP_ERR_TLS_REQD` if TLS has not been initiated; `NBD_OPT_INFO` is included as in this mode, all exports are TLS-only. If the server receives a request to enter transmission mode via `NBD_OPT_EXPORT_NAME` when TLS has not been initiated, then as this request cannot error, it MUST terminate the session. If the server receives a request to enter transmission mode via `NBD_OPT_GO` when TLS has not been initiated, it MUST error with `NBD_REP_ERR_TLS_REQD`.

The server MUST NOT send `NBD_REP_ERR_TLS_REQD` in reply to any option if TLS has already been initiated.

The FORCEDTLS mode of operation has an implementation problem in that the client MAY legally simply send a `NBD_OPT_EXPORT_NAME` to enter transmission mode without previously sending any options. This is avoided by use of `NBD_OPT_INFO` and `NBD_OPT_GO`.

SELECTIVETLS mode

If the server receives `NBD_OPT_STARTTLS` prior to negotiating TLS, it MUST reply with `NBD_REP_ACK` and initiate TLS as set out under 'FORCEDTLS' above. If the

server receives `NBD_OPT_STARTTLS` when TLS has already been negotiated, it it MUST reply with `NBD_REP_ERR_INVALID`.

If the server receives `NBD_OPT_INFO` or `NBD_OPT_GO` and TLS has not been initiated, it MAY reply with `NBD_REP_ERR_TLS_REQD` if that export is non-existent, and MUST reply with `NBD_REP_ERR_TLS_REQD` if that export is TLS-only.

If the server receives a request to enter transmission mode via `NBD_OPT_EXPORT_NAME` on a TLS-only export when TLS has not been initiated, then as this request cannot error, it MUST terminate the session.

The server MUST NOT send `NBD_REP_ERR_TLS_REQD` in reply to any option if TLS has already been negotiated. The server MUST NOT send `NBD_REP_ERR_TLS_REQD` in response to any option other than `NBD_OPT_INFO`, `NBD_OPT_GO` and `NBD_OPT_EXPORT_NAME`, and only in those cases in respect of a TLS-only or non-existent export.

There is a degenerate case of SELECTIVETLS where all exports are TLS-only. This is permitted in part to make programming of servers easier. Operation is a little different from FORCEDTLS, as the client is not forced to upgrade to TLS prior to any options being processed, and the server MAY choose to give information on non-existent exports via `NBD_OPT_INFO` responses prior to an upgrade to TLS.

Client-side requirements

If the client supports TLS at all, it MUST be prepared to deal with servers operating in any of the above modes. Notwithstanding, a client MAY always terminate the session or refuse to connect to a particular export if TLS is not available and the user requires TLS.

The client MUST NOT issue `NBD_OPT_STARTTLS` unless the server set flag `NBD_FLAG_FIXED_NEWSTYLE` and the client replied with `NBD_FLAG_C_FIXED_NEWSTYLE` in the fixed newstyle negotiation.

The client MUST NOT issue `NBD_OPT_STARTTLS` if TLS has already been initiated.

Subject to the above two limitations, the client MAY send `NBD_OPT_STARTTLS` at any time to initiate a TLS session. If the client receives `NBD_REP_ACK` in response, it MUST immediately upgrade the session to TLS. If it receives `NBD_REP_ERR_UNSUP`, `NBD_REP_ERR_POLICY` or any other error in response, it indicates that the server cannot or will not upgrade the session to TLS, and therefore the client MUST either continue the session without TLS, or terminate the session.

A client that prefers to use TLS irrespective of whether the server makes TLS mandatory SHOULD send `NBD_OPT_STARTTLS` as the first option. This will ensure option haggling is subject to TLS, and will thus prevent the possibility of options being compromised by a Man-in-the-Middle attack. Note that the `NBD_OPT_STARTTLS` itself may be compromised - see 'downgrade attacks' for more details. For this reason, a client which only wishes to use TLS SHOULD terminate the session if the `NBD_OPT_STARTTLS` replies with an error.

If the TLS handshake is unsuccessful (for instance the server's certificate does not validate) the client MUST terminate the session as by this stage it is too late to continue without TLS.

If the client receives an `NBD_REP_ERR_TLS_REQD` in response to any option, it implies that this option cannot be executed unless a TLS upgrade is performed. If the option is any option other than `NBD_OPT_INFO` or `NBD_OPT_GO`, this indicates that no option will succeed unless a TLS upgrade is performed; the client MAY therefore choose to issue an `NBD_OPT_STARTTLS`, or MAY terminate the session (if for instance it does not support TLS or does not have appropriate credentials for this server). If the client receives `NBD_REP_ERR_TLS_REQD` in response to `NBD_OPT_INFO` or `NBD_OPT_GO` this indicates that the export referred to within the option is either non-existent or requires TLS; the client MAY therefore choose to issue an `NBD_OPT_STARTTLS`, MAY terminate the session (if for instance it does not support TLS or does not have appropriate credentials for this server), or MAY continue in another manner without TLS, for instance by querying or using other exports.

If a client supports TLS, it SHOULD use `NBD_OPT_GO` (if the server supports it) in place of `NBD_OPT_EXPORT_NAME`. The reason for this is set out in the final paragraphs of the sections under 'FORCEDTLS' and 'SELECTIVETLS': this gives an opportunity for the server to transmit that an error going into transmission mode is due to the client's failure to initiate TLS, and the fact that the client may obtain information about which exports are TLS-only through `NBD_OPT_INFO`.

Security considerations

TLS versions

NBD implementations supporting TLS MUST support TLS version 1.2, SHOULD support any later versions. NBD implementations MAY support older versions but SHOULD NOT do so by default (i.e. they SHOULD only be available by a

configuration change). Older versions SHOULD NOT be used where there is a risk of security problems with those older versions or of a downgrade attack against TLS versions.

Protocol downgrade attacks

A danger inherent in any scheme relying on the negotiation of whether TLS should be employed is downgrade attacks within the NBD protocol.

There are two main dangers:

- A Man-in-the-Middle (MitM) hijacks a session and impersonates the server (possibly by proxying it) claiming not to support TLS. In this manner, the client is confused into operating in a plain-text manner with the MitM (with the session possibly being proxied in plain-text to the server using the method below).
- The MitM hijacks a session and impersonates the client (possibly by proxying it) claiming not to support TLS. In this manner the server is confused into operating in a plain-text manner with the MitM (with the session being possibly proxied to the client with the method above).

With regard to the first, any client that does not wish to be subject to potential downgrade attack SHOULD ensure that if a TLS endpoint is specified by the client, it ensures that TLS is negotiated prior to sending or requesting sensitive data. To recap, the client MAY send `NBD_OPT_STARTTLS` at any point during option haggling, and MAY terminate the session if `NBD_REP_ACK` is not provided.

With regard to the second, any server that does not wish to be subject to a potential downgrade attack SHOULD either used FORCEDTLS mode, or should force TLS on those exports it is concerned about using SELECTIVE mode and TLS-only exports. It is not possible to avoid downgrade attacks on exports which may be served either via TLS or in plain text unless the client insists on TLS.

Block size constraints

During transmission phase, several operations are constrained by the export size sent by the final `NBD_OPT_EXPORT_NAME` or `NBD_OPT_GO`, as well as by three block size constraints defined here (minimum, preferred, and maximum).

If a client can honour server block size constraints (as set out below and under `NBD_INFO_BLOCK_SIZE`), it SHOULD announce this during the handshake phase by

using `NBD_OPT_GO` (and `NBD_OPT_INFO` if used) with an `NBD_INFO_BLOCK_SIZE` information request, and MUST use `NBD_OPT_GO` rather than `NBD_OPT_EXPORT_NAME` (except in the case of a fallback where the server did not support `NBD_OPT_INFO` or `NBD_OPT_GO`).

A server with block size constraints other than the default SHOULD advertise the block size constraints during handshake phase via `NBD_INFO_BLOCK_SIZE` in response to `NBD_OPT_INFO` or `NBD_OPT_GO`, and MUST do so unless it has agreed on block size constraints via out of band means.

Some servers are able to make optimizations, such as opening files with `O_DIRECT`, if they know that the client will obey a particular minimum block size, where it must fall back to safer but slower code if the client might send unaligned requests. For that reason, if a client issues an `NBD_OPT_GO` including an `NBD_INFO_BLOCK_SIZE` information request, it MUST abide by the block size constraints it receives. Clients MAY issue `NBD_OPT_INFO` with `NBD_INFO_BLOCK_SIZE` to learn the server's constraints without committing to them.

If block size constraints have not been advertised or agreed on externally, then a server SHOULD support a default minimum block size of 1, a preferred block size of 2^12 (4,096), and a maximum block size that is effectively unlimited (0xffffffff, or the export size if that is smaller), while a client desiring maximum interoperability SHOULD constrain its requests to a minimum block size of 2^9 (512), and limit `NBD_CMD_READ` and `NBD_CMD_WRITE` commands to a maximum block size of 2^25 (33,554,432). A server that wants to enforce block sizes other than the defaults specified here MAY refuse to go into transmission phase with a client that uses `NBD_OPT_EXPORT_NAME` (via a hard disconnect) or which uses `NBD_OPT_GO` without requesting `NBD_INFO_BLOCK_SIZE` (via an error reply of `NBD_REP_ERR_BLOCK_SIZE_REQD`); but servers SHOULD NOT refuse clients that do not request sizing information when the server supports default sizing or where sizing constraints can be agreed on externally. When allowing clients that did not negotiate sizing via NBD, a server that enforces stricter block size constraints than the defaults MUST cleanly error commands that fall outside the constraints without corrupting data; even so, enforcing constraints in this manner may limit interoperability.

A client MAY choose to operate as if tighter block size constraints had been specified (for example, even when the server advertises the default minimum block size of 1, a client may safely use a minimum block size of 2^9 (512)).

The minimum block size represents the smallest addressable length and alignment within the export, although writing to an area that small may require

the server to use a less-efficient read-modify-write action. If advertised, this value MUST be a power of 2, MUST NOT be larger than 2^16 (65,536), and MAY be as small as 1 for an export backed by a regular file, although the values of 2^9 (512) or 2^12 (4,096) are more typical for an export backed by a block device. If a server advertises a minimum block size, the advertised export size SHOULD be an integer multiple of that block size, since otherwise, the client would be unable to access the final few bytes of the export.

The preferred block size represents the minimum size at which aligned requests will have efficient I/O, avoiding behaviour such as read-modify-write. If advertised, this MUST be a power of 2 at least as large as the maximum of the minimum block size and 2^9 (512), although larger values (such as 4,096, or even the minimum granularity of a hole) are more typical. The preferred block size MAY be larger than the export size, in which case the client is unable to utilize the preferred block size for that export. The server MAY advertise an export size that is not an integer multiple of the preferred block size.

The maximum block size represents the maximum length that the server is willing to handle in one request. If advertised, it MAY be something other than a power of 2, but MUST be either an integer multiple of the minimum block size or the value 0xffffffff for no inherent limit, MUST be at least as large as the smaller of the preferred block size or export size, and SHOULD be at least 2^20 (1,048,576) if the export is that large. For convenience, the server MAY advertise a maximum block size that is larger than the export size, although in that case, the client MUST treat the export size as the effective maximum block size (as further constrained by a nonzero offset).

Where a transmission request can have a nonzero *offset* and/or *length* (such as `NBD_CMD_READ`, `NBD_CMD_WRITE`, or `NBD_CMD_TRIM`), the client MUST ensure that *offset* and *length* are integer multiples of any advertised minimum block size, and SHOULD use integer multiples of any advertised preferred block size where possible. For those requests, the client MUST NOT use a *length* which, when added to *offset*, would exceed the export size. Also for NBD_CMD_READ, NBD_CMD_WRITE, NBD_CMD_CACHE and NBD_CMD_WRITE_ZEROES (except for when NBD_CMD_FLAG_FAST_ZERO is set), the client MUST NOT use a *length* larger than any advertised maximum block size. The server SHOULD report an `NBD_EINVAL` error if the client's request is not aligned to advertised minimum block size boundaries, or is larger than the advertised maximum block size. Notwithstanding any maximum

block size advertised, either the server or the client MAY initiate a hard disconnect if the payload of an `NBD_CMD_WRITE` request or `NBD_CMD_READ` reply would be large enough to be deemed a denial of service attack; however, for maximum portability, any *length* less than 2^25 (33,554,432) bytes SHOULD NOT be considered a denial of service attack (even if the advertised maximum block size is smaller). For all other commands, where the *length* is not reflected in the payload (such as `NBD_CMD_TRIM` or `NBD_CMD_WRITE_ZEROES`), a server SHOULD merely fail the command with an `NBD_EINVAL` error for a client that exceeds the maximum block size, rather than initiating a hard disconnect.

Metadata querying

It is often helpful for the client to be able to query the status of a range of blocks. The nature of the status that can be queried is in part implementation dependent. For instance, the status might represent:

- in a sparse storage format, whether the relevant blocks are actually present on the backing device for the export; or
- whether the relevant blocks are 'dirty'; some storage formats and operations over such formats express a concept of data dirtiness. Whether the operation is block device mirroring, incremental block device backup or any other operation with a concept of data dirtiness, they all share a need to provide a list of ranges that this particular operation treats as dirty.

To provide such classes of information, the NBD protocol has a generic framework for querying metadata; however, its use must first be negotiated, and one or more metadata contexts must be selected.

The procedure works as follows:

- First, during negotiation, if the client wishes to query metadata during transmission, the client MUST select one or more metadata contexts with the `NBD_OPT_SET_META_CONTEXT` command. If needed, the client can use `NBD_OPT_LIST_META_CONTEXT` to list contexts that the server supports.
- During transmission, a client can then indicate interest in metadata for a given region by way of the `NBD_CMD_BLOCK_STATUS` command, where *offset* and *length* indicate the area of interest. The server MUST then respond with the requested information, for all contexts which were selected during negotiation. For every metadata context, the server sends one set of

extent chunks, where the sizes of the extents MUST be less than or equal to the length as specified in the request. Each extent comes with a *flags* field, the semantics of which are defined by the metadata context.

- A server MUST reply to `NBD_CMD_BLOCK_STATUS` with a structured reply of type `NBD_REPLY_TYPE_BLOCK_STATUS`.

A client MUST NOT use `NBD_CMD_BLOCK_STATUS` unless it selected a nonzero number of metadata contexts during negotiation, and used the same export name for the subsequent `NBD_OPT_GO` (or `NBD_OPT_EXPORT_NAME`). Servers SHOULD reply with `NBD_EINVAL` to clients sending `NBD_CMD_BLOCK_STATUS` without selecting at least one metadata context.

The reply to the `NBD_CMD_BLOCK_STATUS` request MUST be sent as a structured reply; this implies that in order to use metadata querying, structured replies MUST be negotiated first.

Metadata contexts are identified by their names. The name MUST consist of a namespace, followed by a colon, followed by a leaf-name. The namespace must consist entirely of printable non-whitespace UTF-8 characters other than colons, and be non-empty. The entire name (namespace, colon, and leaf-name) MUST follow the restrictions for strings as laid out earlier in this document.

Namespaces MUST be consist of one of the following: - `base`, for metadata contexts defined by this document; - `nbd-server`, for metadata contexts defined by the implementation that accompanies this document (none currently); - `x-*`, where `*` can be replaced by an arbitrary string not containing colons, for local experiments. This SHOULD NOT be used by metadata contexts that are expected to be widely used. - A third-party namespace from the list below.

Third-party implementations can register additional namespaces by simple request to the mailing-list. The following additional third-party namespaces are currently registered: * `qemu`, maintained by qemu.org

Save in respect of the `base:` namespace described below, this specification requires no specific semantics of metadata contexts, except that all the information they provide MUST be representable within the flags field as defined for `NBD_REPLY_TYPE_BLOCK_STATUS`. Likewise, save in respect of the `base:` namespace, the syntax of query strings is not specified by this document, other than the recommendation that the empty leaf-name makes sense as a wildcard for a client query during `NBD_OPT_LIST_META_CONTEXT`, but SHOULD NOT select any contexts during `NBD_OPT_SET_META_CONTEXT`.

Server implementations SHOULD ensure the syntax for query strings they support and semantics for resulting metadata context is documented similarly to this document.

The `base:` metadata namespace

This standard defines exactly one metadata context; it is called `base:allocation`, and it provides information on the basic allocation status of extents (that is, whether they are allocated at all in a sparse file context).

The query string within the `base:` metadata context can take one of two forms:

- `base:` - the server MUST ignore this form during `NBD_OPT_SET_META_CONTEXT`, and MUST support this as a wildcard during `NBD_OPT_LIST_META_CONTEXT`, in which case the server's reply will contain a response for each supported metadata context within the `base:` namespace (currently just `base:allocation`, although a future revision of the standard might return multiple contexts); or
- `base:[leaf-name]` to select `[leaf-name]` as a context leaf-name that might exist within the `base` namespace. If a `[leaf-name]` requested by the client is not recognized, the server MUST ignore it rather than report an error.

`base:allocation` metadata context

The `base:allocation` metadata context is the basic "allocated at all" metadata context. If an extent is marked with `NBD_STATE_HOLE` at that context, this means that the given extent is not allocated in the backend storage, and that writing to the extent MAY result in the `NBD_ENOSPC` error. This supports sparse file semantics on the server side. If a server supports the `base:allocation` metadata context, then writing to an extent which has `NBD_STATE_HOLE` clear MUST NOT fail with `NBD_ENOSPC` unless for reasons specified in the definition of another context.

It defines the following flags for the flags field:

- `NBD_STATE_HOLE` (bit 0): if set, the block represents a hole (and future writes to that area may cause fragmentation or encounter an `NBD_ENOSPC` error); if clear, the block is allocated or the server could not otherwise determine its status. Note that the use of `NBD_CMD_TRIM` is related to this status, but that the server MAY report a hole even where `NBD_CMD_TRIM` has not been

requested, and also that a server MAY report that the block is allocated even where `NBD_CMD_TRIM` has been requested.

- `NBD_STATE_ZERO` (bit 1): if set, the block contents read as all zeroes; if clear, the block contents are not known. Note that the use of `NBD_CMD_WRITE_ZEROES` is related to this status, but that the server MAY report zeroes even where `NBD_CMD_WRITE_ZEROES` has not been requested, and also that a server MAY report unknown content even where `NBD_CMD_WRITE_ZEROES` has been requested.

It is not an error for a server to report that a region of the export has both `NBD_STATE_HOLE` set and `NBD_STATE_ZERO` clear. The contents of such an area are undefined, and a client reading such an area should make no assumption as to its contents or stability.

For the `base:allocation` context, the remainder of the flags field is reserved. Servers SHOULD set it to all-zero; clients MUST ignore unknown flags.

Values

This section describes the value and meaning of constants (other than magic numbers) in the protocol.

When flags fields are specified, they are numbered in network byte order.

Handshake phase

Flag fields

Handshake flags

This field of 16 bits is sent by the server after the `INIT_PASSWD` and the first magic number.

- bit 0, `NBD_FLAG_FIXED_NEWSTYLE`; MUST be set by servers that support the fixed newstyle protocol
- bit 1, `NBD_FLAG_NO_ZEROES`; if set, and if the client replies with `NBD_FLAG_C_NO_ZEROES` in the client flags field, the server MUST NOT send the 124 bytes of zero at the end of the negotiation.

The server MUST NOT set any other flags, and SHOULD NOT change behaviour unless the client responds with a corresponding flag. The server MUST NOT set any of these flags during oldstyle negotiation.

Client flags

This field of 32 bits is sent after initial connection and after receiving the handshake flags from the server.

- bit 0, `NBD_FLAG_C_FIXED_NEWSTYLE`; SHOULD be set by clients that support the fixed newstyle protocol. Servers MAY choose to honour fixed newstyle from clients that didn't set this bit, but relying on this isn't recommended.
- bit 1, `NBD_FLAG_C_NO_ZEROES`; MUST NOT be set if the server did not set `NBD_FLAG_NO_ZEROES`. If set, the server MUST NOT send the 124 bytes of zeroes at the end of the negotiation.

Clients MUST NOT set any other flags; the server MUST drop the TCP connection if the client sets an unknown flag, or a flag that does not match something advertised by the server.

Transmission flags

This field of 16 bits is sent by the server after option haggling, or immediately after the handshake flags field in oldstyle negotiation.

Many of these flags allow the server to expose to the client which features it understands (in which case they are documented below as "`NBD_FLAG_XXX` exposes feature `YYY`"). In each case, the server MAY set the flag for features it supports. The server MUST NOT set the flag for features it does not support. The client MUST NOT use a feature documented as 'exposed' by a flag unless that flag was set.

The field has the following format:

- bit 0, `NBD_FLAG_HAS_FLAGS`: MUST always be 1.
- bit 1, `NBD_FLAG_READ_ONLY`: The server MAY set this flag to indicate to the client that the export is read-only (exports might be read-only in a manner undetectable to the server, for instance because of permissions). If this flag is set, the server MUST error subsequent write operations to the export.
- bit 2, `NBD_FLAG_SEND_FLUSH`: exposes support for `NBD_CMD_FLUSH`.
- bit 3, `NBD_FLAG_SEND_FUA`: exposes support for `NBD_CMD_FLAG_FUA`.
- bit 4, `NBD_FLAG_ROTATIONAL`: the server MAY set this flag to 1 to inform the client that the export has the characteristics of a rotational medium, and

the client MAY schedule I/O accesses in a manner corresponding to the setting of this flag.

- bit 5, `NBD_FLAG_SEND_TRIM`: exposes support for `NBD_CMD_TRIM`.
- bit 6, `NBD_FLAG_SEND_WRITE_ZEROES`: exposes support for `NBD_CMD_WRITE_ZEROES` and `NBD_CMD_FLAG_NO_HOLE`.
- bit 7, `NBD_FLAG_SEND_DF`: do not fragment a structured reply. The server MUST set this transmission flag to 1 if the `NBD_CMD_READ` request supports the `NBD_CMD_FLAG_DF` flag, and MUST leave this flag clear if structured replies have not been negotiated. Clients MUST NOT set the `NBD_CMD_FLAG_DF` request flag unless this transmission flag is set.
- bit 8, `NBD_FLAG_CAN_MULTI_CONN`: Indicates that the server operates entirely without cache, or that the cache it uses is shared among all connections to the given device. In particular, if this flag is present, then the effects of `NBD_CMD_FLUSH` and `NBD_CMD_FLAG_FUA` MUST be visible across all connections when the server sends its reply to that command to the client. In the absence of this flag, clients SHOULD NOT multiplex their commands over more than one connection to the export.
- bit 9, `NBD_FLAG_SEND_RESIZE`: defined by the experimental `RESIZE` extension.
- bit 10, `NBD_FLAG_SEND_CACHE`: documents that the server understands `NBD_CMD_CACHE`; however, note that server implementations exist which support the command without advertising this bit, and conversely that this bit does not guarantee that the command will succeed or have an impact.
- bit 11, `NBD_FLAG_SEND_FAST_ZERO`: allow clients to detect whether `NBD_CMD_WRITE_ZEROES` is faster than a corresponding write. The server MUST set this transmission flag to 1 if the `NBD_CMD_WRITE_ZEROES` request supports the `NBD_CMD_FLAG_FAST_ZERO` flag, and MUST set this transmission flag to 0 if `NBD_FLAG_SEND_WRITE_ZEROES` is not set. Servers MAY set this this transmission flag even if it will always use `NBD_ENOTSUP` failures for requests with `NBD_CMD_FLAG_FAST_ZERO` set (such as if the server cannot quickly determine whether a particular write zeroes request will be faster than a regular write). Clients MUST NOT set the `NBD_CMD_FLAG_FAST_ZERO` request flag unless this transmission flag is set.

Clients SHOULD ignore unknown flags.

Option types

These values are used in the “option” field during the option haggling of the newstyle negotiation.

- `NBD_OPT_EXPORT_NAME` (1)

  Choose the export which the client would like to use, end option haggling, and proceed to the transmission phase.

  Data: String, name of the export, as free-form text. The length of the name is determined from the option header. If the chosen export does not exist or requirements for the chosen export are not met (e.g., the client did not initiate TLS for an export where the server requires it), the server MUST terminate the session.

  A special, “empty”, name (i.e., the length field is zero and no name is specified), is reserved for a “default” export, to be used in cases where explicitly specifying an export name makes no sense.

  This is the only valid option in nonfixed newstyle negotiation. A server which wishes to use any other option MUST support fixed newstyle.

  A major problem of this option is that it does not support the return of error messages to the client in case of problems. To remedy this, `NBD_OPT_GO` has been introduced (see below). A client thus SHOULD use `NBD_OPT_GO` in preference to `NBD_OPT_EXPORT_NAME` but SHOULD fall back to `NBD_OPT_EXPORT_NAME` if `NBD_OPT_GO` is not supported (not falling back will prevent it from connecting to old servers).

- `NBD_OPT_ABORT` (2)

  The client desires to abort the negotiation and terminate the session. The server MUST reply with `NBD_REP_ACK`.

  The client SHOULD NOT send any additional data with the option; however, a server SHOULD ignore any data sent by the client rather than rejecting the request as invalid.

  Previous versions of this document were unclear on whether the server should send a reply to `NBD_OPT_ABORT`. Therefore the client SHOULD gracefully handle the server closing the connection after receiving an `NBD_OPT_ABORT` without it sending a reply. Similarly the server SHOULD gracefully handle the client sending an `NBD_OPT_ABORT` and closing the connection without waiting for a reply.

- `NBD_OPT_LIST` (3)

Return zero or more `NBD_REP_SERVER` replies, one for each export, followed by `NBD_REP_ACK` or an error (such as `NBD_REP_ERR_SHUTDOWN`). The server MAY omit entries from this list if TLS has not been negotiated, the server is operating in SELECTIVETLS mode, and the entry concerned is a TLS-only export.

The client MUST NOT send any additional data with the option, and the server SHOULD reject a request that includes data with `NBD_REP_ERR_INVALID`.

- `NBD_OPT_PEEK_EXPORT` (4)

  Was defined by the (withdrawn) experimental `PEEK_EXPORT` extension; not in use.

- `NBD_OPT_STARTTLS` (5)

  The client wishes to initiate TLS.

  The client MUST NOT send any additional data with the option. The server MUST either reply with `NBD_REP_ACK` after which point the connection is upgraded to TLS, or an error reply explicitly permitted by this document (for example, `NBD_REP_ERR_INVALID` if the client included data).

  See the section on TLS above for further details.

- `NBD_OPT_INFO` (6) and `NBD_OPT_GO` (7)

  Both options have identical formats for requests and replies. The only difference is that after a successful reply to `NBD_OPT_GO` (i.e. one or more `NBD_REP_INFO` then an `NBD_REP_ACK`), transmission mode is entered immediately. Therefore these commands share common documentation.

  `NBD_OPT_INFO`: The client wishes to get details about an export with the given name for use in the transmission phase, but does not yet want to move to the transmission phase. When successful, this option provides more details than `NBD_OPT_LIST`, but only for a single export name.

  `NBD_OPT_GO`: The client wishes to terminate the handshake phase and progress to the transmission phase. This client MAY issue this command after an `NBD_OPT_INFO`, or MAY issue it without a previous `NBD_OPT_INFO`. `NBD_OPT_GO` can thus be used as an improved version of `NBD_OPT_EXPORT_NAME` that is capable of returning errors.

  Data (both commands):

  - 32 bits, length of name (unsigned); MUST be no larger than the option data length - 6

- String: name of the export
- 16 bits, number of information requests
- 16 bits x n - list of `NBD_INFO` information requests

The client MAY list one or more items of specific information it is seeking in the list of information requests, or it MAY specify an empty list. The client MUST NOT include any information request in the list more than once. The server MUST ignore any information requests it does not understand. The server MAY reply to the information requests in any order. The server MAY ignore information requests that it does not wish to supply for policy reasons (other than `NBD_INFO_EXPORT`). Equally the client MAY refuse to negotiate if not supplied information it has requested. The server MAY send information requests back which are not explicitly requested, but the server MUST NOT assume that such information requests are understood and respected by the client unless the client explicitly asked for them. The client MUST ignore information replies it does not understand.

If no name is specified (i.e. a zero length string is provided), this specifies the default export (if any), as with `NBD_OPT_EXPORT_NAME`.

The server replies with a number of `NBD_REP_INFO` replies (as few as zero if an error is reported, at least one on success), then concludes the list of information with a final error reply or with a declaration of success, as follows:

- `NBD_REP_ACK`: The server accepts the chosen export, and has completed providing information. In this case, the server MUST send at least one `NBD_REP_INFO`, with an `NBD_INFO_EXPORT` information type.
- `NBD_REP_ERR_UNKNOWN`: The chosen export does not exist on this server. In this case, the server SHOULD NOT send `NBD_REP_INFO` replies.
- `NBD_REP_ERR_TLS_REQD`: The server requires the client to initiate TLS before any revealing any further details about this export. In this case, a FORCEDTLS server MUST NOT send `NBD_REP_INFO` replies, but a SELECTIVETLS server MAY do so if this is a TLS-only export.
- `NBD_REP_ERR_BLOCK_SIZE_REQD`: The server requires the client to request block size constraints using `NBD_INFO_BLOCK_SIZE` prior to entering transmission phase, because the server will be using non-default block sizes constraints. The server MUST NOT send this error if block size

constraints were requested with `NBD_INFO_BLOCK_SIZE` with the `NBD_OPT_INFO` or `NBD_OPT_GO` request. The server SHOULD NOT send this error if it is using default block size constraints or block size constraints negotiated out of band. A server sending an `NBD_REP_ERR_BLOCK_SIZE_REQD` error SHOULD ensure it first sends an `NBD_INFO_BLOCK_SIZE` information reply in order to help avoid a potentially unnecessary round trip.

Additionally, if TLS has not been initiated, the server MAY reply with `NBD_REP_ERR_TLS_REQD` (instead of `NBD_REP_ERR_UNKNOWN`) to requests for exports that are unknown. This is so that clients that have not initiated TLS cannot enumerate exports. A SELECTIVETLS server that chooses to hide unknown exports in this manner SHOULD NOT send `NBD_REP_INFO` replies for a TLS-only export.

For backwards compatibility, clients SHOULD be prepared to also handle `NBD_REP_ERR_UNSUP` by falling back to using `NBD_OPT_EXPORT_NAME`.

Other errors (such as `NBD_REP_ERR_SHUTDOWN`) are also possible, as permitted elsewhere in this document, with no constraints on the number of preceeding `NBD_REP_INFO`.

If there are no intervening option requests between a successful `NBD_OPT_INFO` (that is, one where the reply ended with a final `NBD_REP_ACK`) and an `NBD_OPT_GO` with the same parameters (including the list of information items requested), then the server MUST reply with the same set of information, such as transmission flags in the `NBD_INFO_EXPORT` reply, although the ordering of the intermediate `NBD_REP_INFO` messages MAY differ. Otherwise, due to the intervening option requests or the use of different parameters, the server MAY send different data in the successful response, and/or MAY fail the second request.

The reply to an `NBD_OPT_GO` is identical to the reply to `NBD_OPT_INFO` save that if the reply indicates success (i.e. ends with `NBD_REP_ACK`), the client and the server both immediately enter the transmission phase. The server MUST NOT send any zero padding bytes after the `NBD_REP_ACK` data, whether or not the client negotiated the `NBD_FLAG_C_NO_ZEROES` flag. The client MUST NOT send further option requests unless the final reply from the server indicates an error.

- `NBD_OPT_GO` (7)

  See above under `NBD_OPT_INFO`.

- `NBD_OPT_STRUCTURED_REPLY` (8)

  The client wishes to use structured replies during the transmission phase. The client MUST NOT send any additional data with the option, and the server SHOULD reject a request that includes data with `NBD_REP_ERR_INVALID`.

  The server replies with the following, or with an error permitted elsewhere in this document:

  - `NBD_REP_ACK`: Structured replies have been negotiated; the server MUST use structured replies to the `NBD_CMD_READ` transmission request. Other extensions that require structured replies may now be negotiated.
  - For backwards compatibility, clients SHOULD be prepared to also handle `NBD_REP_ERR_UNSUP`; in this case, no structured replies will be sent.

  It is envisioned that future extensions will add other new requests that may require a data payload in the reply. A server that supports such extensions SHOULD NOT advertise those extensions until the client negotiates structured replies; and a client MUST NOT make use of those extensions without first enabling the `NBD_OPT_STRUCTURED_REPLY` extension.

- `NBD_OPT_LIST_META_CONTEXT` (9)
  Data:

  Return a list of `NBD_REP_META_CONTEXT` replies, one per context, followed by an `NBD_REP_ACK` or an error.

  This option MUST NOT be requested unless structured replies have been negotiated first. If a client attempts to do so, a server SHOULD send `NBD_REP_ERR_INVALID`.

  - 32 bits, length of export name.
  - String, name of export for which we wish to list metadata contexts.
  - 32 bits, number of queries
  - Zero or more queries, each being:
    - 32 bits, length of query.
    - String, query to list a subset of the available metadata contexts. The syntax of this query is implementation-defined, except that it MUST start with a namespace and a colon.

  For details on the query string, see the “Metadata querying” section; note that a namespace may document that a different set of queries are valid

for `NBD_OPT_LIST_META_CONTEXT` than for `NBD_OPT_SET_META_CONTEXT`, such as when using an empty leaf-name for wildcarding.

If the option request is syntactically invalid (such as a query length that would require reading beyond the original length given in the option header), the server MUST fail the request with `NBD_REP_ERR_INVALID`. For requests that are semantically invalid (such as lacking the required colon that delimits the namespace, or using a leaf name that is invalid for a known namespace), the server MAY fail the request with `NBD_REP_ERR_INVALID`. However, the server MUST ignore query strings belonging to an unknown namespace. If none of the query strings find any metadata contexts, the server MUST send a single reply of type `NBD_REP_ACK`.

The server MUST reply with a list of zero or more `NBD_REP_META_CONTEXT` replies, followed by either a final `NBD_REP_ACK` on success or by an error (for instance `NBD_REP_ERR_UNSUP` if the option is not supported). If an error is returned, the client MUST disregard any context replies that may have been sent.

If zero queries are sent, then the server MUST return all the metadata contexts that are available to the client to select on the given export. However, this list may include wildcards that require a further `NBD_OPT_LIST_META_CONTEXT` with the wildcard as a query, rather than an actual context that is appropriate as a query to `NBD_OPT_SET_META_CONTEXT`, as set out below. In this case, the server SHOULD NOT fail with `NBD_REP_ERR_TOO_BIG`.

If one or more queries are sent, then the server MUST return those metadata contexts that are available to the client to select on the given export with `NBD_OPT_SET_META_CONTEXT`, and which match one or more of the queries given. The support of wildcarding within the leaf-name portion of the query string is dependent upon the namespace. The server MAY send contexts in a different order than in the client's query. In this case, the server MAY fail with `NBD_REP_ERR_TOO_BIG` if too many queries are requested.

In either case, however, for any given namespace the server MAY, instead of exhaustively listing every matching context available to select (or every context available to select where no query is given), send sufficient context records back to allow a client with knowledge of the namespace to select any context. This may be helpful where a client can construct algorithmic queries. For instance, a client might reply simply with the

namespace with no leaf-name (e.g. 'x-FooBar:') or with a range of values (e.g. 'x-ModifiedDate:20160310-20161214'). The semantics of such a reply are a matter for the definition of the namespace. However each namespace returned MUST begin with the relevant namespace, followed by a colon, and then other UTF-8 characters, with the entire string following the restrictions for strings set out earlier in this document.

The metadata context ID in these replies is reserved and SHOULD be set to zero; clients MUST disregard it.

- `NBD_OPT_SET_META_CONTEXT` (10)
  Data:

  Change the set of active metadata contexts. Issuing this command replaces all previously-set metadata contexts (including when this command fails); clients must ensure that all metadata contexts they are interested in are selected with the final query that they sent.

  This option MUST NOT be requested unless structured replies have been negotiated first. If a client attempts to do so, a server SHOULD send `NBD_REP_ERR_INVALID`.

  A client MUST NOT send `NBD_CMD_BLOCK_STATUS` unless within the negotiation phase it sent `NBD_OPT_SET_META_CONTEXT` at least once, and where the final time it was sent, it referred to the same export name that was ultimately selected for transmission phase, and where the server responded by returning least one metadata context without error.

  - 32 bits, length of export name.
  - String, name of export for which we wish to list metadata contexts.
  - 32 bits, number of queries
  - Zero or more queries, each being:
    - 32 bits, length of query
    - String, query to select metadata contexts. The syntax of this query is implementation-defined, except that it MUST start with a namespace and a colon.

  If zero queries are sent, the server MUST select no metadata contexts.

  The server MAY return `NBD_REP_ERR_TOO_BIG` if a request seeks to select too many contexts. Otherwise the server MUST reply with a number of

`NBD_REP_META_CONTEXT` replies, one for each selected metadata context, each with a unique metadata context ID, followed by `NBD_REP_ACK`. The server MAY ignore queries that do not select a single metadata context, and MAY return selected contexts in a different order than in the client's request. The metadata context ID is transient and may vary across calls to `NBD_OPT_SET_META_CONTEXT`; clients MUST therefore treat the ID as an opaque value and not (for instance) cache it between connections. It is not an error if a `NBD_OPT_SET_META_CONTEXT` option does not select any metadata context, provided the client then does not attempt to issue `NBD_CMD_BLOCK_STATUS` commands.

Option reply types

These values are used in the "reply type" field, sent by the server during option haggling in the fixed newstyle negotiation.

- `NBD_REP_ACK` (1)

  Will be sent by the server when it accepts the option and no further information is available, or when sending data related to the option (in the case of `NBD_OPT_LIST`) has finished. No data.

- `NBD_REP_SERVER` (2)

  A description of an export. Data:

  - 32 bits, length of name (unsigned); MUST be no larger than the reply packet header length - 4
  - String, name of the export, as expected by `NBD_OPT_EXPORT_NAME`, `NBD_OPT_INFO`, or `NBD_OPT_GO`
  - If length of name < (reply packet header length - 4), then the rest of the data contains some implementation-specific details about the export. This is not currently implemented, but future versions of nbd-server may send along some details about the export. Therefore, unless explicitly documented otherwise by a particular client request, this field is defined to be a string suitable for direct display to a human being.

- `NBD_REP_INFO` (3)

  A detailed description about an aspect of an export. The response to `NBD_OPT_INFO` and `NBD_OPT_GO` includes zero or more of these messages prior to a final error reply, or at least one before an `NBD_REP_ACK` reply indicating

success. The server MUST send an `NBD_INFO_EXPORT` information type at some point before sending an `NBD_REP_ACK`, so that `NBD_OPT_GO` can provide a superset of the information given in response to `NBD_OPT_EXPORT_NAME`; all other information types are optional. A particular information type SHOULD only appear once for a given export unless documented otherwise.

A client MUST NOT rely on any particular ordering amongst the `NBD_OPT_INFO` replies, and MUST ignore information types that it does not recognize.

The acceptable values for the header *length* field are determined by the information type, and includes the 2 bytes for the type designator, in the following general layout:

- 16 bits, information type (e.g. `NBD_INFO_EXPORT`)
- *length* - *2* bytes, information payload

The following information types are defined:

- `NBD_INFO_EXPORT` (0)

  Mandatory information before a successful completion of `NBD_OPT_INFO` or `NBD_OPT_GO`. Describes the same information that is sent in response to the older `NBD_OPT_EXPORT_NAME`, except that there are no trailing zeroes whether or not `NBD_FLAG_C_NO_ZEROES` was negotiated. *length* MUST be 12, and the reply payload is interpreted as follows:

  - 16 bits, `NBD_INFO_EXPORT`
  - 64 bits, size of the export in bytes (unsigned)
  - 16 bits, transmission flags

- `NBD_INFO_NAME` (1)

  Represents the server's canonical name of the export. The name MAY differ from the name presented in the client's option request, and the information item MAY be omitted if the client option request already used the canonical name. This information type represents the same name that would appear in the name portion of an `NBD_REP_SERVER` in response to `NBD_OPT_LIST`. The *length* MUST be at least 2, and the reply payload is interpreted as:

  - 16 bits, `NBD_INFO_NAME`

- String: name of the export, *length* - *2* bytes

- `NBD_INFO_DESCRIPTION` (2)

  A description of the export, suitable for direct display to the human being. This information type represents the same optional description that may appear after the name portion of an `NBD_REP_SERVER` in response to `NBD_OPT_LIST`. The *length* MUST be at least 2, and the reply payload is interpreted as:

  - 16 bits, `NBD_INFO_DESCRIPTION`
  - String: description of the export, *length* - *2* bytes

- `NBD_INFO_BLOCK_SIZE` (3)

  Represents the server's advertised block size constraints; see the "Block size constraints" section for more details on what these values represent, and on constraints on their values. The server MUST send this info if it is requested and it intends to enforce block size constraints other than the defaults. After sending this information in response to an `NBD_OPT_GO` in which the client specifically requested `NBD_INFO_BLOCK_SIZE`, the server can legitimately assume that any client that continues the session will support the block size constraints supplied (note that this assumption cannot be made solely on the basis of an `NBD_OPT_INFO` with an `NBD_INFO_BLOCK_SIZE` request, or an `NBD_OPT_GO` without an explicit `NBD_INFO_BLOCK_SIZE` request). The *length* MUST be 14, and the reply payload is interpreted as:

  - 16 bits, `NBD_INFO_BLOCK_SIZE`
  - 32 bits, minimum block size
  - 32 bits, preferred block size
  - 32 bits, maximum block size

- `NBD_REP_META_CONTEXT` (4)

  A description of a metadata context. Data:

  - 32 bits, NBD metadata context ID.
  - String, name of the metadata context. This is not required to be a human-readable string, but it MUST be valid UTF-8 data.

There are a number of error reply types, all of which are denoted by having bit 31 set. All error replies MAY have some data set, in which case that data is an error message string suitable for display to the user.

- `NBD_REP_ERR_UNSUP` (2^31 + 1)

  The option sent by the client is unknown by this server implementation (e.g., because the server is too old, or from another source).

- `NBD_REP_ERR_POLICY` (2^31 + 2)

  The option sent by the client is known by this server and syntactically valid, but server-side policy forbids the server to allow the option (e.g., the client sent `NBD_OPT_LIST` but server configuration has that disabled)

- `NBD_REP_ERR_INVALID` (2^31 + 3)

  The option sent by the client is known by this server, but was determined by the server to be syntactically or semantically invalid. For instance, the client sent an `NBD_OPT_LIST` with nonzero data length, or the client sent a second `NBD_OPT_STARTTLS` after TLS was already negotiated.

- `NBD_REP_ERR_PLATFORM` (2^31 + 4)

  The option sent by the client is not supported on the platform on which the server is running, or requires compile-time options that were disabled, e.g., upon trying to use TLS.

- `NBD_REP_ERR_TLS_REQD` (2^31 + 5)

  The server is unwilling to continue negotiation unless TLS is initiated first. In the case of `NBD_OPT_INFO` and `NBD_OPT_GO` this unwillingness MAY (depending on the TLS mode) be limited to the export in question. See the section on TLS above for further details.

- `NBD_REP_ERR_UNKNOWN` (2^31 + 6)

  The requested export is not available.

- `NBD_REP_ERR_SHUTDOWN` (2^31 + 7)

  The server is unwilling to continue negotiation as it is in the process of being shut down.

- `NBD_REP_ERR_BLOCK_SIZE_REQD` (2^31 + 8)

  The server is unwilling to enter transmission phase for a given export unless the client first acknowledges (via `NBD_INFO_BLOCK_SIZE`) that it will

obey non-default block sizing requirements.

- `NBD_REP_ERR_TOO_BIG` (2^31 + 9)

  The request or the reply is too large to process.

Transmission phase

Flag fields

Command flags

This field of 16 bits is sent by the client with every request and provides additional information to the server to execute the command. Refer to the "Request types" section below for more details about how a given flag affects a particular command. Clients MUST NOT set a command flag bit that is not documented for the particular command; and whether a flag is valid may depend on negotiation during the handshake phase.

- bit 0, `NBD_CMD_FLAG_FUA`; This flag is valid for all commands, provided `NBD_FLAG_SEND_FUA` has been negotiated, in which case the server MUST accept all commands with this bit set (even by ignoring the bit). The client SHOULD NOT set this bit unless the command has the potential of writing data (current commands are `NBD_CMD_WRITE`, `NBD_CMD_WRITE_ZEROES` and `NBD_CMD_TRIM`), however note that existing clients are known to set this bit on other commands. Subject to that, and provided `NBD_FLAG_SEND_FUA` is negotiated, the client MAY set this bit on all, no or some commands as it wishes (see the section on Ordering of messages and writes for details). If the server receives a command with `NBD_CMD_FLAG_FUA` set it MUST NOT send its reply to that command until all write operations (if any) associated with that command have been completed and persisted to non-volatile storage. If the command does not in fact write data (for instance on an `NBD_CMD_TRIM` in a situation where the command as a whole is ignored), the server MAY ignore this bit being set on such a command.

- bit 1, `NBD_CMD_FLAG_NO_HOLE`; valid during `NBD_CMD_WRITE_ZEROES`. SHOULD be set to 1 if the client wants to ensure that the server does not create a hole. The client MAY send `NBD_CMD_FLAG_NO_HOLE` even if `NBD_FLAG_SEND_TRIM` was not set in the transmission flags field. The server MUST support the use of this flag if it advertises `NBD_FLAG_SEND_WRITE_ZEROES`.

- bit 2, `NBD_CMD_FLAG_DF`; the "don't fragment" flag, valid during `NBD_CMD_READ`. SHOULD be set to 1 if the client requires the server to send at most one content chunk in reply. MUST NOT be set unless the transmission flags include `NBD_FLAG_SEND_DF`. Use of this flag MAY trigger an `NBD_EOVERFLOW` error chunk, if the request length is too large.
- bit 3, `NBD_CMD_FLAG_REQ_ONE`; valid during `NBD_CMD_BLOCK_STATUS`. If set, the client is interested in only one extent per metadata context. If this flag is present, the server MUST NOT send metadata on more than one extent in the reply. Client implementors should note that using this flag on multiple contiguous requests is likely to be inefficient.
- bit 4, `NBD_CMD_FLAG_FAST_ZERO`; valid during `NBD_CMD_WRITE_ZEROES`. If set, but the server cannot perform the write zeroes any faster than it would for an equivalent `NBD_CMD_WRITE`, then the server MUST fail quickly with an error of `NBD_ENOTSUP`. The client MUST NOT set this unless the server advertised `NBD_FLAG_SEND_FAST_ZERO`.

Structured reply flags

This field of 16 bits is sent by the server as part of every structured reply.

- bit 0, `NBD_REPLY_FLAG_DONE`; the server MUST clear this bit if more structured reply chunks will be sent for the same client request, and MUST set this bit if this is the final reply. This bit MUST always be set for the `NBD_REPLY_TYPE_NONE` chunk, although any other chunk type can also be used as the final chunk.

The server MUST NOT set any other flags without first negotiating the extension with the client, unless the client can usefully react to the response without interpreting the flag (for instance if the flag is some form of hint). Clients MUST ignore unrecognized flags.

Structured reply types

These values are used in the "type" field of a structured reply. Some chunk types can additionally be categorized by role, such as *error chunks* or *content chunks*. Each type determines how to interpret the "length" bytes of payload. If the client receives an unknown or unexpected type, other than an *error chunk*, it MUST initiate a hard disconnect.

- `NBD_REPLY_TYPE_NONE` (0)

length MUST be 0 (and the payload field omitted). This chunk type MUST always be used with the `NBD_REPLY_FLAG_DONE` bit set (that is, it may appear at most once in a structured reply, and is only useful as the final reply chunk). If no earlier error chunks were sent, then this type implies that the overall client request is successful. Valid as a reply to any request.

- `NBD_REPLY_TYPE_OFFSET_DATA` (1)

  This chunk type is in the content chunk category. *length* MUST be at least 9. It represents the contents of *length* - *8* bytes of the file, starting at the absolute *offset* from the start of the export. The data MUST lie within the bounds of the original offset and length of the client's request, and MUST NOT overlap with the bounds of any earlier content chunk or error chunk in the same reply. This chunk MAY be used more than once in a reply, unless the `NBD_CMD_FLAG_DF` flag was set. Valid as a reply to `NBD_CMD_READ`.

  The payload is structured as:

  64 bits: offset (unsigned)*length* - *8* bytes: data

- `NBD_REPLY_TYPE_OFFSET_HOLE` (2)

  This chunk type is in the content chunk category. *length* MUST be exactly 12. It represents that the contents of *hole size* bytes, starting at the absolute *offset* from the start of the export, read as all zeroes. The hole MUST lie within the bounds of the original offset and length of the client's request, and MUST NOT overlap with the bounds of any earlier content chunk or error chunk in the same reply. This chunk MAY be used more than once in a reply, unless the `NBD_CMD_FLAG_DF` flag was set. Valid as a reply to `NBD_CMD_READ`.

  The payload is structured as:

  64 bits: offset (unsigned)
  32 bits: hole size (unsigned, MUST be nonzero)

- `NBD_REPLY_TYPE_BLOCK_STATUS` (5)

  length MUST be 4 + (a positive integer multiple of 8). This reply represents a series of consecutive block descriptors where the sum of the length fields within the descriptors is subject to further constraints documented below. This chunk type MUST appear exactly once per metadata ID in a structured reply.

  The payload starts with:

32 bits, metadata context ID

and is followed by a list of one or more descriptors, each with this layout:

32 bits, length of the extent to which the status below applies (unsigned, MUST be nonzero)
32 bits, status flags

If the client used the `NBD_CMD_FLAG_REQ_ONE` flag in the request, then every reply chunk MUST contain exactly one descriptor, and that descriptor MUST NOT exceed the *length* of the original request. If the client did not use the flag, and the server replies with N extents, then the sum of the *length* fields of the first N-1 extents (if any) MUST be less than the requested length, while the *length* of the final extent MAY result in a sum larger than the original requested length, if the server has that information anyway as a side effect of reporting the status of the requested region.

Even if the client did not use the `NBD_CMD_FLAG_REQ_ONE` flag in its request, the server MAY return fewer descriptors in the reply than would be required to fully specify the whole range of requested information to the client, if looking up the information would be too resource-intensive for the server, so long as at least one extent is returned. Servers should however be aware that most clients implementations will then simply ask for the next extent instead.

All error chunk types have bit 15 set, and begin with the same *error*, *message length*, and optional *message* fields as `NBD_REPLY_TYPE_ERROR`. If nonzero, *message length* indicates that an optional error string message appears next, suitable for display to a human user. The header *length* then covers any remaining structured fields at the end.

- `NBD_REPLY_TYPE_ERROR` (2^15 + 1)

  This chunk type is in the error chunk category. *length* MUST be at least 6. This chunk represents that an error occurred, and the client MAY NOT make any assumptions about partial success. This type SHOULD NOT be used more than once in a structured reply. Valid as a reply to any request.

  The payload is structured as:

  32 bits: error (MUST be nonzero)
  16 bits: message length (no more than header *length* - 6)*message length* bytes: optional string suitable for direct display to a human being

- `NBD_REPLY_TYPE_ERROR_OFFSET` (2^15 + 2)

This chunk type is in the error chunk category. *length* MUST be at least 14. This reply represents that an error occurred at a given offset, which MUST lie within the original offset and length of the request; the client can use this offset to determine if request had any partial success. This chunk type MAY appear multiple times in a structured reply, although the same offset SHOULD NOT be repeated. Likewise, if content chunks were sent earlier in the structured reply, the server SHOULD NOT send multiple distinct offsets that lie within the bounds of a single content chunk. Valid as a reply to `NBD_CMD_READ`, `NBD_CMD_WRITE`, `NBD_CMD_TRIM`, and `NBD_CMD_BLOCK_STATUS`.

The payload is structured as:

32 bits: error (MUST be nonzero)
16 bits: message length (no more than header *length* - 14)*message length* bytes: optional string suitable for direct display to a human being
64 bits: offset (unsigned)

If the client receives an unknown or unexpected type with bit 15 set, it MUST consider the current reply as errored, but MAY continue transmission unless it detects that *message length* is too large to fit within the *length* specified by the header. For all other messages with unknown or unexpected type or inconsistent contents, the client MUST initiate a hard disconnect.

Request types

The following request types exist:

- `NBD_CMD_READ` (0)

  A read request. Length and offset define the data to be read. The server MUST reply with either a simple reply or a structured reply, according to whether the structured replies have been negotiated using `NBD_OPT_STRUCTURED_REPLY`. The client SHOULD NOT request a read length of 0; the behavior of a server on such a request is unspecified although the server SHOULD NOT disconnect.

  Simple replies

  If structured replies were not negotiated, then a read request MUST always be answered by a simple reply, as documented above (using magic 0×67446698 `NBD_SIMPLE_REPLY_MAGIC`, and containing length bytes of data according to the client's request).

If an error occurs, the server SHOULD set the appropriate error code in the error field. The server MAY then initiate a hard disconnect. If it chooses not to, it MUST NOT send any payload for this request.

If an error occurs while reading after the server has already sent out the reply header with an error field set to zero (i.e., signalling no error), the server MUST immediately initiate a hard disconnect; it MUST NOT send any further data to the client.

Structured replies

If structured replies are negotiated, then a read request MUST result in a structured reply with one or more chunks (each using magic 0×668e33ef `NBD_STRUCTURED_REPLY_MAGIC`), where the final chunk has the flag `NBD_REPLY_FLAG_DONE`, and with the following additional constraints.

The server MAY split the reply into any number of content chunks; each chunk MUST describe at least one byte, although to minimize overhead, the server SHOULD use chunks with lengths and offsets as an integer multiple of 512 bytes, where possible (the first and last chunk of an unaligned read being the most obvious places for an exception). The server MUST NOT send content chunks that overlap with any earlier content or error chunk, and MUST NOT send chunks that describe data outside the offset and length of the request, but MAY send the content chunks in any order (the client MUST reassemble content chunks into the correct order), and MAY send additional content chunks even after reporting an error chunk. Note that a request for more than 2^32 - 8 bytes MUST be split into at least two chunks, so as not to overflow the length field of a reply while still allowing space for the offset of each chunk. When no error is detected, the server MUST send enough data chunks to cover the entire region described by the offset and length of the client's request.

To minimize traffic, the server MAY use a content or error chunk as the final chunk by setting the `NBD_REPLY_FLAG_DONE` flag, but MUST NOT do so for a content chunk if it would still be possible to detect an error while transmitting the chunk. The `NBD_REPLY_TYPE_NONE` chunk is always acceptable as the final chunk.

If an error is detected, the server MUST still complete the transmission of any current chunk (it MUST use padding bytes which SHOULD be zero, for any remaining data portion of a chunk with type

`NBD_REPLY_TYPE_OFFSET_DATA`), but MAY omit further content chunks. The server MUST include an error chunk as one of the subsequent chunks, but MAY defer the error reporting behind other queued chunks. An error chunk of type `NBD_REPLY_TYPE_ERROR` implies that the client MAY NOT make any assumptions about validity of data chunks (whether sent before or after the error chunk), and if used, SHOULD be the only error chunk in the reply. On the other hand, an error chunk of type `NBD_REPLY_TYPE_ERROR_OFFSET` gives fine-grained information about which earlier data chunk(s) encountered a failure; as such, a server MAY still usefully follow it with further non-overlapping content chunks or with error offsets for other content chunks. The server MAY send an error chunk with no corresponding content chunk, but MUST ensure that the content chunk is sent first if a content and error chunk cover the same offset. Generally, a server SHOULD NOT mix errors with offsets with a generic error. As long as all errors are accompanied by offsets, the client MAY assume that any data chunks with no subsequent error offset are valid, that chunks with an overlapping error offset errors are valid up until the reported offset, and that portions of the read that do not have a corresponding content chunk are not valid.

A client MAY initiate a hard disconnect if it detects that the server has sent invalid chunks (such as overlapping data, or not enough data before claiming success).

In order to avoid the burden of reassembly, the client MAY set the `NBD_CMD_FLAG_DF` flag ("don't fragment"). If this flag is set, the server MUST send at most one content chunk, although it MAY still send multiple chunks (the remaining chunks would be error chunks or a final type of `NBD_REPLY_TYPE_NONE`). If the area being read contains both data and a hole, the server MUST use `NBD_REPLY_TYPE_OFFSET_DATA` with the zeroes explicitly present. A server MAY reject a client's request with the error `NBD_EOVERFLOW` if the length is too large to send without fragmentation, in which case it MUST NOT send a content chunk; however, the server MUST support unfragmented reads in which the client's request length does not exceed 65,536 bytes.

- `NBD_CMD_WRITE` (1)

  A write request. Length and offset define the location and amount of data to be written. The client MUST follow the request header with *length* number of bytes to be written to the device. The client SHOULD NOT

request a write length of 0; the behavior of a server on such a request is unspecified although the server SHOULD NOT disconnect.

The server MUST write the data to disk, and then send the reply message. The server MAY send the reply message before the data has reached permanent storage, unless `NBD_CMD_FLAG_FUA` is in use.

If an error occurs, the server MUST set the appropriate error code in the error field.

- `NBD_CMD_DISC` (2)

  A disconnect request. The server MUST handle all outstanding requests, shut down the TLS session (if one is running), and close the TCP session. A client MUST NOT send anything to the server after sending an `NBD_CMD_DISC` command.

  The values of the length and offset fields in a disconnect request MUST be zero.

  There is no reply to an `NBD_CMD_DISC`.

- `NBD_CMD_FLUSH` (3)

  A flush request. The server MUST NOT send a successful reply header for this request before all write requests for which a reply has already been sent to the client have reached permanent storage (using fsync() or similar).

  A client MUST NOT send a flush request unless `NBD_FLAG_SEND_FLUSH` was set in the transmission flags field.

  For a flush request, *length* and *offset* are reserved, and MUST be set to all-zero.

- `NBD_CMD_TRIM` (4)

  A hint to the server that the data defined by length and offset is no longer needed. A server MAY discard *length* bytes starting at offset, but is not required to; and MAY round *offset* up and *length* down to meet internal alignment constraints so that only a portion of the client's request is actually discarded. The client SHOULD NOT request a trim length of 0; the behavior of a server on such a request is unspecified although the server SHOULD NOT disconnect.

  After issuing this command, a client MUST NOT make any assumptions about the contents of the export affected by this command, until

overwriting it again with `NBD_CMD_WRITE` or `NBD_CMD_WRITE_ZEROES`.

A client MUST NOT send a trim request unless `NBD_FLAG_SEND_TRIM` was set in the transmission flags field.

- `NBD_CMD_CACHE` (5)

  A cache request. The client is informing the server that it plans to access the area specified by *offset* and *length*. The server MAY use this information to speed up further access to that area (for example, by performing the actions of `NBD_CMD_READ` but replying with just status instead of a payload, by using posix_fadvise(), or by retrieving remote data into a local cache so that future reads and unaligned writes to that region are faster). However, it is unspecified what the server's actual caching mechanism is (if any), whether there is a limit on how much can be cached at once, and whether writes to a cached region have write-through or write-back semantics. Thus, even when this command reports success, there is no guarantee of an actual performance gain. A future version of this standard may add command flags to request particular caching behaviors, where a server would reply with an error if that behavior cannot be accomplished.

  If an error occurs, the server MUST set the appropriate error code in the error field. However failure on this operation does not imply that further read and write requests on this area will fail, and, other than any difference in performance, there MUST NOT be any difference in semantics compared to if the client had not used this command. When no command flags are in use, the server MAY send a reply prior to the requested area being fully cached.

  Note that client implementations exist which attempt to send a cache request even when `NBD_FLAG_SEND_CACHE` was not set in the transmission flags field, however, these implementations do not use any command flags. A server MAY advertise `NBD_FLAG_SEND_CACHE` even if the command has no effect or always fails with `NBD_EINVAL`; however, if it advertised the command, the server MUST reject any command flags it does not recognize.

- `NBD_CMD_WRITE_ZEROES` (6)

  A write request with no payload. *Offset* and *length* define the location and amount of data to be zeroed. The client SHOULD NOT request a write

length of 0; the behavior of a server on such a request is unspecified although the server SHOULD NOT disconnect.

The server MUST zero out the data on disk, and then send the reply message. The server MAY send the reply message before the data has reached permanent storage, unless `NBD_CMD_FLAG_FUA` is in use.

A client MUST NOT send a write zeroes request unless `NBD_FLAG_SEND_WRITE_ZEROES` was set in the transmission flags field. Additionally, a client MUST NOT send the `NBD_CMD_FLAG_FAST_ZERO` flag unless `NBD_FLAG_SEND_FAST_ZERO` was set in the transmission flags field.

By default, the server MAY use trimming to zero out the area, even if it did not advertise `NBD_FLAG_SEND_TRIM`; but it MUST ensure that the data reads back as zero. However, the client MAY set the command flag `NBD_CMD_FLAG_NO_HOLE` to inform the server that the area MUST be fully provisioned, ensuring that future writes to the same area will not cause fragmentation or cause failure due to insufficient space.

If the server advertised `NBD_FLAG_SEND_FAST_ZERO` but `NBD_CMD_FLAG_FAST_ZERO` is not set, then the server MUST NOT fail with `NBD_ENOTSUP`, even if the operation is no faster than a corresponding `NBD_CMD_WRITE`. Conversely, if `NBD_CMD_FLAG_FAST_ZERO` is set, the server MUST fail quickly with `NBD_ENOTSUP` unless the request can be serviced in less time than a corresponding `NBD_CMD_WRITE`, and SHOULD NOT alter the contents of the export when returning this failure. The server's determination on whether to fail a fast request MAY depend on a number of factors, such as whether the request was suitably aligned, on whether the `NBD_CMD_FLAG_NO_HOLE` flag was present, or even on whether a previous `NBD_CMD_TRIM` had been performed on the region. If the server did not advertise `NBD_FLAG_SEND_FAST_ZERO`, then it SHOULD NOT fail with `NBD_ENOTSUP`, regardless of the speed of servicing a request, and SHOULD fail with `NBD_EINVAL` if the `NBD_CMD_FLAG_FAST_ZERO` flag was set. A server MAY advertise `NBD_FLAG_SEND_FAST_ZERO` whether or not it will actually succeed on a fast zero request (a fast failure of `NBD_ENOTSUP` still counts as a fast response); similarly, a server SHOULD fail a fast zero request with `NBD_ENOTSUP` if the server cannot quickly determine in advance whether proceeding with the request would be fast, even if it turns out that the same request without the flag would be fast after all.

One intended use of a fast zero request is optimizing the copying of a sparse image source into the export: a client can request fast zeroing of the entire export, and if it succeeds, follow that with write requests to just

the data portions before a single flush of the entire image, for fewer transactions overall. On the other hand, if the fast zero request fails, the fast failure lets the client know that it must manually write zeroes corresponding to the holes of the source image before a final flush, for more transactions but with no time lost to duplicated I/O to the data portions. Knowing this usage pattern can help decide whether a server's implementation for writing zeroes counts as fast (for example, a successful fast zero request may start a background operation that would cause the next flush request to take longer, but that is okay as long as intermediate writes before that flush do not further lengthen the time spent on the overall sequence of operations).

If an error occurs, the server MUST set the appropriate error code in the error field.

The server SHOULD return `NBD_ENOSPC` if it receives a write zeroes request including one or more sectors beyond the size of the device. It SHOULD return `NBD_EPERM` if it receives a write zeroes request on a read-only export.

- `NBD_CMD_BLOCK_STATUS` (7)

  A block status query request. Length and offset define the range of interest. The client SHOULD NOT request a status length of 0; the behavior of a server on such a request is unspecified although the server SHOULD NOT disconnect.

  A client MUST NOT send `NBD_CMD_BLOCK_STATUS` unless within the negotiation phase it sent `NBD_OPT_SET_META_CONTEXT` at least once, and where the final time that was sent, it referred to the same export name used to enter transmission phase, and where the server returned at least one metadata context without an error. This in turn requires the client to first negotiate structured replies. For a successful return, the server MUST use a structured reply, containing exactly one chunk of type `NBD_REPLY_TYPE_BLOCK_STATUS` per selected context id, where the status field of each descriptor is determined by the flags field as defined by the metadata context. The server MAY send chunks in a different order than the context ids were assigned in reply to `NBD_OPT_SET_META_CONTEXT`.

  The list of block status descriptors within the `NBD_REPLY_TYPE_BLOCK_STATUS` chunk represent consecutive portions of the file starting from specified *offset*. If the client used the `NBD_CMD_FLAG_REQ_ONE` flag, each chunk contains exactly one descriptor where the *length* of the descriptor MUST NOT be

greater than the *length* of the request; otherwise, a chunk MAY contain multiple descriptors, and the final descriptor MAY extend beyond the original requested size if the server can determine a larger length without additional effort. On the other hand, the server MAY return less data than requested. However the server MUST return at least one status descriptor (and since each status descriptor has a non-zero length, a client can always make progress on a successful return). The server SHOULD use different *status* values between consecutive descriptors where feasible, although the client SHOULD be prepared to handle consecutive descriptors with the same *status* value. The server SHOULD use descriptor lengths that are an integer multiple of 512 bytes where possible (the first and last descriptor of an unaligned query being the most obvious places for an exception), and MUST use descriptor lengths that are an integer multiple of any advertised minimum block size. The status flags are intentionally defined so that a server MAY always safely report a status of 0 for any block, although the server SHOULD return additional status values when they can be easily detected.

If an error occurs, the server SHOULD set the appropriate error code in the error field of an error chunk. However, if the error does not involve invalid usage (such as a request beyond the bounds of the file), a server MAY reply with a single block status descriptor with *length* matching the requested length, rather than reporting the error; in this case the context MAY mandate the status returned.

A client MAY initiate a hard disconnect if it detects that the server has sent an invalid chunk. The server SHOULD return `NBD_EINVAL` if it receives a `NBD_CMD_BLOCK_STATUS` request including one or more sectors beyond the size of the device.

- `NBD_CMD_RESIZE` (8)

  Defined by the experimental `RESIZE` extension.

- Other requests

  Some third-party implementations may require additional protocol messages which are not described in this document. In the interest of interoperability, authors of such implementations SHOULD contact the maintainer of this document, so that these messages can be listed here to avoid conflicting implementations.

Error values

The error values are used for the error field in the reply message. Originally, error messages were defined as the value of `errno` on the system running the server; however, although they happen to have similar values on most systems, these values are in fact not well-defined, and therefore not entirely portable.

Therefore, the allowed values for the error field have been restricted to set of possibilities. To remain intelligible with older clients, the most common values of `errno` for that particular error has been chosen as the value for an error.

The following error values are defined:

- `NBD_EPERM` (1), Operation not permitted.
- `NBD_EIO` (5), Input/output error.
- `NBD_ENOMEM` (12), Cannot allocate memory.
- `NBD_EINVAL` (22), Invalid argument.
- `NBD_ENOSPC` (28), No space left on device.
- `NBD_EOVERFLOW` (75), Value too large.
- `NBD_ENOTSUP` (95), Operation not supported.
- `NBD_ESHUTDOWN` (108), Server is in the process of being shut down.

The server SHOULD return `NBD_ENOSPC` if it receives a write request including one or more sectors beyond the size of the device. It also SHOULD map the `EDQUOT` and `EFBIG` errors to `NBD_ENOSPC`. It SHOULD return `NBD_EINVAL` if it receives a read or trim request including one or more sectors beyond the size of the device, or if a read or write request is not aligned to advertised minimum block sizes. Finally, it SHOULD return `NBD_EPERM` if it receives a write or trim request on a read-only export.

The server SHOULD NOT return `NBD_EOVERFLOW` except as documented in response to `NBD_CMD_READ` when `NBD_CMD_FLAG_DF` is supported.

The server SHOULD NOT return `NBD_ENOTSUP` except as documented in response to `NBD_CMD_WRITE_ZEROES` when `NBD_CMD_FLAG_FAST_ZERO` is supported.

The server SHOULD return `NBD_EINVAL` if it receives an unknown command.

The server SHOULD return `NBD_EINVAL` if it receives an unknown command flag. It also SHOULD return `NBD_EINVAL` if it receives a request with a flag not explicitly documented as applicable to the given request.

Which error to return in any other case is not specified by the NBD protocol.

The server SHOULD NOT return `NBD_ENOMEM` if at all possible.

The client SHOULD treat an unexpected error value as if it had been `NBD_EINVAL`, rather than disconnecting from the server.

Experimental extensions

In addition to the normative elements of the specification set out herein, various experimental non-normative extensions have been proposed. These may not be implemented in any known server or client, and are subject to change at any point. A full implementation may require changes to the specifications, or cause the specifications to be withdrawn altogether.

These experimental extensions are set out in git branches starting with names starting with the word 'extension'.

Currently known are:

- The `STRUCTURED_REPLY` extension.
- The `BLOCK_STATUS` extension (based on the `STRUCTURED_REPLY` extension).
- The `RESIZE` extension.

Implementors of these extensions are strongly suggested to contact the mailinglist in order to help fine-tune the specifications before committing to a particular implementation.

Those proposing further extensions should also contact the mailinglist. It is possible to reserve command codes etc. within this document for such proposed extensions. Aside from that, extensions are written as branches which can be merged into master if and when those extensions are promoted to the normative version of the document in the master branch.

Compatibility and interoperability

Originally, the NBD protocol was a fairly simple protocol with few options. While the basic protocol is still reasonably simple, a growing number of extensions has been implemented that may make the protocol description seem overwhelming at first.

In an effort to not overwhelm first-time implementors with various options and features that may or may not be important for their use case, while at the

same time desiring maximum interoperability, this section tries to clarify what is optional and what is expected to be available in all implementations.

All protocol options and messages not explicitly mentioned below should be considered optional features that MAY be negotiated between client and server, but are not required to be available.

Baseline

The following MUST be implemented by all implementations, and should be considered a baseline:

- NOTLS mode
- The fixed newstyle handshake
- During the handshake:
  - the `NBD_OPT_INFO` and `NBD_OPT_GO` messages, with the `NBD_INFO_EXPORT` response.
  - Servers that receive messages which they do not implement MUST reply to them with `NBD_OPT_UNSUP`, and MUST NOT fail to parse the next message received.
  - the `NBD_OPT_ABORT` message, and its response.
  - the `NBD_OPT_LIST` message and its response.
- During the transmission phase:
  - Simple replies
  - the `NBD_CMD_READ` message (and its response)
  - the `NBD_CMD_WRITE` message (and its response), unless the implementation is a client that does not wish to write
  - the `NBD_CMD_DISC` message (and its resulting effects, although no response is involved)

Clients that wish to use more messages MUST negotiate them during the handshake phase, first.

Maximum interoperability

Clients and servers that desire maximum interoperability SHOULD implement the following features:

- TLS-encrypted communication, which may be required by some implementations or configurations;
- Servers that implement block constraints through `NBD_INFO_BLOCK_SIZE` and desire maximum interoperability SHOULD NOT require them. Similarly, clients that desire maximum interoperability SHOULD implement querying for block size constraints. Since some clients default to a block size of 512 bytes, implementations desiring maximum interoperability MAY default to that size.
- Clients or servers that desire interoperability with older implementations SHOULD implement the `NBD_OPT_EXPORT_NAME` message in addition to `NBD_OPT_INFO` and `NBD_OPT_GO`.
- For data safety, implementing `NBD_CMD_FLUSH` and the `NBD_CMD_FLAG_FUA` flag to `NBD_CMD_WRITE` is strongly recommended. Clients that do not implement querying for block size constraints SHOULD abide by the rules laid out in the section “Block size constraints”, above.

Future considerations

The following may be moved to the “Maximum interoperability” or “Baseline” sections at some point in the future, but some significant implementations are not yet ready to support them:

- Structured replies; the Linux kernel currently does not yet implement them.

About this file

This file tries to document the NBD protocol as it is currently implemented in the Linux kernel and in the reference implementation. The purpose of this file is to allow people to understand the protocol without having to read the code. However, the description above does not come with any form of warranty; while every effort has been taken to avoid them, mistakes are possible.

In contrast to the other files in this repository, this file is not licensed under the GPLv2. To the extent possible by applicable law, I hereby waive all copyright and related or neighboring rights to this file and release it into the public domain.

The purpose of releasing this into the public domain is to allow competing implementations of the NBD protocol without those implementations being considered derivative implementations; but please note that changing this

document, while allowed by its public domain status, does not make an incompatible implementation suddenly speak the NBD protocol.