Distributed File Streamer: A Framework for Distributed Application Data Coupling

K. Chen #1, Z. Huang #2, B. Li #3, E. Huang *4, H. L. Rajic *5,R. H. Kuhn *6, W. Chen † 7

#*Intel China Research Center Ltd.*
Beijing 100080 China
[1]kang.chen@intel.com
[2]zhiteng.huang@intel.com
[3]bingchen.li@intel.com

**KSL, Software Products Division, Intel*
Champaign, IL 61820 USA
[4]eric.huang@intel.com
[5]hrabri.rajic@intel.com
[6]bob.kuhn@intel.com

†*Department of Computer Science and Technology, Tsinghua University*
Beijing 100084 China
[7]cwg@tsinghua.edu.cn

***Abstract*— File transfer is very common in a modern distributed computing environment. Protocols such as HTTP and FTP are designed for downloading or uploading files from/to servers. Some other tools such as 'secure copy' are used to transfer files among hosts securely. In this paper, the file transfer is considered in the context of connecting distributed applications, what is an output of a data producer on one node would be an input of a data consumer on another node. Intermediate files are used as a medium to connect workflow computational phases, which is a common paradigm used in grid environments. Distributed File Streamer a.k.a. DFS, as its name implies, uses data streaming to couple distributed applications. Instead of waiting for a producer application for output to transfer completely to the consumer node, DFS streams the data over the network directly to a consumer program, managing the data flow efficiently and providing a framework for partial file consumption. This paper describes the architecture of the DFS framework, gives its performance model analysis, and provides results demonstrating DFS advantages over the traditional way on several examples.**

I. Introduction

Communication mechanisms are very important in distributed computing environments. Higher level communication protocols, like the MPI [1] and the PVM [2] are designed for the cluster computing environments, so they do not fit the grid environment that well because they require a very reliable infrastructure and a uniform environment. Sockets [3] are the very basic method for transferring data over network. People often wrap sockets for their special network communication purposes. In this paper, we propose sockets wrapper for file transfers over the network. Thus the user can deal easily with the files without worrying about how to transfer them. File transferring over the network is often substituted as a more convenient communication method when data needs to be exchanged between different nodes.

In a file communication model, applications get their input from a file that was an output from an earlier stage of the workflow. Those files can be stored in the shared file system for applications to access, but could also be transferred from one host to another even across clusters.

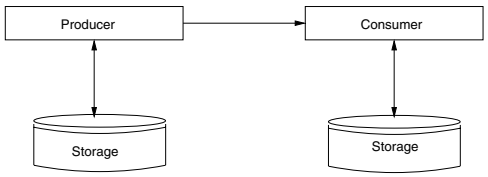

Fig. 1. Basic Producer/Consumer Data Model

Fig. 1 shows a basic producer/consumer data model that uses files as communication media. Many complex data workflows are comprised of several such basic producer/consumer data models.

We propose DFS, a Distributed File Streamer, to improve performance of such models. DFS is an optimized framework, which can speed up the overall workflow execution time by overlapping the output data stage of the producer programs with the input data stage of the consumer programs. In a typical data-driven workflow, computing tasks could generate one or many output files on the local storage. These files are then transferred to the remote nodes, so the next stage of the workflow on the remote node can start. Fig. 2 shows how DFS can improve the distributed application by overlapping I/O and computations. Files will be transferred in the background and the application can keep working. Although such mechanism requires that the application should work in the same way, a lot of applications can fit such model. DFS framework has a capability of delivering multiple files to multiple targets concurrently in an asynchronous manner. It uses parallel TCP connections, like GridFTP [4] to maximize network bandwidth

utilization. It reuses the established TCP channels in a non-blocked mode to lessen the system load and reduce the data transfer overhead.

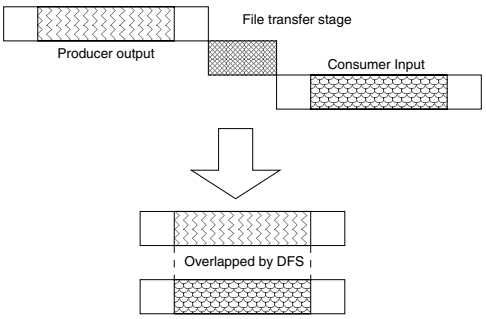

Fig. 2. DFS: Distributed File Streamer Way to Overlap I/O and Computations

We have also implemented very flexible buffering and caching capabilities on both producer and consumer parts in DFS. Depending on the network and its bandwidth availability it is possible to guard against losing data, slowing down the producer application, or impacting user experience if the consumer application is a media player.

The rest of the paper is organized as follows. In Section II of the paper we describe the DFS framework in details, presenting the producer and the consumer parts of the framework separately. We present the DFS performance model in Section III.

In Section IV we provide DFS benchmarking results comparing raw DFS performance to HTTP and FTP protocols. In the same section we present DFS integration into NGB (NAS Grid Benchmarks) [5] HC code and VideoLAN media client [6] showing how DFS framework could improve workflow performance and application functionality.

In Section V we describe related efforts. Section VI serves as a conclusion, where we summarize the DFS work.

II. Design and Implementation of the Distributed File Streamer Framework

A. Basic Components

DFS is an application level framework providing data streaming that can be integrated into distributed applications. It provides an optimized transport layer for connecting distributed nodes, a POSIX compliant I/O API, flexible data caching and buffering, and adaptive buffering and caching mechanisms that guard against network congestion, even temporary unavailability.

As described in the previous section, the framework is split into producer and consumer parts. The producer part intercepts the output data, optionally saves it on the local storage system, and then sends the data to the consumer part, which is usually running on a remote host. The consumer part receives the data from the producer part and stores them into its shared memory buffers before forwarding them to the consumer application. It optionally saves the data on the local disk too.

Producer and consumer module both have a daemon component that manages the shared memory buffers and interacts with the network and the other side daemon, and a file I/O API library that is used by user applications for reading and writing data to and from the DFS framework.

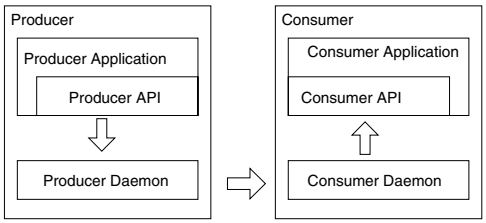

Fig. 3. Basic DFS Framework Components

Fig. 3 shows the basic DFS framework components. Producer application uses the producer API to 'write' the generated data to the DFS framework. The producer daemon then gets notified about the data arrival and is responsible for sending the data to the consumer destination. The consumer daemon gets the data from the producer daemon. Its API is used by the consumer application to read the data from the DFS framework. DFS supports disk file and memory (data generated by applications) file transfers across network. At the high level the framework is augmented with a set of parallel data transfer protocols and flexible buffer and cache data control mechanisms. Generally speaking, files or data are transferred from producer to consumer nodes in a 'push' mode.

DFS daemons maintain two kinds of communication control links, one with the remote daemons, and the other between a local daemon and the application DFS API library. Parallel TCP connections, reusable TCP channels, and a non-blocking socket communication are used to maximize network bandwidth utilization and to minimize system overhead associated with the network communication layers.

B. DFS File APIs

Developers use the DFS library API to redirect their program I/O to the DFS framework. Considering that the vast majority of the scientific computing applications and legacy applications use standard POSIX file manipulation semantics, we have made the DFS APIs compliant with the POSIX file I/O semantics and the calling sequence. The set of APIs on both the consumer and the producer are summarized in Table I. 'Shm' stands for shared memory, 'pdh' for the producer data handler, while 'cdh' for the consumer data handler.

TABLE I
Comparison of DFS APIs with POSIX's

POSIX	DFS Producer	DFS Consumer
open/create	shm_pdh_open	shm_cdh_open
read		shm_cdh_read
write	shm_pdh_write	
close	shm_pdh_close	shm_cdh_close

To make the interface usable for codes written in FORTRAN language, DFS library API has a wrapper interface available according to the FORTRAN function call conventions.

C. DFS Data Streams

DFS data streams are transferred from the producer daemon to the consumer daemon. They consist of messages. Each message has its own message type and format. The message packages and the communication semantics between daemons are designed to support both parallel and striped file transfer. As mentioned before, the DFS framework supports a 'push' mode for the messages. Each transferred file from the producer to the consumer is tagged by its own file ID, which are unique per producer host. There are three variable-size message types. 'MSG_OPEN' is used to open a file. It contains the file name together with the metadata information such as file length, and the assigned file_id. 'MSG_CREATE' is used to create a new file and is similar to the 'MSG_OPEN' message format. 'MSG_STORE' message contains the data. A fixed-size message type 'MSG_CLOSE' is used to close the file.

DFS data streams support striped file transfer by adding 64bits offset value on each 'MSG_STORE' message. Therefore, a single consumer daemon is able to handle both single and multiple files coming from multiple producer daemons concurrently, even when they come from different hosts and carry different file IDs. One such situation is shown in Fig. 4. Producers A, B, C, and D all contribute in sending a striped file over the network. At this moment DFS just provides a simple scheme for striped file transfers.

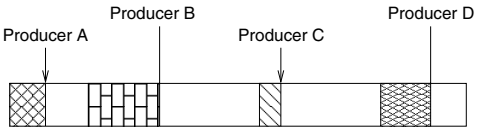

Fig. 4. Striped File Transfer in DFS

The consumer daemon can distinguish file data by a unique file ID tag. An additional number, a serial number is an increasing sequence positive integer that enables the consumer daemon to sort all of the incoming messages. The offset value is used to seek the target file at the right start address when striped I/O or fault recovery events occur. As in the POSIX standard, the only difference of the 'create' and 'open' message is that the 'create' message will truncate the file size to zero while the 'open' message will keep the file size unchanged.

D. Support for Multiple Sources and Destinations

Supporting multiple data destinations or sources is a frequent requirement for distributed applications [7]. The output of a producer program is often streamed to multiple consumers. To support this mode, the data blocks of a specific file will be tagged with different destinations and only when all of the destinations have received the corresponding data, the data blocks will be removed from the DFS framework.

DFS has a parallel data/file transfer support for various distributed computing environments. It provides parallel TCP channels across the network for concurrent multiple file transfers from many producers to many consumers. For a single file transfer, it provides a striped mode transfer where a file could be transferred from multiple producers to multiple or just one consumer. Also supported is file striping on the consumer side.

E. Implementation of the DFS Producer Side

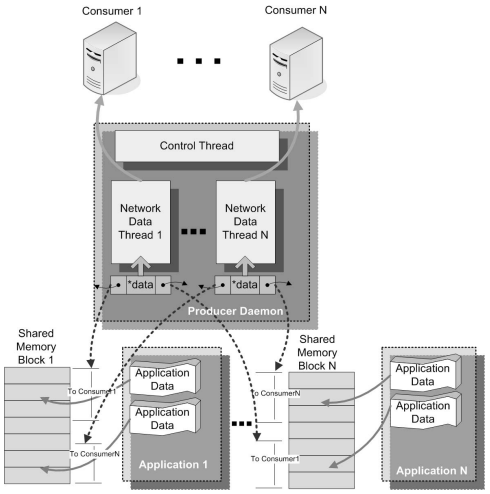

Fig. 5. Detail Implementation of the DFS Producer Side

The DFS producer side consists of two important components: the producer daemon and the producer application API library. There could be only one producer daemon serving producer applications on a single system at the same time. The multiple producer applications are able to co-exist and operate without interfering with each other. In the current implementation, one producer daemon interacts with the producer applications library API through a FIFO (a named pipe) and gets application data via shared memory buffers.

The producer daemon consists of a control thread to manage shared memory, a file hash table, a file destination mapping subsystem, and a support system for creating and destroying network data threads, Fig. 5. Once the control thread starts up, it builds a file and a destination mapping relational table, which we call file-transfer-DB, based on the configuration setup. When a producer application asks to transfer data, the control thread first creates a shared memory block. The DFS producer application library puts the data into the memory block and it notifies the producer daemon that there is data to be transferred. Notice that when the user program calls the producer API, the content of the user provided buffer will be copied to the shared memory. However when transferring the files, the content of files can be copied to the shared memory block directly without the need of extra copies in the memory. It is possible that when the framework provide memory management functions, the user program can access shared memory directly. This is for the future development. Upon receiving the notification, the control thread checks where the data stream is heading by matching it with the file name in file-transfer-DB. The control thread then looks for a network data thread connected to the target machine to handle

the transfer job. If there is no established one it creates a new network data thread to deal with the data transfer.

Each network data thread takes care of a single destination transfer: it communicates with the consumer daemon running on remote machine before sending the data. If the network connection is broken, the network data thread would try to re-connect and continue the transfer.

To use the DFS, user applications call the DFS producer side application programming interfaces instead of using file I/O calls. 'shm_pdh_open(const char *filename, int flag)' DFS routine opens a file that appears on the remote machine, 'shm_pdh_write(file_handle *fh, void * buffer, size_t size)' routine reads data into the shared memory first and then notifies the DFS producer side daemon to send the data through the network. 'shm_pdh_close(file_handle *fh)' will close the file. At the end of the build process the producer applications need to be linked with the DFS producer side library.

F. Implementation of the DFS Consumer Side

The consumer side of the DFS framework is very similar to the producer side. There is a DFS consumer side API, consumer side daemon, and shared memory buffers; only the data flows in the opposite direction.

When the producer and consumer applications are not 100% synchronized there is a need for handling excess data on the consumer node. DFS consumer daemon supports two modes for data storing before passing it to the consumer application, buffering data in the shared memory and caching data on the consumer file system. Storing the data on the producer disk is also an optional feature.

DFS producer and consumer daemons use the same shared memory data management. The management module can manipulate the space in the shared memory for the producer and consumer programs to allocate, use and free the memory blocks.

One consumer daemon process could fork out multiple child processes to accept parallel connections in order to overcome file descriptor limitation of a Linux process. One consumer daemon could also serve several producer daemons, but only one consumer daemon controls the connection with a single DFS producer. We have named it a 'device' in DFS. Besides the necessary non-blocked TCP communication data buffers, the TCP 'devices' also maintain a file hash table, which is used for data sorting. The entries of the hash table are linked arrays populated by file indexes, see Fig. 6. A file index owns all the file related information such as file name, file ID, and a serial number indicating the last serial number which has been processed. When a data block of the shared memory is inserted into the hash table, the offset value enables the sorting routine to position it to the right place. Only continuous data can be delivered to the application or be written into disk files.

A local UNIX domain socket is used to exchange messages between the consumer daemon and the consumer application library. As soon as the producer daemon creates a new TCP connection with the consumer daemon, one local UNIX domain socket with the producer IP address in its name is also created in a pre-determined directory, provided the data only needs to be buffered in the shared memory. In that case the consumer application can connect to the consumer daemon via the DFS library to obtain data from the shared memory chunks by simply using the POSIX compliant DFS I/O read interface.

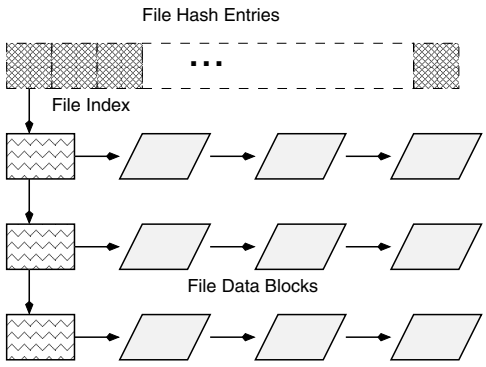

Fig. 6. Consumer Daemon File Hash Structure

In disk file caching mode, the consumer daemon gets the data from the network connections and stores them in the shared memory temporally. Previously mentioned buffer control mechanism makes it possible to do network operations and the disk I/O operations in the alternating fashion, so the data are cached in a local file. The consumer daemon would create a new same named metadata file as long as a new 'create/open' operation happened on the producer side. The existence of such metadata file indicates that the target file has not been transferred completely. The consumer library reads the data from the local files on the consumer application behalf. The library tests the existence of the metadata file when it reaches the end of the file to determine if more data is coming.

G. Improving the Framework Reliability

Unpredictable events may result in broken network connections. Existing application network protocols such as HTTP and FTP, will simply return an error to the caller, so it is the programmer's duty to consider all possible exceptions in order to maintain a communication context in the application such as a failure point in order to properly re-establish a broken session. DFS maintains the communication context on both daemon sides. Therefore, network problems are masked by the DFS library APIs. From the end user point of view, the DFS communication channel is as an unbroken pipe. In this way, the programmers are able to concentrate on their application logic instead of the intricacies of the data transfers. For the time critical applications, such as real time video playing, DFS users can ensure good end user experience with the environment adjusted buffer size value.

In a common Video on Demand (VoD) environment running a fat sever and multiple thin clients, DFS reconnection module takes care of the network faults and the suddenly inaccessible

media. When such faults happen, both DFS framework sides will try to re-establish the connection. The producer side daemon takes responsibility for resuming the data transfer from the failure point. In fact, a broken connection can be viewed as a kind of congestion leading to no or just limited frame drops.

III. DFS Performance model

Overlapping and/or hiding I/O with computation stages is a well exploited way [7],[8],[9] to speed up either single node or distributed nodes computational tasks.

In the traditional way, see the upper half of the Fig. 7 the data producer application needs to finish writing all the output data and close the output file, so that the file could be transferred to the consumer node. Very often, it is necessary for the producer application to finish its execution before the file transfer could start. In a Grid setting [10] there is also a delay before the data are ready and the scheduler could start the transfer task. On the other hand, when both applications are collocated or both applications use a shared file system the transfer time is avoided. Only after the output file has been transferred to the consumer node; adding there a scheduling delay, the consumer program is started and the input file read.

As seen in the bottom part of Fig. 7, the DFS framework cuts the total turnaround time considerably, by streaming the producer data once they are started being written. Shortly afterwards, the DFS consumer side enables data consumption by the consumer application.

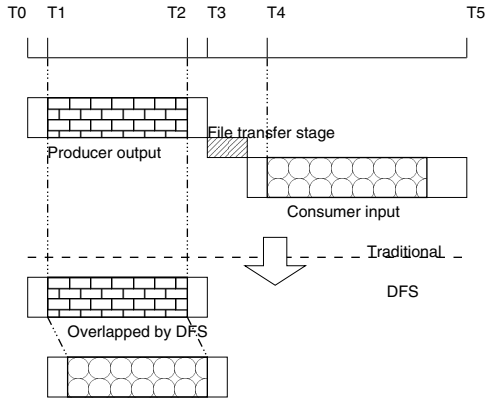

Fig. 7. Traditional and DFS Distributed Application Execution

Looking into Fig. 7, many different speedup models could be derived based on what kind of delays are taken into an account. In a more realistic scenario, there could be several output files stored at the regular intervals, for example they could be written at the end of compute stage iterations, Fig. 8. In a traditional distributed system, the consumer applications will not begin to run until all the output files from the producer program are transferred to the consumer's host. But, through DFS framework the consumer can start to run just after the first output file or files are streamed that leads to large time savings as shown on Fig. 8.

Of course, there are other I/O models, not all of them suitable for optimization using the DFS framework, but the flexibility of DFS could enable many different scenarios to be efficiently handled.

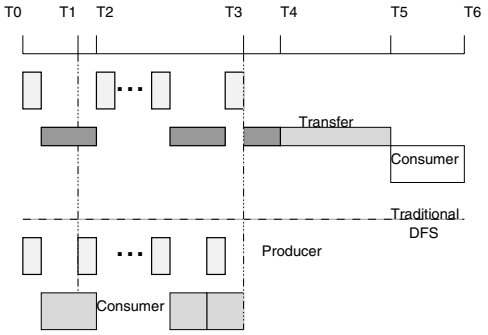

Fig. 8. How DFS Can Improve More Realistic Workflow Execution

IV. DFS performance Evaluation

In this section, we present DFS results, including performance comparison of different file size transfers over the DFS/FTP/HTTP protocols. We show how DFS could help speed up NAS Grid Benchmark (NGB) Helical Chain benchmark. Lastly, we integrate DFS into VideoLAN media client and show how a flexible framework like DFS could help improve digital video user experience.

A. File Transfer Performance

Our file transfer testing environment consists of two Intel Xeon servers, running Red Hat EL AS 4.0 with 2.6.8 Linux kernel. We had used Vsftpd and Apache 1.3 as comparison utilities, for DFS among several of FTP/HTTP servers because of their superior file transfer performance. For measuring protocols overheads we used a zero size file, Table II. The results came as expected, considering that DFS has very light proprietary protocol. The result should not be read how well DFS is performing, but just to get information about DFS overhead.

TABLE II
PROTOCOL OVERHEAD OF DFS/FTP/HTTP WHEN TRANSFERRING 10240 0-BYTE FILES

DFS	FTP	HTTP
2.460 seconds	14.96 seconds	2.58 seconds

In Fig. 9 we show file transfer comparisons for DFS, HTTP, and FTP protocols for a variety of file data sizes. The experiments were performed transferring large number of identical files: 5120 for files sizes less than 1 MB, 512 for 1 and 2 MBs, and just 2 runs for 4GB files. We made sure to clean the memory and avoid data caching. It could be seen that the DFS transport layer has been fully optimized and compares quite well to HTTP and FTP based utilities. It comes as no surprise no surprise that FTP overheads dominate file transfers for small file sizes.

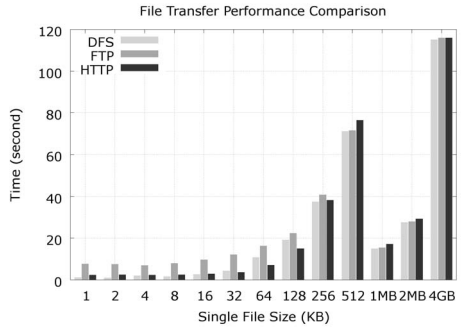

Fig. 9. Performance Comparison of file transfer using DFS/FTP/HTTP

B. Potential for DFS Speedup

A common scenario amenable to DFS framework usage happens when there is a continuous producer output that constitutes consumer input, as shown in Fig. 10. As soon as the producer output gets written it is transferred to the consumer to serve as its input. Depending on the output stage durations, overlap of the I/O and computation stages, beginning of the producer output stage, and various DFS overheads the final theoretical speedup is easily seen as application and resource dependent.

It is this inherent application limitation that dictates the amount of available speedup that DFS could achieve. So, in the following section we emphasize how close to theoretical speedups we could come to, instead of the absolute numbers.

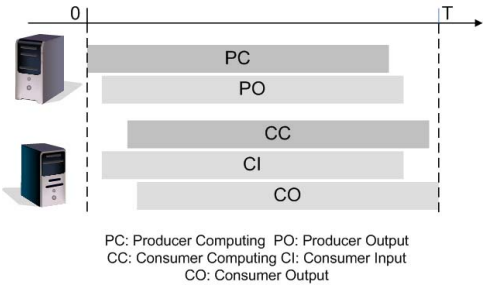

Fig. 10. Continuous data streaming

The applications could be also made of a series of above depicted computational and I/O phases which leads to more involved analysis [9].

C. NGB DFS Experiments

In scientific benchmarks, where the computation steps are followed with data intensive I/O intervals it is possible to effectively utilize higher network bandwidths to get advantage over the rate available of the file I/O system. To demonstrate the performance impact of the DFS framework on a producer/consumer type of applications that use files as communication media we have used Helical Chain (HC) benchmark, Fig. 11 from the NGB suite [5],[10],[11].

Besides class W, that involves mesh filter (MF), due to different input and output matrix sizes, there are NGB class S, A, B, and C data sets, which do not need to filter data from one stage to another and are therefore suitable for our experiments.

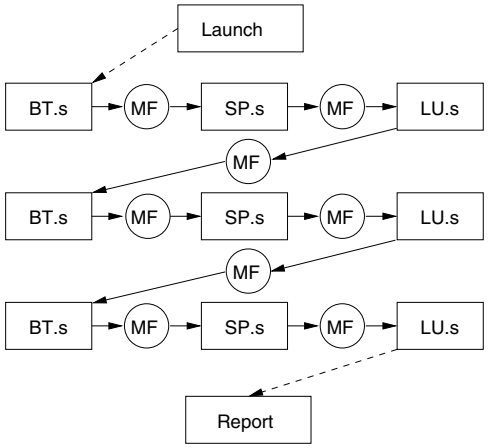

Fig. 11. Helical Chain benchmark

We have performed HC benchmark I/O characterization which has uncovered a little bit more than 10% I/O overlap with computations, provided the data is streamed from one stage of the benchmark to another.

We have run the original and the modified HC code ten times to obtain reliable average turnaround times for each problem class. The results are presented in Table III.

TABLE III
TURNAROUND TIMES FOR HC CODE

Class	Type	Time (Seconds)	Improvement
S	GridFTP	20.1	21.39%
	DFS	15.8	
A	GridFTP	98	10.41%
	DFS	87.8	
B	GridFTP	474.6	9.84%
	DFS	427.9	
C	GridFTP	2053.8	9.73%
	DFS	1853.9	

The benchmark results match our experimental performance improvement estimates of 10% possible speed improvements. 21% speed-up for the class S is not very reliable indicator of the DFS performance impact because of the small problem size and the proportionally large scheduling overhead for the unmodified workflow.

D. Video Streaming Experiments

For digital media playback, user experience is of a paramount importance. Our intention in this section is to show how DFS could be effectively used in digital video streaming situations when there are unpredictable network traffic problems.

To show DFS framework benefits we have integrated it in VideoLAN, a publicly available GPL licensed media client [6]. Using a 240 seconds, 1440x1080 resolution, 13.227 Mb/s,

396.8 MB large high definition video file we have performed a couple of experiments introducing artificially induced disturbances into our network.

In the first experiment, we have focused on the consumer media player capability for resisting severe network congestions. We have designed a network traffic interferer, a UDP package flooding program, which was able to very effectively disturb the network TCP/IP traffic. We have experimented with the variety of the pre-fetch buffer sizes introducing slight and severe network congestions.

Note that by streaming the video at the higher bit rates than the video bit rate, the video frames are buffered on the client side by DFS, resulting in shorter total transfer times than the video playing times. VideoLAN Read Speed curves follow data transfer bandwidths from the DFS consumer side to the VideoLAN client.

For milder UDP interferer interferences, Fig. 12, DFS bit rate was impacted, but VideoLAN kept playing smoothly. The DFS rate dips indicate network congestions. VideoLAN has a similar capability. It is specified by setting a number of seconds that VideoLAN needs to keep in its buffer. Any decrease in video bit rate is followed by very large spikes that produce large demands on the network bandwidth. Occasionally, VideoLAN would drop playing the video.

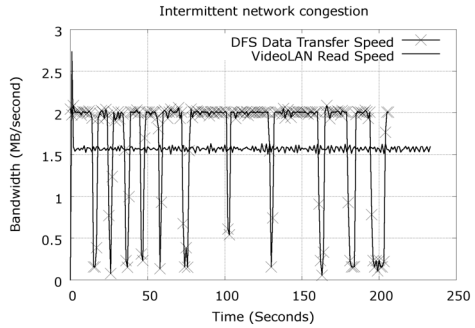

Fig. 12. Moderately intermittent network congestion

Even if a severe network situation exceeds the capability of pre-fetched data buffer (for example 20 seconds disturbances in Fig. 13), the perceptible impact duration (dropped width of 'VideoLAN Read Speed') is much shorter than the real congestion duration visible from the DFS streaming rate curve.

The experiments have shown that DFS can keep a transparent and reliable communication channel for VideoLAN, while other protocols make VideoLAN give up the ongoing session. These experiments prove that VoD (Video on Demand) is quite a viable media delivery provided adequate network speeds are available. Frameworks like DFS could greatly help overcome sporadic network problems by flexible data buffering or even disk caching.

In the second experiment we have investigated how DFS could shield end users from network faults, i.e. broken connections. In this case VideoLAN is forced to drop the connection for good, resulting in lost video feed. To mitigate network

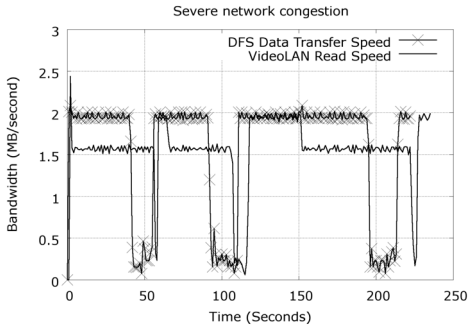

Fig. 13. Severe network congestion

disconnects DFS has a recovery mechanism on both sides of the framework with a flexible reconnection polling mechanism.

We have examined DFS reconnection capability for 20 second network disconnect intervals. DFS data transfer speed was kept at 4MBs except for occasional spikes.

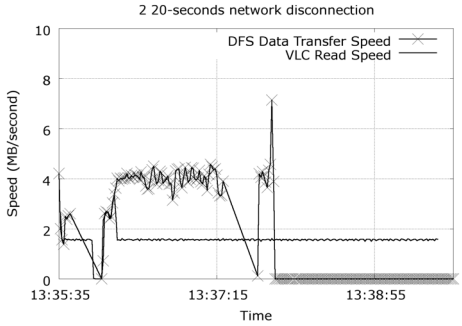

Fig. 14. 2 20-seconds network disconnects

In Fig. 14, two long network disconnects happened, the first was from 13:35:42 to 13:36:02, and the second was from 13:37:19 to 13:37:40. Using one second polling intervals in DFS, data retransmitting started at 13:36:03 and 13:37:41, resulting in only one second latencies.

We can see that the first network disconnect interval is longer than the duration of the pre-fetched video. 21.717MB pre-fetched content makes VideoLAN player continue for 14 seconds (to 13:35:56) before the buffered data is gone from the DFS cache file. Therefore, after the network connection is reestablished, VideoLAN has to skip few seconds. From 13:36:03 to 13:36:12, VideoLAN client tried to recover using the highest bandwidth available. But, after buffering tens of seconds of the video, the following 20-seconds network disconnect was handled by DFS without problems, again proving how useful DFS flexibility could be by substantially improving the user video media experience. VideoLAN, without DFS has simply stopped playing the video after the network has been cut.

V. Related Work

Producer-consumer data exchange models appear frequently in application coupling general area [12]. Data handling components are just one part of the solution, so they are not presented or built in as elaborate and optimized way as the DFS framework. DFS is also not concerned with data transformations associated with geometry mesh issues coming from different physics solution codes, an issue that is of greatest importance in application coupling solutions.

DFS operates in a distributed producer-consumer setting as does Distributed Data Broker (DDB) [13]. DDB is a toolkit that manages distributed communication in multi-application systems in high performance computing environments. It has been tested in UCLA Earth System Model (ESM) under the NASA/ESS HPPC program. DDB is reducing the number of synchronization points and the need of global reductions operations in order to improve high level computational efficiency.

There are also similar I/O solutions like DFS that overlap I/O and computations or hide I/O under computations, like active buffering and background I/O in parallel large-scale multi-component rocket simulations [8].

When we look into how DFS manages data in the shared memory parallels could be drawn to MPI parallel I/O collectives or MPI-IO [14].

Removing the need for data staging is a classic example of remote I/O [15]. Serving data to multiple device remote viewers in addition to data filtering also requires data streaming solutions in data visualization [16].

DFS could be viewed as an API to enable transferring data in complex workflows [8], where API of utilities like Globus Toolkit's [17] GridFTP [4] is used.

VI. Conclusions

We have introduced a Distributed File Streamer framework for optimization of consumer/producer distributed models that supports different data transport modes and gives lots of flexibility for use in digital media applications.

It has been demonstrated that the DFS transport layer is fully optimized by comparing its performance to the FTP and HTTP protocols on different file sizes.

It was shown that the DFS provides a convenient framework for optimizing distributed computing tasks with data dependencies between the computational stages. 10% speedups have been obtained on the DFS enabled NGB code Helical Chain benchmark on class A, B, and C problems. In addition to I/O overlapping with computations, additional DFS framework advantage, albeit small was the elimination of the job scheduling overheads between the remote tasks.

Computer networks have started playing significant roles in digital media distribution. High Definition video streaming needs a reliable and robust data transfer protocols to satisfy high end user expectations. In a non-exclusive network domain, network resource contention coming from different network applications has a potential to introduce intermittent and unpredictable network congestions.

DFS provides a reliable and effective support for high definition video streaming. The VideoLAN experiments show that the DFS framework can effectively utilize disk storage space and spare network bandwidth to provide flexible buffer space for fending severe network congestions. It was demonstrated that DFS can be very effective in recovering from intermittent network faults by its both sides reconnection mechanisms.

References

[1] M. P. Forum, "Mpi: A message-passing interface standard," Knoxville, TN, USA, Tech. Rep., 1994.

[2] V. S. Sunderam, "Pvm: a framework for parallel distributed computing," *Concurrency: Pract. Exper.*, vol. 2, no. 4, pp. 315–339, 1990.

[3] D. E. Comer, *Internetworking with TCP/IP, Vol III: Client-Server Programming and Applications, BSD Socket Version, Second Edition*. Prentice Hall, 1996, cOM d 96:1 1.Ex.

[4] W. A. et al., "Gridftp: Protocol extensions to ftp for the grid," Apr 2003. [Online]. Available: http://www.gridforum.org/

[5] R. F. V. der Wijngaart and M. A. Frumkin, "Nas grid benchmarks version 1.0," NASA Ames Research Center, Moffett Field, CA, 2002, Tech. Rep., 2002.

[6] VideoLAN, "http://www.videolan.org."

[7] G. Allen, T. Dramlitsch, I. Foster, N. T. Karonis, M. Ripeanu, E. Seidel, and B. Toonen, "Supporting efficient execution in heterogeneous distributed computing environments with cactus and globus," in *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*. New York, NY, USA: ACM Press, 2001, pp. 52–52.

[8] X. Ma, X. Jiao, M. Campbell, and M. Winslett, "Flexible and efficient parallel i/o for large-scale multi-component simulations," in *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*. Washington, DC, USA: IEEE Computer Society, 2003, p. 255.1.

[9] E. Rosti, G. Serazzi, E. Smirni, and M. S. Squillante, "The impact of i/o on program behavior and parallel scheduling," in *SIGMETRICS '98/PERFORMANCE '98: Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*. New York, NY, USA: ACM Press, 1998, pp. 56–65.

[10] M. Frumkin and R. F. V. der Wijngaart, "Nas grid benchmarks: A tool for grid space exploration," *Cluster Computing*, vol. 5, no. 3, pp. 247–255, 2002.

[11] L. Peng, S. See, J. Song, A. Stoelwinder, and H. Neo, "Benchmark performance on cluster grid with ngb," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, 2004, p. 275.

[12] R. W. Ford and G. D. Riley, "Model coupling review," Jan 2002. [Online]. Available: http://www.metoffice.gov.uk/research/interproj/flume/pdf/d3_r8.pdf

[13] L. A. Drummond, J. Demmel, C. R. Mechoso, H. Robinson, K. Sklower, and J. A. Spahr, "A data broker for distributed computing environments," in *ICCS '01: Proceedings of the International Conference on Computational Sciences-Part I*. London, UK: Springer-Verlag, 2001, pp. 31–40.

[14] P. Corbett, D. Feitelson, Y. Hsu, J.-P. Prost, M. Snir, S. Fineberg, B. Nitzberg, B. Traversat, and P. Wong, "Mpi-io: A parallel file i/o interface for mpi," NASA Ames Research Center, Moffett Field, CA, 2002, Tech. Rep., 2005.

[15] I. Foster, J. David Kohr, R. Krishnaiyer, and J. Mogill, "Remote i/o: fast access to distant storage," in *IOPADS '97: Proceedings of the fifth workshop on I/O in parallel and distributed systems*. New York, NY, USA: ACM Press, 1997, pp. 14–25.

[16] M. Wolf, Z. Cai, W. Huang, and K. Schwan, "Smartpointers: personalized scientific data portals in your hand," in *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*. Los Alamitos, CA, USA: IEEE Computer Society Press, 2002, pp. 1–16.

[17] I. Foster, "Globus toolkit version 4: Software for service-oriented systems," in *IFIP International Conference on Network and Parallel Computing*, vol. 3779. Springer-Verlag GmbH, 2005, pp. 2–13.