Future Generation Computer Systems 27 (2011) 977–985

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

CSTP: A parallel data transfer protocol using cross-stream coding

Shaofeng Liu [a,*], Jürgen P. Schulze [b], Thomas A. DeFanti [b]

[a] Department of Computer Science and Engineering, University of California San Diego (UCSD), USA

[b] California Institute for Telecommunications and Information Technology (Calit2), University of California San Diego (UCSD), USA

ARTICLE INFO

Article history:
Received 24 March 2010
Received in revised form
18 August 2010
Accepted 29 November 2010
Available online 16 December 2010

Keywords:
Cross-stream coding
4K
Streaming
Optical network

ABSTRACT

Transferring very high quality digital objects over optical networks is critical in many scientific applications, such as video streaming/conferencing, remote rendering on tiled display walls, or 3D virtual reality. Current data transfer protocols rely on UDP as well as a variety of compression techniques. None of the existing transfer protocols, however, scale well to many parallel data connections. Existing parallel streaming protocols have limited synchronization mechanisms for multiple streams, and they are prone to be slowed down significantly if one stream experiences significant packet loss. In this paper, we propose a new parallel streaming protocol which can stream many parallel data streams over optical networks: CSTP, the Cross-Stream Transfer Protocol. It not only tolerates random UDP packet loss, but also aims to tolerate unevenly distributed packet loss patterns across multiple streams to achieve synchronized parallel streams with limited coding overhead. We simulated the approach, and the results show that CSTP can generate steady throughput with fluctuating data streams of different data loss patterns, and can transfer data in parallel at a higher speed than multiple independent UDP streams.

1. Introduction

In recent years, CineGrid [1] has been a leader in using high bandwidth networks to transfer very high quality digital content for real-time movie showings and digital preservation. Although CineGrid has performed many successful demonstrations of 4K video [2,3] streamed over fiber networks, we foresee challenges in designing new protocols which scale to applications demanding higher bandwidth and higher resolution digital content. Today, ultra-high resolution displays have become standard infrastructure in scientific research. These displays are typically built by tiling an array of standard LCD displays into a display wall, using a PC cluster to drive it [4]. Figs. 1a and 1b show different types of display walls: the HiPerSpace uses sixty 30' LCD displays to form a display wall with more than 200 million pixels, the StarCAVE [5] uses 34 high-definition projectors to construct a 3D virtual space in which people can work with 3D virtual reality objects. Meanwhile, high-speed research optical networks [6] make it possible for scientists to use these ultra-high resolution displays over long distances in their scientific applications like very-high-definition video streaming or conferencing [7], real-time data visualization generated by remote scientific instruments, etc. As a perfect example of the combination of display walls and a high-speed network, the OptIPuter [7,8] research project, funded by the American National Science Foundation, constructed 1–10 Gbps optical network infrastructure and middleware aiming to give interactive access to remote gigabyte to terabyte visualization data objects and bring them to a visual interface, the OptIPortal [4].

However, scaling up visualization devices from a single PC with a single display to a cluster of PCs with a cluster of displays has brought up challenges of how to feed data to these display devices. These challenges are based on an obsolete traditional single data source communication model, and create the need for a new communication model with multiple-to-multiple end points, which achieves very-high-bandwidth parallel data streaming between cluster-based display devices while still appearing as a point-to-point communication protocol to the application programmer. Traditional data transport protocols have limitations which prevent their use for such multi-endpoint connections. The popular Transport Control Protocol (TCP) is slow on long distance networks because of its window-based congestion control mechanism. Alternatives like RBUDP [9], UDT [10], and LambdaStream [11] are recently developed UDP-based protocols focusing on high-speed file transfer or real-time data streaming between two end nodes. These protocols are point-to-point rate-based, which means they support one sender and one receiver, and recover lost UDP packets by resending them, and the sender controls the sending rate to minimize packets loss. RBUDP, for instance, uses a bitmap to maintain a list of lost UDP packets and do a multi-round communication to recover lost packets, which usually takes 2–5 Round Trips Time (RTT) to retrieve a GB file correctly; LambdaStream detects packet

* Corresponding author.
E-mail addresses: s8liu@ucsd.edu, liushaofeng@gmail.com (S. Liu), jschulze@ucsd.edu (J.P. Schulze), tdefanti@ucsd.edu (T.A. DeFanti).

0167-739X/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.future.2010.11.028

Fig. 1a. HIPerSpace, one of the world's largest display wall in Calit2, UC San Diego, has 286,720,000 effective pixels.

Fig. 1b. The StarCAVE, a third-generation CAVE and virtual reality OptIPortal in Calit2, UC San Diego, has ∼68,000,000 pixels.

loss based on the gap between the receiving time of two consecutive UDP packets, and if that gap is bigger than expected, the sender will reduce the sending rate accordingly. None of these protocols scale to a multiple-to-multiple communication model in a straightforward way. Existing parallel data streaming protocols do not sufficiently address the synchronization issue of multiple parallel data streams. Existing multiple-to-multiple communication models [12,13] synchronize senders and receivers by adding a synchronization module to senders and/or receivers. This synchronization module waits until all senders/receivers have the next frame of data, and then broadcasts the command of sending/displaying to all senders/receivers. In these cases, the behaviors of individual data streams will affect the overall protocol performance or data integrity, for instance by packet loss of one data stream.

In summary, point-to-point or point-to-multipoint protocols are not sufficient to feed increasingly larger displaying devices, and multipoint-to-multipoint protocols are difficult to efficiently synchronize. In this paper, we examine the multiple-to-multiple communication model, and present the new Cross-Stream Transfer Protocol (CSTP), which focuses on the synchronization challenge of the parallel communication model. We ran experiments with our new protocol, and the results show that even with fluctuating network throughput, or unevenly distributed packet loss in parallel streams, our approach can achieve reliable synchronized throughput. In Section 2, we introduce related work; in Sections 3 and 4, we discuss our approach for reliable synchronized parallel streaming. We show some of our simulation results in Section 5. In Section 6, we conclude and suggest future research directions.

2. Forward Error Correction

Forward Error Correction (FEC) is "*a system of error control for data transmission, whereby the sender adds redundant data to its messages, also known as an error correction code. This allows the receiver to detect and correct errors (within some bound) without the need to ask the sender for additional data*" [14]. Since most high bit-rate transfer protocols use unreliable UDP packets to deliver data, they often use a certain type of FEC to recover lost data. FEC can correct a small percentage of random packet loss, depending on the amount of FEC data sent, but is not always effective for burst packet loss over a certain bound.

In recent years, a new series of coding techniques have been used in FEC, called ***Fountain Code*** [15–18], which "*can generate a limitless sequence of encoding symbols from a given set of original symbols such that the original symbols can be recovered from* ***any subset*** *of the encoding symbols of size equal or only slightly larger than the number of original symbols*" [15]. In our case, a symbol represents a UDP packet.

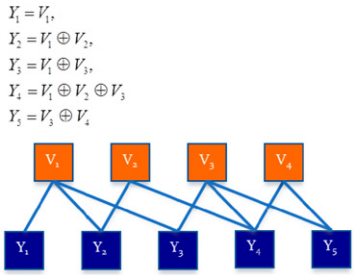

Fig. 2a. V_1–V_4 are original data packets; Y_1–Y_5 are encoded data packets. The formula and bipartite graph have the same meanings.

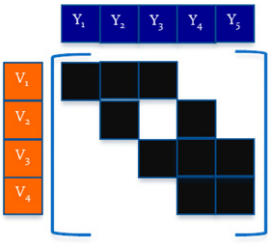

Fig. 2b. Matrix element $M_{[i,j]}$ is marked to black iff data packet V_i is included in code packet Y_j.

All FEC codes come with an overhead. The overhead of the fountain code is defined as α, meaning if N is the number of original symbols, $(1 + \alpha) \cdot N$ is the number of encoding symbols sufficient to recover the N original symbols. Fountain code only works for large N, so it is impossible to give a simple fountain code example. But in principle, the encoding/decoding processes are based on a bipartite graph, in which the edges are generated by a degree distribution function [LTCode]. The three forms of representation of fountain codes are shown in Figs. 2a and 2b: the formula, the bipartite graph or the matrix. In this article, we use the matrix representation because it best explains the encoding and decoding processes. Examples of fountain code include online code, LTCode, and Raptor Code [16].

Compared with other fountain codes, raptor codes are the first known classes of fountain codes with linear encoding and decoding. Sender and receiver must use the same data structures to encode or decode the data, for instance the same pseudo-random number generator, and the same coding matrices.

In the following section, we will explain why synchronized parallel data transfer is a hard problem, and discuss how FEC codes are helpful but not sufficient for large scale parallel data streaming protocols.

3. Parallel data transfer challenges

4K video streaming is very bandwidth intensive and a good sample case for our streaming algorithm. 4K was defined by the Digital Cinema Initiative (DCI) consortium of Hollywood studios in 2003. It describes video with up to 4096 pixels per line and up to 2160 lines per frame. In many practical applications, and in the remainder of this article, 4K refers to a frame size of 3840 × 2160 pixels, which is exactly four times the resolution of 1080p HDTV. 4K video can be captured by 4K cameras, which store every frame as an image file, for instance TIFF. 4K projectors can display this material natively. Streaming uncompressed 4K video is challenging

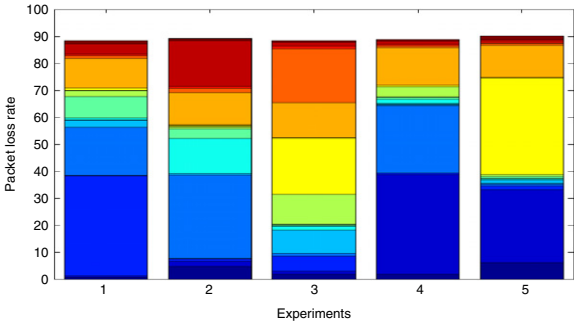

Fig. 3a. Parallel iperf experiments between Chicago and San Diego. • 14 streams, 750 Mbps/stream (10.5 Gbps) over 10 Gbps optical network. • Average packet loss rate is around 6% in all experiments. • *X*-axis: five independent experiments.

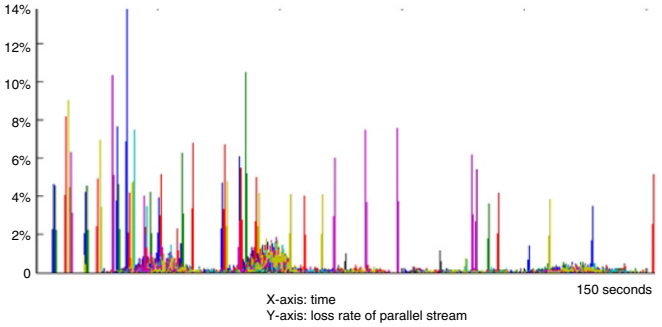

Fig. 3b. Parallel video streaming experiments between Chicago and San Diego. • 13 streams, 700 Mbps/stream (9.1 Gbps) over 10 Gbps optical network. • The test lasted 150 s. • *X*-axis: frame serial number. • *Y*-axis: loss rate of each frame in each stream (14%–0%).

because the data bit-rate is very high, namely, the bit-rate for 32 bit per pixel 4K images at 24 fps is over 6 Gbps, which does not fit into one Gigabit channel, but could be sent through multiple parallel Gigabit connections. So alternatively, assuming we have a large number of 4K frames to send, we can use multiple senders to stream these images to multiple remote receivers. Each of the senders sends out a portion of the images in UDP packets, and each of the receivers receives the corresponding parts of the images and displays these parts on part of a tiled display wall.

Ideally, if all parallel streams are reliably delivered, there will be no problem with this approach. But in reality, UDP packet loss is likely in such high bandwidth applications, which will lead to missing data and synchronization problems at the destination. To our knowledge, there are no publications on the UDP packet loss pattern of large scale parallel data streams over optical networks; but based on our experience and experiments over the OptIPuter network, UDP packet loss behavior is nondeterministic, and a function of many variables, such as the destination nodes' hardware configuration, their workload, OS type and version, OS scheduler, NIC configurations, switch/router scheduling, switch/router queuing strategy, switch/router buffer size, and many more. In particular, if there are multiple streams on the same fiber link nearly saturating the capacity of the link, we found the following typical UDP packet loss behavior over the link:

(1) Very frequently, burst packet loss for an individual stream may happen and will not be recovered by simple FEC code. We can either ignore that which will lead to missing data, or we can go back to the sender to request resending the lost packets, which is time consuming and will make other streams wait.
(2) Bandwidth is not evenly shared by the streams for many reasons. For instance, the UDP protocol does not have constant execution time, neither does switch/router scheduling or queues in switches/routers; the end nodes' capability may vary with time; occasional interference from other network applications may make data streams fluctuate, etc.
(3) The overall/average packet loss rate of all data streams is stable, which means the switching capabilities are stable.

Fig. 3 shows experiments verifying packet loss behavior and packet loss distribution of parallels streams over CaveWave, a 10 Gbps dedicated optical network between Chicago and San Diego. In Fig. 3a, the result of fourteen parallel "iperf" tests on an oversaturated 10 Gbps connection is shown. As expected, an average packet loss rate of 6% is observed in each of the five independent experiments, but packet loss rate patterns and distribution vary significantly between experiments. Fig. 3b is a plot of a similar experiment using video streaming software to measure the frame-by-frame packet loss rate and to understand the burst packet loss pattern. It shows that burst packet loss occurred frequently and in arbitrary streams.

Our experiments confirm that using multiple data streams to transfer data between multiple senders/receivers brings along new challenges. FEC code is helpful in many cases, but not sufficient to handle burst packet loss for individual streams. In the next section, we will discuss our new, parallel Cross-Stream Transfer Protocol (CSTP).

4. CSTP—Cross-Stream Transfer Protocol

Loosely coupled streams are hard to coordinate; linking data streams allows considering new methods to correct for single stream burst packet loss. If one data stream experiences burst packet loss, the receiver does not have to fetch the lost packets from its sender, instead, it gets the data from its peers on the receiving end. Considering the long Round Trip Time (RTT) to fetch a bit from remote senders, being able to get it locally is a significant advantage. However, it is difficult to couple parallel data streams because the independent data streams carry uncorrelated data. To do that, we first need to correlate the data in the data streams. The encoding strategy of our parallel streaming protocol needs to be able to:

(1) *tolerate a certain percentage of packet loss*;
(2) *encode and decode efficiently*;
(3) *encode multiple streams to carry information between them.*

We selected a fountain code as the basis of our work. Fountain code is a type of FEC code which is often used in wireless communication environments where receivers are sometimes not capable of sending back acknowledgements to request the resending of lost packets. In computer science, fountain codes have been used for high bandwidth multicast [19], the robust storage system RubuSTORE [20], etc. But to our knowledge, it has not been used in high bandwidth parallel data transfer applications. In the remainder of this article, we will explore and analyze the coupling of parallel streams using fountain code, introduce our new CSTP coding scheme, and we will discuss the implementation of it in detail.

The input of the protocol is a sequence of original data frames, for example, a series of video frames, and the goal is to transfer these data frames to multiple receivers synchronously. Each original data frame is divided into ***S*** equally sized sub-frames that will be sent out by ***S*** individual senders to their corresponding receivers. This subdivision does not always mean additional computational cost, for instance when the sub-frames are acquired independently, for example, by ***S*** independent cameras. Each sub-frame is further divided into ***N*** equally sized ***original data packets***, and the original data packets are encoded into ***regular code packets*** using fountain code. In addition to the regular code packets, we encode a certain percentage of cross-stream code

Table 1
Definitions of CSTP.

Symbol	Definition
S	Number of data streams
N	Number of original packets of each sub-frame
α	Overhead of fountain code
β	Overhead of transfer protocol
N'	Number of packets (regular code packets and pipe packets) sent to each receiver $N' = N \cdot (1+\alpha) \cdot (1+\beta)$
$M_{1\ldots S}$	Fountain code generating matrices of S data streams which are **known** to all senders and receivers beforehand
$V_{[1\ldots S][1\ldots N]}$	The original data packets
$Y_{[1\ldots S][1\ldots N']}$	The encoded code packets
d_F	*Code degree*: the number of *XOR*ed data packets in a regular code packet
D_F	*Code degree distribution*: the degree distribution function of the fountain code
p	*Pipe width*: percentage of pipe packets
d_S	*Stream degree:* equals 1 for a regular code packet, or $2 \ldots S$ for a pipe packet, meaning how many streams a pipe packet connects
D_S	*Stream degree distribution*: the distribution of d_S. For example, when S is small, we use $D_S(x) = p \cdot x^S + (1-p) \cdot x$

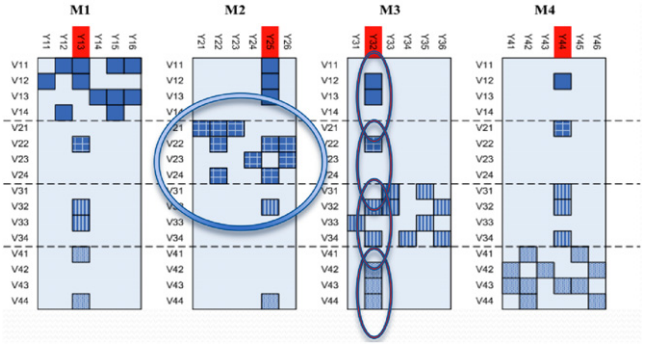

Fig. 4. Sample matrices. The portion in the big circle marked on M2 identified a regular fountain code matrix; the chained four small circles identified a pipe packet in M3. • $S = 4, N = 4, N' = 6, p = 16.7\%$. • X-axis: code data $Y_{[4][6]}$. • Y-axis: original data $V_{[4][4]}$. • Pipe packets (marked in red): Y_{13}, Y_{25}, Y_{32}, Y_{44}.

packets, which are ***XOR*** versions of regular code packets from different sub-frames, and we call them ***pipe packets***. The senders send out the combination of both regular code packets and pipe code packets, and upon receiving those code packets, the receivers use the regular code packets to decode the original data and use pipe packets to communicate between streams to help them reconstruct missing packets.

4.1. Definitions

The definitions used in our protocol are summarized in Table 1.

4.2. Generating $M_{[1\ldots S]}$

$M_{[1\ldots S]}$ are pre-generated generating matrices for CSTP. Similar to fountain code generating matrices like those shown in Figs. 2a and 2b, $M_{[i]}$ is basically a standard fountain code matrix with extra pipe packet columns. So, to generate $M_{[i]}$, we first create a standard fountain code matrix, then randomly insert pipe columns into the matrix. Each pipe column is an assembled column of multiple columns from the S standard fountain code matrices of S streams. A sample is shown in Fig. 4. A pipe packet can consist of code packets from all streams, or only two streams. In the DS function in Table 1, all pipe packets consist of code packets from all S streams. We will use a different DS in our experiments in Sections 6.3 and 6.4. With that distribution function, 50% of the pipe packets have a stream degree of S, and 50% have stream degree 2. A pipe packet with a stream degree of 2 will consist of one regular code packet of its own and one regular code packet randomly selected from the other $S-1$ data streams.

Fig. 5a. Assuming R_1, R_2 and R_4 have received all the packets and successfully recovered their original data (marked as green), but R_3 lost two code packets: Y_{34} and Y_{35} and need help.

4.3. Encoding and decoding algorithms

4.3.1. Encoding algorithm

For each encoder $[1 \ldots S]$:
Step 1: Break its original sub-frame into equally sized data packets.
Step 2: Get next column from M_i, if it is a regular code packet, XOR all packets that are valid in that column; if it is a pipe packet, fetch data from other encoders and XOR all packets that are valid in that column.
Step 3: If enough packets were generated, stop, else go to Step 2.

4.3.2. Decoding algorithm

For each decoder $[1 \ldots S]$
Step 1: Run a native fountain code decoding algorithm.
Step 2: Use pipe packet code to communicate with other decoders: if data needed, request data from other decoders, if data available send to other decoders.
Step 3: Repeat Step 2 until all data are decoded or the decoding process halts.

Fig. 5 illustrates how the decoding process works, and how pipe packets can help receiving nodes recover lost packets. Assuming receivers R1, R2 and R4 have received enough packets, and R3 has lost a big portion of the packets, R1, R2 and R4 then can recover their data by applying the standard fountain code decoding algorithm, as illustrated in Fig. 5a. When they finish the decoding process, all the code packets in the their pipe packets have also been decoded through local communication, shown in Fig. 5b, and the remaining parts of these pipe packets are extra code packets for R3. These extra code packets are sent to R3 so that R3 now will have enough code packets for decoding, as shown in Fig. 5c.

Fig. 5b. The decoded original data packets are distributed to each other via pipe packets, and all pipe packets columns now become regular columns of M_3.

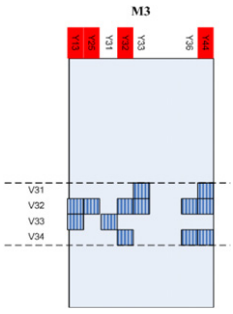

Fig. 5c. R_3 collects pipe packets from R_1, R_2 and R_4, and now can decode its original data.

5. CSTP implementation

CSTP has multiple senders and multiple receivers. The goal is to send sequential data frames from the senders to the receivers synchronously. Each data frame is initially divided into multiple sub-frames, and each sender sends out just one sub-frame to its receiver. The senders keep reading sequential data sub-frames, encode them and send the encoded sub-frames to their corresponding receivers; the receivers receive the data perform cross-stream decoding to reconstruct the original data. The senders and receivers communicate via a high-speed network. This topology, known as a dumbbell topology, characterizes the connections between end clusters.

5.1. CSTP senders

The sending process consists of three major procedures: reading, encoding and sending, which are controlled by a timer-based synchronizer. Due to the deterministic behavior of the senders, the senders are not the bottleneck in our protocol.

The CSTP encoding process is implemented using a coding matrix, as shown in Fig. 6. The coding matrix is a two dimensional (2D) linked list, associated with four memory buffers: DataBuf, CodeBuf, PipeBuf and ExtraBuf. Each node (i, j) in the 2D linked list means code packet $\boldsymbol{i}$ contains data packet $\boldsymbol{j}$ in its "XOR" list. DataBuf stores the original data sub-frame read from the input devices, e.g., a partition of the video frame file, or input from one of multiple cameras. CodeBuf and ExtraBuf are generated from DataBuf using the coding matrix (also see Figs. 2a and 2b). CodeBuf includes regular code packets and pipe packets; ExtraBuf includes only regular code packets for the sending nodes to construct their pipe packets. PipeBuf are regular code packets received from the senders. Each packet in ExtraBuf will be sent to the PipeBuf of one of the other senders.

A regular code packet (only composed of local data packets) in CodeBuf is encoded immediately from DataBuf. A pipe packet that contains packets in PipeBuf will wait until that packet is received from the peers. Practically, the senders will encode ExtraBuf first and distribute the ExtraBuf packets to its peer senders so that the encoding of CodeBuf will not need to wait for the PipeBuf packets. Therefore, the encoding process is a deterministic process. Finally, the sender only sends out the code packets in CodeBuf to the receiver, while ExtraBuf is not sent.

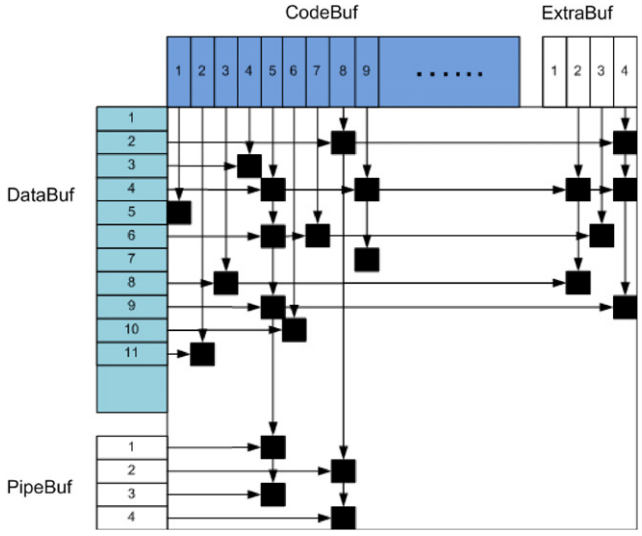

Fig. 6. Code matrix data structure.

5.2. CSTP receivers

The receivers have two major threads: a streaming thread and a decoding thread. They operate on two major data structures: a circular buffer and a coding matrix.

5.2.1. Circular buffer

A circular buffer is used to buffer the received data sub-frames. It has two pointers: a reader's pointer and a writer's pointer, controlled by the reading and writing thread respectively. The writing thread (the stream thread) reads UDP packets from the network and saves them into the blocks pointed to by the current writer's pointer. The reading thread (the decoding thread) fetches a block from the circular buffer and sends it to the decoder for decoding. Both threads will increase their circular buffer pointers after finishing writing/reading.

5.2.2. Decoding matrix

Although the decoding matrix data structure is the same as the encoding matrix data structure, the decoding algorithm is much more involved. After the reading thread fetches a sub-frame from the circular buffer, it will fill the decoding matrix's CodeBuf and associate with it a Mask Variable, which is an array of "Boolean" values, marking each packet in the CodeBuf either "Lost" or "Received" so that only "Received" packets will be used for decoding. Because the packet loss is not known before hand, the decoding process is nondeterministic.

In the decoding process of standard LTCode or Raptor code, the decoder keeps looking for packets in CodeBuf whose remaining code degree equals one, decoding that packet by copying the code packet to the data packet, and distributing the decoded data packet to all other code packets containing that data packet, until the process stops without any further possible movement.

In our algorithm, we first perform this standard decoding process based on CodeBuf and the Mask Variable using the decoding

matrix. After this is done, a decoder either has recovered all its packets in DataBuf, or only partially decodes DataBuf due to the packet loss in CodeBuf. If all decoders have successfully decoded their sub-frames, the whole data frame is successfully received and the algorithm can proceed to the next data frame.

If some receivers cannot recover all their data packets, the decoders will need to work together to recover the un-decoded parts. Extrabuf and PipeBuf are used to communicate between decoders to decode those parts through pipe packets.

(a) *From PipeBuf to ExtraBuf*

Assuming receiver A has received a pipe packet $Y = V_{A,1} \oplus V_{A,2} \oplus \cdots \oplus V_{A,k} \oplus Y_B$, and has decoded all its data packets $V_{A,[1\ldots k]}$, it can now decode Y_B by doing k XOR operations on Y, and $Y_B = Y \oplus V_{A,1} \oplus V_{A,2} \oplus \cdots \oplus V_{A,k}$.

Y_B becomes a new regular code packet in A's PipeBuf for receiver B, so that it will be sent to B's ExtraBuf to help B decode un-decoded data packets as if B receives one more regular code packet from its sender.

(b) *From ExtraBuf to PipeBuf*

In each decoder, ExtraBuf is set to "zero" before the decoding process. During the first phase of the decoding process, in addition to the standard decoding process, all the decoded data packets are also distributed to the associated code packets in ExtraBuf, and the code packets in ExtraBuf will "XOR" the data packet to itself. Therefore, for those decoders which have successfully decoded all their data packets, they also have generated the code packets in ExtraBuf. As explained before, ExtraBuf are used by the encoder to encode a pipe packet during the encoding process and are not sent to the receivers. Just as the encoder sends the code packet in ExtraBuf to its peers to generate a pipe packet in the peers, now the newly generated code packets in ExtraBuf are sent to the same pipe packet. For example, assuming a pipe packet Y is located in receiver A, and $Y = V_{A,1} \oplus V_{A,2} \oplus \cdots \oplus V_{A,k} \oplus Y_B \oplus Y_C \oplus Y_D$, once B, C and D generate Y_B, Y_C and Y_D in their ExtraBuf, Y_B, Y_C, and Y_D will be sent to A, so that the pipe packet Y in A can be converted into a new regular code packet Y', $Y' = Y \oplus Y_B \oplus Y_C \oplus Y_D = V_{A,1} \oplus V_{A,2} \oplus \cdots \oplus V_{A,k}$. Y' is a new code packet for A, as if A had received a new code packet from its sender.

ExtraBuf and PipeBuf are mutually beneficial. In practice, Step (a) and Step (b) will start as early as possible, overlapping with the self-decoding process, and will repeat in the post decoding process.

5.3. Reliable delivery

In our CSTP implementation, the goal is to tolerate unevenly distributed burst packet loss below a certain threshold, but when significant packet loss happens for many streams, CSTP cannot guarantee the reliable delivery of the data. Based on our experiments, CSTP tolerates burst packet loss quite well and is sufficient for reliable delivery in many practical scenarios. However, in case packet loss is significant for many data streams, and after the cross-stream decoding process there are still unrecovered data packets left, more data would have to be requested from the senders if reliable delivery is required. Two categories of issues have to be discussed: dynamic sending rate and lost packets resending. We have not covered these in our CSTP implementation, but they could be integrated in future versions of the algorithm. Dynamic sending rates have been studied in [9–11], and we have a similar issue in CSTP. In order to resend lost packets, other approaches maintain a bitmap for lost packets and retransfer lost packets, which is straightforward but ineffective, because the senders have to keep all data in main memory to avoid random disk access. For big file transfers, RAM will be used to keep the original data quickly accessible, like in RBUDP. In CSTP, instead, we only need to keep a small percentage of extra code packets in the RAM of the senders. So when receivers ask for more data, the senders simply send out these new code packets. This is less expensive than saving all original data for retransfer.

5.4. Synchronization

Synchronization mechanisms are widely used in a cluster environment. In CSTP, both senders and receivers use a synchronizer to do frame-by-frame synchronization for video transfer. The synchronization scheme is simple: all senders/receivers send a "sync" message to a master node, and when the master node collected all "sync" messages, it broadcasts a confirmation signal to all other nodes, which makes them continue with the next frame. This mechanism is similar to the synchronization in computer graphics clusters, like COVISE [21], where frame buffer swaps need to be synchronized in a similar way.

5.5. Scalability of CSTP

The scalability of CSTP is primarily subject to the communication cost of CSTP. In CSTP, both encoding and decoding algorithms will incur certain communication cost among the nodes of a cluster. In the encoding process, the communication is deterministic, while decoding communication is more nondeterministic because the decoding process is subject to packet loss. But on both sides, the communication cost of the CSTP is reasonably small when S is small because only pipe packets are needed to communicate among the nodes. In the worst case, the total communication cost has an upper bound of $(\boldsymbol{S} - \boldsymbol{1}) \cdot \boldsymbol{p}$, e.g., if $\boldsymbol{p} = 0.05$, $\boldsymbol{S} = 4$, the decoding communication cost is about 15% of the size of the original data.

The scalability will be more important when S is bigger. For small numbers of parallel streams, the stream degree distribution function in Table 1 works quite well. However, for larger scale applications, which use more than ten parallel streams, the communication cost with that distribution function will increase linearly with the number of streams. Therefore, we are designing and experimenting with different distribution functions. Our preliminary results show that with constant communication overhead, the CSTP protocol scales very well to larger numbers of parallel machines, with reasonable assumption of parallel data loss pattern.

Another factor that can affect the scalability of CSTP is the synchronization of large numbers of nodes. In applications requiring frequent synchronization operations, more thought needs to be put into this. In our current experiments, the synchronizer in CSTP can complete more than 600 synchronizations per second among 15 machines. With 100 machines, that number is about 360 per second. Therefore, we do not foresee this as a limiting factor.

6. Experimental results

In our experiments, we evaluated the decoding performance of the most recent version of the fountain code, and simulated multiple data streams using cross-stream coding algorithms, test the stability of decoding algorithms with fluctuating data flows and various packet loss rates, and compare the transfer speed of CSTP with parallel RBUDP. The fountain code we selected is the Raptor Code, which has linear time encoding and decoding algorithms. The precode and LTCode we used are those described in [17]:

Precode: LDGM + HDPC

LTCode degree distribution:

$D_F(x) = 0.0156 \cdot x^{40} + 0.0797 \cdot x^{11} + 0.111 \cdot x^{10} + 0.113 \cdot x^4 + 0.210 \cdot x^3 + 0.456 \cdot x^2 + 0.00971 \cdot x.$

The average code degree of LTCode is 4.6116, and the average degree of Raptor Code is the sum of LTCode degree and precode degree, which is a function of N, e.g., when $N = 1024$, Raptor Code has an average code degree of 13. The average code degree

Table 2
Decoding speed of the Raptor Code.

# of original packets	# of Raptor Code packets	Decoding percentage (%)	Decoding speed (7680 B/pkt) (Gbps)	Decoding speed (3840 B/pkt) (Gbps)
512	650	100	2.51	3.17
1024	1200	100	2.00	2.22
2048	2400	100	1.68	1.82
4096	4800	100	1.45	1.52

determines the complexity of the decoding algorithms. Theoretically, assuming the packet size is fixed, the complexity of the decoding algorithm is $O(D \cdot N)$, where D is the average degree of Raptor Code, and N is the number of original data packets.

The platform we used for our experiments is a 2006 Dell XPS 720, with:

CPU: Intel(R) Core(TM)2 Quad CPU @ 2.40 GHz
Cache size: 4096 KB
Memory: 4 GB
OS: 64 bit Version of CentOS 5

6.1. Raptor Code decoding performance

An important measurement is with how much overhead Raptor Code can recover the original data. Although this has been discussed in prior work, we have done our own experiments. In our test, N cannot be too small since the Raptor Code is based on probability theory and will not hold for small values of N. When N is around 1000, Raptor Code can recover original data with an overhead of around 6%–8%.

To test the decoding speed of the Raptor Code, we run our implementation of it (there is no open source implementation of the Raptor Code) with one thread to decode native Raptor Code, and the decoding speed is shown in Table 2. We tested 7680 and 3840 bytes packets.

The decoding algorithm is CPU intensive, and consumes significant memory bandwidth. Currently in our single thread experiments, the decoding algorithm is capable of decoding a single HD stream. The results also suggest that the cache miss rate is an important factor. As shown, when we increased the number of original packets, the decoding speed went down. We believe this is because the cache miss rate became higher with the larger data set.

6.2. CSTP recovery performance

First, we need to understand whether streams can help each other when one stream loses significantly more packets than others. We use the following settings:

$p = 5\%$

$S = 4$

$N = 1024$

$D_s(x) = p \cdot x^S + (1 - p) \cdot x.$

Our next experiment we tested how CSTP works with random packet loss across all streams. We encoded 1000 frames into four sets of code packets using four generating matrices, and then simulated sending a certain number N' of code packets to each receiver. We did not assume all the packets were correctly received, but purposely dropped some of them. And based on our network assumption, the packet loss rate of each receiver is randomly generated with every new frame. The average packet loss rate of the receivers is set to 2%, 3% and 4% respectively.

We then plotted a figure with the X-axis representing N', and the Y-axis representing the possibility of successful recovery of the full frame. The result shows that with a reasonable overhead, e.g., 5%–15%, our approach can recover significant packet loss in all streams and the possibility of fully recovered frames is ∼99.99% (Fig. 7).

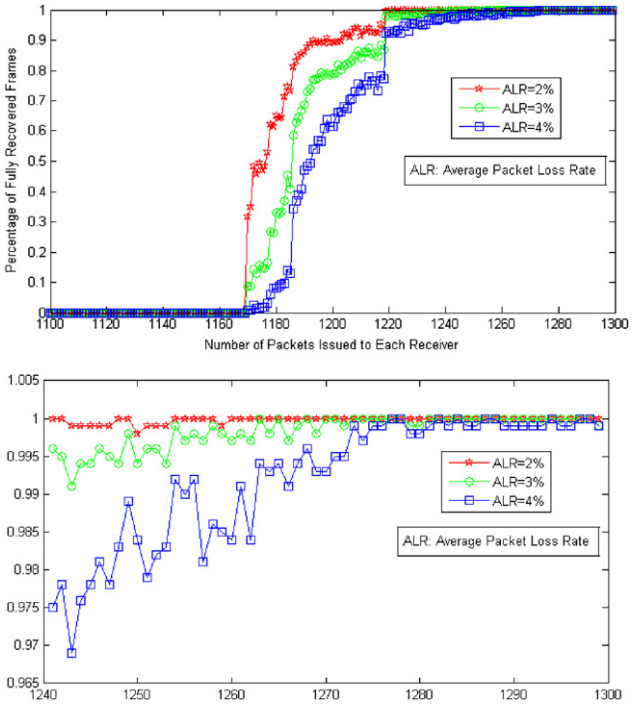

Fig. 7. These lines show the function of frame recovery percentage to the number of issued packet to receivers, and the lower figure is the zoom-in of the top-right part of the upper figure. • X-axis: number of code packets. • Y-axis: percentage of recovered original packets.

6.3. Burst packet loss recovery

By carefully controlling the sending rate, we are able to deliver data with only a small average packet loss percentage. Ideally, if all packet loss were completely random and evenly distributed across all streams, simple FEC would solve the problem. But in practice as shown in Figs. 3a and 3b, packet loss patterns are not uniformly distributed, and FEC cannot recover burst packet loss. Here we compare the recovery capability of native FEC code and CSTP under a variety of assumed packet loss patterns. We use 10 parallel data streams to do the comparison with an assumption of an average packet loss (APL) of 7.5%. We chose this APL to make the comparison more interesting. The settings of CSTP are:

$p = 5\%$

$S = 10$

$N = 1024$

$D_s(x) = p \cdot (50\% \cdot x^S + 50\% \cdot x^2) + (1 - p) \cdot x.$

The meaning of $D_s(x)$ *is: for the pipe packets,* 50% *have stream degree* S, 50% *have stream degree* 2.

The four packet loss patterns we experimented with are:

(1) Unified pattern: all packet loss is evenly distributed across all data streams.
(2) Linear pattern: packet loss of all data streams fits into a linear pattern.
(3) Polynomial pattern: packet loss of all data streams forms a polynomial pattern.
(4) Random chop: we treat the overall packet loss like a pie, and randomly chop the pie to divide it into 10 parts.

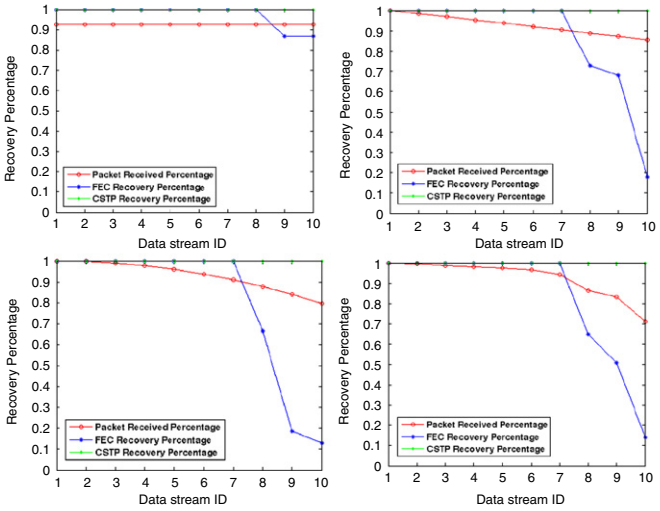

Fig. 8. CSTP vs. pure FEC. • X-axis: data stream ID from 1 to 10. • Y-axis: packet recovery percentage.

Our experiments show a significant advantage of CSTP over native FEC code (Fig. 8). In all four cases, CSTP can recover 100% of the original data in all 10 data streams. But the independent FEC code fails in some data streams in all cases. With packets loss evenly distributed, the FEC code only recovers about 90% of the original data in two streams. With more unevenly distributed burst packet loss the FEC code's performance is much worse than CSTP.

6.4. CSTP Parallel transfer performance

We then compared the transfer speed of CSTP to the popular UDP transfer protocol RBUDP. To measure the protocol performance and avoid disk operations, we modified the RBUDP protocol by eliminating the disk I/O operations, and compared both protocols using 2–10 parallel streams between EVL in Chicago and Calit2 in San Diego. The link between them was a 10 Gbps optical network. The settings of CSTP are:

$p = 5\%$

$S = 2 \ldots 14$

$N = 1024$

$D_s(x) = p \cdot (50\% \cdot x^S + 50\% \cdot x^2) + (1 - p) \cdot x.$

As illustrated in Fig. 9, the overall transfer speed of CSTP is about 30% higher than the overall transfer speed of parallel RBUDP streams; the maximum throughput of CSTP is about 20% higher than that of RBUDP.

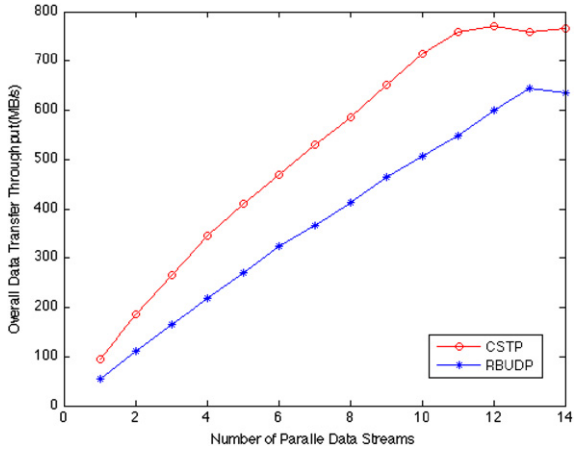

Fig. 9. CSTP vs. RBUDP in parallel data transfer. • X-axis: number of parallel data streams. • Y-axis: overall data transfer speed in MB/s.

7. Conclusion

We studied the problems which come along with large scale data transfer/streaming, and present a solution, the CSTP protocol. Traditional point-to-point protocols are not suitable for today's cluster-to-cluster communication scenarios, used by scientific applications, and existing multi-to-multi-point protocols do not sufficiently address the synchronization issue of parallel data streaming and cannot tolerate burst packet loss [22]. In this paper, we propose our new Cross-Stream Transfer Protocol (CSTP), which used cross-stream fountain code and pipe packets to tightly couple streams and help lossy streams to decode correctly and tolerate burst packet loss. Our experimental results show that the CSTP meets our expectation in terms of burst packet loss tolerance, decoding efficiency, communication overhead, and overall data throughput.

Numerous challenges remain to be addressed. Our future work will mainly look at the scalability issue of our CSTP algorithm. We also intend to add modules for reliable delivery to the current CSTP implementation.

References

[1] CineGrid: http://www.cinegrid.org.
[2] Daisuke Shirai, Tetsuo Kawano, Tatsuya Fujii, Kunitake Kaneko, Naohisa Ohta, Sadayasu Ono, Sachine Arai, Terukazu Ogoshi, Real time switching and streaming transmission of uncompressed 4K motion pictures, Future Generation Computer Systems 25 (2) (2009) 192–197.
[3] Laurin Herr, et al. International real-time streaming of 4K digital cinema, demonstration, in: iGrid 2005. http://www.igrid2005.org/program/applications/videoservices_rtvideo.html.
[4] Thomas A. DeFanti, Jason Leigh, Luc Renambot, Byungil Jeong, Larry L. Smarr, et al., The OptIPortal, a scalable visualization, storage, and computing interface device for the OptiPuter, Future Generation Computer Systems 25 (2) (2009) 114–123.
[5] Thomas A. DeFanti, Gregory Dawe, Daniel J. Sandin, Jurgen P. Schulze, Peter Otto, Javier Girado, Falko Kuester, Larry Smarr, Ramesh Ral, The StarCAVE, a third-generation CAVE and virtual reality OptIPortal, The International Journal of FGCS (ISSN: 0167-739X) 25 (2) (2009) 169–178.
[6] GLIF: http://www.glif.is.
[7] OPIPUTER: http://www.optiputer.net.
[8] Larry Smarr, The OptIPuter and its applications, in: 2009 IEEE LEOS Summer Topicals Meeting on Future Global Networks, July 22, 2009, pp. 151–152, doi:10.1109/LEOSST.2009.5226201.
[9] E. He, J. Leigh, O. Yu, T.A. DeFanti, Reliable blast UDP: predictable high performance bulk data transfer, in: IEEE Cluster Computing 2002, Chicago, Illinois, September 2002.
[10] Yunhong Gu, Robert L. Grossman, UDT: UDP-based data transfer for high-speed wide area networks, Computer Networks 51 (7) (2007).
[11] V. Vishwanath, J. Leigh, E. He, M.D. Brown, L. Long, L. Renambot, A. Verlo, X. Wang, T.A. DeFanti, Wide-area experiments with LambdaStream over dedicated high-bandwidth networks, in: IEEE INFOCOM 2006.
[12] SAGE: http://www.evl.uic.edu/cavern/sage/index.php.
[13] 4KStreaming: http://www.envision.purdue.edu/4kstream/.
[14] Farhan Aadil, Shahzada Khayyam Nisar, Wajahat Abbas, Asim Shahzad, Reusable IP core for forward error correcting codes, International Journal of Basic & Applied Sciences IJBAS-IJENS (ISSN: 2077-1223) 10 (01) (2010) 24–30.
[15] Fountain code: http://en.wikipedia.org/wiki/Fountain_code.
[16] Amin Shokrollahi, Raptor codes, IEEE Transactions on Information Theory 52 (2006) 2551–2567.
[17] Raptor code: http://algo.epfl.ch/contents/output/presents/.
[18] M. Luby, LT-codes, in: Proc. 43rd Annu. IEEE Symp. Foundations of Computer Science, FOCS, Vancouver, BC, Canada, November 2002, pp. 271–280.
[19] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, Atul Singh, Splitstream: high-bandwidth multicast in cooperative environments, in: SOSP'03 ACM Symposium on Operating Systems Principles, vol. 19, Bolton Landing, Lake George, New York, ETATS-UNIS (19/10/2003).
[20] Huaxia Xia, Andrew Chien, RobuSTore: a distributed storage architecture with robust and high performance, in: Proceedings of Supercomputing, SC07, Reno, NV, 2007.
[21] COVISE: http://www.hlrs.de/organization/av/vis/covise/.
[22] Shaofeng Liu, Jurgen P. Schulze, Tomas A. Defanti, Synchronizing parallel data streams via cross-stream coding, in: 2009 IEEE International Conference on Networking, Architecture, and Storage, NAS, 2009, pp. 333–340.

Shaofeng Liu is a Ph.D. Candidate in the Computer Science and Engineering Department at the University of California, San Diego. His research interests include networking protocols, Distributed System, Job schedule, etc. Shaofeng holds a B.S degree and a M.S. degree from Tsinghua University in Beijing, China.

He has worked for IBM Company for two years in Beijing.

Jürgen P. Schulze is a Research Scientist at the California Institute for Telecommunications and Information Technology, and a Lecturer in the computer science department at the University of California San Diego. His research interests include scientific visualization in virtual environments, human–computer interaction, real-time volume rendering, and graphics algorithms on programmable graphics hardware. He holds an M.S. degree from the University of Massachusetts and a Ph.D. from the University of Stuttgart, Germany. After his graduation he spent two years as a post-doctoral researcher in the Computer Science Department at Brown University.

Thomas A. DeFanti received a B.A. in Mathematics from Queens College in 1969, a M.S. in Computer Information Science from Ohio State University in 1970, and here three years later in 1973 a Ph.D. in Computer Information Science. He did his Ph.D. work under Charles Csuri in the Computer Graphics Research Group. For his dissertation, he created the GRASS programming language.

In 1973, he joined the faculty of the University of Illinois at Chicago. In the next 20 years at the University, DeFanti has amassed a number of credits, including: use of EVL hardware and software for the computer animation produced for the Star Wars movie. With Daniel J. Sandin, he founded the Circle Graphics Habitat, now known as the Electronic Visualization Laboratory (EVL). DeFanti contributed greatly to the growth of the SIGGRAPH organization and conference. He served as Chair of the group from 1981 to 1985, co-organized early film and video presentations, which became the Electronic Theatre, and in 1979 started the SIGGRAPH Video Review, a video archive of computer graphics research.

DeFanti is a Fellow of the Association for Computing Machinery. He has received the 1988 ACM Outstanding Contribution Award, the 2000 SIGGRAPH Outstanding Service Award, and the UIC Inventor of the Year Award.