NBD 协议简介

# 1. 两个阶段

1、nbd协议有两个阶段：

（1）握手阶段（也叫协商阶段，在这个阶段，nbd客户端与nbd服务端建立socket连接，并对服务端export的设备进行协商，这包括设备名称、容量以及协议自身的一些选项的协商）

（2）数据传输阶段（客户端与服务端进行数据传输）。

Linux主机上，第（1）个阶段在用户态完成，第（2）个阶段在内核态实现。

2、握手阶段，客户端至少需要执行两个ioctl命令：

**ioctl(nbd, NBD\_SET\_SOCK, sock)**

该命令主要将sock拷贝到内核，并设置到客户端内核nbd设备的数据结构上，以便第二阶段中，客户端内核态直接与服务端内核态进行数据传输。（参数nbd是client端打开的文件描述符（一般对应/dev/nbdx设备），sock是客户端建立起来的与服务端的socket连接）

**ioctl(nbd, NBD\_DO\_IT)**

该命令启动客户端内核的数据循环，不会返回，直到客户端断开与服务端的连接才返回。

此外，nbd握手阶段还会视客户端启动参数情况，调用其它的ioctl命令，设置内核选项参数。

3. 数据传输阶段的两类消息

在数据传输阶段，nbd支持以下两类消息格式：

1. request请求(nbd客户端发送)

request请求有以下五种：

* NBD\_CMD\_READ

读请求

* NBD\_CMD\_WRITE

写请求

* NBD\_CMD\_DISC

断开连接

* NBD\_CMD\_FLUSH

刷盘请求

* NBD\_CMD\_TRIM

丢弃数据（discard）

request请求的头部（header）格式如下：

32位的magic数字

32位的域字段（请求类型：见上述五种，例如：NBD\_CMD\_READ）

64位的handle（’handle’）

64位的偏移（请求的偏移offset地址，from）

32位的长度(请求操作的长度，len)

例如：写请求（NBD\_CMD\_WRITE）中，header后紧跟的就是len长度字节的数据。在NBD\_CMD\_FLUSH请求中，from和len被设置为0，只有头部(header)没有数据，表示整个设备刷数据。

1. response回复（nbd服务器发送）

Response回复的头部格式如下：

32位的magic数字

32位的error code

64位handle（'handle’）

例如：对读请求的回复中，error设置为0，回复的头部后面紧跟客户端请求长度的数据。

其中request和response头部的handle为待发送用户态数据的内存地址。

# 2 nbd协商阶段

nbd协商阶段实现在用户态，在nbd客户端连接到nbd服务端后立即进行nbd设备的协商阶段。由于历史原因，nbd协商存在两种：传统格式的协商和新格式的协商。Nbd用户态的实现各自不一，但一般都兼容两类协商格式，本文只介绍新格式的协商方式。

nbd协商阶段，又可以细分为两个阶段

* flags标志位的协商
* nbd选项的协商

当nbd服务端接收到nbd服务端的连接请求，且连接成功后，立即进入协商阶段，**nbd服务端首先发起协商。**以下假设‘S’代表nbd服务端，‘C’代表nbd客户端:

（1）flags标志位的协商(nbd服务端主导)

S: “NBDMAGIC” （8B，发送nbd协议的密钥）

S: 0x49484156454F5054 （8B，发送新格式的协商magic）

S: flags （2B，发送16位的全局标志位）

C: flags （4B，发送32位的标志位）

以上完成了nbd协商的第一个阶段，nbd服务端发送的全局标志位当前仅支持第0位设置为1，代表使用new-style的协商。nbd客户端收到并检查到该位被置位，如果支持新格式的协商，则同样需要将自己的标志位置位第0位，并发送回nbd服务端。至此，nbd双方在协议上达成一致，后续会进行一些选项的协商。

（2）nbd选项的协商（nbd客户端主导）

在nbd选项的协商阶段，当前支持的选项有限，如下：

* NBD\_OPT\_LIST （请求nbd服务端的“导出设备”列表）
* NBD\_OPT\_EXPORT\_NAME （nbd服务端请求某一个“导出设备”）
* NBD\_OPT\_ABORT （结束nbd协商并关闭socket连接）

新格式的协商，在选项的协商阶段，nbd客户端首先要请求nbd服务端的“导出设备”列表（NBD\_OPT\_LIST），在验证自己请求设备在nbd服务端“导出设备”列表中后，再正式发出自己请求的“导出设备”（NBD\_OPT\_EXPORT\_NAME）。

<1> NBD\_OPT\_LIST协商

**nbd客户端开始协商：**

C: 0x49484156454F5054 （8B，发送新格式的协商magic）

C：NBD\_OPT\_LIST （4B,发送选项标识，此处为“请求列表”）

C: length （4B,选项数据长度，NBD\_OPT\_LIST选项无数据，此处为0）

[C: option data] （length长度的选项数据，此处无数据，故不发送）

**nbd服务端回复NBD\_OPT\_LIST协商选项：**

nbd服务端读到选项信息后，遍历“导出设备”列表，逐个发送，例如对一个“导出设备”的回复如下：

S: 0x3e889045565a9LL （8B，nbd服务端的回复magic）

S: NBD\_OPT\_LIST （4B,发送选项标识，此处为“请求列表”）

S： NBD\_REP\_SERVER （4B,发送nbd服务端回复的类型**rep type，具体见后文，此处为**NBD\_REP\_SERVER）

S: length （4B，发送nbd回复的数据长度，数据长度=4 +导出设备字符串长度，这里的4表示下面的name len是占用8B的数据类型）

S: name len （4B， “导出设备”名称的字符串字节长度）

S: data （name len长度的字节，代表“导出设备”名称字符串）

…

…

每一个nbd“导出设备”，nbd服务端都写上面这样一组数据给nbd客户端，nbd客户端读到这些信息，会检查自己要请求的设备是否在这些导出设备之中。

发送完导出设备列表，nbd服务端发送下列ACK回复：

S：0x3e889045565a9LL （8B，nbd服务端的回复magic）

S：NBD\_OPT\_LIST （4B,发送选项标识，此处为“请求列表”）

S: NBD\_REP\_ACK （4B,发送nbd服务端回复的类型）

S: length （4B，发送nbd回复的数据长度，此处为0）

至此，nbd服务端完成了nbd客户端的“导出设备列表”的请求处理。

<2> NBD\_OPT\_EXPORT\_NAME协商

**nbd客户端开始协商：**

C: 0x49484156454F5054 （8B，发送新格式的协商magic）

C：NBD\_OPT\_EXPORT\_NAME （4B,发送选项标识，此处为“请求导出设备”）

C: length （4B,选项数据长度，此处为“导出设备”名称字符串长度）

C: option data （length长度的选项数据，此处为“导出设备”名称字符串）

**nbd服务端回复NBD\_OPT\_EXPORT\_NAME协商选项：**

nbd服务端接受到以上NBD\_OPT\_EXPORT\_NAME的协商数据，回查找“导出设备”，并直接回复如下：

S: size （8B，“导出设备”的容量大小）

S： flags （2B，16位全局标志位&“导出设备”支持的标志位，这些标志位后文介绍）

至此，nbd协商完成。可以看出nbd协商主要对一下进行协商：

* nbd协议（全局标志位来支持）
* nbd导出设备（nbd客户端在协商之后，获取了导出设备的大小）

# 3. nbd协议中的flags标志位

nbd协商过程中支持两类标志位，如下：

* 全局标志位
* nbd导出设备标志位(表示该设备支持的特性，例如：FUA等)

1. 全局标志位(16位)

位0：NBD\_FLAG\_FIXED\_NEWSTYLE位，nbd服务端设置第0位，标志使用fixed-newstyle的协商格式。nbd客户端如果支持同样设置该位；

位1： NBD\_FLAG\_NO\_ZEROES位，只有在传统nbd握手协议中，有该标志位。若客户端设置了此位，则服务端不发送握手协议的124位尾0。

1. nbd设备的标志位(16位)

位0：NBD\_FLAG\_HAS\_FLAGS位，永远为1，表示有flags标记位；

位1：NBD\_FLAG\_READ\_ONLY位，当服务端export的设备只读时被设置；

位2：NBD\_FLAG\_SEND\_FLUSH位，当服务端设备支持 NBD\_CMD\_FLUSH 命令时设置；

位3：NBD\_FLAG\_SEND\_FUA位， 当服务端设备支持NBD\_CMD\_FLAG\_FUA 时被设置；

位4： NBD\_FLAG\_ROTATIONAL位，设置此位让客户端调度I/O访问一个循环设备。

位5： NBD\_FLAG\_SEND\_TRIM位， 服务端支持NBD\_CMD\_TRIM命令时设置。

**4. nbd服务端回复信息类型**

前文描述了部分nbd服务端的回复类型rep type，nbd服务端支持的回复类型包含两类，一类是正常时的回复，另一类是出错回复。具体支持的回复类型如下：

**正常回复**

* NBD\_REP\_ACK（1）

当nbd服务端接受nbd客户端发送过来的协商“选项”或者nbd服务端发送完对某个“选项”的回复时，发送该回复类型。

* NBD\_REP\_SERVER（2）

nbd服务端发送“导出设备”描述信息时，发送该回复类型。

**错误回复**

* NBD\_REP\_ERR\_UNSUP (2^31 + 1)

客户端发送的“选项”，nbd服务端不认识，可能nbd服务端的实现版本太老旧，不支持新版本协议添加的选项。

* NBD\_REP\_ERR\_POLICY (2^31 + 2)

客户端发送的“选项”被服务端识别，在语义上也是有效的，但当前服务端禁止了该“选项”的回复。

* NBD\_REP\_ERR\_INVALID (2^31 + 3)

客户端发送的“选项”被服务端识别，但在语义上是无效的。例如：客户端发送NBD\_OPT\_LIST选项时，带有了非0长度的数据，被服务端认为非法选项协商。

* NBD\_REP\_ERR\_PLATFORM (2^31 + 4)

客户端发送的“选项”在nbd服务端平台上不支持，当前不使用该回复类型。

* NBD\_REP\_ERR\_TLS\_REQD` (2^31 + 5)

服务端需要使用TLS才能进行协商。

nbd的协议是不断变化的，附录中给出nbd 3.14版本的英文描述协议。

**附录**

# 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](https://www.ietf.org/rfc/rfc2119.txt).

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.

### 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).

#### 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`). 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 two message types in the transmission phase: the request,

and the reply. The

transmission phase consists of a series of transactions, where the

client submits requests and the server sends corresponding replies.

The phase continues until

either side terminates transmission; this can be performed cleanly

only by the client.

Replies need not be sent in the same order as requests (i.e., requests

may be handled by the server asynchronously).

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

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`,

and `NBD\_CMD\_TRIM`) that the server

completes (i.e. replies to) prior to processing to 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 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\_FLUSH` 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](https://www.kernel.org/doc/Documentation/block/writeback\_cache\_control.txt)

may be useful.

#### Request message

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

C: 32 bits, 0x25609513, 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`)

#### Reply message

The reply message MUST be sent by the server in response to all

requests (save for `NBD\_CMD\_DISC`). The message looks as

follows:

S: 32 bits, 0x67446698, magic (`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`)

#### 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 `ESHUTDOWN`.

If the client receives an `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](https://tools.ietf.org/html/rfc5246)

as updated by

[RFC6176](https://tools.ietf.org/html/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`

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, 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, it MUST reply with

`NBD\_REP\_ERR\_TLS\_REQD` if TLS has not been initiated.

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.

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.

#### 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 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.

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.

The SELECTIVETLS mode of operation has an implementation problem

in that unless the INFO extension is supported, the client that

does not use TLS may have its access to exports denied without

it being able to ascertain the reason. For instance it may

go into transmission mode using `NBD\_OPT\_EXPORT\_NAME` - which

does not return an error as no options will be denied with

`NBD\_REP\_ERR\_TLS\_REQD`. Further there is no way to remotely

determine whether an export requires TLS, and therefore this

must be initiated between client and server out of band.

### 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. 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).

### 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.

### Status

This functionality has not yet been implemented by the reference

implementation, but was implemented by qemu and subsequently

by other users, so has been moved out of the "experimental" section.

## 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`: defined by the

experimental `WRITE\_ZEROES` [extension](https://github.com/yoe/nbd/blob/extension-write-zeroes/doc/proto.md).

- bit 7, `NBD\_FLAG\_SEND\_DF`: defined by the experimental `STRUCTURED\_REPLY`

[extension](https://github.com/yoe/nbd/blob/extension-structured-reply/doc/proto.md).

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.

- `NBD\_OPT\_ABORT` (2)

The client desires to abort the negotiation and terminate the

session. The server MUST reply with `NBD\_REP\_ACK`.

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.

- `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 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.

See the section on TLS above for further details.

- `NBD\_OPT\_INFO` (6)

Defined by the experimental `INFO` [extension](https://github.com/yoe/nbd/blob/extension-info/doc/proto.md).

- `NBD\_OPT\_GO` (7)

Defined by the experimental `INFO` [extension](https://github.com/yoe/nbd/blob/extension-info/doc/proto.md).

- `NBD\_OPT\_STRUCTURED\_REPLY` (8)

Defined by the experimental `STRUCTURED\_REPLY` [extension](https://github.com/yoe/nbd/blob/extension-structured-reply/doc/proto.md).

- `NBD\_OPT\_BLOCK\_SIZE` (9)

Defined by the experimental `INFO` [extension](https://github.com/yoe/nbd/blob/extension-info/doc/proto.md).

#### 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`

- 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)

Defined by the experimental `INFO` [extension](https://github.com/yoe/nbd/blob/extension-info/doc/proto.md).

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 invalid. For instance,

the client sent an `NBD\_OPT\_LIST` with nonzero data length.

\* `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. Not currently used.

\* `NBD\_REP\_ERR\_TLS\_REQD` (2^31 + 5)

The server is unwilling to continue negotiation unless TLS is

initiated first.

\* `NBD\_REP\_ERR\_UNKNOWN` (2^31 + 6)

Defined by the experimental `INFO` [extension](https://github.com/yoe/nbd/blob/extension-info/doc/proto.md).

\* `NBD\_REP\_ERR\_SHUTDOWN` (2^32 + 7)

The server is unwilling to continue negotiation as it is in the

process of being shut down.

\* `NBD\_REP\_ERR\_BLOCK\_SIZE\_REQD` (2^32 + 8)

Defined by the experimental `INFO` [extension](https://github.com/yoe/nbd/blob/extension-info/doc/proto.md).

### Transmission phase

#### 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`

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`; defined by the experimental `WRITE\_ZEROES`

[extension](https://github.com/yoe/nbd/blob/extension-write-zeroes/doc/proto.md).

- bit 2, `NBD\_CMD\_FLAG\_DF`; defined by the experimental `STRUCTURED\_REPLY`

[extension](https://github.com/yoe/nbd/blob/extension-structured-reply/doc/proto.md).

#### 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 a reply header,

followed immediately by \*length\* bytes

of data, read from \*offset\* bytes into the file, unless an error

condition has occurred.

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.

\* `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 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.

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; a write barrier. 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 len and offset is no

longer needed. A server MAY discard len bytes starting at offset, but

is not required to.

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`.

A client MUST NOT send a trim request unless `NBD\_FLAG\_SEND\_TRIM`

was set in the transmission flags field.

\* `NBD\_CMD\_WRITE\_ZEROES` (6)

Defined by the experimental `WRITE\_ZEROES` [extension](https://github.com/yoe/nbd/blob/extension-write-zeroes/doc/proto.md).

\* 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.

Currently one such message is known: `NBD\_CMD\_CACHE`, with type set to

5, implemented by xnbd.

#### 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:

\* `EPERM` (1), Operation not permitted.

\* `EIO` (5), Input/output error.

\* `ENOMEM` (12), Cannot allocate memory.

\* `EINVAL` (22), Invalid argument.

\* `ENOSPC` (28), No space left on device.

\* `EOVERFLOW` (75), defined in the experimental `STRUCTURED\_REPLY`

[extension](https://github.com/yoe/nbd/blob/extension-structured-reply/doc/proto.md).

\* `ESHUTDOWN` (108), Server is in the process of being shut down.

The server SHOULD return `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 `ENOSPC`. Finally, it

SHOULD return `EPERM` if it receives a write or trim request on a

read-only export.

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

The server SHOULD return `EINVAL` if it receives an unknown command flag. It

also SHOULD return `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 `ENOMEM` if at all possible.

## 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 `WRITE\_ZEROES` [extension](https://github.com/yoe/nbd/blob/extension-write-zeroes/doc/proto.md).

\* The `STRUCTURED\_REPLY` [extension](https://github.com/yoe/nbd/blob/extension-structured-reply/doc/proto.md)

\* The `INFO` [extension](https://github.com/yoe/nbd/blob/extension-info/doc/proto.md).

Implementors of these extensions are strongly suggested to contact the

[mailinglist](mailto:nbd-general@lists.sourceforge.net) in order to help

fine-tune the specifications before committing to a particular

implementation.

Those proposing further extensions should also contact the

[mailinglist](mailto:nbd-general@lists.sourceforge.net). 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.

## 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.