Internet Draft	W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, L. Liming, S. Tuecke
Document:draft GridFTP protocol	ANL
Category: ?	March 2001
Expires: August 2001	Page 1 of 21

GridFTP: Protocol Extensions to FTP for the Grid

1. Status of this Memo

2. Conventions used in this document

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 [5].

3. Abstract

This document fully describes the GridFTP protocol. This protocol combines portions of RFC 959 “FILE TRANSFER PROTOCOL (FTP)”, RFC 2228 “FTP Security Extensions”, RFC 2389 “Feature negotiation mechanism for the File Transfer Protocol”, an IETF draft “FTP Extensions”, and several additional proposed extensions. This combination of features will allow secure, fast, efficient, flexible, and extensible data transfer and data access.

4. Table of Contents

5. Introduction

5.1. Background

In Grid environments, access to distributed data is typically as important as access to distributed computational resources. Distributed scientific and engineering applications require:

• *transfers* of large amounts of data (terabytes or petabytes) between storage systems, and
• *access* to large amounts of data (gigabytes or terabytes) by many geographically distributed applications and users for analysis, visualization, etc.

Unfortunately, the lack of standard protocols for transfer and access of data in the Grid has led to a fragmented Grid storage community. Users who wish to access different storage systems are forced to use multiple protocols and/or APIs, and it is difficult to efficiently transfer data between these different storage systems.

We propose a common data transfer and access protocol called GridFTP that provides secure, efficient data movement in Grid environments. This protocol, which extends the standard FTP protocol, provides a superset of the features offered by the various Grid storage systems currently in use. We chose the FTP protocol because it is the most commonly used protocol for data transfer on the Internet, and of the existing candidates from which to start it comes closest to meeting the Grid's needs. The GridFTP protocol includes the following features:

• Grid Security Infrastructure (GSI) and Kerberos support
• Third-party control of data transfer
• Parallel data transfer
• Striped data transfer
• Partial file transfer
• Automatic negotiation of TCP buffer/window sizes
• Support for reliable and restartable data transfer
• Integrated instrumentation

5.2. Motivation

There are already a number of storage systems in use by the Grid community. These storagesystems have been created in response to specific needs for storing and accessing large datasets.They each focus on a distinct set of requirements and provide distinct services to their clients.

For example, some storage systems (DPSS, HPSS) focus on high-performance access to dataand utilize parallel data transfer streams and/or striping across multiple servers to improveperformance. Other systems (DFS) focus on supporting high-volume usage and utilize datasetreplication and local caching to divide and balance server load. The SRB system connectsheterogeneous data collections and provides a uniform client interface to these repositories, andalso provides metadata for use in identifying and locating data within the storage system. Stillother systems (HDF5) focus on the structure of the data, and provide client support foraccessing structured data from a variety of underlying storage systems.

Unfortunately, most of these storage systems utilize incompatible, an often unpublishedprotocols for accessing data, and therefore require the use of their own client libraries to accessdata. The use of multiple incompatible protocols and client libraries for accessing storageeffectively partitions the datasets available on the grid. Applications that require access to datastored in different storage systems must either choose to only use a subset of storage systems, ormust use multiple methods to retrieve data from the various storage systems.

One approach to breaking down partitions created by these mutually incompatible storagesystem protocols is to build a layered client or gateway which can present the user with oneinterface, but which translates requests into the various storage system protocols and/or clientlibrary calls. This approach is attractive to existing storage system providers because it does notrequire them adopt support for a new protocol. But it also has significant disadvantages,including:

• Performance: Costly translations are often required between the layered client and storage system specific client libraries and protocols. In addition, it can be challenging to efficiently transfer a dataset from one storage system to another.

• Complexity: Building and maintaining a client or gateway that supports numerous storage systems is considerable work. In addition, staying up to date as each storage system independently evolves is very difficult. This is further exacerbated by the need to provide support for multiple client languages, such as C/C++, Java, Perl, Python, shells, etc.

It would be mutually advantageous to both storage providers and users to have a common levelof interoperability between all of these disparate systems: a common—but extensible—underlying data transfer protocol. Storage providers would gain a broader user base, becausetheir data would be available to any client. Storage users would gain access to a broader rangeof storage systems and data. In addition, these benefits can be gained without the performanceand complexity problems of the layered client or gateway approach.

5.3. Requirements

This section defines extensions to the FTP specification STD 9, RFC959, FILE TRANSFER PROTOCOL (FTP) (p.~rfc959) (October 1985)These extensions provide striped data transfer, parallel datatransfer, extended data transfer, data buffer size configuration, and data channel authentication.
Do I want to add Grid definition, talk about FTP shortcomings for Grid, WebDav, etc..

6. Overview

6.1. History

RFC 959 has an excellent review of the RFCs which lead up to it. In this section, we review the RFCs that have corrected, modified, or extended the FTP protocol since RFC 959.
RFC 2228: FTP Security Extensions
RFC 2389: Feature Negotiation for the FileTransfer Protocol
Draft: FTP Extensions

6.2. Terminology

Parallel transfer: From a single data server, splitting file data for transfer over multiple data connections.
Striped transfer: Distributing a file's data over multiple independent data nodes, and transerring over multiple data connections.
Data Node: In a striped data transfer, a data node is one of the stripe destinations returned in the SPAS command, or one of the stripe destinations sent in the SPOR command.
DTP: The data transfer process establishes and manages the data connection. The DTP can be passive or active.
PI: The protocol interpreter. The user and server sides of the protocol have distinct roles implemented in a user-PI and a server-PI.
Features: A response from a server indicating it supports a set of specified functionality. This is in accordance with RFC 2389.
Options: A command to a server defining alternative behavior. This is in accordance with RFC 2389.

6.3. The FTP Model

7. The Extensions

7.1. Summary

This section describes the extensions toRFC 959. These extensions consist of commands, options, features, and a new mode. The commands are as follows:

SPAS:	Striped Passive.	This enables striping and parallelism.
SPOR:	Striped Port.	This enables striping and parallelism.
ERET:	Extended Retrieve.	This enables server side processing on a retrieved file.
ESTO:	Extended Store.	This enables server side processing on a stored file.
SBUF:	Set TCP Buffer Size:	Allows the TCP buffer size to be set explicitly.
ABUF:	Auto Negotiate TCP Buffer Size.	Automatically determines and sets the TCP buffer size.
DCAU:	Data Channel Authentication.	Enables authentication on the data connection.

Feature resposes have been defined so that a client may determine if an implementation supports these commands. A new mode, EBLOCK, or extended block mode has been defined to support parallel and striped transfers. Also, new options were defined for the RETR command that allows parallelism and striping information to be specified.

7.2. Commands

7.2.1. Striped Passive (SPAS)

This extension is used to establish a vector of data socket listeners for each stripe of the data. To simplify interaction with the parallel data transfer extensions, the SPAS MUST only be done on a control connection when the data is to be stored onto the file space served by that control connection. The SPAS command request the FTP server to "listen" on a data port (which is not the default data port) and to wait for one or more data connections, rather than initiating a connection upon receipt of a transfer command. The response to this command includes a list of host and port addresses the server is listening on. This command MUST always be used in conjunction with the extended block mode.

7.2.1.1. Syntax

The syntax of the SPAS command is:

```
    spas = "SPAS" <CRLF>
```

7.2.1.2. Responses

The server-PI will respond to the SPAS command with a 229 reply giving the list of host-port strings for the remote server-DTP or user-DTP to connect to.

```
    spas-response = "229-Entering Striped Passive Mode" CRLF
```

```
                    1*(<SP> host-port CRLF)
                    229 End
```

Where the command is correctly parsed, but the server-DTP cannot process the SPAS request, it must return the same error responses as the PASV command.

7.2.1.3. OPTS for SPAS

There are no options in this SPAS specification, and hence there is no OPTS command defined.

7.2.2. Striped Data Port (SPOR)

This extension is to be used as a complement to the SPAS command to implement striped third-party transfers. To simplify interaction with the parallel data transfer extensions, the SPOR MUST only be done on a control connection when the data is to be retrieved from the file space served by that control connection for a third-party transfer. This command MUST always be used in conjunction with the extended block mode.

7.2.2.1. Syntax

The syntax of the SPOR command is:

```
SPOR 1*(<SP> <host-port>) <CRLF>
```

The host-port sequence in the command structure MUST match the host-port replies to a SPAS command.

7.2.2.2. Responses

The server-PI will respond to the SPOR command with the same response set as the PORT command described in the ftp specification.

7.2.2.3. OPTS for SPOR

There are no options in this SPOR specification, and hence there is no OPTS command defined.

7.2.3. Extended Retrieve (ERET)

The extended retrieve extension is used to request that a retrieve be done with some additional processing on the server. This command an extensible way of providing server-side data reduction or other modifications to the RETR command. This command is used in place of OPTS to the RETR command to allow server side processing to be done with a single round trip (one command sent to the server instead of two) for latency-critical applications.

7.2.3.1. Syntax

The syntax of the ERET command is

```
ERET <SP> <retrieve-mode> <SP> <filename>

retrieve-mode ::= P <SP> <offset> <SP> <size>
offset ::= 64 bit integer
size ::= 64 bit integer
```

The retrieve-mode defines behavior of the extended-retrieve mode. There is one mode defined by this specification, but others may be added later.

7.2.3.2. Extended Retrieve Modes

Partial Retrieve Mode (P): A section of the file will be retrieved from the data server. The section is defined by the starting offset and extent size parameters.

7.2.3.3. Responses

The response to the ERET command should be per RFC 959 for the RETR command.

7.2.3.4. Options

There are no options in this ERET specification, and hence there is no OPTS command defined.

7.2.4. Extended Store (ESTO)

The extended store extension is used to request that a store be done with some additional processing on the server.

7.2.4.1. Syntax

The format of the ESTO command is

```
ESTO <SP> <store-mode> <filename>

store-mode ::= A <SP> <offset>
```

The store-mode defines the behavior of the extended store. There is one mode defined by this specification, but others may be added later.

7.2.4.2. Store Modes

Adjusted store (A): The data in the file is to stored with **offset** added to the file pointer before storing the blocks of the file. In extended block mode, this value is added to the offset in the extended block header, and may be a positive or negative value. In block, compressed, or stream modes modes, the offset is added to the implicit offset of 0 for the beginning of the data.

7.2.4.3. Responses

The response to the ESTO command should be per RFC 959 for the STOR command.

7.2.4.4. Options

There are no options in this ERET specification, and hence there is no OPTS command defined.

7.2.5. Set Buffer Size (SBUF)

This extension adds the capability of a client to set the TCP buffer size for subsequent data connections to a value. This replaces the server-specific commands SITE RBUFSIZE, SITE RETRBUFSIZE, SITE RBUFSZ, SITE SBUFSIZE, SITE SBUFSZ, and SITE BUFSIZE

7.2.5.1. Syntax

The syntax of the SBUF command is:

```
sbuf = SBUF <SP> <buffer-size>

buffer-size ::= <number>
```

The buffer-size value is the TCP buffer size in bytes. The TCP window size should be set accordingly by the server.

7.2.5.2. Response Codes

If the server-PI is able to set the buffer size state to the requested buffer-size, then it will return a 200. Note: Even if the SBUF is accepted by the server, an error may occur later when the data connections are actually created.

7.2.6. Auto-Negotiate Buffer Size (ABUF)

This extension adds the capability to automatically determine and set the optimal TCP buffer size for data connections.

7.2.6.1. Syntax

The syntax of the ABUF command is:

```
ABUF <SP> <autobuffer-mode> <CRLF>

autobuffer-mode = A <initial-buffer> <minimum-buffer>
                    <maximum-buffer> <test-msg-size>
initial-buffer ::= <number>
minimum-buffer ::= <number>
maximum-buffer ::= <number>
test-msg-size ::= <number>
```

The autobuffer-mode defines behavior of the ABUF command. There is one mode defined by this specification, but others may be added later.

7.2.6.2. Buffer Auto-Negotiation Modes

Negotiate based on a RTT and BW test (A): A new data connection will be established using the standard PORT/PASV method. This command will close any previously-opened data ports on the FTP server(s) involved in the experiment. After a network experiment is run, the buffer sizes on each server will be set to the computed buffer size value. The value will be returned using the same responses as the SBUF message. The experiment will be run with the buffer size of the data connection set to initial-buffer. Once the experiment is complete, the buffer size will be set to the computed optimal buffer size, restricted to the range [minimum-buffer, maximum-buffer]. The proposed data channel protocol for this style of buffer negotiation is

1. open data channel with <start> buffer size
2. send a 1 byte message to the PASV side of the connection.
3. when the message arrives at PASV, it will send 1 byte response
4. when response arrives at PORT, it will send <test-size> message
5. when message arrives at PASV, it will send 1 byte response
6. when response arrives, PORT will send ASCII string
   <round-trip-time-in-usec> <SP> <bandwidth-in-bytes-per-second>
7. Both sides of the socket close the connection.

7.2.7. Data Channel Authentication (DCAU)

This extension provides a method for specifying the type of authentication to be performed on FTP data channels. This extension may only be used when the control connection was authenticated using RFC 2228 Security extensions.

7.2.7.1. Syntax

The format of the DCAU command is

```
DCAU <SP> <authentication-mode> <CRLF>

authentication-mode ::= <no-authentication>
                      | <authenticate-with-self>
                      | <authenticate-with-subject>

no-authentication ::= N
authenticate-with-self ::= A
```

```
authenticate-with-subject ::= S <subject-name>

subject-name ::= string
```

7.2.7.2. Authentication Modes

- No authentication (**N**)
  No authentication handshake will be done upon data connection establishment.
- Self authentication (**S**)
  A security-protocol specific authentication will be used on the data channel. The identity of the remote data connection will be the same as the identity of the user which authenticated to the control connection.
- Subject-name authentication (**S**)
  A security-protocol specific authentication will be used on the data channel. The identity of the remote data connection MUST match the supplied **subject-name** string.

The default data channel authentication mode is S for FTP sessions which are RFC 2228 authenticated. If the security handshake fails, the server must return the error response 432 (Data channel authentication failed).

7.3. Features

RFC 2389 provides for the addition of the FEAT and OPTS commands to allow for the negotiation of feature sets. The following new feature names are to be included in the FTP server's response to FEAT if it implements the following sets of functionality

PARALLEL: The server supports the SPOR and SPAS commands, the RETR options, and extended block mode as described in this document.
ESTO: The server implements the ESTO command as described in this document.
ERET: The server implements the ERET command as described in this document.
SBUF: The server implements the SBUF command as described in this document.
ABUF: The server implements the ABUF command as described in this document.
DCAU: The server implements the DCAU command as described in this document, including the requirement that data channels are authenticated by default, if RFC 2228 authentication is used to establish the control channel.
PIPE: The server supports pipelining (i.e. queueing) of commands.

7.4. Mode

7.4.1. Extended Block Mode

The striped and parallel data transfer methods described above require an extended transfer mode to support out-of-sequence data delivery, and partial data transmission per data connection. The extended block mode described here extends the block mode header to provide support for these as well as large blocks, and end-of-data synchronization. Clients indicate that they want to use extended block mode by sending the command:

MODE <SP> E <CRLF>

on the control channel before a transfer command is sent. The structure of the extended block header is:

```
Extended Block Header

+----------------+-------/-----------+------/------------+
| Descriptor     |    Byte Count     |    Offset Count   |
|         8 bits |        64 bits    |           64 bits |
+----------------+-------/-----------+------/------------+
```

The descriptor codes are indicated by bit flags in the descriptor byte. Six codes have been assigned, where each code number is the decimal value of the corresponding bit in the byte.

```
 Code     Meaning

  128     End of data block is EOR (Legacy)
   64     End of data block is EOF
   32     Suspected errors in data block
   16     Data block is a restart marker
    8     End of data block is EOD for a parallel/striped transfer
  4    Sender will close the data connection
```

With this encoding, more than one descriptor coded condition may exist for a particular block. As many bits as necessary may be flagged. Some additional protocol is added to the extended block mode data channels, to properly handle end-of-file detection in the presence of an unknown number of data streams.

- When no more data is to be sent on the data channel, then the sender will mark the last block, or send a zero-length block after the last block with the EOD bit (8) set in the extended block header.
- After receiving an EOD the data connection can be cached for use in a subsequent transfer. To signifiy that the data connection will be closed the sender sets the close bit (4) in the header on the last message sent.
- The sender communicates end of file by sending an EOF message to all servers receiving data. The EOF message format follows.

Extended Block EOF Header +----------------+-------/--------+------/---------------+ | Descriptor | unused | EOD count expected | | 8 bits | 64 bits | 64 bits | +----------------+-------/--------+------/---------------+

EOF Descriptor. The EOF header descriptor has the same definition as the regular data message header described above.

EOD Count Expected. This 64 bit field represents the total number of data connections that will be established with the server receiving the file. This number is used by the receiver to determine it has received all of the data. When the number of EOD messages received equals the number represented by the "EOD Count Expected" field the receiver has hit end of file.

Simply waiting for EOD on all open data connections is not sufficient. It is possible that the receiver reads an EOD message on all of its open data connects while an additional data connection is in flight. If the receiver were to assume it reached end of file it would fail to receive the data on the in flight connection.

To handle EOF in the multi-striped server case a 126 response has been introduced. When receiving data from a striped server a client makes a control connection to a single host, but several host may create several data connections back to the client. Each host can independently decide how many data connections it will use, but only a single EOF message may be sent to back to the client, therefore it must be possible to aggregate the total number of data connections used in the transfer across the stripes. The 126 response serves this purpose.

The 126 is an intermediate response to RETR command. It has the following format.

"126" <SP> 1*(count of data connections)

Several "Count of data connections" can be in a single reply. They correspond to the stripes returned in the response to the SPAS command.

Discussion of protocol change to enable bidirectional data channels brought up the following problem if doing bidirectional data channels

If the client is pasv, and sending to a multi-stripe server, then the server creates data connections connections; since the client didn't do SPAS, it cannot associate HOST/PORT pairs on the data connections with stripes on the server (it doesn't even know how many there are). it cannot reliably determine which nodes to send data to. (Becomes even more complex in the third-party transfer case, because the sender may have multiple stripes of data.) The basic problem is that we need to know logical stripe numbers to know where to send the data.

EOF Handling in Extended Block Mode

If you are in either striped or parallel mode, you will get exactly one EOF on each SPAS-specified ports (stripes). Hosts in extended block mode must be prepared to accept an arbitrary number of connections on each SPOR port before the EOF block is sent.

7.5. Options

Options to RETR

The options described in this section provide a means to convey striping and transfer parallelism information to the server-DTP. For the RETR command, the Client-FTP may specify a parallelism and striping mode it wishes the server-DTP to use. These options are only used by the server-DTP if the

retrieve operation is done in extended block mode. These options are implemented as RFC 2389 extensions.
The format of the RETR OPTS is specified by:

```
     retr-opts        = "OPTS" <SP> "RETR" [<SP> option-list] CRLF
  option-list   = [ layout-opts ";" ] [ parallel-opts ";" ]
  layout-opts   = "StripeLayout=Partitioned"
                | "StripeLayout=Blocked;BlockSize=" <block-size>
  parallel-opts = "Parallelism=" <starting-parallelism> ","
                             <minimum-parallelism>  ","
                             <maximum-parallelism>

  block-size          ::= <number>
  starting-parallelism ::= <number>
  minimum-parallelism  ::= <number>
  maximum-parallelism  ::= <number>
```

Layout Options

The layout option is used by the source data node to send sections of the data file to the appropriate destination stripe. The various StripeLayout parameters are to be implemented as follows:

Partitioned

A partitioned data layout is one where the data is distributed evenly on the destination data nodes.Only one contiguous section of data is stored on each data node. A data node is defined here a single host-port mentioned in the SPOR command

Blocked

A blocked data layout is one where the data is distributed in round-robin fashion over the destination data nodes. The data distribution is ordered by the order of the host-port specifications in the SPOR command. The **block-size** defines the size of blocks to be distributed.

Parallelism Options

The parallelism option is used by the source data node to control how many parallel data connections may be established to each destination data node. This extension option provides for both a fixed level of parallelism, and for adapting the parallelism to the host/network connection, within a range. If the **starting-parallelism** option is set, then the server-DTP will make **starting-parallelism** connections to each destination data node. If the **minimum-parallelism** option is set, then the server may reduce the number of parallel connections per destination data node to this value. If the **maximum-parallelism** option is set, then the server may increase the number of parallel connections to per destination data node to at most this value.

8. Declaritive Specifications

8.1. Minimum Implementation

The extensions described in this document are designed to provide a data access and transport mechanism that is secure, fast, reliable, flexible, and extensible. However, not all applications require all these features and it is desireable that they still be able to be “part of the grid”. This means, that in fact, none of the extensions described here are required for the minimum implementation. Our recommendation for a minimum implementation is as recommended in RFC 959 with the addition of the RFC 2228 Security extensions. Clear text passwords simply are no longer acceptable. We have listed the details below:

Per RFC 959:

TYPE:	ASCII Non-print
MODE:	Stream
STRUCTURE:	File, Record
COMMANDS:	USER, QUIT, PORT, TYPE, MODE, STRU,
COMMANDS:	RETR, STOR, NOOP (these commands with default values only)

The default values for transfer parameters are:

TYPE:	ASCII Non-print
MODE:	Stream
STRU:	File

All hosts must accept the above as the standard defaults.

Per RFC 2228:
COMMANDS:AUTH , ADAT, MIC, CONF, ENC

8.2. Recommended Implementation

In order to gain all the benefits and to fully take advantage of the grid, we recommend the following for a full featured implementation. Note that there are some commands, modes, features, etc, that are being deprecated as they are seldom implemented and in some cases simply no longer apply:

RFC 959, FILE TRANSFER PROTOCOL (FTP), J. Postel, R. Reynolds (October 1985)
Commands used by GridFTP

USER	PASS	ACCT	CWD	CDUP	QUIT
REIN	PORT	PASV	TYPE	MODE	RETR
STOR	STOU	APPE	ALLO	REST	RNFR
RNTO	ABOR	DELE	RMD	MKD	PWD
LIST	NLST	SITE	SYST	STAT	HELP
NOOP					

- Features used by GridFTP
    - Type:ASCII, Image
    - Mode: Stream, EBlock
    - Structure: File structure

- RFC 2228, FTP Security Extensions, Horowitz, M. and S. Lunt (October 1997)
    - Commands used by GridFTP
        - AUTH
        - ADAT
        - MIC
        - CONF
        - ENC
    - Features used by GridFTP
        - GSSAPI authentication

- RFC 2389, Feature negotiation mechanism for the File Transfer Protocol, P. Hethmon , R. Elz (August 1998)
    - Commands used by GridFTP
        - FEAT
        - OPTS
    - Features used by GridFTP
- FTP Extensions, R. Elz, P. Hethmon (September 2000)
    - Commands used by GridFTP
        - SIZE
    - Features used by GridFTP
        - Restart of a stream mode transfer

8.3. Sequencing of Commands and Replies

9. Security Considerations

Security is one of the key considerations for the grid and what makes FTP as defined by RFC 959 unacceptable for use today. GridFTP was designed with security in mind from the start and was, in fact, the driving force that started this effort. While we will retain anonymous FTP, and support for the USER and PASS commands, we strongly discourage their use, particularly PASS. We incorporate the GSS API extensions defined in RFC 2228, with both GSI and Kerberos bindings.

10. Known Issues

10.1. Unidirectional data transfer in EBLOCK mode:

Currently if you are in eblock mode, PASV must be paired with STOR, and PORT must be paired with RETR. In other words, the direction of the connection on the data channels must go from the sending (RETR) to the receiving (STOR) side. While this works, it raises the following issues:

1) The current FTP protocol does not have this restriction.

2) Firewalls: It can help you traverse some firewalls more easily to be able to set the direction to connect out from behind the firewall.

3) A mixture of partial gets/puts currently requires two control channel connections. This is less than ideal.

10.2. Order dependency between PASV/SPAS and STOR/RETR:

RETR is currently not a problem in EBLOCK mode, but will be if #1 is fixed, and is if you are using stream mode. We believe that it will be necessary to support partitioning and load balancing amongst FTP servers in large installations. A site should be able to setup an arbitrary number of FTP servers, place file across them in some manner, and then have a load balancing front end in front of all this that redirects connections to the appropriate server. This is similar to the redirect capability of HTTP.

Both points 1 and 2 share the same basic problem, which is the response sequence to the PASV command. In order to perform a redirection, I need to know what file, URL, etc. is needed. In the existing FTP protocol, this information is obtained via the STOR or RETR commands. The problem is that the client will have previously sent a PASV command to find out the ip/port to connect to, but we don't know what ip/port to connect to until the STOR/RETR issued. In other words, we currently have a circular dependency. In order to issue the STOR/RETR, you must first issue a PASV command, but the server does not know what to return for the PASV until it receives the STOR/RETR. Note that always using PORT is not a solution, because if you are doing a 3rd party transfer, one side must use PASV.

Two solutions to this problem have been proposed:

A new response should be defined for the PASV command: This response would a “delayed IP/Port”. This response would indicate that the IP/Port information would be returned as an intermediate response in the

STOR/RETR command. Then when the STOR/RETR command is received, a decision can be made about the ultimate host to form the data connection with based on the filename/URL provided. With this information now available, IP/Port information can be provided and then normal STOR/RETR behavior can follow.

Redefine the state machine to allow PORT/PASV and STOR/RETR in any pairs, but unordered: Currently, the state machine is such that the STOR/RETR command knows that the data connection MUST already exist and therefore it can immediately begin transmission. If instead the state machine were redefined so that a state of "OK TO BEGIN TRANSMISSION" were defined and that state was reached by receiving one each of PORT/PASV and STOR/RETR, then there would no longer be an ordering restriction.

10.3. Reuse of eblock data channels:

In order to amoritize the overhead of authentication and channel setup, it is desireable to reuse existing data channels if multiple transfers are to be made to the same host. The issue is whether or not modifications are required to the protocol itself in order to support this, perhaps as OPTS or FEATURES.

10.4. Pipelining of commands & reuse of eblock data channels:

In order to get maximal efficiency when issuing multiple RETR/ERET/STOR/ESTO commands, in addition to reusing the data channels, you would also want to pipeline the issuing of commands. That is, for example, while the data for one RETR is still being sent by the server, the client could issue another RETR command. This would, in theory, allow the server to keep the data channel pipes full. As soon as it finishes sending the data for the first RETR, it can immediately start sending the data for the next RETR, at the same time as it sends the control channel response to the first RETR.

The issue here is making sure that the receiving end can figure out where the data for the first RETR ends, and the second begins. The obvious complication is when there are multiple data channels.

10.5. Support for file info:

Do we wish to incorporate direct support for the following file info: File existence, file size, various date information, cryptographic checksum

10.6. Support for disk resource management:

How will we incorporate features such as advanced reservation, space allocation, pinning, delivery estimation, time to live estimation, etc..

11. Appendix I

Restarting

In general, opaque restart markers passed via the block header should not be used in extended block mode. Instead, the destination server should send extended data marker responses over the control connection, in the following form:

```
    extended-mark-response = "111" <SP> "Range Marker" <SP> <byte-ranges-list>

  byte-ranges-list      = <byte-range> [ *("," <byte-range>) ]
  byte-range            = <start-offset> "-" <end-offset>

  start-offset        ::= <number>
  end-offset          ::= <number>
```

The byte ranges in the marker are an incremental set of byte ranges which have been stored to disk by the data server. The complete restart marker is a concatenation of all byte ranges received by the client in 111 responses.

The client MAY combine adjacent ranges received over several range responses into any number of ranges when sending the REST command to the server to restart a transfer.

For example, the client, on receiving the responses:

```
111 Range Marker 0-29
111 Range Marker 30-89
```

may send, equivalently,

```
REST 0-29,30-89
REST 0-89
REST 30-59,0-29,60-89
```

to restart the transfer after those 90 bytes have been received.

The server MAY indicate that a given range of data has been received in multiple subsequent range markers. The client MUST be able to handle this. For example:

```
111 Range Marker 30-59
111 Range Marker 0-89
```

is equivalent to

```
111 Range Marker 30-59
111 Range Marker 0-29,60-89
```

Similarly, the client, if it is doing no processing of the restart markers, MAY send redundant information in a restart.

Performance Monitoring

In order to monitor the performance of extended block mode transfer, an additional preliminary reply MAY be transmitted over the control channel. This reply is of the form:

```
    extended-perf-response  =  "112-Perf Marker" <SP> <timestamp> CRLF
                         *( perf-line CRLF )
                         "112 End" CRLF
  perf-line              = <SP> "AllTransferred:" <SP> <bytes-transferred>
                       | <SP> "AllConnections:" <SP> <num-data-connections>
                       | <SP> "AllThroughput:"  <SP> <throughput>
                       | <SP> "StripeTransferred:" <SP> <stripe-num>
                                   <SP> <bytes-transferred>
                       | <SP> "StripeConnections:" <SP> <stripe-num>
```

```
                                        <SP> <num-data-connections>
                    | <SP> "StripeThroughput:" <SP> <stripe-num>
                                        <SP> <throughput>
   throughput          ::= 1*<digit> [ "." 1*<digit> ]
```

The performance marker can contain any subset of the perf-line facts about the current performance state. (The types of performance data can/should be extended; perhaps an OPTS STOR should be implemented to choose what type of performance data is interesting to an application... eventually).
All perf-line facts represent an instantaneous state of the transfer at the given timestamp. The meaning of the facts are

- AllTransferred - Total amount of data transferred for this file transfer (in bytes)
- AllConnections - Total number of data connections used for this transfer
- AllThroughput - Aggregate throughput of the transfer over all data stripes (in bytes/second)
- StripeTransferred - Per-stripe amouont of data transferred
- StripeConnections - Per-stripe number of data connections used for this transfer
- StripeThroughput - Per-stripe throughput of the transfer (in bytes/second)

12. References

[1] Postel, J. and Reynolds, J., " FILE TRANSFER PROTOCOL (FTP)", STD 9, RFC 959, October 1985.
[2] Hethmon, P. and Elz, R., " Feature negotiation mechanism for the File Transfer Protocol", RFC 2389, August 1998.
[3] Horowitz, M. and Lunt, S., " FTP Security Extensions", RFC 2228, October 1997.
[4] Elz, R. and Hethom, P., " FTP Extensions", IETF Draft, September 2000.