Fast Send Protocol – Minimizing Sending Time in High-Speed Bulk Data Transfers

Christoph Koch, Tilmann Rabl, Günther Hölbling and Harald Kosch
Chair of Distributed Information Systems
University of Passau
94032 Passau, Germany
{koch,rabl,hoelblin,kosch}@fim.uni-passau.de

Abstract

Over the last decades Internet traffic has grown dramatically. Besides the number of transfers, data sizes have risen as well. Traditional transfer protocols do not adapt to this evolution. Large-scale computational applications running on expensive parallel computers produce large amounts of data which often have to be transferred to weaker machines at the clients' premises. As parallel computers are frequently charged by the minute, it is indispensable to minimize the transfer time after computation succeeded to keep down costs. Consequently, the economic focus lies on minimizing the time to move away all data from the parallel computer whereas the actual time to arrival remains less (but still) important. This paper describes the design and implementation of a new transfer protocol, the Fast Send Protocol (FSP), which employs striping to intermediate nodes in order to minimize sending time and to utilize the sender's resources to a high extent.

1. Introduction

Increasing quantities of data produced and stored in grid environments in combination with high-speed wide area networks (WANs) stoke the desire for transferring tremendous amounts of data between dispersed sites. 10 Gbit fiber networks provide a theoretic throughput of more than 1 GB/s. High-performance servers are able to saturate such networks using specialized protocols. Application-layer protocols like GridFTP [1] are widely used and transport-layer approaches such as FAST TCP [16] and pTCP [7] have been proposed.

The weakest link on the path from sender to receiver determines the achievable throughput. A slow network link, e.g. a 100 Mbit WAN connecting two fiber Gbit LANs, may be the bottleneck, as well as a poor receiving machine limited by its CPU, hard drive or NIC.

FSP aims at overcoming the weakest link by introducing *intermediate nodes*. The sender partitions the data into smaller blocks and starts a striped transfer to distribute the blocks to several intermediate nodes. As soon as all blocks are distributed, the server retreats from the transfer, while the receiver collects the blocks from the intermediate nodes. The striping and collecting phase may be overlapped to reduce overall transfer time. Intermediate nodes are chosen from a preconfigured static list or from a self-organizing peer-to-peer (P2P) overlay.

Application areas may be found wherever resources at a server have to be freed by moving data as fast as possible. There are several concrete usage scenarios which benefit from minimizing the sending time:

disaster recovery: Disasters (e.g. fire, flooding, earthquakes) may threaten a building containing information storage servers. The servers may be rendered physically inaccessible, but still be technically intact for some time. The data has to be "evacuated" as fast as possible. A good solution is to stripe the data to servers nearby before bundling it at a safe and distant server.

server maintenance: Before server hardware is changed or when a server has to be shutdown, a data backup has to be performed. To keep down backup times, FSP may be used to move the data to a backup server.

parallel computers: High-performance parallel computers are frequently shared between many users and are becoming more and more widespread with the advent of computing and data grids. Computing nodes are likely to be geographically far away from their users, especially in the academic science community. Each user has a certain time-slot it has to adhere to, and costs are directly associated with the time the resources are used. After computations have finished, moving the resulting data from the parallel computer has top priority to reduce costs. Once again, FSP can be used to stripe the data to cheaper nodes as fast as possible.

The remainder of this paper is structured as follows. Section 2 gives a technical description of FSP and section 3 shows experimental results. We discuss related work in section 4 before concluding in section 5.

2. FSP protocol

Our design has been motivated by the goal of minimizing the sending time when transferring data from a fast server. We observed several limitations that might hurt throughput in traditional direct file transfers:

slow network link: WAN links somewhere on the path between sender and receiver are likely to offer less bandwidth than the server is able to saturate. Additionally, cross traffic on shared links limits throughput.

slow receiver: The receiving machine may not be able to cope with the server's sending rate. Limiting factors are slow CPUs, hard disks and NICs.

transport protocol limitations: Most file transfers use TCP on the transport layer. TCP Reno, the predominant TCP implementation, performs badly on so-called long fat pipes, i.e. connections with a large bandwidth-delay-product (BDP). Its flow-control parameters are often statically tuned for small BDPs. Additionally, packet loss triggers large TCP window reductions.

The Fast Send Protocol (FSP) addresses all these limitations. The main idea is to replace the direct transfer from sender to receiver by introducing intermediate nodes. Data is striped to several intermediate nodes. These nodes are closer to the sender or have more resources than the receiver. Finally, the data is collected by the receiver. Striping allows to use the combined bandwidth of several nodes and network links to overcome slow direct network links and slow receivers. The transport protocol limitations are addressed by using automatic TCP parameter tuning (see section 2.2).

FSP is an application-layer protocol extending FTP [11], the most ubiquitous data transfer protocol. FTP has been chosen as the basis of our protocol for quite the same reasons as in GridFTP design [3]: FTP is a widespread and mature protocol. With control and data channel being two separate TCP connections, third-party transfers can be implemented and extensibility is eased. A number of extension have already been proposed by the IETF[1], e.g. in the domain of security. For the sake of clarity the figures in this paper do not distinguish between control and data connections.

FSP has been designed to be completely compatible with standard FTP and may be used as a drop-in replacement for FTP. Additionally, FSP takes advantage of many GridFTP concepts which address shortcomings of FTP with respect to performance and data integrity.

[1]The Internet Engineering Task Force – `http://www.ietf.org/`

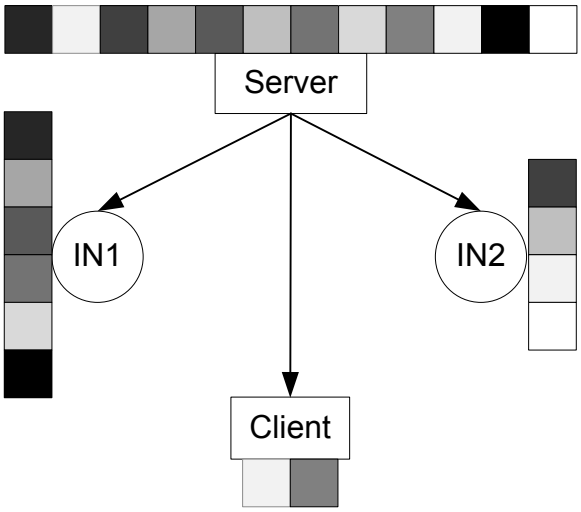


Figure 1. Partitioning into blocks and striping to intermediate nodes using a first-come, first-served distribution strategy

An FSP data transfer is logically separated into several phases:

request: The data transfer starts with a client requesting a file or directory from an FSP server.

partitioning: The server partitions the data into blocks of configurable size. A block header containing a descriptor flag, header length, payload length, file offset and file name is prepended to each block. The file name is represented as the relative path with respect to the current directory. Including the file name in each header allows to send complete directories with one single request, as each block can be associated with its file and the offset within the file. Typical header sizes range between 20 and 50 bytes which results in a negligible per-block overhead with sufficiently large block sizes.

intermediate node selection: After that, the intermediate nodes have to be selected from a static list or from a dynamic P2P overlay (see section 2.1 for details). Clever dynamic selection is crucial to the performance and accomplished by considering inter-node distances.

striping: The server opens a connection to each of the intermediate nodes and stripes the blocks according to a distribution strategy (see figure 1). Distribution strategies include *first-come, first-served* (i.e. faster intermediate nodes receive more blocks), *cyclic distribution* (i.e. blocks are allocated round-robin) and *partitioned distribution* (i.e.

each intermediate node receives an equally-sized contiguous share of each file). The different strategies enhance load-balancing if it is expected that the data has to be transferred to several clients later on.

collection: The server continuously informs the client of the block locations via the control channel, leaving it to the client when to start collecting the distributed blocks. Typically, the distribution includes the client to keep down the total time of transfer and the client starts collecting the distributed blocks while distribution is still in progress. The client opens a connection to each intermediate node and requests the respective blocks. Due to the block format, out-of-order delivery of the blocks does not pose a problem. When the client uses the data for streaming purposes, the block size has to be set quite small, so that a slower intermediate node does not delay the reception of a needed block.

The following subsections describe FSP's features and techniques in more detail.

2.1. Intermediate node selection

Selecting those intermediate nodes which offer the highest bandwidth is crucial to minimizing the sending time. It does not help to use intermediate nodes with scarce resources located farther away than the receiver. Currently, FSP selects nodes according to their network location, taking into account their distances reflected by the latency (round-trip time) to both sender and receiver. The actual distance constraints are configurable. One may choose intermediate nodes very close to the sender (maybe on the same LAN) without constraints regarding their distances to the receiver. While this approach is optimal to reduce sending time, it may hurt the total transfer time. A compromise is to choose nodes in the middle between sender and receiver.

FSP organizes the intermediate nodes using a *Meridian* P2P overlay as its network location service [18]. Meridian allows to select nodes based on their location in a network. Each Meridian node keeps track of a logarithmic number of nodes organized in a set of concentric rings centered around itself with exponentially increasing radii. The round-trip time is used to place the nodes in the rings. A node selection query consists of two constraints, the distance constraints to the sender d_s and receiver d_r. The resulting nodes are located in the intersection of the two constraint circles with the radii d_s around the sender and d_r around the receiver respectively as shown in figure 2.

Given the distance constraints, the FSP server first determines the nodes that potentially match the distance constraints. After that, it issues measurement requests to these nodes. Each of the nodes replies with its distance to both

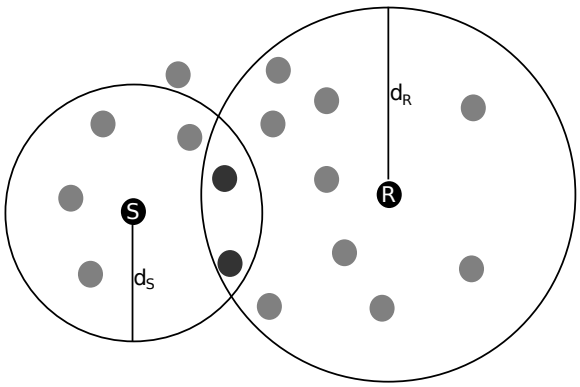


Figure 2. Intermediate node selection using Meridian

sender and receiver, so that the FSP server can determine whether the node actually matches the constraints. If there are no (or too few) nodes exactly matching a constraint, the query is forwarded to the node closest to the solution. This may lead to a multi-hop search with each hop reducing the distance to the target exponentially. A nice property of Meridian is that a node does not have to be a member of the overlay to issue a query. It is sufficient to know at least one Meridian node as an entry point. The exact algorithm is beyond the scope of this paper and we refer to [18] for the excellent, original description.

2.2. TCP tuning

FSP is expected to transfer data between geographically dispersed nodes via high-speed networks. These kind of networks exhibit large round-trip times (RTT) implying a large BDP with $BDP = RTT * Bandwidth$. TCP, the underlying transport protocol, has not been designed to adapt to such networks. It uses window based flow and congestion control. The sender is only allowed to send up to W_s (send window size) bytes without receiving an acknowledgement from the receiver. When W_s is small compared to the BDP, the sender has to wait for acknowledgements most of the time and the throughput T is limited by $T = W_s/RTT$. For instance, a transatlantic TCP connection on a 100 Mbit/s line with 90 ms RTT (and $BDP = 1.125\ MB$) and a standard $W_s = 64\ KB$ yields a throughput of only $T = 64\ KB/90\ ms = 5.56\ Mbit/s$. A W_s larger than the BDP wastes system memory. Best results are achieved with $W_s = BDP$. W_s is the minimum of the congestion window, the receive window and the sender's socket buffer size. The window sizes are hidden in the TCP implementation and cannot be set directly from within the application layer. The window size is set indirectly by changing the send and receive buffer sizes via the socket API. The maximum pos-

sible buffer sizes are limited by operating system settings and have to be adjusted if necessary [2].

Automatic buffer tuning. TCP's buffer-sizes are often statically tuned for slow networks. Manual tuning is perceived tedious and sub-optimal if RTTs vary over time. Several techniques have been proposed to automatically tune buffer sizes. For an excellent comparison of the most important ones we refer to [17]. AutoNcFTP measures the BDP at connection set-up and sets buffers accordingly [10]. However, it suffers from a potentially fluctuating BDP. FSP implements a different approach called dynamic right-sizing (DRS) that has been designed for GridFTP [6]. DRS continually determines both current bandwidth and round-trip time. Bandwidth is estimated by the receiver by simply calculating the current throughput. To estimate the RTT, the receiver periodically sends a special block header containing the receiver's buffer size to the sender on the data channel (unlike FTP, the data channel is used bi-directional). The sender tries to adjust its own send buffer size to the one received and acknowledges by setting a special descriptor flag in the header of the next outgoing block. Upon receiving the acknowledgement, the receiver can calculate the RTT as the time it took the acknowledgement to arrive. This method slightly overestimates the RTT because of the application layer protocol overhead, but serves as a good estimate. After calculating the BDP, the receiver updates its buffer sizes and informs the sender with the next RTT measurement header. DRS makes sure that the connection is never limited by the flow-control window.

Parallel streams. Besides buffer tuning, FSP adopts GridFTP's concept of parallel TCP streams to improve TCP throughput. The advantages of parallel TCP streams are threefold [8]. First, using n streams, means that n-times the TCP buffer size is available compared to a single TCP connection. Second, ramping up the transfer rate during slow-start is accelerated. Third, aggregated throughput in the congestion-avoidance phase is increased because recovery from packet losses is faster and competing TCP connections are suppressed. Using parallel streams on networks carrying cross-traffic may hurt the throughput of competing connections. Contrary, DRS connections are TCP-friendly.

Parallel streams are especially useful when buffer sizes can not be tuned due to operating system restrictions. The number of parallel streams has to be set manually by the user. A good rule of thumb is to use 4 parallel streams. Combining DRS and parallel streams may even provide better results [6]. However, using too many streams hurts the throughput because of the overhead associated with driving many connections.

2.3. Compression

FSP optionally supports on-the-fly per-block compression using ZLIB [4] with configurable compression levels. ZLIB has been chosen because it is an open standard with open-source libraries being freely available. However, other algorithms could be integrated easily. As compression is computationally very expensive, it does not increase throughput in most of the cases. Good results may be achieved with highly compressible data such as text files or on slow network connections where compression speed is able to keep up with network speed. Intermediate nodes do not decompress blocks to save CPU cycles and storage space.

2.4. Data integrity

When large amounts of data have to be transferred without any bit errors, it is crucial to detect the range of bytes where an error occurred. This allows to retransmit only the erroneous block instead of the complete file. TCP offers a reliable byte stream with its own error detection mechanism, but the checksums used are rather weak [13]. Standard FTP does not offer any integrated means to ensure data integrity. One has to resort to manually transferring a precomputed checksum and checking the validity. With only a single checksum per file, the complete file has to be retransmitted which is very inefficient.

FSP optionally computes its own per-block checksums and uses the same approach to data integrity as GridFTP. Each block is appended a checksum calculated over the complete bytes of the block. Transmission errors can be tracked down on a per-block basis. As block sizes are relatively small compared to the complete file length in most cases, retransmitting erroneous blocks is very efficient. Retransmission is always triggered by the receiving endpoint. The checksum algorithm may be chosen freely. Our current implementation supports ADLER32, CRC32, MD5 and SHA1 checksum algorithms.

2.5. Security

FSP currently uses username/password authentication over a TLS encrypted control channel. Both server and client have to have a user account at each intermediate node or there must be a common account at each intermediate node, respectively. We are aware of the security implications and FSP can be extended with a more secure mechanism such as X.509 certificates.

Being based on FTP, FSP has both a control and a data channel. Both are point-to-point connections, but the data they carry differs in its security needs. The control data exchanged on the control connection is only relevant to the

two involved parties. Point-to-point security is sufficient and FSP encrypts the control channel with SSL/TLS in the same way as the protocol extension to FTP proposes in RFC4217 [5].

On the other hand, data exchanged on the data channel has end-to-end semantics. While it is processed and stored at several intermediate nodes, only the client and server should be able to read the (confidential) data as clear text. Blocks of data distributed to intermediate nodes are not supposed to be readable at the intermediate nodes or on the wire during transmission. Thus, there is a need for end-to-end security. FSP encrypts the blocks with symmetric block ciphers offering AES [9] and Blowfish [12] encryption with bit sizes of at least 128 bits. The encryption key is securely exchanged between server and client via the TLS protected control channel. Intermediate nodes are not able to decrypt the blocks and are not aware of the content they store.

Encrypting is computationally expensive and has a negative performance impact on servers which are not able to encrypt at the same rate as their network connection provides. The user has to decide upon the tradeoff between security and performance.

2.6. Third-party transfers

Transfers between two servers initiated by a third party are referred to as third-party transfers. They are implemented differently than in FTP or GridFTP. The client opens a control connection to the receiving server and indirectly triggers the transfer between the two servers. It acts like a "remote control" for the transfer. The receiving server retrieves the requested file from the sending server as if it were a client. Unlike FTP, the client does not directly request one of the servers to open any data connections. This approach keeps the complexity of opening connections to the intermediate nodes at the servers.

3. Experimental results

We implemented an FSP client and server prototype in Java 1.6. Java has been chosen for the sake of platform-independence. We use the Java new I/O (NIO) APIs for file and network I/O which benefits performance compared to Java's traditional stream-based I/O. NIO takes advantage of native OS buffers which moves time-consuming I/O operations to the operating system where they are implemented more efficiently.

The test setup is shown in figure 3. The server has a 1 Gbit ethernet connection to a Gbit switch. The intermediate nodes and the client are connected to 100 Mbit switches which are themselves linked to the Gbit switch. The link to intermediate node 4 (IN4) is routed via a VPN and carries cross-traffic. Both the client and IN3 share the same 100 Mbit uplink to the Gbit switch. RTTs between the nodes range between 0.3 ms and 2 ms, typical values for a LAN environment. The machines are not limited by their CPU or harddrive, but only by the network connection. All tests were conducted with transfers of a single 1 GB file using blocks of size 1 MB. DRS was enabled which generally set the TCP buffer sizes to 128 KB.

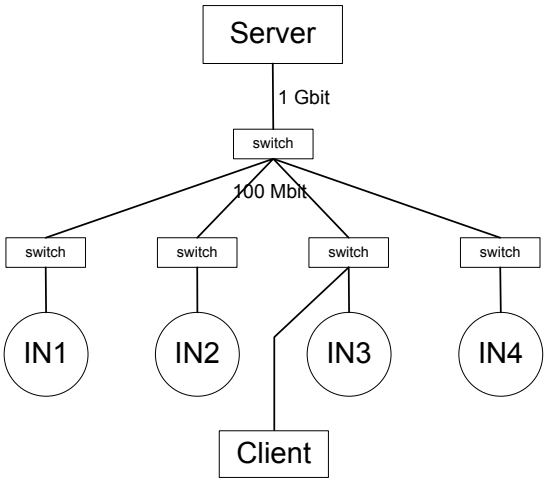
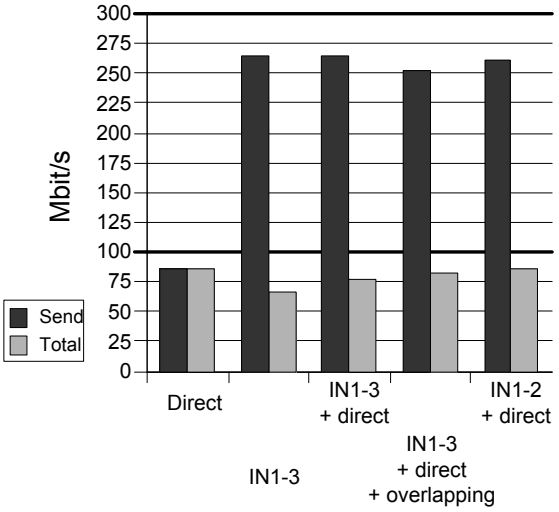


Figure 3. Test setup

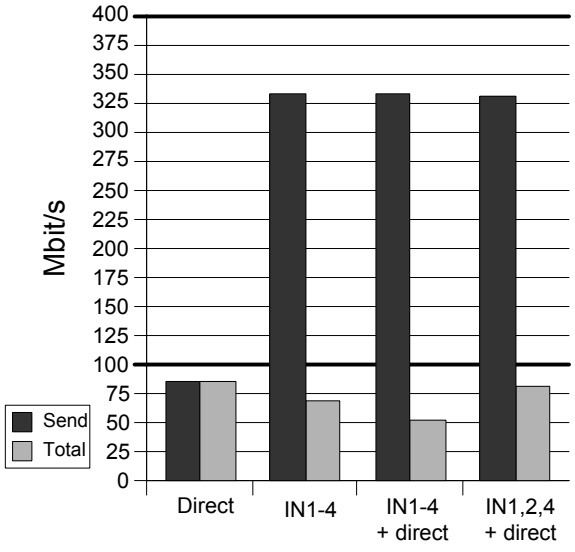
The first test uses intermediate nodes 1 to 3. The maximum (theoretically) achievable bandwidth during striping is 300 Mbit/s while the client is limited by its 100 Mbit/s connection. Figure 4(a) shows the results of our tests. The left bar of each column shows the goodput[2] during striping, the right bar indicates the goodput of the total transfer. A direct transfer between server and client resulted in a goodput of 86.1 Mbit/s. Striping to IN1-3 without including the client in the striping phase yields 264 Mbit/s and 66 Mbit/s in total. Including the client in the striping phase increases the total goodput to 76 Mbit/s (striping bandwidth remains the same as the client and IN3 compete for bandwidth). Additionally, overlapping striping and collection phase reduces the striping goodput to 252 Mbit/s, but increases the total goodput to 82 Mbit/s. Best overall results were obtained by striping to IN1 and IN2 and including the client in the striping phase (thus removing bandwidth competition between client and IN3). With 261 Mbit/s striping goodput and 86.4 Mbit/s total goodput, we obtained 87% of the theoretical 300 Mbit/s maximum striping throughput and even exceeded the total goodput of a direct transfer between client and server.

Our second test takes advantage of striping to all four intermediate nodes. This increases the theoretical striping

[2]Goodput is defined as the application level throughput, that is the amount of useful bits transferred per second. In our case, the overhead of the block headers and the protocol overhead of transport, network, data link and physical layer are excluded.



(a) Using 3*100 Mbit lines



(b) Using 4*100Mbit lines

Figure 4. Experimental results

throughput to 400 Mbit/s (however, the line to IN4 carries cross traffic). We achieved 83.4% of the theoretical striping throughput, but the total goodput ranged between 52 Mbit/s and 80 Mbit/s. We attribute the worse total goodput to varying amounts of cross traffic on the route to IN4.

Our results show, that the striping approach significantly increases the throughput during sending and thus reduces the sending time. It is very promising to note, that under certain circumstances (test *IN1-2 + direct*) the total goodput is even slightly higher than in a direct transfer.

4. Related work

FSP is based on FTP [11] using TLS encryption as described in RFC4217 [5]. It adopts some of the GridFTP concepts such as parallel TCP streams and striping. GridFTP [1] provides point-to-point transfers even though both of the endpoints may span several hosts (N:M striping). In this sense, FSP may be seen as a sequence of 1:N striping during the distribution phase to the intermediate nodes and a following N:1 striping during collection at the client. GridFTP may not be used directly to stripe data from a server to intermediate nodes and collect it at a client automatically. It could be used as a means to transfer the data between the nodes, but a controlling entity would still be needed. GridFTP lacks the feature of dynamically selecting nodes to stripe to. The data nodes that are used for striping are statically registered at a front-end node. Additionally, GridFTP is quite complex to install.

Quite close to the FSP approach is Kangaroo [15]. Kangaroo is a data movement service which hides I/O errors from grid applications by transparently moving data to a storage site, optionally using network and disk capacity on a single path of intermediate sites. However, the routing to and between intermediate sites has to be statically configured at each site for each source and destination address. Contrary to FSP, Kangaroo is limited to a single path of intermediate sites and does not apply TCP optimizations. Swany [14] proposes an approach using network logistics. A single connection is divided into a path of shorter connections. End-to-end throughput is improved because TCP performs better on connections with smaller RTTs and BDPs. A similar effect is achieved when FSP is configured to select intermediate nodes in the middle of client and server. However, FSP is limited to one level of intermediate nodes, while Swany's approach can use several intermediate hosts on a path.

5. Conclusion

In this paper we presented a new approach for minimizing sending time in high-speed bulk data transfers. The Fast Send Protocol uses intermediate nodes and striping mechanisms to maximize the amount of data sent by the server and to reduce the impact of slow network links. The selection of intermediate nodes is either static or based on a P2P overlay. FSP adopts concepts of GridFTP and reaches equal overall transfer rates. It is completely compatible with

FTP. Optionally, data integrity can be ensured by using per-block checksums. Additional mechanisms for encryption, compression and third-party-transfers have been included. Experimental results show that FSP significantly increases the bandwith utilization at the server side, thus reducing the sending time. Another finding is that the overall transfer times of striped transfer and direct transfer are equal.

Further research will address the dynamic adjustment of the number and the selection of intermediate nodes during data transfer. To fulfil stricter security requirements, support of certificates will be added. The dynamic adaptation of the number of parallel streams as described in [8] could further improve the transfer rate.

References

[1] W. Allcock. GridFTP: Protocol Extensions to FTP for the Grid. Technical report, Global Grid Forum, April 2003.

[2] W. Allcock and J. Bresnahan. Maximizing Your Globus Toolkit GridFTP Server. *ClusterWorld*, 2(9):2–7, 2004.

[3] W. Allcock, J. Bresnahan, R. Kettimuthu, and M. Link. The Globus Striped GridFTP Framework and Server. In *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, page 54, Washington, DC, USA, 2005. IEEE Computer Society.

[4] P. Deutsch and J.-L. Gailly. ZLIB Compressed Data Format Specification version 3.3 (RFC1950). Technical report, IETF Network Working Group, USA, 1996.

[5] P. Ford-Hutchinson. Securing FTP with TLS (RFC4217). Technical report, IETF Network Working Group, October 2005.

[6] M. K. Gardner, S. Thulasidasan, and W. Feng. User-space auto-tuning for TCP flow control in computational grids. *Computer Communications*, 27(14):1364–1374, 2004.

[7] H.-Y. Hsieh and R. Sivakumar. pTCP: An End-to-End Transport Layer Protocol for Striped Connections. In *ICNP '02: Proceedings of the 10th IEEE International Conference on Network Protocols*, pages 24–33, Washington, DC, USA, 2002. IEEE Computer Society.

[8] T. Ito, H. Ohsaki, and M. Imase. Automatic Parameter Configuration Mechanism for Data Transfer Protocol GridFTP. In *SAINT '06: Proceedings of the International Symposium on Applications on Internet*, pages 32–38, Washington, DC, USA, 2006. IEEE Computer Society.

[9] National Institute of Standards and Technology. *Federal Information Processing Standard 197, Advanced Encryption Standard (AES)*, Nov. 2001.

[10] National Laboratory for Applied Network Research. Automatic TCP Window Tuning and Applications.

[11] J. B. Postel and J. K. Reynolds. File Transfer Protocol (FTP) (RFC959). Technical report, IETF Network Working Group, October 1985.

[12] B. Schneier. Description of a New Variable-Length Key, 64-bit Block Cipher (Blowfish). *Fast Software Encryption, Cambridge Security Workshop*, pages 191–204, 1994.

[13] J. Stone and C. Partridge. When the CRC and TCP Checksum Disagree. In *SIGCOMM '00: Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 309–319, New York, NY, USA, 2000. ACM Press.

[14] M. Swany. Improving Throughput for Grid Applications with Network Logistics. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 23, Washington, DC, USA, 2004. IEEE Computer Society.

[15] D. Thain, J. Basney, S.-C. Son, and M. Livny. The Kangaroo Approach to Data Movement on the Grid. In *HPDC '01: Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC-10'01)*, page 325, Washington, DC, USA, 2001. IEEE Computer Society.

[16] D. X. Wei, C. Jin, S. H. Low, and S. Hegde. FAST TCP: Motivation, Architecture, Algorithms, Performance. *IEEE/ACM Transactions on Networking*, 14(6):1246–1259, 2006.

[17] E. Weigle and W. Feng. A Comparison of TCP Automatic Tuning Techniques for Distributed Computing. In *HPDC '02: Proceedings of the 11 th IEEE International Symposium on High Performance Distributed Computing HPDC-11 20002 (HPDC'02)*, page 265, Washington, DC, USA, 2002. IEEE Computer Society.

[18] B. Wong, A. Slivkins, and E. G. Sirer. Meridian: A Lightweight Network Location Service without Virtual Coordinates. *ACM SIGCOMM Computer Communication Review*, 35(4):85–96, 2005.