

LambdaStream – a Data Transport Protocol for Streaming Network-intensive Applications over Photonic Networks

Chaoyue Xiong, Jason Leigh, Eric He, Venkatram Vishwanath, Tadao Murata
Luc Renambot, Thomas A. DeFanti
Electronic Visualization Laboratory
University of Illinois at Chicago

Abstract LambdaStream is a transport protocol designed specifically to support gigabit-level streaming, which is required by streaming applications over OptIPuter. The protocol takes advantage of characteristics in photonic networks. It adapts the sending rate to dynamic network conditions while maintaining a constant sending rate whenever possible. One advantage of this scheme is that the protocol avoids deliberately provoking packet loss when probing for available bandwidth, a common strategy used by other congestion control schemes. Another advantage is that it significantly decreases fluctuations in the sending rate. As a result, streaming applications experience small jitter and react smoothly to congestion. Another important feature is that the protocol extends congestion control to encompass an end-to-end scope. It differentiates packet loss and updates the sending rate accordingly, thus increasing throughput. We have implemented and evaluated LambdaStream over the photonic network testbed between Chicago and Amsterdam. Our results show that LambdaStream occupies almost the full bandwidth and exhibits very small application-level jitter, which is very suitable for streaming applications in OptIPuter.

1. INTRODUCTION

The OptIPuter [1] is a National Science Foundation funded project to interconnect distributed storage, computing and visualization resources using photonic networks whose current bandwidth can reach up to 10Gbps. The main goal of the project is to exploit the trend that network capacity is increasing at a rate *far exceeding* processor speed [5], while at the same time plummeting in cost. This allows one to experiment with a new paradigm in distributed computing - where the photonic networks serve as the computer's system bus and compute clusters, taken as a whole, serve as the peripherals in a potentially planetary-scale computer. We differentiate photonic networks from optical networks as networks comprised of optical fibers and MEMS (Micro-Electro-Mechanical Systems) optical switching devices. There is no translation of photons to electrons and hence no routing within photonic

switches. Applications that control these networks will direct photons from the starting point to the end point of a series of photonic switches and hence will have a full control of the available bandwidth in these allocated light paths. Therefore, congestion control is only necessary for applications which require variable numbers of streams. As a result, photonic networks lead to a much lower level of statistical multiplexing in flows.

In order to optimize data delivery in OptIPuter applications such as Vol-a-Tile [2] and TeraVision [3], advances need to be made at several of the OSI network layers. For example, many OptIPuter applications send data in frames, where between-packet jitter is not an issue but jitter between frames needs to be minimized. Existing transport protocols do not adequately meet this requirement. This paper focuses on our work above the transport layer to provide a gigabit-level streaming protocol called LambdaStream. LambdaStream builds on experiences from QUANTA [4] and high performance networking protocol—Reliable Blast User Datagram Protocol (RBUDP) [4]—that transports data using UDP and uses TCP to acknowledge missing packets. Through mathematical modeling and experiments on both national and international links, RBUDP has been shown to effectively utilize available bandwidth for reliable data transfer. For example, over the TeraGrid [19] between UCSD and NCSA, RBUDP has been able to achieve 18Gb/s throughput over 20Gb/s available link [6]. RBUDP is optimal for large payloads and does not perform favorably for small payloads and *continuous* data streams with varying payloads that are crucial for visualization applications. In LambdaStream, we have developed a congestion control scheme to decrease jitter and improve RBUDP's adaptation to network conditions, tailoring the protocol for OptIPuter visualization applications. We target LambdaStream as an application-layer library, for two reasons. Firstly, we believe an application-layer tool makes development easier and simplifies deployment for testing purposes. Secondly, an application-layer protocol can measure end-to-end conditions as applications actually experience them,

allowing the protocol to distinguish packet loss and avoid unnecessarily throttling throughput.

LambdaStream is an application-layer transport protocol designed specifically for streaming applications in OptIPuter. Correspondingly, key characteristics of LambdaStream include a combination loss recovery and a special rate control, which avoids packet loss inherent in other congestion control schemes [7] [8] [11]. To efficiently utilize bandwidth and quickly converge to a new state, the protocol sets the initial sending rate as the quotient of the link capacity over the maximum number of flows, which is easily obtained in a dedicated network.

The remainder of the paper is organized as follows. Metric justifications are given in Section 2. In Section 3, we describe a reliable delivery scheme that is suitable for applications in OptIPuter. Detailed information on the congestion control is given in Section 4. Section 5 provides experimental results of the protocol over the photonic network testbed between Amsterdam and Chicago. We describe the related work in Section 6. Conclusions and future work are given in Section 7.

2. METRIC JUSTIFICATIONS

The key characteristics of the congestion control in LambdaStream are: it is rate based, it uses receiving interval as the primary metric to control the sending rate, it calculates rate decrease/increase at the receiver side during a probing phase, and it maintains a constant sending rate after probing for available bandwidth. LambdaStream uses the receiving interval as a metric because 1) the receiving interval is closely related with the link congestion and the receiver's processing capability; 2) the receiving interval can be used to detect incipient congestion.

A connection is composed of devices like switches, physical links and the end computers. Thus the whole system can be modeled as a set of store-and-forward queues as shown in Figure 1.

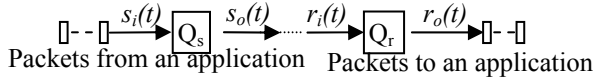


Figure 1: An end-to-end connection

Q_s is the queue at the sender side, and Q_r is the queue at the receiver side. The two queues usually adopt FIFO (First In First Out) mechanism. $s_i(t)$ is an application's sending rate and $r_o(t)$ is an application's receiving rate, which is dependent on the feeding rate as well as the receiver's processing speed. Lower receiving capability usually leads to a smaller

servicing rate (i.e., the receiving rate in this case), $r_o(t)$.

2.1 Incipient congestion

Assume Q is a bottleneck queue, and t_0 is the time instant when congestion begins to appear, so the length of the bottleneck queue is expressed by:

$$q(t) = q_0 + \int_{t_0}^{t_i} (r_i(t) - r_o(t)) dt \quad (1)$$

After congestion occurs, $r_o(t) < r_i(t)$ ($r_o(t)$ is Q 's servicing rate and $r_i(t)$ is the packet incoming rate at Q), so clearly $q(t)$ is a monotonously increasing function. Suppose packet i arrives at Q right before packet $i+1$ does ($t_i < t_{i+1}$), so $q(t_{i+1}) > q(t_i)$. A packet has to wait in the queue until all the previous packets in the queue are delivered, so its waiting time is $w = q * r_o$. We know $q(t_{i+1}) > q(t_i)$, thus $w_i < w_{i+1}$, that is to say, packet $i+1$ has to wait longer than packet i does. So finally, the receiving interval between packet i and $i+1$ is larger than the sending interval when congestion occurs, i.e., $\Delta_r > \Delta_s$. As a consequence, the ratio between the receiving rate and the sending rate is smaller than one. Based on this analysis, we assume a packet delay larger than the sending interval indicates congestion. This assumption and others similar to it are used in many related works [12] [13] [14] [15] [16].

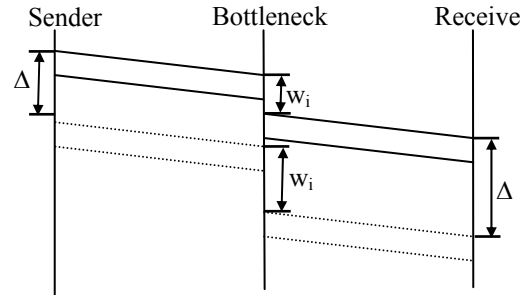


Figure 2: Inter-packet spacing

This assumption is reasonable in photonic networks. In Figure 3, a single stream is sent over 1Gbps photonic network at different speeds. The stream has 200 packets. We measure the average ratio at the receiver's side. When packets are sent at a rate slower than the available bandwidth, the ratio is close to one. But when the rate is higher than the available bandwidth, ratio is greater than one. The results also show that the value of the ratio indicates a degree of congestion. Higher the ratio, more serious the congestion.

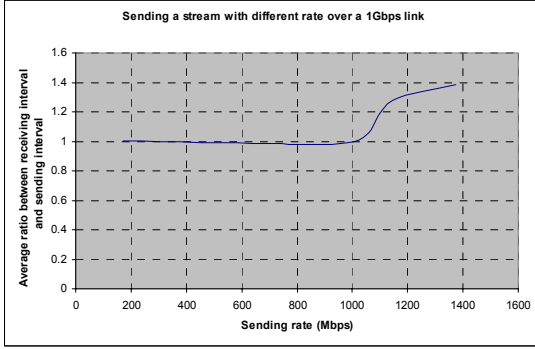


Figure 3: Average ratio VS sending rate

If the operating system reschedules the sender program or the receiver program, the actual packet interval may be longer than the ideal one without disturbance from the operating system. CPU's shift to a small application causes a small spike while a large application causes a big spike in the packet interval. Spikes in the sending interval have a similar reflection at the receiver's side. All these may cause imprecision in the ratio measurement. We adopt two methods to reduce this deviation. First, the protocol uses an average inter-packet delay, which is calculated once every epoch. This method is helpful in filtering out spikes caused by the receiver. Second, the protocol removes samples with a large spike. The algorithm considers a spike large when the spike is 20 times larger than the sending interval.

2.2 Loss differentiation

The protocol decreases the sending rate when the ratio is greater than 1.02 (we choose this threshold by experience). Additional decrease is necessary if congestion is serious or a receiver suffers from a long-lasting low capability. These two cases usually results in packet losses. However, not every packet loss is an indication for these two cases. When a receiver suffers from a low capability for a much shorter time than RTT, packet loss may occur but it is not necessary to decrease the sending rate. To avoid unnecessarily throttling the sending rate, we should firstly differentiate causes for packet loss.

We use *loss spacing* and the *average receiving interval* to distinguish causes for packet loss. Loss spacing is defined as the difference in sequence number between two neighboring loss events. The average packet delay is the quotient of difference in time between two consecutively received packets over difference of their sequence numbers, as given by Equation (2).

$$avg_pkt\ delay = \frac{currt - prevt}{seqno - prevseqno} \quad (2)$$

where *currt* and *seqno* are the receiving time and the sequence number for the last received packet, and *prevt* and *prevseqno* are the receiving time and the sequence number for the packet preceding it.

When congestion occurs, packets en route must wait longer for delivery. This means that the average packet delay observed by the receiver becomes longer than the sending interval. If the congestion is serious and persists for a long time, more loss occurs with a higher frequency, which means the loss spacing is small.

When a receiver has enough capability to handle incoming packets, the average length of Q_r converges to 0. However, when the receiving capability is low, the receiving rate $r_o(t)$ will be smaller than the sending rate and the queue will be filled soon and newly arrived packets are dropped. However, a sudden independent decrease in the receiver's capability rarely affects the application because the receiver recovers its capability very quickly. The receiver immediately drains all the packets in the queue during a short timeframe, and then waits for the oncoming of another packet. As a consequence, the loss event is usually not followed by another loss event and the average packet delay between the two packets with dropped packets in between is close to the sending interval. If the receiver suffers from a continuous decrease in its capability, the queue length accumulates. Loss events occur much more frequently. Because data is always available for the receiver to fetch, the actual average packet delay with lost packets in between, on the contrary, will be much shorter than the sending interval.

Therefore, average packet delay and loss spacing are good metrics to determine the loss type. If the average packet delay falls somewhere outside a range and the loss spacing is small, the protocol decreases the sending rate. Congestion may coincide with a receiver's low performance, making metrics deviate. In spite of this, the experimental results show that the two metrics still produce a decision with a high degree of correctness and acceptability.

3. RELIABLE DELIVERIES

Most applications over OptIPuter are visualization related and can tolerate certain amount of packet loss. So a reliable delivery is not necessary. Actually recovering packet loss is time costly in LFNs and usually compromises performance. Therefore, some applications prefer packet loss to reliable delivery in

order to obtain better performance. The protocol proposes *the combination loss recovery* to meet this requirement. It offers choices of either guaranteed reliable delivery or unreliable delivery. Both congestion and a receiver's capacity are the two main reasons for packet losses in OptIPuter.

Combination loss notification scheme is composed of two independent loss notification schemes. One is called quick loss notification; and another is called epoch loss notification. These two schemes together contribute and guarantee reliable delivery. Epoch loss notification can be easily disabled if an unreliable delivery is preferred.

When a receiver receives a packet with a higher sequence number than the expected sequence number, the receiver immediately sends a quick loss notification. The lost list ranges from the expected packet to the received packet (excluding the packet). Whenever a packet loss is detected, the receiver immediately sends out a loss notification. Upon receiving this negative acknowledgement, the sender immediately retransmits the lost packet. This scheme takes about one RTT to recover the lost packet. The quick loss notification applies only to packets lost in the first transmission.

The protocol notifies retransmission loss by the epoch loss notification scheme. Once every epoch, a receiver checks retransmission loss and sends a retransmission loss notification. In case a retransmission loss occurs, the sequence number of the lost packet will queue in the lost list buffer and blocks the increase of maximal continuous sequence number. A receiver decides a retransmission loss when it detects that the lost packet is not received after a threshold.

This mechanism improves the performance of a loss-tolerant application. For an application needs a reliable delivery, this mechanism may need longer time to recover a packet loss if the packet is lost more than once. However, the probability is small for a retransmitted packet to be discarded over a photonic network. Furthermore, time spent on waiting for a retransmission notification is still small compared with long propagation delay on the network. Therefore, this disadvantage is acceptable for a reliable delivery application.

4 CONGESTION CONTROL

The congestion control is composed of two parts. One part is to distinguish a packet loss and adjusts sending rate accordingly, thus avoiding unnecessarily throttling of the sending rate. Another part is to

update the sending rate based on the ratio between the average receiving interval and the sending interval. Incipient congestion leads to a higher ratio, which triggers the protocol to decrease the sending rate. The protocol increases its sending rate if the ratio is close to one and the available bandwidth is greater than zero.

4.1 An algorithm for distinguishing a packet loss

To fast clear serious congestion and thus avoid more packet losses, the protocol should further decrease its sending rate as soon as possible. However, if a packet loss is caused by light congestion or a *sudden* decrease of the receiver's processing capability, the protocol should not decrease additional amount of the sending rate. Thus the protocol is required to distinguish the two cases and updates sending rate accordingly.

The pseudo-code for this scheme is shown below. When the protocol detects a packet loss (line 2), it first checks its average receiving delay and the loss spacing. If the loss is determined to be caused by serious congestion or continuous low receiver's capacity (line 5), the receiver decreases sending rate and sends the feedback to the sender (line 6 and 7). Otherwise, it neglects the packet loss and does not update the sending rate.

```

rate_cotroll()
1 when (a packet arriving at the receiver)
2   if (seqno>expected sequence number)
3     if  $(1-\alpha)sndDealy < \frac{currt-prevt}{seqno-lastseqno} < (1+\alpha)sndDealy$ 
4       //do nothing
5     else if (seqno-lostSeqno<th)
6       sndPktDelay ← (1+K1)*sndPktDelay
7       ack(sndPktDelay)
8     lostSeqno = seqno

```

4.2 An algorithm for updating the sending rate

The main objective of the algorithm is to obtain a high throughput while maintaining a minimal jitter. Propagation delay does not account for jitter because an end-to-end transmission involves only one transmission route in OptIPuter. However, packet loss and changes of sending rate can not be neglected for jitter because: 1) a change of the sending rate alters the packet receiving interval; 2) recovering lost packets is time-consuming. The mechanism we discussed in this section reduces jitter by preventing scenarios for 1 and 2 from appearing.

Most congestion control schemes detect available bandwidth *reactively*. They send data at a rate exceeding the actual available bandwidth. One advantage of this mechanism is that it fast detects the available bandwidth. However, it imposes packet loss. For example, TCP's congestion control algorithm is designed to deliberately cause occasional loss to provide feedbacks to the sender. This mechanism works very well in a network with small RTT and with frequent variations in the number of flows, but is costly in LFNs. It greatly increases jitter. Photonic networks are characterized by low levels of statistical multiplexing. We take advantage of this property and propose a *proactively* congestion control scheme, making it more suitable for our applications. The algorithm updates sending rate *only* when it detects environmental changes and keeps the sending rate as a constant when no change is detected. Thus, it avoids deliberately invoking packet loss and greatly decreases jitter.

4.2.1 Design

The algorithm considers that network conditions are changing when it observes that the ratio is greater than 1.02 or the ratio is close to one and the end-to-end available bandwidth is greater than zero. End-to-end available bandwidth is defined to be the unused bandwidth on a link [17].

As we mentioned earlier, either incipient congestion or a receiver's capacity affects the average receiving rate. When congestion is about to occur or a receiver has a low processing capacity, the receiving rate is slower than the sending rate. However, with a very limited information, we do not know a convenient expression for the dynamics of the average receiving interval, the degree of congestion and a receiver's capacity. Nonetheless, we know that 1) the average receiving interval never exceeds the sending interval; 2) the lower a receiver's capacity or the more serious the congestion is, the slower the receiving rate. Based on this knowledge, we model the system as shown in Figure 4. In the Figure, s is the sending rate, r is the average receiving rate, b_l is the link allowable rate and b_r is the receiver allowable rate.

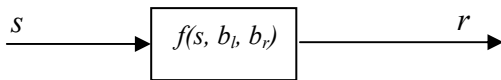


Figure 4: System model

$$f(s, b_l, b_r) = \begin{cases} 1 & s \leq \min(b_l, b_r) \\ < 1 & s > \min(b_l, b_r) \end{cases} \quad (3)$$

The control objective is to obtain a high sending rate without causing congestion or overflowing a receiver, and maintains the appropriate sending rate as stable as possible. The appropriate sending rate is actually equal to $\min(b_l, b_r)$. This actually means the appropriate sending rate is system bandwidth when receiver is a bottleneck. Clearly, when several flows run simultaneously on the link, a receiver with a much lower capacity than others may have a lower share of the link capacity. The following straightforward rules are used to probe for $\min(b_l, b_r)$.

- 1) **Rule 1:**
If:
 - a. no congestion occurs OR
 - b. available bandwidth is greater than zero and receiver has enough extra capacity
 Then increase current sending rate;
- 2) **Rule 2:**
If:
 - a. congestion is predicted OR
 - b. receiver does not have enough capacity to handle current sending rate
 Then decrease current sending rate;
- 3) **Rule 3:**
If:
 - a. Rule 1 does not apply AND
 - b. Rule 2 does not apply
 Then maintain current sending rate.

Clearly, the important part of the rules is to detect the network conditions accurately. We map the two metrics, the estimated available bandwidth and the ratio between the average receiving interval and the sending interval, to the current network conditions. After deciding network conditions, the algorithm should appropriately update the sending rate. A large adjustment is desirable in situations where the protocol must adapt rapidly to changes in the connection environment, but small adjustments are usually a better way to efficiently utilize the bandwidth in a more stable environment. For example, with a small growth, the protocol will not push the system into serious congestion if the system is already on the verge of congestion. Therefore, variable growth rate is necessary. We choose an equation that allows the protocol to change the growth rate easily. The equation $s(n+1) = (1 \pm k)s(n)$ meets this requirement. When the protocol chooses a small k , say k is close to 0, the growth rate is small. When the protocol chooses a large k , the growth rate will be exponential. The larger k , the more the

sending rate varies. So by adaptively changing k , we can change the growth rate as expected. The sending rate is updated as follows:

$$s(n+1) = \begin{cases} s(n) + \tilde{k}_1(n)s(n) & f(s, b_1, b_r) = 1 \text{ and } b_{est} > 0 \\ s(n) - \tilde{k}_2(n)s(n) & f(s, b_1, b_r) < 1 \end{cases} \quad (4)$$

For the practical meaning, $\tilde{k}_1(n)$ and $\tilde{k}_2(n)$ should satisfy: $1 > \tilde{k}_2(n) > 0$ and $\tilde{k}_1(n) > 0$. b_{est} is an estimated available bandwidth and is obtained based on the packet train method [17]. Clearly, $\tilde{k}_1(n)$ and $\tilde{k}_2(n)$ affects the system performance. If they are not properly chosen, the system may not be able to maintain the sending rate at a constant. Instead, the system may oscillate executing between Rule1 and Rule2, resulting in high jitter.

4.2.2 Choices for parameter $\tilde{k}_1(n)$ and $\tilde{k}_2(n)$

At any instant, the system may work in one of the following scenarios in a probing phase:

- Scenario 1: working under Rule1;
- Scenario 2: working under Rule2.

In either scenario, parameters $\tilde{k}_1(n)$ and $\tilde{k}_2(n)$ should be selected in a way so that the overall sending rate converges to $\min(b_r, b_l)$. When the algorithm spots that the network is in congestion or the end-to-end available bandwidth is greater than zero, the algorithm initiates a probing phase. It firstly chooses an initial value of parameter $\tilde{k}_1(0)$ or $\tilde{k}_2(0)$ based on the current network conditions. All the following values of $\tilde{k}_1(i+1)$ and $\tilde{k}_2(i+1)$ are determined by preceding values $\tilde{k}_1(i)$ or $\tilde{k}_2(i)$. The farther the sending rate away from $\min(b_r, b_l)$, the larger the initial value $\tilde{k}_1(0)$ or $\tilde{k}_2(0)$ should be.

When the algorithm starts a probing phase by increasing the sending rate, $\tilde{k}_1(0)$ is chosen as such that the new sending rate reaches somewhere between the current sending rate s and s plus the predicted available bandwidth. So

$$s + l_m b_{est} = (1 + k_1(0))s$$

l_m is called *bandwidth proportional share*. The network before the probing phase is stable, which means every connection has a proper share of the bandwidth. So the available bandwidth should be distributed proportionally to the previous share among all flows. We choose $l_m = s/L$. Clearly, the sum of l_m for all connections is one and

$$k_1(0) = \frac{l_m b_{est}}{s} = \frac{b_{est}}{L}$$

When congestion triggers a probing phase, the initial value of $\tilde{k}_2(0)$ is chosen in the following way:

If the connection is in serious congestion (ratio > 1.2), the algorithm determines that the congestion is caused by participation of a new flow and directly sets the sending rate to a fair share among all flows. The number of flows before the probing phase is estimated as $\lfloor L/s \rfloor$, and thus the number of current flows is $\lceil L/s \rceil$. So the new sending rate should be $\frac{L}{\lceil L/s \rceil}$

so

$$k_2(0) = 1 - \frac{L}{s \cdot \lceil L/s \rceil}$$

Figure 3 shows that the ratio can well indicate congestion degree when the connection is in a light congestion. Based on experience, we consider that a connection is in a light congestion if $1 < \text{ratio} < 1.2$. Therefore, the initial value of \tilde{k}_2 is chosen so that the sending rate will be $s(n+1) = s(n)/\text{ratio}$, so $k_2(0) = 1 - 1/\text{ratio}$

After updating the sending rate, the system may be in the same Scenario or in another Scenario. If the system is in the same Scenario, we update the parameter so that the sending rate increases/decreases in a smaller amount than the previous increase/decrease. But if the update moves the system to another Scenario as shown in Figure 5, we need to choose a parameter so that the system has a decaying oscillation. After one oscillation, actually we know the ideal sending rate, $s(i+2)$ lies somewhere within the range between $s(i)$ and $s(i+1)$. So let's obtain some conditions to guarantee $s(i+2)$ to be within $s(i)$ and $s(i+1)$.

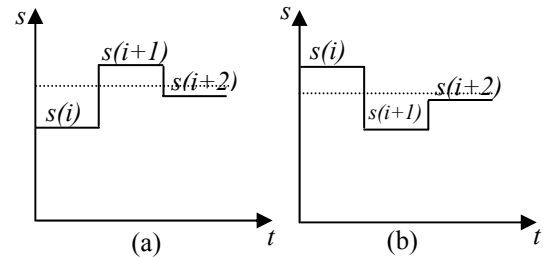


Figure 5: Update the sending rate

First let's assume that at time i , the system satisfies the condition in Rule1, shown in Figure 5(a). So the sending rate will be increased to:

$$s(i+1) = (1 + k_1(i))s(i) \quad (5)$$

Then $s(i+1)$ moves the system to under the control of Rule2, resulting in oscillation. So

$$s(i+2) = (1 - k_2(i+1))s(i+1) \quad (6)$$

For decaying oscillation, the ideal $s(i+2)$ should lie between

$$s(i) < s(i+2) < s(i+1) \quad (7)$$

as shown in Figure 5(a). Since $k_2(i+1) > 0$, so clearly,

$$s(i+2) < s(i+1);$$

From Equations (5) and (6), we get

$$\begin{aligned} s(i+2) &= (1 - k_2(i+1))s(i+1) \\ &= (1 - k_2(i+1))(1 + k_1(i))s(i) \\ &= (1 + k_1(i) - k_2(i+1) - k_1(i)k_2(i+1))s(i) \end{aligned}$$

For (7) to be held, we have:

$$k_1(i) - k_2(i+1) - k_1(i)k_2(i+1) > 0$$

so

$$k_2(i+1) < \frac{k_1(i)}{1 + k_1(i)} \quad (8)$$

Equation (8) guarantees a decaying oscillation for the case shown in Figure 5(a).

Second, let's assume that at time i , the system satisfies the condition in Rule2, as shown in Figure 5(b). So the sending rate will be reduced to:

$$s(i+1) = (1 - k_2(i))s(i)$$

$s(i+1)$ moves the system to under the control of Rule1. So

$$s(i+2) = (1 + k_1(i+1))s(i+1)$$

For decaying oscillation, the ideal $s(i+2)$ should lie between

$$s(i+1) < s(i+2) < s(i) \quad (9)$$

Since $k_1(i+1) > 0$, so clearly $s(i+1) < s(i+2)$

$$\begin{aligned} s(i+2) &= (1 + k_1(i+1))s(i+1) \\ &= (1 + k_1(i+1))(1 - k_2(i))s(i) \\ &= (1 + k_1(i+1) - k_2(i) - k_1(i+1)k_2(i))s(i) \end{aligned}$$

For satisfying Equation (9), we have

$$k_1(i+1) - k_2(i) - k_1(i+1)k_2(i) < 0$$

i.e.,

$$k_1(i+1) < \frac{k_2(i)}{1 - k_2(i)} \quad (10)$$

Equation (10) guarantees a decaying oscillation for the case shown in Figure 5(b).

Therefore, if oscillation occurs and the next step parameter is chosen according to Equation (8) or (10), the system will guarantee a decaying oscillation, that means the system will have a convergence sending rate.

For simplicity, this algorithm chooses the parameters as

$$\tilde{k}_1(n+1) = \begin{cases} \tilde{k}_1(n)/e & \text{oscillation} = \text{false} \\ \frac{0.5\tilde{k}_2(n)}{1 - \tilde{k}_2(n)} & \text{oscillation} = \text{true} \end{cases}$$

$$\tilde{k}_2(n+1) = \begin{cases} \tilde{k}_2(n)/e & \text{oscillation} = \text{false} \\ \frac{0.5k_1(n)}{1 + k_1(n)} & \text{oscillation} = \text{true} \end{cases}$$

The initial parameters are determined as described above.

4.2.3. Pseudo-code algorithm

This section describes the pseudo-code for the above algorithm. At the end of every sampling interval, the algorithm checks if any packet is received in the new interval. If no packet is received, the algorithm directly starts another sampling period. Otherwise, the algorithm generates the rate control algorithm (Line 1). If the algorithm detects that a probing phase is on, the algorithm calculates the following value of the parameter (Line 13~17 or Line 28~32), or if a probing phase is off but the algorithm detects that the connection conditions are changing (Line 4 or 23), the algorithm starts a probing phase (Line 5~12 or Line 24~32) and calculates the initial value of the parameter. After updating the parameter accordingly, the algorithm requests the sender to update its sending rate (Line 19 or Line 34). The algorithm finally checks if the probing phase is complete or not (Line 38~43).

```

rate_control2()
1 if (totalNoOfSample != 0) {
2   avgDRatio ← totalDRatio/totalNoOfSample
3   if (avgDRatio > 1.02) {
4     if (!bProbe) {
5       if (avgDRatio > 1.2) {
6         k0 = 1 - L / (rL / sndRateγ . sndRate)
7         k = k0
8       }
9     } else {
10       k0 = 1 - 1 / avgDRatio
11       k = k0
12     }
13   } else {
14     if (bInc)
15       k = 0.5k / (1 + k)
16     else
17       k = k / e
18   }
19   newSndRate = (1 - k) . sndRate
20   bProbe = true;
21   bInc = false;
22 }
23 else if (estBandwidth > 0) {
24   if (!bProbe) {
25     k0 = best / L;
26     k = k0
27   }
28   else {
29     if (bInc)
30       k = k / e
31     else
32       k = 0.5k / (1 - k)
33   }

```

```

34     newSndRate=(1+k)sndRate
35     bInc = true;
36     bProbe = true;
37   }
38   if (newSndRate ≅ sndRate) {
39     bProbe = false;
40     bInc=false
41     totalNoOfSample = 0
42     totalDRatio = 0
43   }
44 }

```

4.4 Estimate available bandwidth

To avoid intrusiveness, the protocol uses data packets to form a measurement train. So the resulting available bandwidth is actually A-S (A is the measured available bandwidth and S is the current sending rate). The sender sends a measurement train once every $T(8000)$ packets. The packet train is sent at different rate with the current sending rate as the starting rate. The train has $K(300)$ packets and every 50 packets is grouped into one category. Starting with the current sending rate, the following group is sent at an increasing sending rate with the increasing amount Δ . Receiver is responsible for calculating the available bandwidth. If the receiver finds the connection is already in congestion, it sets the available bandwidth to zero. Otherwise, it calculates inter-packet delay and chooses the mean in every group as the representative inter-packet delay for the group. It uses representative inter-packet delays to calculate a bandwidth for each group. So finally we obtain five groups of (s_i, r_i) . The final available bandwidth is the breakpoint where $r_i/s_i < 1$.

5. EXPERIMENTAL RESULTS

We have evaluated LambdaStream over the photonic network testbed between Chicago (Scylla) and Amsterdam (Vangogh). As shown in Figure 7, this Testbed consists of three all-optical switches, located at University of Amsterdam, downtown Chicago StarLight site and University of Illinois at Chicago. The transatlantic link is composed of one 1Gbps fiber and its RTT is 110ms.

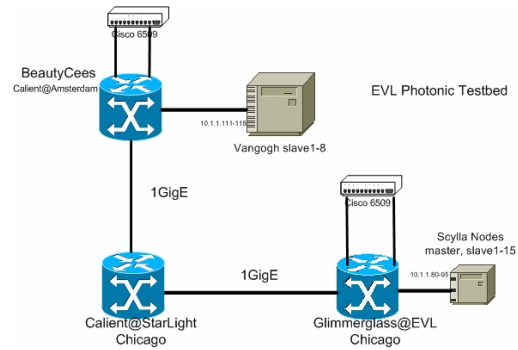


Figure 7: Photonic network testbed between Amsterdam and Chicago

Vangogh	Dual Intel Xeon 2.8GHz, Linux 2.4.26 with web100 kernel patch, 2GB RAM, Intel Pro 1000 Gigabit Adaptor
Scylla	Dual Intel Xeon 1.8Hz, Linux 2.4.18-3smp with web100 kernel patch, 1.5GB RAM, , Intel Pro 1000 Gigabit Adaptor

Table 1 : System Configuration

Figure 8 describes throughput for one single flow running on the Testbed. One line represents a TCP flow, and the rest of the lines represent LambdaStream flows, each of which starts with a different initial sending rate. TCP window is set to 35Mbytes in this experiment. The Figure shows that the LambdaStream protocol’s throughput converges very well for a single flow. Even if we set the initial sending rate to either 1720Mbps or 172Mbps, values far exceeding or much lower than the available bandwidth, the protocol manages to maintain the throughput at an almost fixed sending rate, about 950Mbps. Figure 8 also shows that a single TCP flow obtains about 800Mbps throughput, which is a little bit lower than but still comparable to LambdaStream. However, because of TCP’s slow-start scheme, it takes about 1.2 seconds to obtain this stable throughput, which is much longer than the worst case with LambdaStream, which is 0.8 seconds. The graph also shows that the TCP throughput varies much more than LambdaStream does. Higher variation in throughput usually indicates a higher jitter.

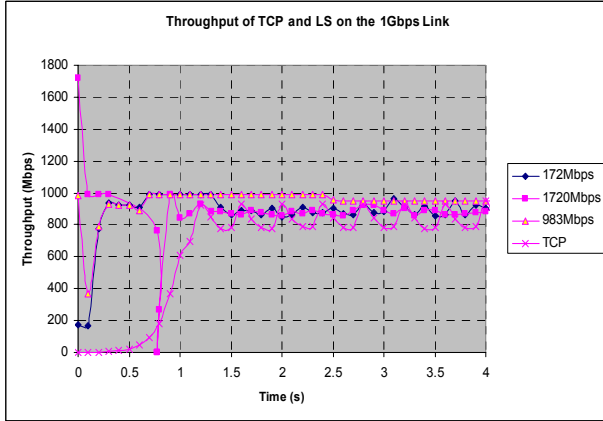


Figure 8: Throughput for a single flow on a 1Gbps network (showing multiple cases)

Our OptIPuter applications use frames to send and receive data, so all we care about is the frame-level (or called application-level) jitter. Figure 9 describes the frame-level jitter when the frame size is 2MB. In Figure 9 (below), the Y-axis represents the interval between received packets, so the variations indicate jitter. Higher variation in packet interval represents higher jitter. LambdaStream *exhibits* much smaller jitter than TCP does. The big spike in the LambdaStream line indicates that the protocol recovered a lost packet. With smaller payloads (e.g. 2MB), every lost packet increases jitter. For example, to transmit a 2MB payload without loss takes $2M \cdot 8 / 1G = 16ms$ for a complete transmission. However, the time needed for lost packet recovery is at least one RTT (110ms). As a result, every lost packet causes a huge spike in jitter.

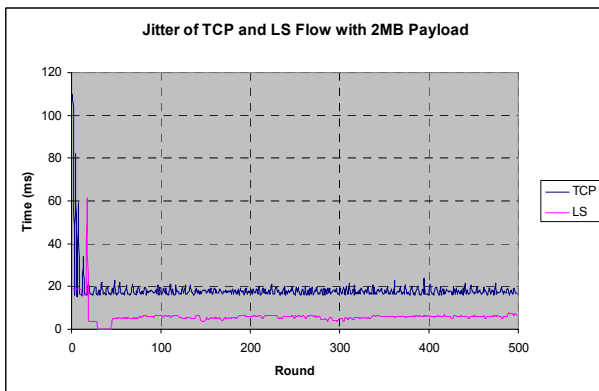
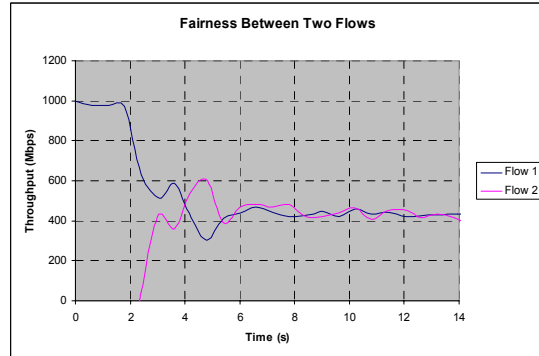


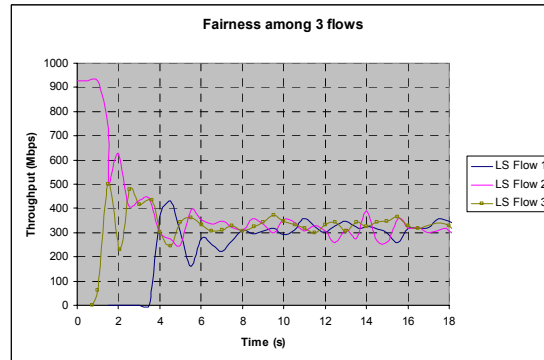
Figure 9: Jitter Comparison between TCP and LambdaStream with identical 2 Mbyte Payloads

Figure 10 describes fairness between multiple LambdaStream flows. Experiments show that LambdaStream can achieve either converging fairness (Figure 10a and Figure 10b) or min-max

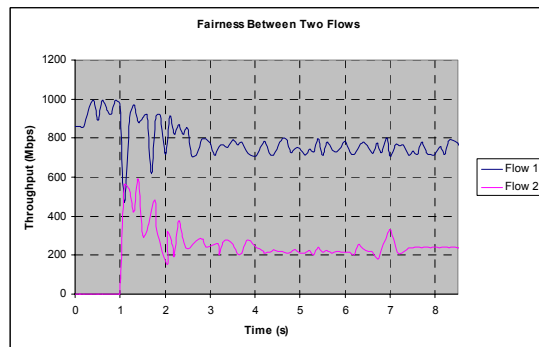
fairness (Figure 10c and Figure 10d) among flows. In both cases, LambdaStream can fully occupy the available bandwidth. The differences result from scheduling imprecision in non real-time operating systems, which sometimes delay LambdaStream packets by returning control to the protocol tens of milliseconds later than requested. We are currently investigating this.



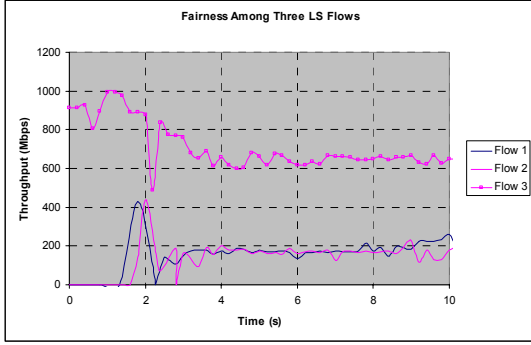
a. Two LambdaStream flows with converging fairness



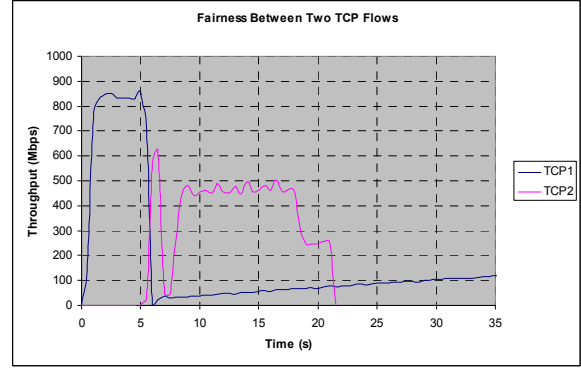
b. Three LambdaStream flows with converging fairness



c. Two LambdaStream flows with min-max fairness



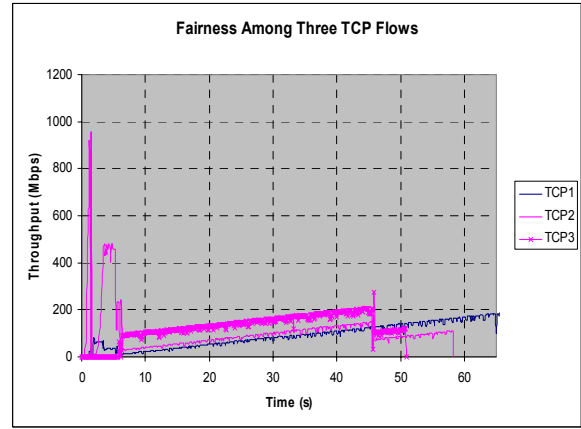
d. Three LambdaStream flows with min-max fairness



b. Two TCP flows with new flows added later

Figure 10: Fairness experiments for LambdaStream flows over 1Gbps link

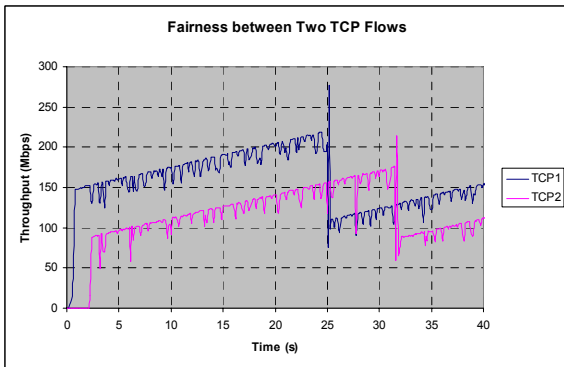
It is difficult to compare fairness between TCP with a high performance protocol because TCP usually cannot utilize the full bandwidth. But since the single revised TCP uses almost the full bandwidth, the paper also shows the situations when several TCP flows run simultaneously. Figure 11 shows fairness achieved by TCP in the same environment. In Figure 11(a), a new TCP flow is started *before* the first flow reaches maximum throughput. In this case, the flows achieve average fairness. In Figure 11(b), a new flow is started *after* the first flow reaches maximum throughput. In this case, TCP cannot achieve fairness. The first flow is affected greatly when a second flow is added. Even after the second flow completes, the first flow continues to increase its throughput at the same rate and does not probe the highest available bandwidth for more than fifteen seconds. In both cases, the results show that multiple TCP flows cannot fully occupy the available bandwidth.



c. Three TCP flows

Figure 11: Fairness experiment for TCP flows

Figure 12 describes how a LambdaStream flow may affect TCP flows. In this experiment, a LambdaStream flow lasts about 10s and then disappears. We see TCP's throughput is similar to the situation when another TCP joins in. Therefore, we can say that the way LambdaStream influences TCP flows is similar to the way when another TCP joins in. LambdaStream occupy the rest of the bandwidth. We cannot say LambdaStream is fair to TCP.



a. Two TCP flows with new flow added earlier

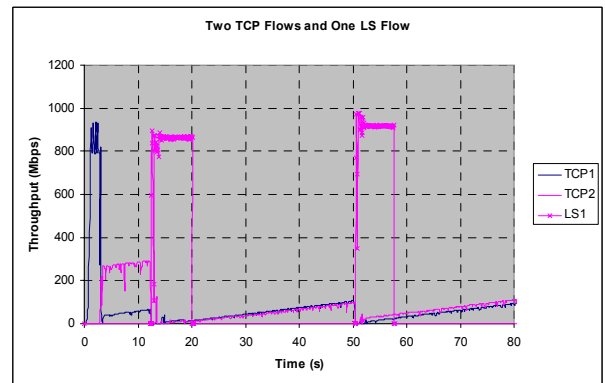


Figure 12: Influence

6. RELATED WORK

Network researchers have reached a consensus that the current TCP implementations are not suitable for long distance high performance data transfer. Either TCP needs to be modified radically or new transport protocols should be introduced. Long Fat Networks (LFNs), such as those between the U.S., Europe and Asia, have extremely high round-trip latencies (at best 120ms). This latency results in gross bandwidth under-utilization when TCP is used for data delivery. Several solutions have been proposed. One solution is to provide revised versions of TCP [8] [11] with better utilization of the link capacity. Another solution is to develop UDP-based protocols to improve bandwidth usage. The Simple Available Bandwidth Utilization Library (SABUL) [9], Tsunami [10], Reliable Blast UDP (RBUDP) [4] and the Group Transport Protocol (GTP) [20] are few recent examples.

However, none of these solutions, except GTP, works well when *packet loss* occurs. TCP interprets packet loss as an evidence of congestion and halves its congestion window when detecting a packet loss. This mechanism ensures fairness among connections sharing the same channel, and works well in the current Internet, where most of the packet losses are due to congestion. However, in OptIPuter, other factors for packet loss can not be ignored. Most visualization-related applications for the OptIPuter are both computation and network-intensive, processing datasets to generate geometries for real time rendering. Additionally, high bandwidth of LFNs allows packets to reach a receiver at speeds that may exceed the receiver's capability, forcing the receiver to discard packets (this often occurs when the receiver's CPU usage is high). Thus the receiver usually becomes the bottleneck to maintaining a high throughput [18]. However, TCP and many of its revised versions such as SCTP (Stream Control Transmission Protocol) [8] and FAST (Fast Active queue management Scalable TCP) [11] use windowing mechanisms and do not differentiate the causes for packet loss, which limits the complete utilization of link capacity. The latest version of SABUL--UDT [9] has the similar problem. GTP is designed as a request transfer data model and our work is designed as a real time data streaming model.

7 CONCLUSIONS

LambdaStream extends the congestion control to encompass an end-to-end scope. It distinguishes packet loss and adjusts the sending rate accordingly. The protocol also applies a ratio sampling approach to detect incipient congestion and combines it with a bandwidth estimation method for *proactively* probing for an appropriate sending rate. The experimental results show that LambdaStream achieves 950Mbps throughput in a 1Gbps channel. It exhibits small application-level jitter and react smoothly to congestion, which is very suitable for streaming applications in OptIPuter. LambdaStream works well for continuous data streams of varying payloads that are representative characteristics of visualization applications.

8. REFERENCE

- [1] <http://www.evl.uic.edu/cavern/optiputer>
- [2] <http://www.evl.uic.edu/cavern/optiputer/volatile.html>
- [3] Rajvikram Singh, Jason Leigh, Thomas A. DeFanti and Fotis Karayannis, "TeraVision: a high resolution graphics streaming device for amplified collaboration environments", Future Generation Computer Systems, Special Issue: IGRID 2002. Volume 19(2003), Number 6, August 2003, Elsevier B. V., The Netherlands.
- [4] E. He, J. Alimohideen, J. Eliason, N. K. Krishnaprasad, J. Leigh, O. Yu and T.A. DeFanti, "QUANTA: A toolkit for high performance data delivery over photonic networks", Future Generation Computer Systems, Special Issue: IGRID 2002. Page 919-933, Volume 19(2003), Number 6, August 2003, Elsevier B. V., The Netherlands.
- [5] J. S. Chase, A. J. Gallatin, and K. G. Yocum, "End-system optimizations for high-speed TCP", In IEEE Communications, special issue on TCP Performance in Future Networking Environments, vol. 39 no. 4, April 2001
- [6] http://www.evl.uic.edu/cavern/rg/20030817_he/
- [7] A. S. Tanenbaum, *Computer Networks*. New Jersey: Prentice-Hall, 1996, ch. 6.
- [8] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang and V. Paxson Stream Control Transmission Protocol (RFC 2960).<http://www.ietf.org/rfc/rfc2960.txt>
- [9] Y. Gu and R. L. Grossman, "UDT: An Application Level Transport Protocol for Grid Computing", Abstract for 2nd International Workshop on Protocols for Fast Long-Distance Networks
- [10] <http://www.indiana.edu/~anml/anmlresearch.html>
- [11] C. Jin, D. X. Wei and S. H. Low, "FAST TCP: motivation, architecture, algorithms, performance", IEEE Infocom, March 2004

- [12] A.A. Awadallah and C. Rai. TCP-BFA: Buffer Fill Avoidance. In Proceeding of IFIP high performance networking conference, sep. 1998
- [13]. L.S. Brakmo, S.W. O'Malley, and L. L. Peterson. TCP Vegas: new techniques for congestion detection and avoidance. In proceesings of ACM SIGCOMM, Aug. 1994
- [14] R. Jain, A delay-based approach for congestion avoidance in interconnected heterogeneous computer networks, ACM computer communications review, 19(5):56-71, Oct, 1989
- [15] D. Mitra and J.B. Seery. dynamic adaptive windows for high-speed data networks: theory and simulations. In proceedings of ACM SIGCOMM, Aug. 1990.
- [16] M. Jain, C. Dovrolos. end-to-end available bandwidth: measurement methodology, dynamics, and relation with TCP throughput. In proceedings of SIGCOMM Aug. 2002.
- [17] B. Melander, M. Bjorkman, P. Gunningberg, "A new end-to-end probing and analysis method for estimating bandwidth bottlenecks," in IEEE Global Internet Symposium, 2000.
- [18] J. S. Chase, A. J. Gallatin, and K. G. Yocum, "End-system optimizations for high-speed TCP", In IEEE Communications, special issue on TCP Performance in Future Networking Environments, vol. 39 no. 4, April 2001
- [19] D. A. Reed, Grids, the TeraGrids, and Beyond. IEEE Computer, 2003. 36(1):p 62-68
- [20] Ryan X. Wu, and Andrew Chien, "*GTP: Group Transport Protocol for Lambda-Grids*", in Proceedings of the 4th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid), April 2004