I. Mandrichenko, editor
Fermi National Accelerator Laboratory
May 2004

GridFTP v2 Protocol Description

Editor: Igor Mandrichenko, FNAL

I. Mandrichenko, editor
Fermi National Accelerator Laboratory
May 2004

Status of this Document

This document is a Global Grid Forum Draft. It presents the result of research and development work performed by GridFTP WG based on the analysis of GridFTP v1 [gftp] protocol summarized in GridFTP Protocol Improvements document [gftp-impr] It describes extensions and modifications of GridFTP v1 protocol. GridFTP v1 definition together with this document should be used as definition of GridFTP v2 protocol.

Abstract

GridFTP protocol has become popular data movement tool used to build distributed grid-oriented applications. GridFTP protocol extends FTP protocol defined by RFC959 [rfc959] and other IETF documents by adding certain features designed to improve performance of data movement over wide area network, to allow the application to take advantage of “long fat” communication channels, to help build distributed data handling applications.

Several groups have developed independent implementations of GridFTP v1 [gftp] protocol for different types of applications. The experience gained by these groups uncovered several drawbacks of GridFTP v1 protocol. They were summarized in GGF draft [gftp-impr]. This document proposes modifications of the protocol which are supposed to address majority of issues found.

Contents

eXtended Block (X-Block) Mode

eXtended block mode (or simply X mode) is further development of Extended block mode (E mode) introduced in GridFTP v1.0 standard [gftp]. X-mode is developed to fix certain drawbacks of E mode and add some useful features such as:

- Remove so called unidirectional data transfers limitation
- Add more flexibility in dynamic management of network connections as resource
- Add data integrity verification on the network transport level

Standard RFC959 MODE command should be used with "X" argument to switch into this mode:

```
MODE X CRLF
```

If this mode is supported by the server, it replies with 2xx response.

Basic Ideas

The proposed solution is based on the following ideas:

- Use robust handshake schema to open and close each data channel. This is achieved by introducing "READY", "CLOSE" and "BYE" messages sent in the beginning and at the end of the transfer on the *data channel* in the direction opposite to the data flow;
- Do not use EODC to send number of used data channels. Instead, send EOF message on one or more data channels open between two hosts. The same bit 64 can be used for EOF message;
- Send checksum value along with each data block so that the receiver can verify data integrity and immediately request retransmission of the block if an error is detected. The receiver sends "RESEND" message back to the sender on the same data channel but in the direction opposite to the data flow. Sender and receiver will use OPTS/FEAT mechanism to negotiate concrete type of the checksum prior to the data transfer.

Data Block Format

Data block format is almost the same as for E mode. The only difference is that if data integrity verification is turned on (using OPTS mechanism), each data block is followed by checksum value calculated over the block header and data. Length of the checksum value is determined by previously negotiated checksum type. If checksum calculation is not turned on, then no checksum value is appended to the end of the block. Data block format is:

Field	**Length, bytes**	**Contents**
Descriptor	1	Block descriptor. Bits in the descriptor are: 64 – End of file (EOF) 8 – End of data (EOD) – request to close this data channel 4 – Sender will close this data channel instead of reusing it (?)
Byte count	8	Length of data

Offset	8	Offset of the block in the file
Data	<byte count>, can be 0	Data
Checksum	depends on the type, can be 0	Value of the checksum calculated over header and data

Data Channel Protocol

Proposed data channel protocol is outlined in Fig. 1 and 2 for active and passive sender cases respectively

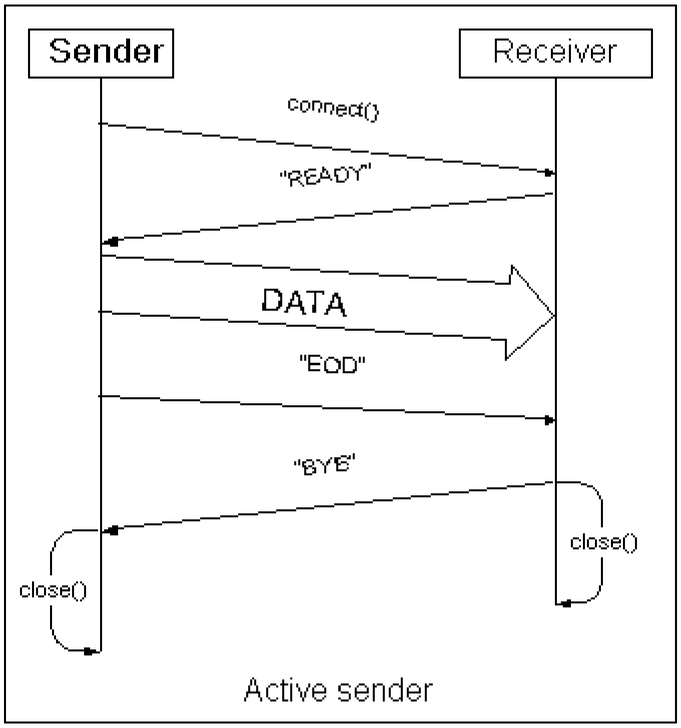

Fig. 1

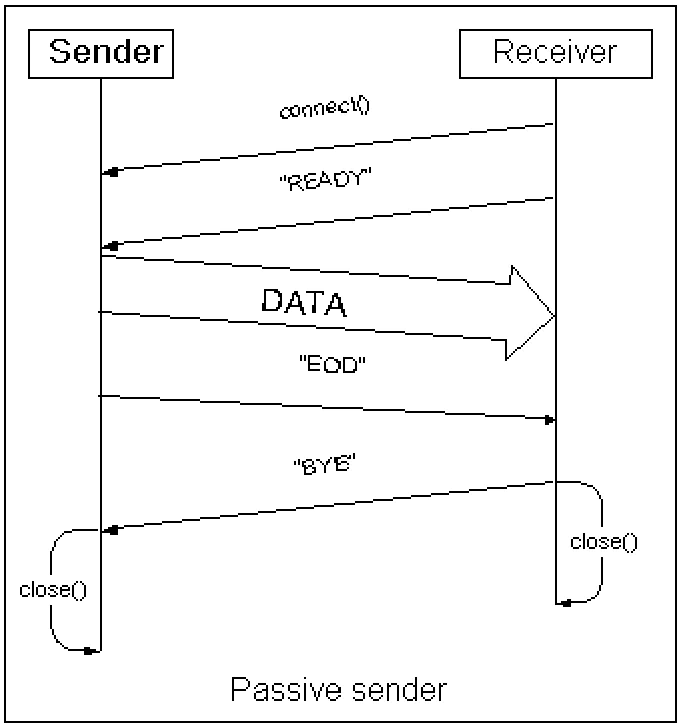

Fig. 2

Opening Data Channel

When passive data receiver accepts new incoming connection on the data socket, it must acknowledge data channel opening with “READY” message sent on newly created data channel socket to data source. Active data sender does not send any data on the data channel until it receives “READY” message. This procedure ensures that active sender and passive receiver hosts use the same number of data channels for the transaction and essentially makes it unnecessary to send channel count in EODC message.

Passive receiver may close new data socket without sending “READY” message or even stop accepting new connections. No data will be lost in such cases because the sender will not send any data before receiving “READY” message.

In case of passive sender and active receiver, there is no need for “READY” message. Sender can immediately begin sending data on newly accepted data channel socket.

In general case of many-to-many striped transfer, active peer must open at least one data channel to each passive peer host. This is necessary to make sure that , even if there is no data to be sent to or received from one of passive hosts, it does not have to wait forever for the transfer to begin.

Closing Data Channel

There are two cases when a data channel may be closed under normal circumstances:

- There is no more data to send on the channel, i.e. the sender has reached end of file
- Either sender or receiver closes one or more data channels in the middle of transfer, e.g. to control bandwidth utilization

Data Channel Closed by the Sender

Before closing a data channel socket (either at the end of the file or in the middle of the transfer), data sender (active or passive) must send EOD message as defined by extended block mode protocol on the data channel. Data receiver acknowledges EOD message with "BYE" message sent back on the data channel. Data sender may choose to wait for "BYE" message to make sure the receiver successfully received all data sent over the data channel. Failure to send "BYE" message to the sender should not be considered an error by the receiver as the sender may choose not to wait for data channel closure confirmations. After sending "BYE" message the receiver may close the data channel or keep it open to reuse in future transfers. Likewise, after receiving "BYE" the sender may choose to close the data channel or keep it open.

Data Channel Closed by the Receiver

If the receiver wishes to close a data channel in the middle of the transfer, it must send "CLOSE" message on the data channel (see Fig. 3). After sending "CLOSE" message, the receiver must continue receiving the data on the data channel until it receives EOD block. After receiving EOD block, the receiver sends "BYE" message on the channel.

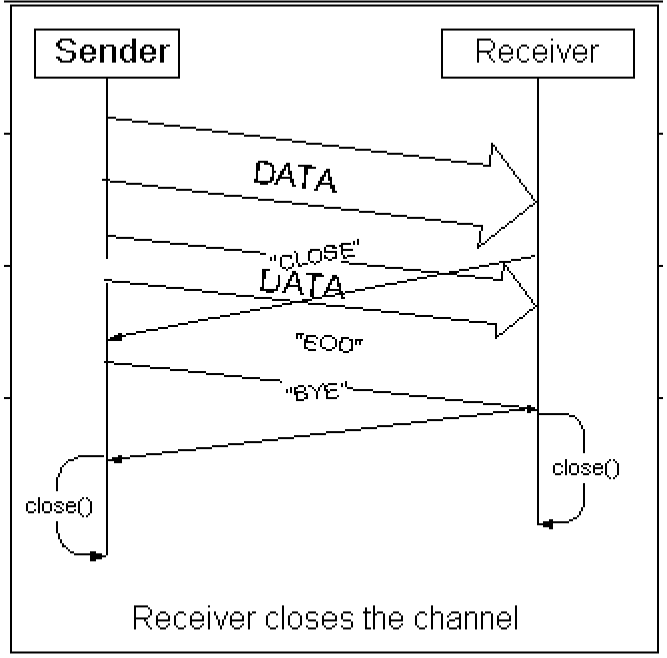

Fig. 3

Data Retransmission

If the receiver detects an error in a block transmission, it can request that the sender resends the block. To request block retransmission the receiver sends "RESEND" command on the same data channel where the erroneous block was received:

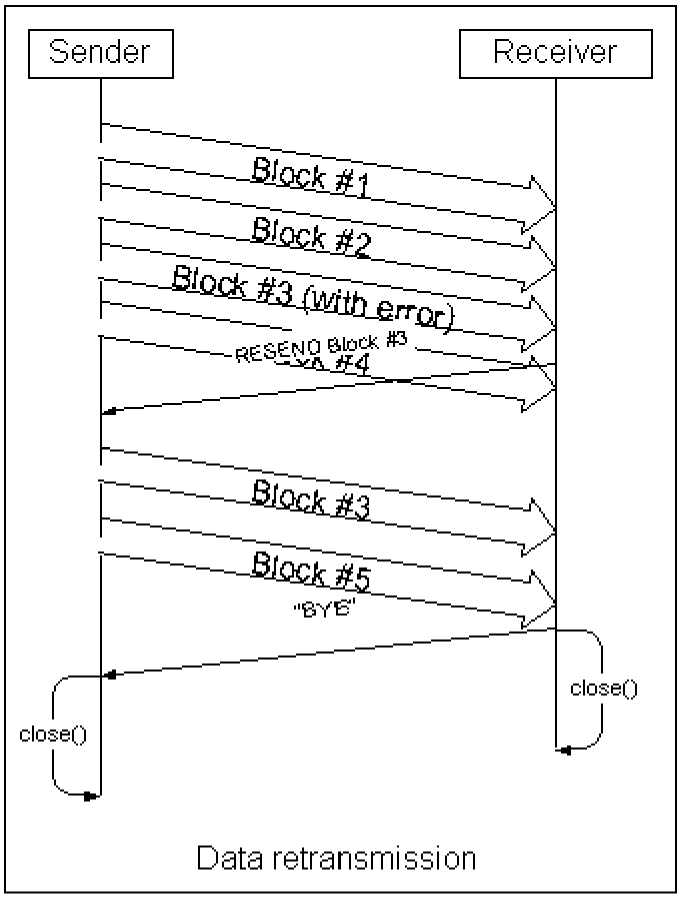

Fig. 4

After detecting a transmission error in one of data blocks and sending "RESEND" command, the receiver should continue receiving data.

It is possible that the bad block will be retransmitted *after* EOD is received. The receiver should not send final "BYE" and close the data channel until it receives all blocks it requested to be retransmitted.

The sender does not necessarily have to resend requested data in single block or even on the same data channel. It may split it into several blocks if necessary.

If the sender does not support resend functionality, it should abort the transfer in any way, for example by closing the data channel without waiting for "BYE" message. The receiver will treat this condition as a transmission error.

Host Pairs

In most general case of striped transfers, data is sent from N *sender hosts* to M *receiver hosts.* Therefore, there are N*M sender-receiver *host* pairs. Each host pair may open zero or more data channels (see Fig. 4).

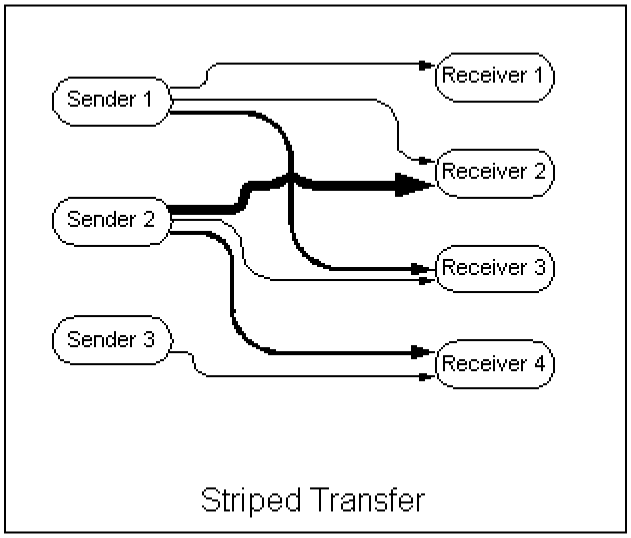

Fig. 4

The protocol allows for dynamic management of such resources as network bandwidth and socket file descriptors by allowing hosts in each pair to open and close data channels dynamically during data transfer without any data loss.

End of File Communication

End of file is signaled by sending (possibly empty) block with EOD and EOF flags set in the block descriptor. The transfer between individual sender and receiver hosts is considered finished successfully after last data channel between them is closed with EOD and the receiver host received at least one EOF block on at least one data channel established between the two hosts. In general case of many-to-many striping, EOF block must be sent on at least one data channel for every sender-receiver pair. EOF communication is described in more details for each type of host.

Active Receiver

After receiving EOF block from a sender host, active data receiver host must not try to open any new data channels to that sender host. It must continue receiving data on all previously open data channels until it receives EOD block on the channel. The receiver host may try to open new data channels to other sender hosts, those it has not received EOF from. In case of striped transfer, the receiver must attempt to open at least one data channel to each sender host. As long as at least one data channel to at least one of sender hosts was open successfully, failure to initiate other channels should not be considered an error by the receiver.

The transfer is considered finished successfully by active receiver after all data channels are closed and at least one EOF block was received from each sender host.

Passive Receiver

Passive receiver host must be receiving data on all open channels until it receives EOD on all channels with EOF on at least one of them. When EOF is received on one of data channels, passive receiver is allowed to stop accepting new data channel connections.

The transfer is considered finished successfully after all open data channels were closed with EOD and at least one EOF block was received by each receiving host.

Passive Sender

Passive sender sends EOD on all open data channels with EOF bit set on at least one data channel per receiver host. In case when it is impossible for the sender to distinguish between connections coming from different receiving hosts, sender may simply send EOF on all open data channels.

The transfer is considered successfully finished when all data was sent and all data channels were closed and the receiver acknowledged all channel closures with "BYE" messages.

Active Sender

Active sender sends EOD on all open data channels and EOF at least on one per receiving host. Sender must not send EOF on any data channel until it receives "READY" on all open channels.

In case of striped transfer, the sender must open at least one data channel to each receiver host and send at least EOD and EOF block to each host even if there is no data to be sent to the host.

The transfer is considered successfully finished when all data was sent and all data channels were closed and the receiver acknowledged all channel closures with "BYE" messages.

Dynamic Resource Allocation

For some applications, it is desired that such resources as network bandwidth, CPU power and open I/O channels (file descriptors) can be dynamically allocated and reallocated between concurrent transfers. Proposed protocol allows for new data channels to be open and closed in the middle of transfer without data loss or corruption. There are provisions for active or passive sender or receiver to open, close or refuse to open new data channel at any time during transfer.

Active Sender

Active sender can control number of open data channels by opening and closing them at any time. The receiver acknowledges new data channel with "READY" message that allows the sender to start using the new channel. At any time active sender can close any data channel after sending EOD block and optionally receiving "BYE" as the acknowledgement.

Active Receiver

Active receiver can control number of open data channels by opening and closing them at any time. The sender may or may not send any data on newly open channel. Once the channel is open by the active receiver, it may close it at any time, but only after sending "CLOSE" message and receiving EOD block. The receiver must keep the channel open and continue receiving data until EOD block is received on the channel.

Passive Sender

Passive sender, naturally, cannot open new data channels, so it cannot increase bandwidth utilization by adding new channels. It can only decrease bandwidth utilization by:

- Closing data socket port thus refusing new data connections
- Closing newly opened data connection before sending any data on the channel
- Sending EOD and closing the data channel
- Sending EOD, waiting for "BYE" and closing the data channel

Existing data channel can be closed at any time after sending EOD block and optionally waiting for "BYE".

Passive Receiver

Passive receiver can decrease bandwidth utilization by:

- Closing data socket port and refusing new data connections
- Closing newly opened data connection before sending "READY"
- Sending "CLOSE" as a request to close the data channel

Once "READY" message was sent to the sender, passive receiver must receive all data sent on the channel until it receives EOD block or the sender closes the channel.

Data Channel Command Syntax

This section describes the format of commands sent by the data receiver on the data channel socket. General format is text terminated with carriage return, linefeed combination or just linefeed:

```
<DC command> = <keyword> [<parameters>] [CR] LF
```

Commands and their parameters are:

```
READY                   (no parameters)
```
The receiver sends READY command after the data channel is open to allow the sender to start sending the data.

```
CLOSE                   (no parameters)
```
Data receiver sends this command when it needs to close the data channel. The receiver must continue receiving data even after sending CLOSE command until it receives EOD block.

```
BYE                     (no parameters)
```
This command is sent by the receiver to allow the sender to close the data channel. It acknowledges that the receiver has successfully received all the data sent on this

data channel. The sender must not close the data channel until it receives "BYE" command. The receiver closes the channel right after it sends "BYE".

```
RESEND      <offset>    <length>
```

The receiver uses this command to request retransmission of a data block. Offset and length are ASCII strings representing decimal numbers for block offset within the file and its length.

GET/PUT Commands

GET and PUT commands are introduced as an alternative to RETR and STOR in order to eliminate the drawback of RFC959 FTP protocol that requires that the server sends the address of data channel socket in response to PASV command before it even knows what file is about to be transferred. The idea is to include all necessary information for the server to be able to initiate the transfer into single command, and have the server use (multiple) 1xx replies to convey such information as data socket address before actual transfer begins.

GET and PUT commands combine functionality of PRET, PORT/PASV, and then STOR and RETR respectively.

Command Syntax

```
GET <parameter> [=<value>]; […] CRLF
PUT <parameter> [=<value>]; […] CRLF
```

<parameter> is a single keyword wihtout spaces in the middle. <value> is optional, it can be either one word or multiple words, terminated with semicolon. Single command can have multiple parameter/value pairs on the same line. The command is terminated with CRLF sequence. This document introduces the following parameters:

- mode – Possible values are "S", "B", "E" or "X" – for Stream, Block, Extended and eXtended block transfer modes. Unless MODE command was sent prior to the GET/PUT, mode argument must be specified.
- port – used by the client to convey data socket address in "active" mode. Parameter value is data port address specification in standard form: "a.b.c.d.e.f" where "a.b.c.d" is IP address of the data host and "e.f" are upper and lower bytes of the port number. If some other port address was previously sent with separate PORT command, that value gets discarded and the value of this argument will be used as the port address.
- pasv – specifies that the transfer should be performed in "passive" mode, and that the server must send "1xx PORT=a.b.c.d.e.f" before actual transfer can begin. This parameter provides the same functionality as RFC959 PASV command. If PASV command was used prior to the GET or PUT command, and the server has already replied to it with some data port address, it still must send the data port address in 1xx response. In this case, the port address sent in 1xx response overrides the one sent previously. Unless PORT or PASV command was sent prior to the GET/PUT command, GET/PUT

command must include either "port" or "pasv" argument. Currently, "1xx PORT=…" is the only 1xx response with documented syntax, which must be recognized by the client. All other 1xx responses are purely informational, and can be ignored.

- cksum – is mode X is used for this transfer, cksum parameter specifies what algorithm should be used for block checksum calculation. Parameter value is the keyword specifying the algorithm. Keyword "NONE" should be used for none. This argument overrides any arrangement previously negotiated using OPTS mechanism for single transfer.
- path – path to the file to be transferred. This is required parameter.

Examples of Communication

Active file retrieval in Stream mode:

```
Client             Server
GET path=/tmp/file.dat;port=34,23,45,12,48,14;mode=s;
                   1xx Data connection established
                   2xx Transfer complete
```

Passive file retrieval in E mode:

```
Client             Server
GET path=/tmp/file.dat;pasv;mode=e;
                   1xx wait
                   1xx wait
                   1xx PORT=134,23,145,2,48,114
                   1xx Data connection established
                   2xx Transfer complete
```

Passive file upload in X mode with MD5 signature calculation:

```
Client             Server
PUT path=/tmp/file.dat;pasv;mode=x;chksum=md5;
                   1xx wait
                   1xx wait
                   1xx PORT=134,23,145,2,48,114
                   1xx Data connection established
                   2xx Transfer complete
```

This is equivalent to the following exchange:

```
Client             Server
OPTS CKSUM NONE
                   2xx OK, will not do checksums any more
PASV
                   2xx PORT=134,23,145,2,48,110
MODE X
                   2xx OK, Will use X mode
PUT path=/tmp/file.dat;pasv;mode=x;cksum=md5;
                   1xx will use MD5 for this single transfer
                   1xx wait for port
                   1xx PORT=134,23,145,2,48,114
                   1xx Data connection established
                   2xx Transfer complete
```

as you can see, the client used OPTS to set default checksum algorithm for the session to NONE, and then overrode this default for this single transfer using "cksum" argument. Client sent PASV command and received some data port address, but then used "pasv" argument with PUR command. Apparently, server discarded previously assigned data port and created new one and sent it with 1xx PORT=… reply.

Explicit EOF Communication in Stream mode

In order to allow the server to detect client shutdown in the middle of file upload and not to treat data socket disconnection as normal end of file, EOF command is introduced. This command has no parameters, so its syntax is simple:

```
EOF CRLF
```

If the EOF command functionality is switched on by previous OPTS command, then the client must issue EOF command after successful file upload before closing control channel:

```
Client            Server
STOR file.dat
                  1xx Opening data socket

(client sends data)

(client closes data channel at the end of file)

                  2xx data transferred

EOF
                  2xx OK, transfer acknowledged
                  (server considers the transfer successful)

(client disconnects control channel)
```

Here is an example of how the server would detect abnormal client disconnection:

```
Client            Server
STOR file.dat
                  1xx Opening data socket

(client sends some data)

(client crashes in the middle of transfer)

                  (server detects end of data socket)
                  2xx data transferred

                  (server detects end of control socket.
                  because EOF was never received,
                  server discards the file as incomplete)
```

In order to maintain backward compatibility with old clients, this functionality is turned off by default. Client turns it on using OPTS mechanism.

I. Mandrichenko, editor
Fermi National Accelerator Laboratory
May 2004

Checksum Transmission

Two commands are introduced for data integrity verification

CKSM

This command is used by the client to request checksum calculation over a portion or whole file. The syntax is:

```
CKSM <algorithm> <offset> <length> <path> CRLF
```

Server executes this command by calculating specified type of checksum over portion of the file starting at the offset and of the specified length. If length is –1, the checksum will be calculated through the end of the file. On success, the server replies with

```
2xx <checksum value>
```

Actual format of checksum value depends on the algorithm used, but generally, hexadecimal representation should be used.

SCKSM

This command is sent prior to upload command such as STOR, ESTO, PUT. It is used to convey to the server that the checksum value for the next uploaded file. At the end of transfer, server will calculate checksum for the received file, and if it does not match, will consider the transfer to have failed. Syntax of the command is:

```
SCKSM <algorithm> <value> CRLF
```

Actual format of checksum value depends on the algorithm used, but generally, hexadecimal representation should be used.

Options and Features Negotiation

OPTS

OPTS command should be used by the client and server to negotiate options for further transfers.

Checksum Calculation

Client negotiates specific checksum calculation algorithm to be used for all subsequent transfers performed during this session with the following OPTS subcommand:

```
OPTS (RETR|STOR|ERET|ESTO) CKSM <algorithm> CRLF
```

Where <algorithm> is the keyword specifying actual checksum calculation algorithm to be used in X-block mode. Suggested keywords are:

- ADLER32 – for Adler32 algorithm
- MD5 – for MD5
- NONE – to specify that no checksum calculation should be performed

If the server supports specified algorithm, it replies with 2xx response. Otherwise – with 5xx or 4xx, in which case, no checksum will be calculated during subsequent transfers.

EOF Communication in Stream Mode

Client requests the server to use explicit EOF notification at the end of Stream mode upload using

```
            OPTS STOR EOF CRLF
```

If the server supports this option, it replies with 2xx response, and explicit EOF confirmation as described above will be turned on for subsequent Stream mode uploads.

FEAT

As described in [rfc2389], FEAT is used by the client to find out what features are supported by the server. This document introduces the following FEAT items:

```
            CKSUM <algorithm>[, …]
            EOF
            MODEX
            GETPUT
```

References

[gftp] Bill Allcock, et all, GridFTP v1.0 Draft
http://www-isd.fnal.gov/gridftp-wg/draft/GridFTPRev3.htm

[gftp-impr] Igor Mandrichenko, GridFTP Protocol Improvements,
http://www.ggf.org/documents/GWD-I-E/GFD-E.021.pdf

[rfc959] IETF RFC959 http://www.ietf.org/rfc/rfc0959.txt?number=959

[rfc2389] IETF RFC2389
http://www.ietf.org/rfc/rfc0959.txt?number=2389

[crc] Timur Perelmutov, GridFTP Data Integrity Verification
Draft proposal
http://home.fnal.gov/~timur/gridftp/index.html