

Toward Seamless Grid Data Access: Design and Implementation of GridFTP on .NET

Jun Feng, Lingling Cui, Glenn Wasson, and Marty Humphrey

Department of Computer Science, University of Virginia, Charlottesville, VA 22904

Abstract—To date, only Linux-/UNIX-based hosts have been participants in the Grid vision for seamless data access, because the necessary Grid data access protocols have not been implemented on Windows. As part of our larger effort at the University of Virginia to make the Windows platform a first-class participant in all aspects of Grids, this paper describes our experiences and lessons learned while implementing GridFTP on the Microsoft .NET Framework. Our implementation not only supports major extensions of GridFTP v1, it also uniquely implements some features of GridFTP v2 and introduces a new transfer mode specifically designed for transfer of large collection of small files. Our measured performance is comparable to GT4 GridFTP on both single and parallel streams transfer and more efficient than GT4 GridFTP on directory tree transfer. We also identify issues specific to the .NET Framework/Windows platform with regard to security and we identify limitations of current GridFTP protocol. To our knowledge, the work described in this paper is the first comprehensive and evaluated implementation of GridFTP on .NET.

I. INTRODUCTION

The vision for Grid Computing includes easy remote access to a broad variety of computational resources (subject to the appropriate security mechanisms and policies). This vision also includes that Grid users and computations acting on their behalf be able to locate, access, and integrate all the relevant data to enable successful completion of their activities. However, to date, only Linux-/UNIX-based hosts have been participants in the Grid vision for data access because the necessary Grid data access protocols have not been implemented on Windows.

At the University of Virginia, the overall theme of one of our research projects is to make the Windows platform a first-class participant in the Grid. Our goal is to build the necessary tooling and protocol support (e.g., WSRF.NET [1], GRAM support via WSRF.NET [2]) to integrate, leverage, and expose Windows-based capabilities so that there is seamless access from the desktop to high-end computing platforms, and back (we have also developed support for Grids on PocketPC-class machines, Mobile OGSI.NET [3]). We are currently designing, developing, and deploying this software in the University of Virginia Campus Grid (UVACG [4]), which will ultimately consist of hundreds of Windows machines (WSRF.NET) interoperating with Linux-/UNIX-based systems (Globus Toolkit v4 [5]) in a seamless, integrated Grid.

In previous research in which we leveraged WSRF for remote execution on .NET platforms in Grid environments [6], we had to rely on SOAP over TCP for the stage in/out for job execution service. Although it works in our environment, it

does not interoperate with the Globus-based Grids, in which the data transfer is taken care by the GridFTP protocol [7] and the Reliable File Transfer (RFT) service [8].

To address this problem, we have designed and implemented the GridFTP protocol on the .NET platform. Our implementation includes a GridFTP Windows service, a GridFTP client which can work in both interactive and batch mode, and a WSRF.NET-based file transfer service. Our implementation not only supports the major extensions of GridFTP v1 [7], it also implements some features of GridFTP v2 [9] and introduces a new transfer mode specifically designed for the transfer of large collection of small data objects. The performance evaluation shows our implementation is comparable to GT4 GridFTP on both single and parallel streams transfers, and significantly better than GT4 GridFTP on directory tree transfer. For example, our measured performance in transferring a large directory consisting of many small files required 69 seconds while taking 82 seconds for GT4.

The rest of this paper is organized as follows. Section II gives a brief introduction to the GridFTP protocol. Section III presents the design and implementation of our .NET GridFTP server and client. Section IV is the performance evaluation. Section V describes the related work. Section VI contains a brief discussion and the conclusion.

II. GRIDFTP PROTOCOL

GridFTP [7] is a data transfer protocol for accessing distributed data on the Grid. Its first—and still major—implementation is in the Globus Toolkit. It is based on the RFC 959 “File Transfer Protocol”, RFC 2228 “FTP Security Extensions”, and RFC 2389 “Feature Negotiation Mechanism for the File Transfer Protocol”. The GridFTP protocol is optimized for high-performance, secure, and reliable data transfer in high-bandwidth wide-area networks. The following is a summary of the key GridFTP features:

Parallel data transfer: multiple TCP streams to improve transfer rate, supported through FTP command extensions and data channel extensions.

Authentication, data integrity and confidentiality: Generic Security Service (GSS)-API authentication on the control channel (RFC 2228) and data channel (GridFTP extensions). It also supports user-controlled levels of data integrity and/or confidentiality on the data channel.

Third party control of data transfer: between two storage servers. Third party transfer is very common on Grids, and is the foundation of the Reliable File Transfer service.

Striped data transfer: partition and access data across multiple servers to utilize the aggregated bandwidth.

Partial file transfer: transfer of arbitrary regions of a file, not supported in standard FTP.

Reliable data transfer: restart and performance markers can be sent by the sender to the receiver on the data channel for reliability and instrumentation purposes. This makes it possible to handle transient network failures and server outages.

Control of TCP buffer size: both manual and automatic configuration of optimal TCP buffer size at the receiver side to achieve maximum bandwidth with TCP/IP. Current implementations tend to only support manual settings of TCP buffer size.

The GridFTP v1 defines following two transfer modes. *Stream mode* (Mode S) is the regular FTP transfer mode. In this mode, the data is transmitted as a stream of bytes. There is no FTP encoding in this mode to indicate the end of transfer; instead, the sending host must close the data connection to implicitly end the transfer. Due to this, stream mode is inherently unreliable and it is not possible to reuse the data connection in this mode. *Extended block mode* (Mode E) is the extension made by GridFTP protocol v1 to the original block mode in RFC 959. The extended block mode extends the block mode header to provide support for striped and parallel data transfer, as well as large blocks, and end-of-data synchronization. In this mode, each data block is preceded by a fixed-size header. Inside this header, there are descriptor bits to indicate the nature of this data block, the size of the following data block, and the offset in the original file to which this data block corresponds. The end of transfer in this mode is implicitly indicated when the number of “end of transfer (eof)” messages received equals the number in the “expected eof” field in one particular data header.

Based on the implementation experience of GridFTP v1, people have proposed several points of improvements. These are all summarized in the GridFTP v2 [9] draft protocol, containing two key features. First, *eXtended mode* (Mode X) is a new mode introduced in the GridFTP v2. X mode is developed to fix certain drawbacks of the E mode and add some useful features including the removal of so-called unidirectional data transfer limitations, add more flexibility in dynamic management of network connections and add data integrity verification on the network transport level. As for the data header format, X mode adds two more fields, one is a 4 byte transaction ID and another one is a checksum. The checksum algorithm can be Adler32, MD5 or CRC32. X mode also introduces a robust handshake protocol on the data channel to explicitly open and close each data channel. Unlike extended block mode, X mode explicitly sends EOF messages on data channels to indicate the end of data transfer. Second, the *GET/PUT* commands are also introduced in the GridFTP v2 protocol. They are intended to replace the original *RETR/STOR* commands. These commands enable the server to create data channels after it knows what file to transfer along with other transfer parameters.

III. .NET GRIDFTP

A. Architecture Overview

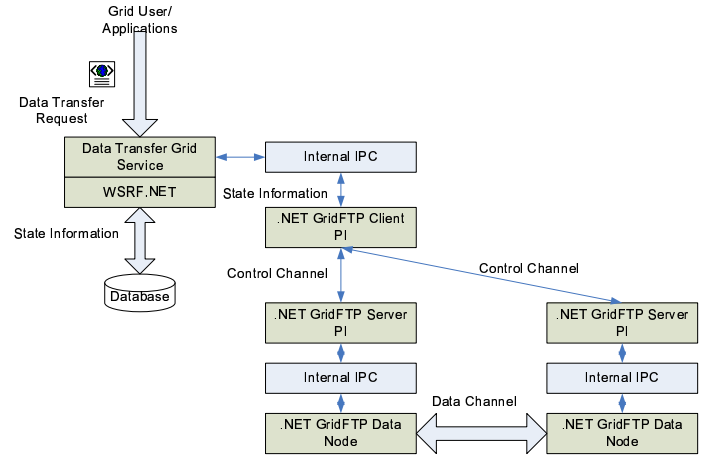


Fig. 1. Architecture of .NET GridFTP

Figure 1 shows our .NET GridFTP, which is architecturally similar to the Globus Toolkit V4 (GT4) GridFTP implementation [10], both to facilitate interoperability and to reduce the learning curve for new .NET GridFTP users who already are familiar with the GT4 architecture and services. There are three major components: a file transfer Grid service based on WSRF.NET, .NET GridFTP client, and .NET GridFTP service. The file transfer Grid service takes the file transfer request in the form of XML file from the Grid users or applications. For each request, this service will create a WS-Resource using the resource creation mechanism in the WSRF.NET. Using the WS-Resource End Point Reference (EPR) returned by the transfer service, grid users or applications can query the status or terminate this transfer. The WS-Resource lifetime is the duration of corresponding file transfer. The actual data transfer is handled by the .NET GridFTP client process created by the file transfer service instance.

The .NET GridFTP client can work in either interactive mode or batch mode—when the client is “forked” by the ASP.NET account under which the file transfer service runs, it will work in batch mode and communicate with the file transfer service thread through a Windows pipe. The file service gathers the returned information from the client and the corresponding resource properties are updated.

The .NET GridFTP service can be either a Windows service or a command line utility. It has two parts: the server protocol interpreter (PI) and data node (DN). The server PI handles the control channel command exchange and the establishment of GridFTP transfer session. The DNs handle the actual data transfer. The server PI and DNs can run together in a single multi-threaded process—each connection from the client will create one server PI thread and at least one DN thread, and they communicate through shared variables. Server PI and DNs can also run in different processes or even different machines, in this case they will communicate through TCP sockets.

B. Security Challenges

Implementing the GridFTP protocol on the .NET platform proved to be rather challenging because the security mechanism that are available in UNIX/Linux are not present in Windows. We have encountered two major challenges: the lack of the Grid Security Infrastructure (GSI) support on Windows and the creation of processes on Windows machines without knowing their passwords.

GSI is a widely-used mechanism for authentication and message integrity/confidentiality and is used by GridFTP. The Globus implementation of GSI is written in C and is based on the *openssl* library. It also adheres to the Generic Security Service API (GSS-API), which is a standard API for security systems promoted by the Internet Engineering Task Force (IETF). The most straightforward way to ensure interoperability between Windows and UNIX/Linux is to use the *openssl* libraries (and the Globus wrapper code) on Windows in our implementation of GridFTP. Although these libraries can be successfully compiled on Windows with some effort, they are not .NET “managed code”, meaning that they would not benefit from the enhanced security provided by the .NET Common Language Runtime (CLR). Simply, they would not be as secure, so this is not the best option.

There are certainly some alternative solutions for this problem. One is to use the Windows Security Service Provider Interface (SSPI)—a native Win32 interface for obtaining integrated security service for authentication, message integrity and message privacy—to implement a whole GSI library all by ourself. Although SSPI and GSS-API are very similar, implementing a whole GSI library is not practical as it requires the extensive understanding of both SSPI and GSS-API. Even it can be done, the result would still be an unmanaged Win32 library! Another option is to build a GSI library on top of some .NET based cryptography libraries. At the time this project started, there was an ongoing project to port the Java based BouncyCastle Crypto APIs [11] to C#. However this code was not ready for use at that time. Realizing the technical difficulties to implement a .NET GSI library all by ourself, we have chosen another strategy: to wrap only the necessary *openssl* functions in a C# assembly on which we then build those absolutely needed GSI functions, anticipating our future use of BouncyCastle when it matured. It turned out to be a better way for us to focus on the data movement problem itself and start the project earlier. At the time of this writing, BouncyCastle has released its beta 2 version of the C# crypto API with the active involvement of one of our project members. Based on that, we have built our own C# implementation of proxy certificates, and our ultimate goal is to replace the whole GSI library with the help from the BouncyCastle C# port.

The second issue involved creating processes on Windows as specific users, without knowing their passwords. A GridFTP mechanism generally spawns a process as the target user in order to simplify access control (the process just naturally inherits the access control mechanisms provided by the under-

lying operating system). On Windows, the GridFTP service can run either as a “Local Service” or “Network Service”. The working thread must be able to impersonate a grid user to inherit this user’s security privilege. Otherwise the working thread will be running under some system accounts and have much higher and in some sense *dangerous* security permissions. From the grid user’s perspective, they also want to see the file they created are in their own security context. The Globus Gridmap-file-based authorization solution only maps certain X.509 Certificate Distinguished Names to certain local user names, so the impersonation can be done without knowing the passwords. This can be achieved in UNIX/Linux system by using the *setuid* mechanism. However, this is not feasible on Windows.

Security in Windows is based, in part, on passing user tokens to authentication methods. The best-available, well-documented methods available to create a new user token or impersonating a user in a thread require a user name, domain and password combination in cleartext. In its normal use, this is not a vulnerability, as the user has generally typed the password on the console, so this password never leaves the machine. If these well-documented methods are used, there are a number of possibilities for implementation. In one case, the user must securely enter the password for each transfer request. This is undesirable because it does not allow a user to do single sign-on. Another option is to include the username and password pair in the gridmap file *in cleartext*, which obviously creates a new vulnerability. Also, this gridmap file will need to be changed every time the user changes his password on the machine, which would be very tedious. We have previously discovered a way to create a user token without knowing password by using an undocumented WIN32 API [2]. We decided to not pursue this approach due to its inherent instability and lack of first-class support.

We have several solutions to this problem. The first solution is simply no impersonation. Instead we limit the GridFTP service access only to a fixed location on the disk, which is what we called “GridFTPRoot”. Inside this location, each grid user will have his own sub-directory, and there is a public directory to store publicly-accessible data. Each grid user’s access is strictly restricted to his own sub-directory and public directory only. This mechanism is at the application-level (specifically, our Windows GridFTP service) and does not rely on the security mechanisms of Windows, which we would like to leverage. The second solution is to have a common user name and password on all machines with the least security privilege. All grid users’ DNs will be mapped to this common user. This is essentially a many-to-one mapping that leverages operating system protection mechanisms. The third solution involves the use of CredEx [12] (“Credential Exchanger”), which is a separate service we developed in the context of WSRF.NET project at the University of Virginia. In this solution, upon GSI authentication between client and GridFTP server, the GridFTP service will do a callout to the CredEx service, effectively “exchanging” an X509-based credential for a username/password credential. After that, the

GridFTP working thread will call some Windows functions to impersonate a user using the returned username and password. All the communications between GridFTP service and CredEx service are through a secure channel and GridFTP services are authenticated by the CredEx services. The integration with CredEx (still under development) is arguably the best in our situation. However, it should be possible to integrate with other similar service, such as MyProxy [13].

C. Our New Mode for Archiving Data

There are many situations where move a directory tree is desired. For example, a grid user may want to migrate his whole home directory to another machine for load balancing or capacity-limit reasons. Replication or migration of a whole data set is often seen on Grids, such as the replication of Sloan Digital Sky Data Set [14]. GridFTP is mostly designed to work well with single large files through its parallel data streams and TCP buffer size settings. However, it turns out the performance of GridFTP on large collection of small files, such as a directory tree, is not good due to the fact that small files benefit only a little from the increase of stream numbers and TCP buffer setting. Also for small files, the GridFTP’s partial file transfer based restartability solution does not make much sense since simply re-transferring the whole file is cheap. For directory tree transfer, the current GT4 GridFTP’s optimizations include reusing data channels and the implementation of “MLSD/MLST” commands on the control channel. However, besides it cannot benefit from parallel streams, it also generates too many commands and responses sent on the control channel. For each subdirectory, there will be a separate “MLSD” command and for each file, there will be a separate “RETR” command as well. More commands and responses will certainly result in more overhead.

One naive approach is to pack/compress all of the directory tree to a single file, then transfer this big single file using parallel streams, and decompress/unpack at the destination place (we investigate this in the next section of this paper). However, this approach first requires significant free space to hold the additional file archive. Second, in many situations, especially those Grids using many-to-one authorization mechanisms, this approach becomes not feasible since the Grid users are only expected to interact with the Grid through Grid services and they are not given or are not aware their local access rights. In this case, it becomes difficult for them to login to the storage resource to create their own archive. Third, in this approach, the creation, transfer and unpacking of the file archive have to be serialized.

Archive mode (Mode T) is specifically designed and introduced by us for moving a large collection of small files over the network (mode T is *not* part of the GridFTP v1 or GridFTP v2 protocols). In the archive mode, we want to move directory tree fast and still retain the restartability in case of failures. This is achieved through the following steps:

- 1) At the sender side, the source directory is traversed recursively to build a list of all files. Then this file list is sorted based on the file size of each individual file.

After the sort, the small files will be at the head of list while bigger files are behind.

- 2) The file list is further partitioned to segments. This is done by summarizing the file sizes starting from the head of the list. If the sum reaches the pre-configured segment size, a new segment is formed, then the sum returns to zero and this process restart from the next file in the file list.
- 3) Starting from the first segment, the sender starts to pack the files in the current segment to a single big temporary file. The pack is done using the *tar* algorithm (the optional *gzip/bzip2* compression is supported). When the packing is done, it is sent to the receiver using parallel streams. The preparation of next segment is started right after the sending of the previous segment begins.
- 4) At the receiver side, the data is unpacked/decompressed, then the receiver sends commands asking for the next segment.

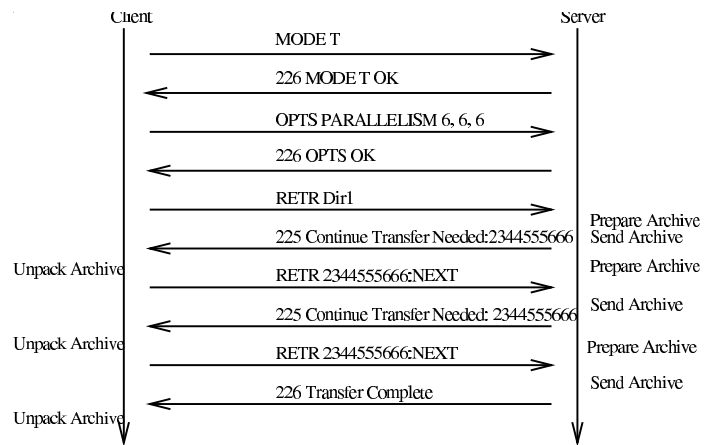


Fig. 2. Archive Mode Representative Message Interaction

Necessary changes are made to the GridFTP protocol. Figure 2 shows a representative message interaction between the client and server. Each directory transfer will be associated with a unique transaction ID by the sender if it can not be finished in one segment transfer. For each successful segment transfer, the receiver only has to store the transaction ID along with the last received segment ID for restartability purpose. Figure 2 shows the client always requests the next segment. However, it is possible for the client to request an arbitrary segment.

The archive mode achieves fast transfer rate by utilizing parallel streams on a rather large segment. The overlapping of preparation and sending segments on the sender side can also contribute to the overall performance increase. Compared with GT4 GridFTP’s one by one file transfer, this archive mode also significantly reduces the number of messages. Because the file list is strictly sorted, the necessary information for restartability purpose is minimal as only the transaction ID and last successfully transferred segment number are needed.

The segment size is a very important factor here. Generally, big segments will lead to the better utilization of parallel streams and fewer messages, but will also incur more cost for temporary disk space and restartability. However, our experiments show that after the segment reaches a certain size which is big enough to benefit from multiple streams, further increasing of segment size does not significantly increase performance as it mainly only reduces the number of messages between sender and receivers. Thus our preference is to use a “just-right” segment size to minimize the temporary space needed and cost for restartability. Justification of this “just-right” segment size will vary from one situation to another, and depend on the result of careful experiments on parallel streams (see the next section).

IV. PERFORMANCE EVALUATION

We performed experiments in both the local area and the wide area, both to test the interoperability of our implementation and to measure the performance (including our new Archive mode). In the LAN, our testbed at the server side contains two hardware-identical machines, one running Windows Server 2003 and another one running Linux with kernel version 2.6.9. Both machines have two Opteron 240 CPUs and 2G memory and they are connected to Gigabit switch. Windows server machine hosts *.NET GridFTP Server* and *Windows IIS FTP Server*, while the Linux machine hosts *Apache web server v2.0*, *sshd server* and *GT4 GridFTP server*. On the client side, we have a Windows XP Pro/Linux dual boot desktop connected to a 100Mbps switch. The machine has one Intel P4 3.2GHz with 1G memory. This machine will run client side applications, including *.NET GridFTP client*, *GT4 globus-url-copy(GUC)* client, FTP client *ncftpget* and http client *wget*. Our wide area testbed contains a Windows Server 2003/Linux dual boot machine at the University of Southern California (USC) in addition to the aforementioned machines at the University of Virginia. This USC machine has one Intel Xeon 2.2GHz and 1GB RAM, and it is connected to a 100Mbps switch. We use this remote machine to run *.NET GridFTP client*, *GT4 client*, *scp*, *wget* and *ncftpget*. Due to the difficulty of finding two Windows clusters on the Internet, we were not able to perform performance evaluation for striping transfer on wide area by the time of this writing.

A. Interoperability Test

We first tested the interoperability between our implementation and GT4 implementation. Our current implementation can interoperate with GT4 GridFTP on most FTP operations and major GridFTP extensions. However, there are three exceptions. First, our *.NET* implementation currently does not support data channel authentication. When use GT4 clients to access *.NET* GridFTP servers, grid users must explicitly disable the data channel authentication by using the “-nodcau” flag. Second, our new Archive mode clearly is not implemented in GT4 (although GT4 and our *.NET*-based implementation can transfer directories by moving files one-by-one). Third, striped data transfer is also not interoperable,

because the communication mechanisms between the server PI and DNs are not specified in the GridFTP specification and it is left up to the implementation. We use a text based protocol over TCP socket while GT4 uses MPI messages to do the inter-process communication (we are currently investigate the utility of attempting interoperability in this respect).

B. Stream Mode Performance on LAN and WAN

Aside from GridFTP, there are a number of alternative protocols to do data transfer including http, scp, and regular FTP. We compare our *.NET* GridFTP with these protocols and GT4 GridFTP implementation in stream mode on both local area and wide area network. In this experiment, as for scp, we create password-less RSA/DSA public key authentication so it does not have the password typing overhead. For FTP, we use *ncftpget* tool and it was set to do anonymous login.

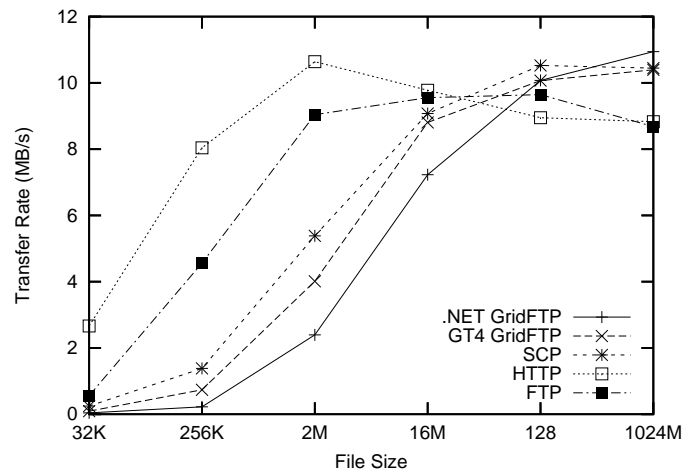


Fig. 3. Single Stream on LAN

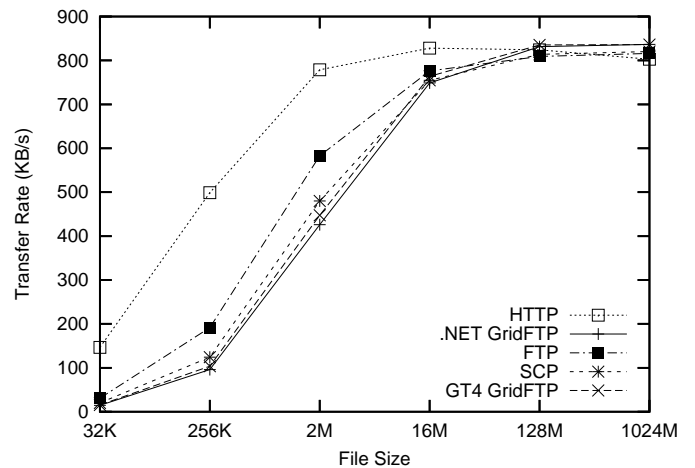


Fig. 4. Single Stream on WAN

Figure 3 and Figure 4 are the results of this comparison. In these experiments, We measured the transfer rate as the function of the file size from 32K to 1024MB. Each value in these two figures are averaged from ten experiments. We see that for small and middle size files (less than 16M in these experiments), HTTP protocol shows superior performance over other protocols. The FTP protocol is a little slower than HTTP but still faster than the rest for small files. This is predictable as HTTP is the simplest protocol here and it does not pay any security overhead at all in these experiments. As for FTP, it still has to exchange some messages to do anonymous authentication/authorization before the transfer begins. In LAN, Figure 3 shows that the .NET GridFTP is a little bit slower than the GT4 GridFTP for small files. In both the LAN and WAN cases, when the file is bigger than 128M, performance difference between these protocols becomes very small. This is because either the client size network interface is saturated or in the WAN case, the bandwidth has reached the maximum transfer rate for a single un-tuned TCP stream.

One interesting observation is that although scp encrypts all the traffic on the channel, it is generally faster than the both GridFTP implementations, especially for the LAN case. We speculate that the ssh authentication protocol has less overhead than the SSL handshake protocol on which GSI depends (note that there are no X509 certificates in scp and thus no certificate processing, although still cryptographic challenge-responses); second, the symmetric key encryption/decryption used by ssh has less overhead. Further investigation of this is out of the scope of this paper.

C. Parallel Stream Performance

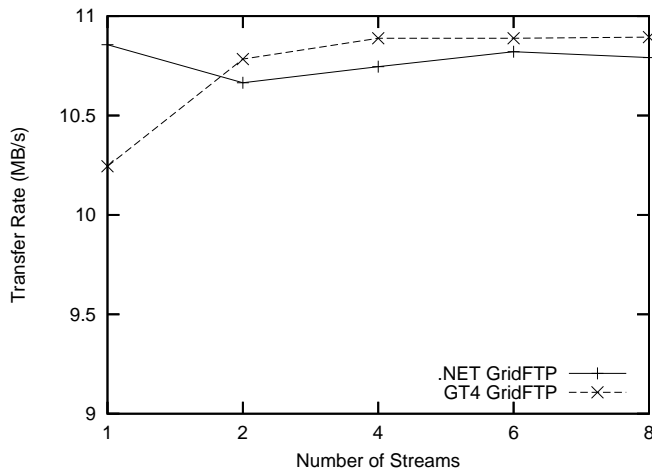


Fig. 5. Multiple Stream on LAN

Figure 5 and Figure 6 are the comparison between the .NET GridFTP and GT4 GridFTP on parallel stream performance on both LAN and WAN. A 512M file was used in these experiments. All values in these figures are averages from 10 runs at different time. Figure 5 shows that in LAN, both

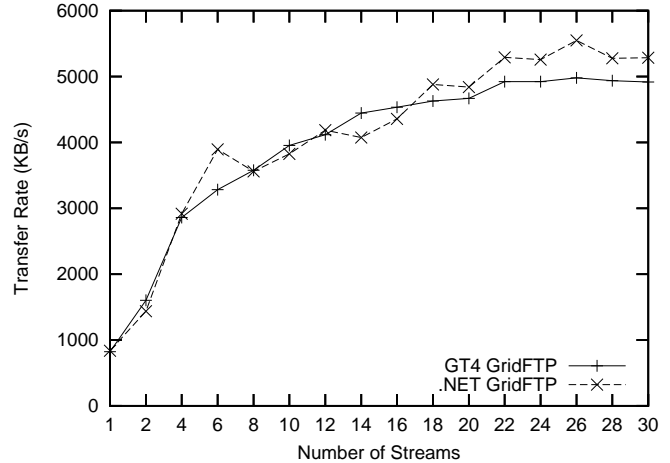


Fig. 6. Multiple Stream on WAN

implementations do not benefit significantly from using multiple streams and confirms the intuition that parallel streams are more effective with high bandwidth latency production networks. In LAN, the .NET implementation performance seems to be even worse when multiple streams are used, especially when two streams are used. Figure 5 suggests that using more than one stream in the local area network will not be useful as the performance gain will be very minimal.

On WAN, both implementations show the advantage of using multiple streams. Increasing the number of streams is very useful initially to boost the transfer rate. However, the performance gain starts to diminish very quickly when more than 4 streams are used. In fact, as streams increased to 20, transfer rate is only improved by about 6 times. The .NET implementation seems to show more variability than the GT4 implementation here, and it can reach a little bit higher on the transfer rate when more streams used. We think it is because the .NET implementation pre-caches when reading files into the buffer, thus it reduces the number of disk seek/read operations on the sender side.

D. Transfer Large Collection of Small Files

In our last reported experiment, we evaluate the directory tree movement utilizing our new Archive mode. In this experiment, we chose a representative directory containing 2874 files in 56 sub-directories that we downloaded from Sloan Digital Sky Survey project Data Release 3 [14]. File size ranges from 0 to around 258KB with mean at 54KB. About 99% files are JPEG images so the compression is not necessary here. The total data volume is about 147MB.

We conduct experiments for GT4 GridFTP, .NET GridFTP, SCP and HTTP. GT4 GridFTP, SCP and HTTP all have two options for directory transfer. One is to simply recursively copy the whole directory, another one is to first create a single archive on the source machine using the tar utility, then send it over the network, and untar it at the destination place. We compare .NET GridFTP in the “Archive” mode with all these

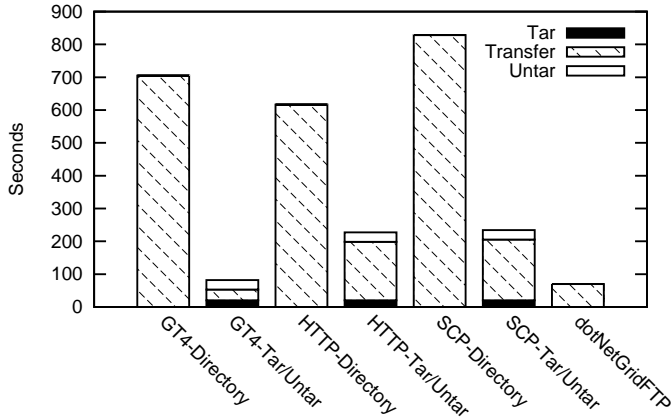


Fig. 7. Comparison of Directory Tree Transfer Performance

variations (GT4-Directory, GT4-Tar/Untar, HTTP Directory, HTTP-Tar/Untar, SCP-Directory, and SCP-Tar/Untar). GT4-Directory is the standard way in the GT4 GridFTP to move directory trees while the GT4-Tar/Untar will only work for those people with local access right and enough temporary space. For HTTP protocol, we use “*wget -no-host-directories -l inf -r -np -continue*” to transfer the entire source directory. This command tells the wget to do recursive copy with infinite recursive depth and don’t ascend to the parent directory.

Figure 7 depicts the total time required for each transfer method. This figure is drawn based on the average values from 10 runs for each mechanism. GT4-Tar/Untar uses 20 parallel streams while .NET GridFTP uses 16MB as segment size and 12 parallel streams (The reason is given in the next paragraph). This figure shows that those methods using recursively directory copy are generally very slow compared with other methods. The best result is achieved by the .NET GridFTP within about 69 seconds, while the GT4 GridFTP uses about totally 82 seconds, including 20 seconds on tar ball preparation, 32 seconds on data transfer and another 28 seconds on untar the archive. Because there is no compression used in the .NET GridFTP, both GridFTP implementations should have roughly transferred the same amount of data. Also the difference of using 20 streams or 12 streams should be very small based on the result of the Figure 6. Then we can conclude that the performance difference between the .NET GridFTP and GT4 GridFTP here should mostly come from the fact that the .NET GridFTP overlaps the tar ball preparation and network transfer, which is strictly serialized in GT4 GridFTP case.

Not demonstrated in Figure 7, all other tar based methods except .NET GridFTP have to use additional space to hold the temporary tar ball. Also the .NET GridFTP archive mode will have less restart cost compared with GT4 GridFTP as only the failed segments need to be re-transferred instead of the whole tar ball in GT4 GridFTP case.

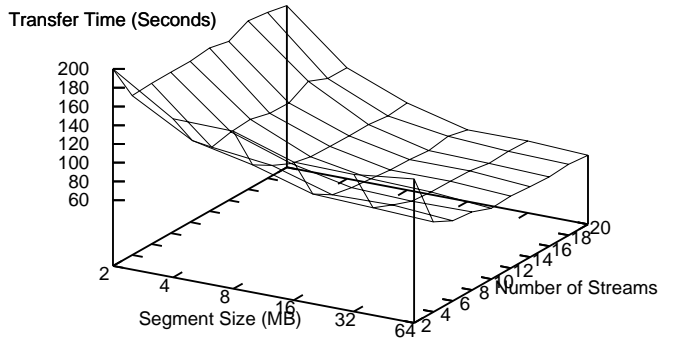


Fig. 8. Effect of Segment Size and Stream Numbers in Archive Mode

As stated in the Section III, the segment size plays an important role in .NET GridFTP Archive mode. Figure 8 shows the total transfer time as the function of both the segment size and the number of streams in our evaluation environment. In this figure, the transfer time generally shortens when more streams and large segments are used. However, when the segment size reaches 16MB and about 12 streams are used, the figure shows that increasing the segment size and stream number shows little improvement, for the following reasons. First, the 12 streams is sufficient to achieve about 85% of the maximum throughput in our test environment as it is clearly shown in the Figure 6. Second, the 16MB segment size is large enough in our test to show the benefits of using more than 12 streams. We see that although the small segment such as 2MB can quickly reach the best transfer time by using only 4 streams, further increasing the stream numbers degrades performance. For bigger segments such as 32MB and 64MB, they almost show the same curve as 16MB and they also reach the best result at about 12 streams. Thus we claim that the 16MB segment size and 12 streams are the “just right” parameters in our test environment as they can achieve acceptable performance and still retain a relatively small segment size to reduce restartability cost in case of failure. Currently, we are investigating ways to automatically tune this at run-time.

V. RELATED WORK

Fast and secure data transfer is of great importance for Grid Computing. There are already some implementations of GridFTP protocol, most notably the GT4 GridFTP [15]. *Ubertftp* [16] is an interactive GridFTP client. *Ubertftp* supports several of GridFTP’s extensions, including parallel transfers and third-party transfers. *rFTP* [17] is a grid data transfer tool. It improves the data transfer rate and reliability on Grids by utilizing multiple replica data sources concurrently to avoid replica selection thus dramatically increase performance to the aggregate of transfer rates of all replicas. *Dynamo* [18] is a

tool built on top of GridFTP implementation to specifically address the scenario of moving large data sets consisting of very large numbers of small objects. Our archive mode is inspired by this work and we have some improvements. First, our implementation is built into GridFTP protocol and it supports data transfer on both directions. Second, in our Archive mode, the information the sender and receiver need to keep for restartability purpose is much less than Dynamo. GDMP [19] is a file and data object replication tool built on GridFTP protocol. For directory transfer, it first makes an archive out of the whole directory, then transfer it to destination place using GridFTP. To our knowledge, the work described in this paper is the first comprehensive and evaluated implementation of GridFTP on .NET.

VI. DISCUSSION/CONCLUSION

During our implementation of the GridFTP protocol on the .NET platform, we have found some limitations of the GridFTP protocol. First, the GridFTP specification does not define the protocol between the protocol interpreter and data transfer node. This part is left to be implementation-dependent. However, this will likely lead to interoperability problems between different implementations. We believe this can be addressed in the future GridFTP specifications. The second, and more problematic, issue concerns the GridFTP and firewalls, especially on Windows. In the GridFTP Extended mode, the data connection must be originated from the data sender to data receiver. This requires the data receiver to listen on a random chosen port. However, this port will often be blocked by the default Windows firewall. It is not even possible to open a particular port on the Windows firewall as the listening port is randomly chosen. Although the GridFTP v2 protocol acknowledges this problem (note that the FTP "PASV" option will not solve this problem), we foresee the problem will remain in the third party transfer situation as one party must work as the passive receiver. This problem will get even worse as the Grid Computing moves to be Web Service based, in which the communication is only expected through the widely-opened HTTP port to across enterprise boundaries. In such a situation, the third party transfer basically will become impractical to use. We believe the future GridFTP protocol must incorporate more mechanisms to work around firewall issues.

In conclusion, to have the ability to seamlessly access/store Grid data regardless of platform, we have designed and implemented the GridFTP protocol on the .NET platform. Our implementation contains all of the features of GridFTP protocol v1 (except data channel authentication), supports "Extended Mode" and "GET/PUT" commands in GridFTP protocol v2, and is interoperable with GT4 GridFTP in most FTP operations and FTP extensions. Performance evaluation shows our implementation is comparable to the GT4 GridFTP on both single and parallel stream transfers, and is better on transfer of large collection of small files. Our future work will include the full implementation of GridFTP protocol v2 and

we hope to exploit emerging Web services specifications to define more powerful and standard-based control interfaces.

ACKNOWLEDGMENT

We gratefully thank John McGee at USC/ISI for his help with our wide-area experiments described in this paper.

REFERENCES

- [1] M. Humphrey and G. Wasson, "Architectural foundations of WSRF.NET," in *International Journal of Web Services Research*, April-June 2005.
- [2] J.V.S.Watson, S.-M. Park, and M. Humphrey, "Towards GT3 and OGSI.NET interoperability: GRAM support on OGSI.NET," in *2005 International Conference on Computational Science (ICCS 2005)*, May 2005.
- [3] D. Chu and M. Humphrey, "Mobile OGSI.NET: Grid computing on mobile devices," in *2004 Grid Computing Workshop (associated with Supercomputing 2004)*, Pittsburgh, PA, November 2004.
- [4] M. Humphrey and G. Wasson, "The University of Virginia campus Grid: Integrating Grid technologies with the campus information infrastructure," in *2005 European Grid Conference (ECG 2005)*, February 2005.
- [5] I. Foster and C. Kesselman, "Globus: A metacomputing toolkit," *International Journal of Supercomputing Applications*, vol. 11, no. 2, pp. 115-128, 1997.
- [6] G. Wasson and M. Humphrey, "Exploiting WSRF and WSRF.NET for remote job execution in grid environments," in *2005 International Parallel and Distributed Processing Symposium (IPDPS 2005)*, April 2005.
- [7] W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, L. Liming, and S. Tuecke, "GridFTP: Protocol extensions to ftp for the Grid," 2001. [Online]. Available: <http://www-fp.mcs.anl.gov/dsl/GridFTP-Protocol-RFC-Draft.pdf>
- [8] W.E.Allcock, I.Foster, and R.Madduri, "Reliable data transport," in *Building Service Based Grids Workshop, Global Grid Forum 11*, June 2004.
- [9] B. Allcock, I. Mandrichenko, and T. Perelmutov, "GridFTP v2 protocol description," 2005. [Online]. Available: <http://www.ggf.org/>
- [10] B. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, and I. Foster, "The Globus striped GridFTP framework and server," 2005. [Online]. Available: <http://www.globus.org/alliance/publications/papers/gridftp-1.1.pdf>
- [11] "Bouncycastle C# port." [Online]. Available: <http://www.bouncycastle.org/csharp/index.html>
- [12] D. Del Vecchio, M. Humphrey, and J. Basney, "CredEx: User-centric credential selection and management for Grid and web services," in *Proceedings of the 2005 IEEE International Conference on Web Services (ICWS 2005)*, July 2005.
- [13] J.Novotny, S.Tuecke, and V.Welch, "An online credential repository for the Grid: MyProxy," in *10th IEEE International Symposium on High Performance Distributed Computing(HPDC-10)*, August 2001.
- [14] "Sloan digital sky survey data release 3." [Online]. Available: <http://www.sdss.org/dr3/>
- [15] "Globus toolkit v4.0 GridFTP," 2005. [Online]. Available: <http://www.globus.org/toolkit/docs/4.0/data/gridftp/>
- [16] "NCSA GridFTP client." [Online]. Available: <http://dims.ncsa.uiuc.edu/set/uberftp/>
- [17] J. Feng and M. Humphrey, "Eliminating replica selection - using multiple replicas to accelerate data transfer on grids," in *Proceedings of the Tenth International Conference on Parallel and Distributed Systems (ICPADS 2004)*, July 2004.
- [18] M. Silberstein, M. Factor, and D. Lorenz, "DYNAMO-directory, net archiver and mover," in *Proceedings of the Third International Workshop on Grid Computing (Grid 2002)*, November 2002.
- [19] H. Stockinger, A. Samar, B. Allcock, I. Foster, K. Holtman, and B. Tierney, "File and object replication in Data Grids," in *Proceedings of the 10th IEEE Symposium on High Performance and Distributed Computing(HPDC-10)*, August 2001.