Synchronizing Parallel Data Streams via Cross-Stream Coding

Shaofeng Liu[a,b], Jurgen P. Schulze[b], Tomas A. Defanti[b]

[a] Dept. Of Computer Science and Engineering, University of California San Diego (UCSD), United States
[b] California Institute for Telecommunications and Information Technology (Calit2), University of California San Diego (UCSD), United States

***Abstract*— Streaming very-high-definition visualization data objects on top of optical networks is critical in many scientific research areas, including video streaming/conferencing, remote rendering on tiled display walls, 3D virtual reality applications, etc. Current data streaming protocols rely on UDP as well as a variety of compression techniques. However, none of the protocols scale well to the parallel streaming model of large scale graphic applications, and the existing parallel streaming protocols have limited synchronization mechanisms to synchronize the streams efficiently, and are prone to be slowed down by just one slow stream. In this paper, we propose a new parallel streaming protocol that can stream synchronized multiple Gbps media content over optical networks through reliable Cross-Stream packet coding, which not only tolerates random UDP packet loss, but also aims to achieve good synchronization performance across multiple parallel data streams with reasonable coding overhead. We simulated the approach, and the results show that our approach can generate steady throughput with fluctuating data streams.**

***Index Terms*—Cross-Stream Coding, Streaming, Optical Network**

I. Introduction

In recent years, ultra-high-resolution displays have become a standard infrastructure in scientific research. These displays are typically achieved by tiling together an array of standard LCD displays into a display wall, using a PC cluster to drive it. [3]. High-speed research optical network [1], on the other hand, make is possible for scientists to use these ultra-high resolution displays over long distance in their scientific applications like very-high-definition video streaming/conferencing [2], real-time data visualization generated by remote scientific instruments, etc. As a perfect example of the combination of display walls and high-speed network, the OpTiPuter [2] research project, funded by the American National Science Foundation, constructed 1Gbps-10Gbps optical network infrastructure and middleware aiming to make interactive access of remote gigabyte to terabyte visualization data objects and bring them to its visual interface-OptIPortals [3], a tiled display wall with hundreds of million pixels, as shown in Figure 1 a). Figure 1 b) shows a different setting of display walls, the StarCAVE [23], which uses 16 high-definition projectors to construct a 3D virtual room where people can navigate 3D virtual reality objects.

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

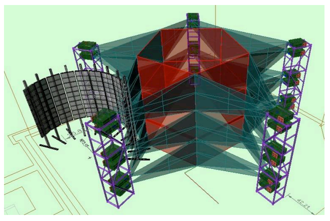


Fig. 1 b): The STARCAVE, a third-generation CAVE and virtual reality OptIPortal in Calit2, UC San Diego, ~68,000,000 pixels

The scaling up of display devices from a single PC with a single display to a cluster of PCs with a cluster of displays has created new multi-to-multi communication models, which require very-high-bandwidth parallel data streaming protocols between termination display devices while appearing as point-to-point communications between them. Those models are a challenge to the transport protocols in-between scientific applications and the hardware infrastructures. The traditional Transport Control Protocol (TCP) is too slow because of its window-based congestion control mechanism, particularly for long distance communications. Alternatives like RBUDP [6], UDT [7], and LambdaStream [12] 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 only 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 loss based on the gap between the receiving time of two consecutive UDP packets, and if that gap is bigger that expected, the sender will reduce the sending rate accordingly. These protocols, however, cannot be scaled to the cluster-to-cluster communication model in a

Shaofeng Liu is a Ph.D candidate in the Computer Science and Engineering Dept. at the University of California San Diego (e-mail: s8liu@ ucsd.edu).
Jurgen Schulze, Ph.D, is a Project Scientist at Calit2, UCSD (e-mail: jschulze@ucsd.edu).
Thomas A. DeFanti, Ph.D., at the University of California, San Diego, is a Senior Research Scientist at Calit2. (email: tdefanti@ucsd.edu).

straightforward way. On the other hand, existing parallel data streaming protocols do not sufficiently address the synchronization issue of multiple parallel data streams. For example, SAGE [5] extended LambdaStream to multiple senders/receivers by adding a simple synchronization module to both senders and receivers, and the 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. Similarly, [11], a parallel 4K(e.g. 3840x1920 image resolution) video streaming model developed by Purdue University, uses six streaming servers to stream data to a tiled display wall driven by twelve PCs, and the servers send frames as quick as possible, and receivers buffer the data and wait until all receivers have received their next frame. However, since both approaches synchronize streams by waiting, if some stream is slowed down by a higher packet loss rate, all other streams have to wait for it, which could be a significant overhead in long distance communication scenarios, and will cause severe jittering problems.

Basically, point-to-point protocols are not scalable, and multipoint-to-multipoint protocols are hard to be efficiently synchronized. In this paper, we study the cluster-to-cluster real time communication model, and will focus on the synchronization challenge brought up by this model. We put forward a new approach aiming to achieve reliable synchronized data transfer, which can be potentially scalable to tens/hundreds of streams that can stream multiple 10Gbps for future graphic applications. We simulated our protocols and the results showed that even with fluctuating network flows, small but uneven distribution of packet losses, our approach could achieve reliable synchronized throughput. In Section 2, we introduce related work. In Section 3&4, we discuss our approach for reliable synchronized parallel streaming. We show some of our simulation results in Section 5, and in Section 6, we conclude and suggest future research directions.

II. 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.*" Since most high bit-rate streaming 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 amount of random packet loss, but is not always effective. For example, most FEC algorithms cannot recover burst packet loss over a certain bound.

In recent years, a new series of coding techniques have been used in FEC, called fountain code [19][20][21][22], which *"can generate 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."* [19] –In our case, a symbol is a UDP packet.

In practice, the overhead of fountain code is defined as α. 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. There are three forms of representations of typical fountain code, and Figure 2 shows how a piece of code is represented by a formula, a bipartite graph or a matrix. In this paper, we use the matrix representation because it is easier to explain the encoding and decoding processes with

$$Y_1 = V_1,$$
$$Y_2 = V_1 \oplus V_2,$$
$$Y_3 = V_1 \oplus V_3,$$
$$Y_4 = V_1 \oplus V_2 \oplus V_3$$
$$Y_5 = V_3 \oplus V_4$$

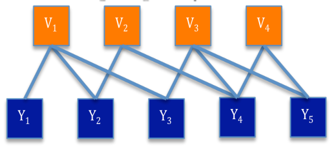


Fig. 2 a). V_1-V_4 are original data packets; Y_1-Y_5 are encoded data packets. The formula and bipartite graph have the same meanings.

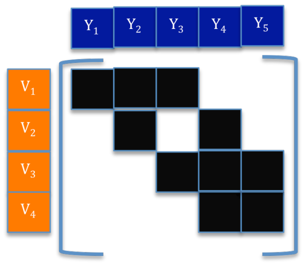


Fig. 2 b). Matrix element M[i,j] is marked to black iff data packet V_i is included in code packet Y_j .

matrixes. Examples of fountain code include online code, LTCode, Raptor Code, etc. Currently, Raptor Code is the fastest known fountain code, which uses linear time encoding and decoding algorithms [20].

In the next section, we will explain why parallel data streaming is a hard problem, and discuss why FEC codes are useful but not sufficient for large-scale parallel data streaming protocols.

III. Parallel Data Streaming Dilemma

To be simple, we use 4K media streaming as an example. **4K** is a high-definition image standard, which roughly specifies an image with 4000 pixels per line and 2000 lines per image. In this paper, by default, a 4K image means an image with 3840x2160 pixels, exactly four times the resolution of HDTV. 4K images are usually captured by 4K cameras, stored in TIFF files, and shown by 4K projectors. Streaming uncompressed 4K video is very difficult because the data bit-rate is very high, for instance, the bit-rate of streaming RGBA (32bit/pixel) 4K images at 24fps is over 6Gbps, which is too high to be handled by one stream and has to be done by multiple data streams. So alternatively, assuming we have infinite 4K images, we can use multiple senders to stream these images to multiple remote receivers. Each of the senders sends out a portion of the images via UDP packets, and each of the receivers receives the corresponding parts of the images and displays the images simultaneously on a display wall.

Ideally, if all parallel streams are reliably delivered, there will be no synchronization issue; but in reality, UDP packet loss is inevitable in our high-bandwidth applications, which will lead to synchronization problems. To our best knowledge,

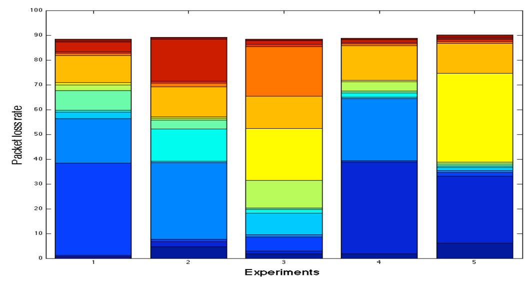


Fig. 3 a). Parallel iperf experiments between Chicago and San Diego.
➢ 14 streams, 750Mbps/stream (10.5Gbps) over 10Gbps optical network
➢ Average packet loss rate is around 6% in all experiments
➢ X-axis: five independent experiments

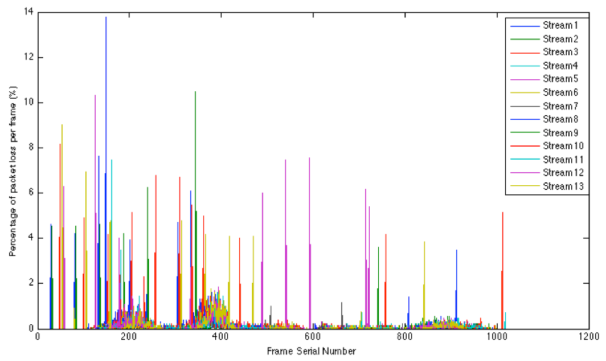


Fig. 3 b). Parallel video-streaming experiments between Chicago and San Diego.
➢ 13 streams, 700Mbps/stream (9.1Gbps) over 10Gbps optical network
➢ The test lasted 150 seconds
➢ X-axis: frame serial No.
➢ Y-axis: loss rate of each frame in each stream (14% - 0%)

the UDP packet loss pattern of large scale parallel data streams over optical network has not been very well studied; but based on our experience and experiments over the OptIPuter network, UDP packet loss behavior is very nondeterministic, and is basically a function of N variables, including end nodes hardware configuration, end nodes real time workload, OS version, OS scheduler, NIC, switch/router scheduling, switch/router queuing strategy, and many others. In particular, if there are multiple streams on the same fiber link nearly saturating the capacity of the link, we found the following UDP packet loss behavior over the link:

1) Very frequently, blast packet loss for some 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 all the way back to the sender to request lost packets, which is time consuming and can cause synchronization problems.

2) Bandwidth is not even shared by streams due to many reasons. E.g. the UDP protocol is not fair; switch/router scheduling may not be fair; queues in switch/router may not be fair; end nodes capability may vary with time; occasional interfere from other network applications may bring fluctuation to data streams, etc.

3) The overall/average packet loss rate of all data streams is stable.

Figure 3 shows some experiments verifying the packet loss behaviors and packet loss distribution of parallels streams over CaveWave, a10Gbps dedicated optical network between Chicago and San Diego. Fig 3 a) run 14 parallel “iperf” tests, 750Mb each totaling 10.5Gbps, which exceeded the 10Gbps capacity of the link. As we expected, an average of 6% packet loss rate occurred in each of the five independent experiments, but the packet loss rate patterns vary a lot and the distribution is very non-deterministic. Fig 3 b) is a plot of a similar experiment using video streaming software to measure frame-by-frame packet loss rate and to understand the burst packet loss property. It shows that burst packet loss occurred to random streams very frequently.

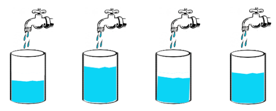


Fig. 4 a). Four faucet are pouring water into four containers. Because the water streams differ, the water levels in the containers differ.

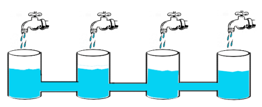


Fig. 4 b). Now, we install pipes between adjacent containers, and the water levels are even despite the water streams remain.

Our experiments showed that streaming strict synchronized parallel data streams is not trivial. FEC code can help in many cases, but apparently not all the time. In movie streaming, losing a few packets might be acceptable and we can use interpolation methods to simulate the lost pixels and probably will not affect the image quality a lot, as long as the packet loss rate is low. But in other scientific applications, for example, earthquake simulation and visualization, lost packets may contain critical data that cannot be simulated.

Our experiments showed that loosely coupled streams make it hard for a health stream to help a faulty stream, so we use tightly coupled parallel data streams.

Let’s considering a model similar to the water-streaming model, as shown in Fig 4. In a), four faucets are pouring water into four containers. Because water can splash out of the containers, the water levels are not even in four containers. Fortunately, we have a simple solution for this problem as shown in Fig 4. b): we add pipes between adjacent containers, so that the water levels are automatically equalized. Although this water is not directly applicable to our parallel data streaming model, it provides an important insight that we might be able to tightly couple data streams to design a correlated parallel data streaming protocol instead of a protocol consisting of multiple independent data streams.

The idea of coupling data streams suggests that once one stream suffers from packet loss, it does not have to fetch the lost packets from the sender, instead, it fetches the data from its peers. Considering the high 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 apply the water pipe coupling method directly to the parallel data streaming model, because water drops are all identical while data packets represent different data. So, to take advantage of

the water pipe model, we must weaken the difference between individual data packets and make them nearly "identical". Therefore, we came up with the idea to use encoding methods in our parallel streaming protocol, and the code should:
1) Be able to tolerate certain percentage of packet loss.
2) Have fast encoding and decoding algorithms to handle multiple Gbps.
3) Have cross-stream encoding capability to communicate between streams

We selected the fountain code as the basis of our work. The fountain code is mostly used in lossy wireless communication environments where receivers are sometimes not capable of sending back acknowledgements to notify lost packets. In computer science, the fountain code has been used for high-bandwidth multicast [14], the robust storage system RubuSTORE [15], etc. But to our best knowledge, it has not been used in high bandwidth parallel data streaming applications to solve synchronization issues. In the next section, we will briefly explore and analyze the problem of coupling parallel streams using the fountain code, and introduce our new 2D Cross-Stream coding approach, which will be discussed in detail in Section 5.

IV. COUPLING PARALLEL DATA STREAMS

Following the analysis in the previous sections, we will use UDP and the fountain code as the basic building blocks to construct our parallel data streaming model, and couple parallel streams to synchronize them. In this section, we will briefly analyze two naïve ways of coupling parallel streams, and explain why they are flawed, and then introduce our new 2D Cross-Stream code.

1. *Unified Parallel Code*

In the 4K streaming example, each image can be divided into equally sized **N** original data packets. The unified approach generate $(1+\chi)\times N$ coded packets from N original data packets, and evenly distributes them to four receivers, and as long as the sum of received packets of all four receivers exceeds $(1+\alpha)\times N$ (α is the overhead of the fountain code, $\alpha<\chi$), the image can be recovered.

The issue of this approach is the very high decoding communication cost. The major decoding communication comes from the decoding algorithm, where once an original data packet is decoded, it has to be distributed to all receivers that have received code packets containing that data packet. Since the possibility of one data packet to be included in multiple code packets is very high in Fountain Code, this overhead will significantly slow down the decoding process. Moreover, once the decoding process is completed, the data needs to be relocated to the right receivers. For example, if we show the 4K images on four HD displays driven by four receivers, we need R1 (Receiver 1) to have the upper-left part of the image, R2 to have the upper-right part of the image, and so on. But, during the decoding process, although the decoders have recovered all the original data packets, those data packets are randomly distributed across the receivers and need to be re-distributed to the right receiver based on their positions in the image. The decoding communication cost is on the order of the total number of original data packets.

2. *Cross-Distributed Code*

We can also first divide each 4K image into sub-images, and **separately** encode them into encoded sub-images. After that, rather than sending each encoded sub-image to one receiver, we can first mix the encoded sub-images together, and send a portion of the mixed data to each receiver. Once the receivers have received their portion of the mixed data, they first communicate to reassemble their corresponding encoded sub-images and then start to decode independently.

The benefit of this approach is that even if one stream lost many packets, those lost packets are likely evenly distributed among all encoded sub-images, so that all receivers can successfully decode their encoded sub-images with high possibility. But the problem is this approach still introduces very high communication cost on both sides.

3. *2D Cross-Stream Code*

The two straightforward approaches to couple parallel data streams have fundamental problems and cannot be practically used in real applications. So we get back to the water stream model and couple parallel data streams in a similar way using pipes. We encode the original data in two dimensions: one dimension is along local original data packets; the other dimension is along parallel data streams. Therefore, each code has two degrees: a local fountain code degree and a stream degree. And, we name this 2D Cross-Stream Code, which is the basis of our Cross-Stream Transfer Protocol.

V. CROSS-STREAM TRANSFER PROTOCOL

In this section, we will thoroughly discuss our new parallel streaming protocol: Cross-Stream Transfer Protocol (CSTP).

We divide a piece of original data, for example, a video frame, into ***S*** equally sized sub-frames that will be sent out by ***S*** individual senders to their corresponding receivers. Each sub-frame is further divided into ***N*** equal sized ***original data packets***, and is encoded separately into ***regular code packets*** using Fountain Code. In addition to regular code packets, we integrate a certain percentage of cross-stream code packets into each of the streams, which are ***XOR*** 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 code packets, and upon receiving those code packets, the receivers use the regular code packets to decode original data and use pipe packets to communicate between streams to help lossy streams decode.

1. *Definitions*

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

2. *Creating $\boldsymbol{M_{[1..S]}}$*

$\boldsymbol{M_{[1...S]}}$ are pre-generated generating matrices of 2D Cross-Stream Fountain Code. $\boldsymbol{M_{[i]}}$ is basically a standard Fountain

Symbol	Definition
S	Number of data streams
N	Number of original packet of each sub-frame
α	Overhead of Fountain Code
β	Overhead of transfer protocol
N'	Number of packet (regular code packets & 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 that are **known** to all senders and receivers beforehand
$V_{[1..S][1\ldots N]}$	The original data packets
$Y_{[1..S][1\ldots N']}$	The encoded code packet
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 to 1 for a regular code packet, or *2…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 $D_S(x) = p\cdot x^S + (1-p)\cdot x$

Table 1: Definitions of CSTP

Code matrix plus a few extra pipe packet columns. So, to

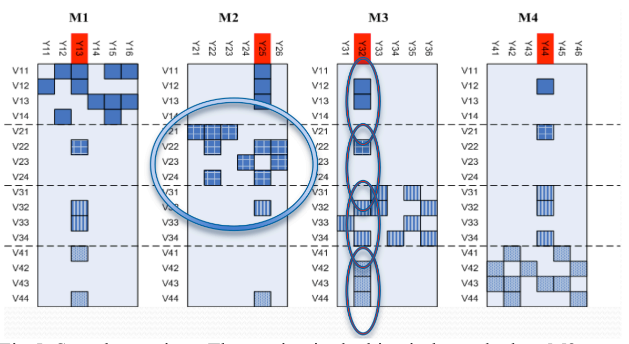


Fig 5: Sample matrices. The portion in the big circle marked on M2 identified a regular fountain code matrix; the chained four small circles identified a pipe packet in M3.

- S=4, N=4, N'=6, p=16.7%
- X-axis: Code data $\mathbf{Y}_{[4][6]}$
- Y-axis: Original data $\mathbf{V}_{[4][4]}$
- Pipe packets (marked as red): $\mathbf{Y}_{13}$, $\mathbf{Y}_{25}$, $\mathbf{Y}_{32}$, $\mathbf{Y}_{44}$

compute $M_{[i]}$, we first create a standard Fountain Code matrix, then insert pipe columns into the matrix. Each pipe column is an assembled column of *S* columns from the *S* standard Fountain Code matrices of S streams. A sample is shown in Figure 5.

3. *Encoding and Decoding Algorithms*

1) *Encoding Algorithm*

```
For each encoder [1…S]:
Step1: break its original sub-frame into equal sized data packets
 Step 2: Get next column from $M_i$, if it's 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
   goes to Step 2.
```

2) *Decoding Algorithm*

```
For each decoder [1…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
   Step3: Repeat Step2 until all data are decoded or the decoding process halts
```

During the encoding and decoding processes, we assume local communication is reliable.

Figure 6 shows how the decoding process works, and how pipe packets can help each other to recover lost packets in one stream. And if only one stream has significant loss, we have the following single stream fault-tolerance theorem.

Theorem 1: 2D Cross-Stream Fountain Code can tolerate the single stream packet loss rate lr, with protocol overhead

$$x = \frac{lr}{S - lr}.$$

Proof:

Assume x is the protocol overhead, which basically means we send out $(1+\alpha)\cdot(1+x)\cdot N$ *code packets to each receiver. Firsts we must ensure that normal receivers have received enough regular code packets to recover their original data, which implies* $(1+x)\cdot(1-p)\geq 1$.

So we have formula one:

$$x \geq \frac{p}{1-p}$$

After all normal receivers have recovered their original packets; only the one that has lost ***lr*** *of the packets remains unfinished. To help the lossy receiver decode, the total number of pipe packets from other receivers plus the number of packets received by itself should be at least* $(1+\alpha)\cdot N$. *That implies,*

$$N\cdot(1+\alpha)\cdot(1+x)\cdot p\cdot(S-1) + N\cdot(1+\alpha)\cdot(1+x)\cdot(1-lr)\geq N\cdot(1+\alpha)$$

So we have formula two:

$$x \geq \frac{1}{p\cdot(S-1)+(1-lr)} - 1$$

Summarizing formula one and two, x gets its smallest value

$$x = \frac{lr}{S-lr},\ \text{when}\ p = \frac{lr}{S}.$$

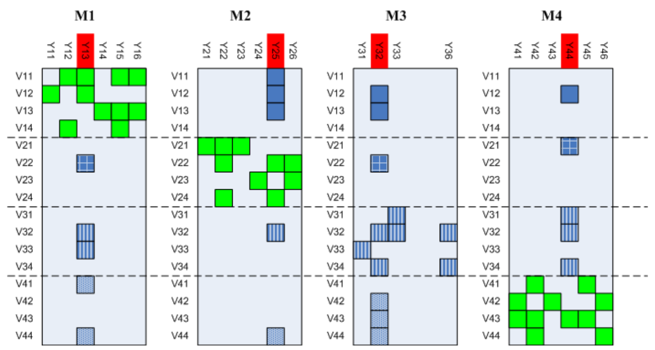


Fig. 6. a) Assuming $\boldsymbol{R_1}$, $\boldsymbol{R_2}$ and $\boldsymbol{R_4}$ have received all the packets and successfully recovered their original data (marked as green), but $\boldsymbol{R_3}$ lost two code packets: $\boldsymbol{Y_{34}}$ and $\boldsymbol{Y_{35}}$ and need help.

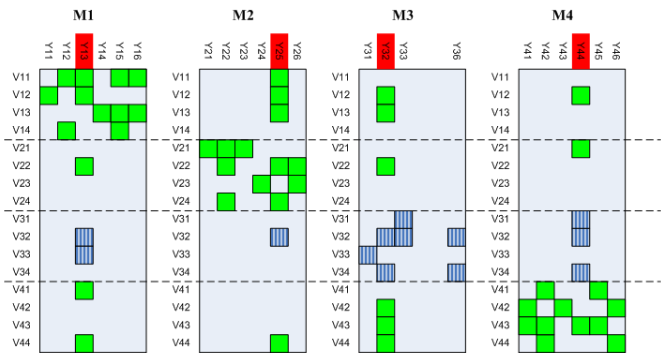


Fig. 6. b) The decoded original data packets are distributed to each other via pipe packets, and all pipe packets columns now become regular columns of $\boldsymbol{M_3}$.

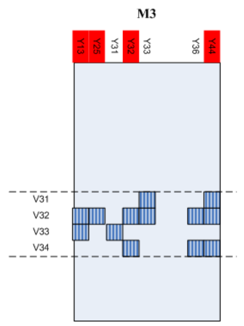


Fig. 6. c) $\boldsymbol{R_3}$ collects pipe packets from $\boldsymbol{R_1}$, $\boldsymbol{R_2}$ and $\boldsymbol{R_4}$, and now can decode its original data.

4. *Decoding Communication Cost*

In CSTP, both encoding and decoding algorithms will incur certain communication cost among nodes within 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 2D Cross-Stream Code is significantly smaller than that of Unified Parallel Code or Cross-Distributed Code because only pipe packets are needed to communicate among nodes. In worst case, since a pipe packet incurs at most ***(S-1)*** messages, the total communication cost's upper bound is $(S-1)\cdot p$, e.g., if $\boldsymbol{p}$=0.05, $\boldsymbol{S}$=4, the decoding communication cost is about 15% of the size of the original data.

5. *Scale CSTP to large number of streams*

The synchronization problem gets harder as the number of parallel data streams increases. For small number of parallel streams, the stream degree distribution function in Table 1 works well. However, for larger scale applications that use tens of parallel streams, the communication cost of the 2D Cross-Stream Code will increase linearly with the number of streams, which can be improved. In that case, we will use a new degree distribution function. We are designing and experimenting different degree distribution functions, and our preliminary research shows that, constant communication overhead is achievable with careful selection of degree distribution functions. A further discussion of more efficient Cross-Stream degree distribution function is beyond the scope of this paper, and we will address it in future work.

VI. 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, and evaluated the stability of decoding algorithm with fluctuating data flows and various packet loss rates. The fountain code we selected is the Raptor Code, which has linear time encoding and decoding algorithms. The pre-code and LTCode we used are those suggested in [21]:

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 LT-Code is 4.6116, and the average degree of Raptor Code is the sum of LTCode degree and pre-code 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 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, N is the number of original data packets.

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

CPU: Intel(R) Core(TM)2 Quad CPU @ 2.40GHz
Cache size: 4096KB
Memory: 4GB
OS: x86_64 Version of CentOS 5.

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 related work, we did some tests here as shown in Figure 7. This picture shows that N cannot be too small since the Raptor Code is based on probability theory and won't hold for small value of N. When N is bigger than 1000, Raptor

Code can recover all original data with an overhead around 4%-8%.

To test the decoding speed of the Raptor Code, we run our implementation of Raptor Code (there is no open source implementation of the fountain 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.

#of Original Packets	#of Raptor Code Packets	Decoding Percent-Age	Decoding Speed (7680B/pkt)	Decoding Speed (3840B/pkt)
512	650	100%	2.51Gbps	3.17Gbps
1024	1200	100%	2.00Gbps	2.22Gbps
2048	2400	100%	1.68Gbps	1.82Gbps
4096	4800	100%	1.45Gbps	1.52Gbps

Table 2: Decode speed of the Raptor Code

The decoding algorithm is very CPU intensive, and consumes significant memory bandwidth. Currently in our single thread experiments, the decoding algorithm is capable of decoding a single HD stream, but in the future we will extend to multiple threads to relieve the CPU bottleneck. 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.

2. *Unified Parallel Code: communication overhead*

To estimate the communication cost, we built a *4096x4096* Raptor Code generating matrix, and use it to generate 4096 code packets. We assume that code packets are alternatively distributed to four receivers:

```
for i=0; i < N; i++
    The ($4 \cdot i$)th packet goes to $R_1$;
    The ($4 \cdot i$ +1)th packet goes to $R_2$;
    The ($4 \cdot i$ +2)th packet goes to $R_3$;
    The ($4 \cdot i$ +3)th packet goes to $R_4$;
```

We then calculate the communication cost among the receivers: the total number of packets exchanged is about 3600 packets, nearly 90% of the original data size.

As we explained before, communication occurs when a data packet was encoded in code packets in multiple receivers. To reduce the overlapping portion of the code packets, we may repartition the matrix to minimize communication cost via integer programming, which however is a NP-hard problem. So instead, we run Zoltan [18], the most recent research result of hyper-graph partitioning, to repartition the matrix, and the improved communication overhead is around 3400 packets, a roughly 8% improvement. We set S=8, and get similar results. Therefore, we can conclude that this approach requires a lot of communication and is not suitable for our applications.

3. *2D Cross-Stream Code Performance*

First, we want to understand whether streams can help each other when one stream is losing significantly more packets than the others. We use these settings:

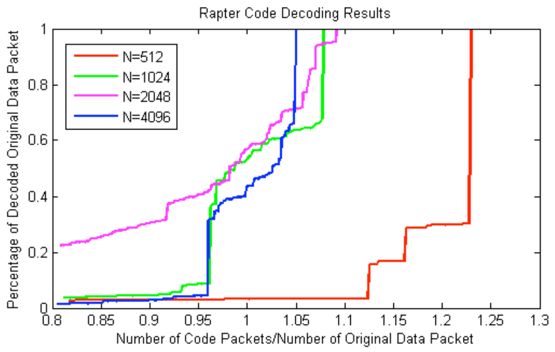


Fig. 7: Raptor Code overhead

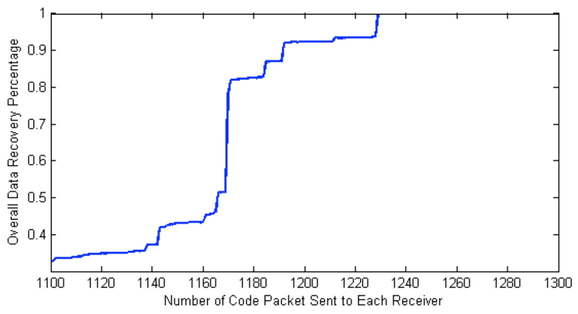
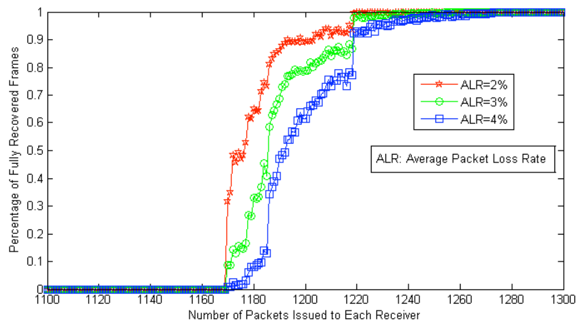


Fig. 8: Protocol overhead can be calculated from this figure. The original number of data packet is 4340 including pre-code, Raptor Code overhead can be read from Fig. 5, which is about 7.5% for N=1024, and from this figure, the overall recovery percentage becomes 1 when N'=1230, so protocol overhead is 1230*4/(4340*1.075) -1=5.45%.

$$p = 5\%$$
$$S = 4$$
$$N = 1024$$
$$D_s(x) = p \cdot x^S + (1-p) \cdot x$$

We verify the performance when single stream packet loss is significant by sending data to four receivers and purposely drop 20% packets from one stream, and test how the other streams can help it recover lost packets to maintain synchronized throughput and measure the protocol overhead of the 2D Cross-Stream Code. Figure 8 shows the lossy stream can be recovered with pipe packets from other streams. By Theorem 1, protocol overhead should be lr/(S-lr) when p=lr/S, which is 5.26%, and our experiments reported 5.45, which verified the theorem.

Our next experiment tested how 2D Cross-Stream Code 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 was randomly generated with every new frame. The average packet loss rate of receivers was 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 is shown in Figure 9, which proves that with a reasonable overhead, e.g. 5% to 15%, our approach can recover significant packet loss in all streams and the possibility of fully recovered frames is ~99.99%.



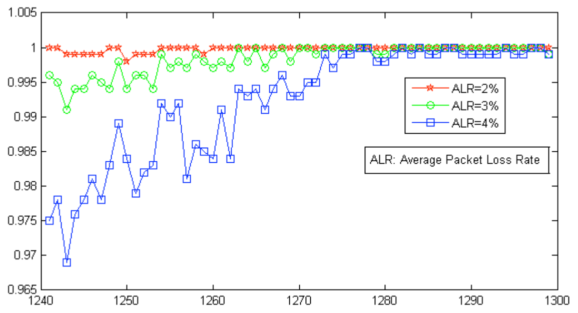


Fig. 9: These lines show the function of frame recovery percentage to the number of issued packet to receivers, and the lower figure is the zoom-in of the top-right part of the upper figure.

All experiments above showed that the 2D Cross-Stream Code met our expectation in tolerating packet loss and stream fluctuation. We performed the same experiments with eight streams, and got very similar results. Regarding the communication cost, we got <15% communication overhead when S=4, and <35% when S=8, both of which were significant better than the coding approach discussed in Section 4. We have implemented our parallel file-transfer software based on CSTP, and are in the progress of comparing our CSTP with other UDP based file transfer protocols, e.g. RBUDP, Lambdastream, and will report the results shortly.

VII. Conclusion and Future Work

In this paper, we studied the problem of large-scale data transfer/streaming. Traditional point-to-point protocols are not scalable to our new cluster-to-cluster communication models in many new scientific applications, and existing multi-to-multi protocols do not sufficiently address the synchronization issue of parallel data streaming. In this paper, we proposed a new Cross-Stream Transfer Protocol, which used 2D Cross-Stream fountain code to tolerate packet loss, and pipe packets to tightly couple streams and help lossy streams to decode correctly. Simulation results met our expectation in terms of packet loss tolerance, decoding efficiency, communication overhead, etc.

There remain many challenges to be solved. We plan on using the CSTP approach to implement a 4K uncompressed streaming system that streams synchronized 4K movie at 6-8Gbps.We are also going to optimize the decoding algorithm to use multi-core processors, eliminate buffer copies to reduce memory consumption, etc. And we are working on designing a more general stream degree distribution function that can easily scale to many streams.

References

[1] GLIF: http://www.glif.is

[2] OPIPUTER: http://www.optiputer.net

[3] Thomas A. DeFanti, Jason Leigh, Luc Renambot, Byungil Jeong, Larry L Smarr, etc. “The OptIPortal, a Scalable Visualization, Storage, and Computing Interface Device for the OptiPuter”, Future Generation Computer Systems 25(2), Elsevier, February 2009, pp. 114-123.

[4] C. Cruz-Neira, D. Sandin, T. DeFanti, R. Kenyon, J. Hart, “The CAVE®: Audio Visual Experience Automatic Virtual Environment,” Communications of the ACM, June 1992

[5] SAGE: http://www.evl.uic.edu/cavern/sage/index.php

[6] E. He, J. Leigh, O. Yu, T. A. DeFanti, “Reliable Blast UDP : Predictable High Performance Bulk DataTransfer”, IEEE Cluster Computing 2002, Chicago, Illinois, Sept, 2002.

[7] Yunhong Gu and Robert L. Grossman**,**UDT: UDP-based Data Transfer for High-Speed Wide Area Networks Computer Networks (Elsevier). Volume 51, Issue 7. May 2007

[8] CALIT2: http://www.calit2.net/newsroom/release.php?id=694

[9] Luc Renambot1, Byungil Jeong, Jason Leigh, “REALTIME COMPRESSION FORHIGH-RESOLUTION CONTENT”, Proceedings of the Access Grid Retreat 2007, Chicago, IL

[10] DMC4K: http://www.dmc.keio.ac.jp/en/topics/071126-4K.html

[11] 4KStreaming: http://www.envision.purdue.edu/4k stream/

[12] Vishwanath, V., 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”, IEEE INFOCOM 2006.

[13] Vishwanath, V., Shimizu, T., Takizawa, M., Obana, K., Leigh, J. Towards Terabit/s Systems: Performance Evaluation of Multi-Rail Systems”, Proceedings of Supercomputing 2007 (SC07), Reno, NV.

[14] Miguel Castro, Peter Druschel, Anne-marie Kermarrec, Animesh Nandi, Antony Rowstron, Atul Singh, “SplitStream: High-bandwidth multicast in cooperative environments”, SOSP'03 ACM Symposium on Operating Systems Principles N°19, Bolton Landing (Lake George), New York , ETATS-UNIS (19/10/2003)

[15] Huaxia Xia, Andrew Chien, “RobuSTore: A Distributed Storage Architecture with Robust and High Performance”, Proceedings of Supercomputing 2007 (SC07), Reno, NV.

[16] U. Catalyurek, E. Boman, K. Devine, D. Bozdag, R. Heaphy, “Hypergraph-based Dynamic Load Balancing for Adaptive Scientific Computations”,Proceedings of IPDPS'07, Best Algorithms Paper Award, March 2007.

[17] Bruce Hendrickson, Tamara G. Kolda, “Partitioning Rectangular and Structurally Unsymmetric Sparse Matrices for Parallel Processing**”,** SIAM Journal on Scientific Computing, volume 21, issue 6 (December 1999), Pages: 2048 - 2072, ISSN: 1064-8275

[18] Zoltan: http://www.cs.sandia.gov/Zoltan/

[19] Fountain Code: http://en.wikipedia.org/wiki/Fountain_code

[20] Amin Shokrollahi, "Raptor Codes," IEEE Transactions on Information Theory, vol. 52, pp. 2551-2567, 2006.

[21] Raptor Code: http://algo.epfl.ch/contents/output/presents/ Raptor-Bangalore.pdf

[22] M. Luby, “LT-codes.” In Proc. 43rd Annu. IEEESymp. Foundations of Computer Science (FOCS), Vancouver, BC, Canada, Nov. 2002, pp.271-280.

[23] 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, volume 25, issue 2 (Feb. 2009), Page: 169-178, ISSN:0167-739X