# Performance-Adaptive Prediction-Based Transport Control over Dedicated Links

Xukang Lu[1], Qishi Wu[1], Nageswara S.V. Rao[2], and Zongmin Wang[3]

[1] Dept of Computer Science, University of Memphis, Memphis, TN 38142, USA
{xlv,qishiwu}@memphis.edu

[2] Computer Sci. Math. Div., Oak Ridge National Lab., Oak Ridge, TN 37831, USA
raons@ornl.gov

[3] Henan Key Lab On Info. Net., Zhengzhou Univ., Zhengzhou, Henan 450052, China
zmwang@zzu.edu.cn

**Abstract.** Several research and production networks now provide multiple Gbps dedicated connections to meet the demands of large data transfers over wide-area networks. End users, however, have not been able to see corresponding increase in application goodputs mainly because (i) such rates have pushed the bottleneck from the network to the end system, and (ii) the traditional transport methods are not optimized for handling host dynamics. Due to the sharing with unknown background workloads, the data receiver oftentimes lacks sufficient system resources to process packets arriving from high-speed dedicated links, therefore leading to significant packet drops at the end system. We propose a rigorous design approach for a new class of transport protocols that explicitly account for the dynamics of the running environment to maximize application goodputs over dedicated connections. The control strategy of the proposed transport method combines two aspects: (i) the receiving bottleneck rate is predicted based on performance modeling, and (ii) the sending rate is stabilized at the estimated bottleneck rate based on stochastic approximation. We test the proposed method on a local dedicated connection and the experimental results illustrate its superior performance over existing methods.

**Keywords:** Transport control, dedicated networks, performance modeling.

## 1 Introduction

Many large-scale scientific, engineering, and e-commerce applications require the rapid transfer of vast amounts of data on the order of terabytes or petabytes. Efforts to improve the data transfer performance in the shared Internet met little success due to the variable limited available bandwidth in response to cross traffic. Dedicated networks provisioning multiple Gbps connections have been recognized to be a promising solution and a number of high-performance network initiatives are currently underway including Dynamic Resource Allocation via

GMPLS Optical Networks (DRAGON) [1], UltraScience Net (USN) [17], Circuit-switched High speed End-to-End Transport ArcHitecture (CHEETAH) [20], and others.

However, end users have not been able to see corresponding goodput[1] increase in their applications mainly because (i) such rates have pushed the bottleneck from the network to the end system, and (ii) the traditional transport methods are not optimized for handling host dynamics. Due to the lack of a system-wide advance reservation scheme, the data receiver running in a shared computing environment with other resource-demanding workloads oftentimes could not obtain sufficient system resources to process packets arriving from high-speed dedicated links, therefore leading to significant packet drops at the end system.

The current research efforts on transport protocol design are mainly focused on TCP enhancements and rate-based application-level protocols over UDP. The widely deployed TCP, which has been proved to be remarkably successful in the Internet, is not adequate to achieve high goodput in wide-area dedicated networks because the Additive Increase Multiplicative Decrease (AIMD)-based congestion control algorithm is not well suited for links with high Bandwidth Delay Product (BDP). In TCP, packet loss is detected either by timeout of an unacknowledged segment or several duplicated acknowledgements. If packet loss is caused by network congestion, TCP is able to achieve a reasonable link utilization. However, many observations have shown that packet loss is a poor indicator of network congestion, especially in high-speed dedicated networks where congestion has been pushed to the end system. Various TCP enhancements have been proposed to improve throughput performance, including TCP vegas [5,16], Scalable TCP [12], High Speed TCP for large congestion windows [14], XCP (eXplicit Control Protocol) [11]. and many others [8]. Diverging from TCP's AIMD control, a number of UDP-based high-performance transport protocols use non-AIMD rate control to overcome TCP's throughput limitation for high BDP networks. These protocols include Hurricane [18], SAUBUL (Simple Available Bandwidth Utilization Library)/UDT (UDP-based Data Transfer) [9], FRTP (Fixed Rate Transport Protocol) [19], RBUDP (Reliable Blast UDP)/LambdaStream [10], and Tsunami [2].

However, the main design goal of the aforementioned transport methods based on either TCP or UDP is still to address the congestion over network links, not to account for the dynamics of the end system. As a matte of fact, besides processing power, many other host factors including NIC-system-application interactions, memory/buffer management, CPU scheduling, and disk I/O, collectively contribute to the complex end-system dynamics and play a significant role in determining the application goodput in high-speed dedicated networks. Therefore, to maximize application goodputs, transport protocols need to incorporate a performance-adaptive mechanism through which the data sender and receiver could suitably adjust their sending and receiving activities in response to the system dynamics.

[1] Goodput only counts the user payload and is equivalent in value to throughput if packet duplicates and protocol headers are negligible.

The goal of our work is to present Performance-Adaptive Prediction-based Transport Control (PAPTC) that explicitly accounts for the dynamics of the end system to maximize application goodputs over dedicated connections. With rigorous design and careful analysis, the control strategy of PAPTC combines two aspects: (i) the receiving bottleneck rate is predicted based on performance modeling, and (ii) the sending rate is stabilized at the estimated bottleneck rate based on a stochastic approximation (SA) method. We construct a mathematical model for the data receiving process and employ an autoregressive method to predict the receiving bottleneck rate, which is sent back to the sender for rate control. To account for both network and host dynamics and achieve quick convergence, we adjust the source rate for goodput stabilization at the estimated receiving bottleneck rate using the Robbins-Monro SA algorithm: the source rate is continuously adjusted to match the bottleneck receiving rate at a strategically selected interval. We test the proposed method on a local dedicated connection and the experimental results illustrate its superior performance over existing methods.

The rest of the paper is organized as follows. In Section 2, we briefly outline the framework of PAPTC structure. In Section 3, we present a performance model for the data receiver, and in Section 4, we describe the rate control algorithm for the data sender. The implementation details and experimental results are provided in Section 5.

## 2 Framework of PAPTC Structure

PAPTC employs a UDP-based transport control structure for disk-to-disk data transfer as shown in Fig. 1. The sender (source) reads data sequentially from its local storage device as a set of UDP datagrams of *Maximum Datagram Size* (MDS), each of which is assigned a unique continuous sequence number and loaded into the sender buffer. The source sending rate $r_S(t)$ at time $t$ is regulated by a pair of congestion window $W(t)$ and sleep or idle time (i.e. inter-window delay) $T(t)$. The receiver (destination) accepts incoming datagrams in the order of their arrival and keeps track of the datagram sequence numbers in a checklist. The received datagrams are immediately forwarded to a disk I/O module that handles datagram reordering if necessary and writes them to the disk in order in the background. Based on the status of the datagram checklist, an either positive or negative acknowledgment (ACK) of lost datagrams during an interval $I(t)$ is generated and sent periodically to the sender for retransmission.

As shown in Fig. 1, the data flow moves from source to destination along the solid lines and the acknowledgment feedback follows the dotted lines from destination to source. In this transport structure, there are two control operations represented by two shaded elliptic boxes: (a) source rate control through idle time and (b) ACK event interval control. The transport performance over high-speed dedicated channels critically depends on the strategies implemented for these two control operations. Many transport control protocols send a positive acknowledgment for a received data packet, which is necessary for shared lossy

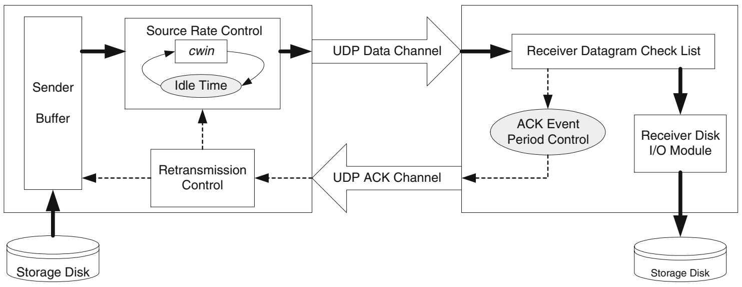

**Fig. 1.** Transport control structure for disk-to-disk data transfer

links in Internet environments. However, dedicated channels usually provide very reliable connections where packet loss rarely occurs. At high data rates, generating and sending acknowledgments at the receiver consumes CPU time and may interfere with the host receiving process. Similarly, accepting and processing acknowledgments at the sender may also affect the host sending process. To achieve peak performance over dedicated channels, we employ a mixed acknowledgment mechanism that sends an either positive or negative acknowledgment after a carefully selected period of time. An appropriate delay time of mixed acknowledgments is adaptively determined for network connections based on link and host properties.

## 3 Performance Model for Data Receiver

We present an analytical study on the impact of system properties on the performance of transport protocols. To instantiate the analysis, we consider the Linux kernel.

### 3.1 Packet Processing Issues

For convenience, we plot in Fig. 2 an overview of Linux packet processing that involves the NIC hardware, device drive, kernel protocol stack, and application [7]. When a new packet arrives, the NIC generates an interrupt and the packet is put into the kernel buffer by the card DMA engine. In general, heavily engaging the CPU in other compute-bound tasks during an interrupt may severely hinder a running process. To avoid flooding the host system with too many interrupts, the interrupt coalescence scheme collects multiple packets and generates one single interrupt for them, therefore reducing the amount of time that the CPU would otherwise have to spend on context switching to serve multiple interrupts. The Linux kernel uses *sk_buff* structure to hold any single packet. The pointers of *sk_buff* are held in a ring buffer in the kernel memory and manipulated through the network stack. If there are no free pointers in the ring buffer, incoming packets will be dropped by the kernel silently. From the ring buffer, the packets are delivered to the corresponding receiving function of the IP layer, which examines

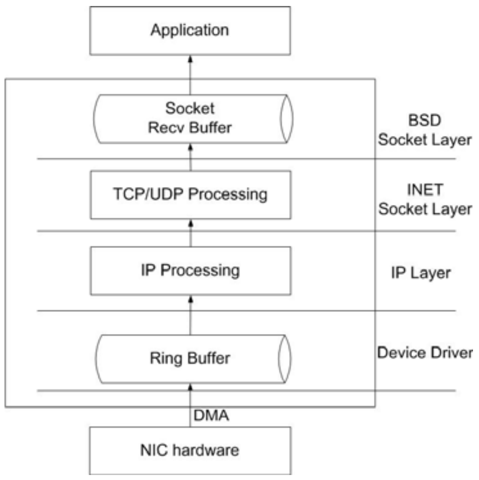

**Fig. 2.** Packet processing flow in Linux

the packets for errors and then forwards them up to the INET Socket layer (such as TCP or UDP), which in turn checks for errors and copies the packets into the socket receive buffer. Then, the waiting application wakes up and returns from a corresponding receive system call that copies the data from the kernel into the application buffer. The flow control mechanism of TCP is implemented to avoid packet drops in the receive buffer. However, the UDP receive buffer might be overflowed if the packet receiving process can not acquire enough CPU cycles to consume the data in the buffer due to CPU contention. In this case, all incoming packets are discarded, hence wasting the protocol processing resources and impairing the application performance.

The Linux packet processing flow shows that packet drops by the kernel could happen in either the ring buffer, or the socket receive buffer, or both. Since the data receiving process has a lower priority than the packet processing by the kernel and the Interrupt Service Routine (ISR), packets are more likely to drop in the socket receive buffer. Although UDP is buffered on both the sender and receiver sides, we focus on the analysis of the receiver side since the receiver is under considerably more system strain than the sender.

## 3.2 Mathematical Model for Data Receiving Process

Linux 2.6 is a preemptive multi-processing kernel whose scheduling policy is priority-based and is explicitly in favor of I/O bound processes in order to provide a fast process response time (interactive processes are I/O bound). Processes are initially assigned with static priorities, which can be modified dynamically by the scheduler to fulfill scheduling objectives. The Linux scheduler calculates a dynamic priority through the static priority and interactivity of the process. A process with a higher interactivity is assigned with a higher dynamic priority

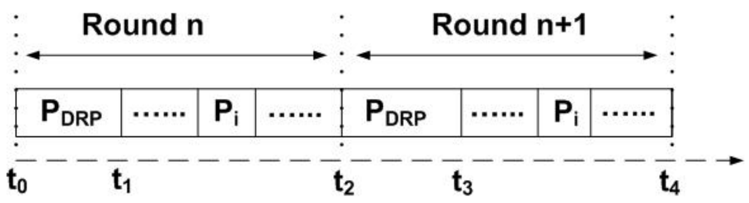

**Fig. 3.** Data receiving process running model ($P_{DRP}$ representing the data receiving process)

and hence runs more frequently. On the contrary, CPU bound processes receive a lower dynamic priority. The timeslice of a process is determined by its dynamic priority per round of execution. Thus, important processes are assigned a longer timeslice that enables these processes to run longer. The old Linux CPU scheduler recalculates each task's timeslice using an $O(n)$ algorithm implemented as a loop over each task; while the newer Linux scheduler maintains two priority arrays, an active array and an expired array, with $O(1)$ complexity for priority updating. Processes move from the active array to the expired array when they exhaust their timeslices. Recalculating all timeslices is just to switch the active and expired arrays [15].

Based on the above analysis, the running behavior of the data receiving process is shown in Fig. 3. Let $t_{DRP}$ and $t_{EXP}$ be the CPU time and the expired time assigned to the data receiving process, respectively, and $t_{TOT}$ be the total CPU time assigned to all the running processes. We have:

$$t_{DRP} = timeslice(P_{DRP}). \tag{1}$$

$$t_{TOT} = timeslice(P_{DRP}) + \sum_{i=1}^{n} timeslice(P_i), \qquad P_i \neq P_{DRP}. \tag{2}$$

The expired time for the data receiving process is:

$$t_{EXP} = t_{TOT} - t_{DRP}. \tag{3}$$

From Eqs. 1, 2 and 3, we know that the running time of the data receiving process is contingent on its own priority and the system load, which includes all interrupt-related processing and handling as well as the load of concurrent processes. Note that interrupt handling has the highest priority and is always scheduled to run before other tasks. Hence, a system with a high interrupt rate is not able to respond to the data receiving process immediately, resulting in a decreased data receiving rate. In an extreme case where the system is completely occupied for handling interrupts, the data receiving process could be temporarily suspended, resulting in significant packet losses in the socket receive buffer. Similarly, a system heavily loaded with concurrent processes could not guarantee enough CPU cycles for the data receiving process because processes with higher priorities may starve the data receiving process. To increase the data receiving rate, one needs to either increase the data receiving process' priority or reduce the system load. However, reducing the system load does not seem to be a viable solution since the data receiving process typically runs with other concurrent resource-intensive workloads in a shared computing environment.

We denote the packet processing rate through the kernel protocol stack as $\lambda$ and the effective service rate of the data receiving process as $\mu$ in the unit of bits per second (bps). Therefore, $\frac{1}{\mu}$ is the time the receiving process takes to copy an incoming packet from the kernel socket receive buffer into the application buffer. We wish to determine an appropriate size of the UDP's receiving buffer to match the kernel packet processing rate $\lambda$ with the data receiving rate $\mu$. Let $t$ be the time in seconds and $m$ be the UDP buffer size in bytes. The time to deplete $m$ when the packet receiving process runs out of its time slice is given by:

$$t = \frac{m}{\lambda}. \tag{4}$$

On the other hand, the time to deplete $m$ when the CPU time is available to process the arriving packets is given by:

$$t = \frac{m}{\lambda - \mu}. \tag{5}$$

At time $t$, the kernel socket receive buffer is not able to accept any new packets and thus will have to drop them. The depleted UDP buffer results in the drop of the UDP datagrams received by the kernel.

### 3.3 Predicting Bottleneck Processing Rate at the Receiver

We collected source rates, goodputs, loss rates and retransmission rates over the USN-ESnet hybrid channel using a UDP-based transport profile generator [18], as shown in Fig. 4. These profiles illustrate how the destination acknowledgement interval together with the source rate affects the transport performance over dedicated channels. We observed that the peak goodput is achieved with low loss and low retransmission rates, which inspires us to derive the desired bottleneck rate.

Let $T_{ts}$ be the timeslice of the data receiving process in one round and $T_{tp}$ be the average time required for copying one packet from the socket receive buffer to the application buffer. The average processing rate $\overline{\mu}$ in this round can be calculated as:

$$\overline{\mu} = \frac{T_{ts}}{T_{tp} \cdot t_{TOT}}. \tag{6}$$

We consider two cases. (i) If $\lambda > \overline{\mu}$, the socket receive buffer will become full after time $t$, as shown in Eqs. 4 and 5. In this case, the data receiving process is not able to consume all the packets arriving from the network, resulting in packet loss in the socket receive buffer. At high data rates, generating and sending packets retransmission requests at the receiver consume CPU time and may significantly interfere with the data receiving process. (ii) If $\lambda < \overline{\mu}$, the data

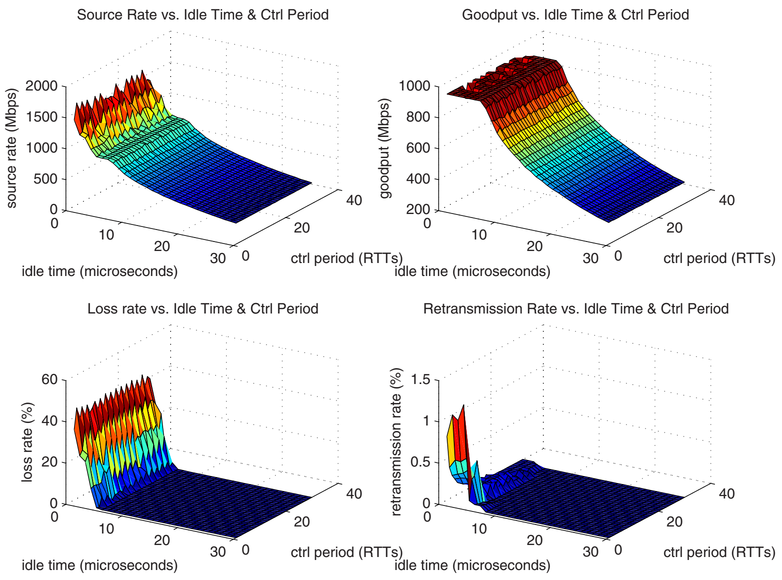

**Fig. 4.** Goodput, loss and re-transmission profiles of PLUT over 9900 mile 1Gbps USN-ESnet hybrid connection

receiving process has sufficient CPU cycles to consume the packets but there are no enough packets in the socket receiving buffer. In this case, the socket receiving buffer could become empty and there are still idle CPU cycles, both of which are a waste of system resources. The transport profiles show that the receiver cannot achieve the peak goodput in either case. So, $\overline{\mu}$ is the corresponding bottleneck processing rate for achieving peak goodput on the receiver side.

We know that Linux 2.6 is a preemptive multi-processing kernel whose scheduling policy is priority-based and is explicitly in favor of I/O bound processes in order to provide a fast process response time (interactive processes are I/O bound). The timeslice of a process is determined by its dynamic priority per round of execution. Thus, important processes are assigned a longer timeslice that enables these processes to run longer. So in order to get a longer timeslice to increase the value of $\overline{\mu}$, the data receiving process should be given a high priority.

In practice, we can sample $\mu$ at an carefully selected interval $\Delta$. We denote such a sequence of $\mu$ samples as $< \mu_T >= ...\mu_{T-1}\mu_T\mu_{T+1}...$. If $\mu_{T+k}$ is known for $k > 0$, we could predict $\mu_{T+k+1}$ in some way. We define the following notations to facilitate the description of the prediction strategy:

- $\mu_T$: the service rate of the data receiving process at the $T$-th measurement.
- $\mu_{T+1}$: the prediction service rate for the $(T+1)$-th measurement.
- $N$: the number of historical data points used for the prediction of $\mu_{T+1}$.

We measure the prediction quality by the *mean squared error*, which is the average of the square of the difference between predicted values and actual values.

We treat the sequence of periodic samples of $\mu$ as a linear time series. We employ the autoregressive (AR) model [3,6], a common approach for modeling univariate time series:

$$X_t = \delta + \phi_1 X_{t-1} + \phi_2 X_{t-2} + ... + \phi_p X_{t-p} + A_t, \tag{7}$$

where $X_t$ is the time series, $A_t$ is the white noise, and $\delta$ is a constant. The value of $p$ is called the order of the AR model. After measuring $\mu_t$, we can predict the value of $\mu_{t+1}$ at time $(t+1)$ using the AR model.

## 4 Rate Control for Data Sender

Based on the performance model, the receiver sends the predicted bottleneck rate back to the sender periodically. If the sender just simply fix data sending rate at the bottleneck processing rate, it will not yield the highest goodput at the receiver which in turn involves accounting at some level for both network and host dynamics. Let $r_S(t)$ be the rate at which packets are sent and let $l(t)$ be the fraction of them that are lost before being read by the receiver, and hence have to be retransmitted. Let $x(t)$ be the fraction of $r_S(t)$ that corresponds to retransmitted packets. Thus the flow $r_S(t)$ is composed of two streams of rates $g_S(t)$ and $x(t)r_S(t)$ corresponding to packets sent for the first time and retransmissions, respectively. In general the data processing rate $\mu_R(t)$ at the receiver depends on $r_S(t)$, $l(t)$ and $x(t)$. The effect of randomness necessitates the utilization of stochastic approximation methods, which has a non-trivial effect on the underlying transport method: the step sizes used in parameter adaptation must be appropriately varied as per conditions such as in classical Robbins-Monro case [13]. To take into account the random effects, we define *processing-rate regression* as

$$G_R(r) = E\left[\hat{\mu}_R(t) | r_S(t) = r\right]. \tag{8}$$

Similarly, we have *loss-fraction* and *retransmission-fraction regressions* defined as

$$L(r) = E\left[\hat{l}(t) | r_S(t) = r\right] \quad \text{and} \quad X(r) = E\left[\hat{x}(t) | r_S(t) = r\right]. \tag{9}$$

Let $\mu^*$ be the attainable bottleneck processing rate at the receiver over a given dedicated connection. The objective of APPTC control is to stabilize $r(.)$ at a suitable rate $r^*$, such that:

$$G_R(r^*) = \mu^* = r^*[1 - X(r^*)], \tag{10}$$

which ensures that peak throughput is attained at low loss rate.

At time step $k$, for the measured source rate $\hat{r}_S(k)$, measured processing rate $\hat{\mu}_R(k)$, and measured retransmission rate $\hat{x}(k)$, the equation $\hat{r}(k) = \hat{\mu}(k)/[1 - \hat{x}(k)]$ is only approximately satisfied. For $\hat{r}_S(k) = a(k) \cdot \mu^*(k)$ and $\hat{\mu}_R(k) = \alpha \cdot \mu^*(k)$, the coefficient function are typically $a(k) >= 1$ and $\alpha(k) <= 1$. Thus there are two possible estimates of $\mu^*(k)$ based on $\hat{r}_S(k)$ and $\hat{\mu}_R(k)$, which yield two different values. We consider the following general form that combines these two estimates:

$$\hat{\mu}^*(k) = [\hat{r}_S(k)(1 - \hat{x}(k))]^{\beta} \hat{\mu}_R(k)^{1-\beta}, \quad 0 \leq \beta \leq 1, \tag{11}$$

where $\beta$ is determined by host and link properties. Typically, $\hat{r}_S(k)$ and $\hat{x}(k)$ are more stable compared to $\hat{\mu}_R(k)$ since the the former are not subject to connection-level variations. For the specific case where $\alpha(k) = 1/a(k)$, we have $\hat{\mu}^*(k) = \sqrt{\hat{r}_S(k)\hat{g}_R(k)}$. To account for randomness in measurements and the effects of delay and its variation of sending rate $\hat{r}_S(k)$ on processing rate measurement $\hat{\mu}_R(k)$, we apply a dynamic version of Robbins-Monro method [13] to adjust the source rate to achieve the target bottleneck processing rate $\mu^*(k)$ at the receiver:

$$\hat{r}_S(k+1) = \hat{r}_S(k) - \rho_k[\hat{\mu}_R(k) - \hat{\mu}^*(k)], \tag{12}$$

where the time step adjustment coefficient is given by $\rho_k = b/k^{\gamma}$ for $0.5 < \gamma < 1.0$ and $b > 0$, a suitably chosen constant. The sending rate will increase if the measured processing rate $\hat{\lambda}_R(k)$ is less than the estimated maximum attainable processing rate $\hat{\mu}^*(k)$ at low sending rates; while in the source rate control zone approaching the peak processing rate, the processing rate measurement may exceed the maximum processing rate estimate due to increased retransmission rate, causing the sender to back off.

The step sizes satisfy the Robbins-Monro property namely, $\sum_{k=1}^{\infty} \rho_k = \infty$ and $\sum_{k=1}^{\infty} \rho_k^2 < \infty$. We assume that the errors satisfy the following martingale property for $\hat{r}_S(k) = r$:

$$E\left[\hat{g}(k) - \hat{g}^*(k) | \hat{r}_S(k) = r\right] = G_R(r) - [r(1 - X(r)]^{\beta} G_R(r)^{1-\beta},$$

which essentially assumes that the errors are not correlated across the time steps other than through $\hat{r}(.)$. Then the limit behavior of Eq. 12 is specified by the Ordinary Differential Equation (ODE) (Chapter 5, [13]):

$$\frac{d\hat{r}}{dt} = E\left[\hat{\mu}^*(k) - \hat{\mu}_R(k)\right] = E\left[\hat{\mu}^*\right] - G_R(\hat{r}).$$

Under low loss condition, we approximate

$$E[\hat{\mu}^*] = [\hat{r}(1 - X(\hat{r})]^{\beta} G_R(\hat{r})^{1-\beta}.$$

Then under the conditions (A.1), (A.3-4), the solution to ODE is given by the stationary point corresponding to

$$G_R(\hat{r}) \left[1 - \left(\frac{\hat{r}[1 - X(\hat{r})]}{G_R(\hat{r})}\right)^{\beta}\right] = 0,$$

which in turn corresponds to $G_R(\hat{r}) = \hat{r}[1 - X(\hat{x})] = \mu^*$. Thus the limit behavior of this algorithm is to stabilize at sending rate $\hat{R}_S(k) \rightarrow \hat{r}$ such that $\hat{\mu}_R(k) \rightarrow \mu^*$

**Table 1.** Goodput performance comparison (Mbps) without concurrent workloads

| 10 cases without load | #1 | #2 | #3 | #4 | #5 | #6 | #7 | #8 | #9 | #10 | Std. Dev. |
|---|---|---|---|---|---|---|---|---|---|---|---|
| PAPTC | 878 | 860 | 867 | 863 | 860 | 864 | 872 | 860 | 869 | 868 | 5.915 |
| UDT | 835 | 861 | 836 | 860 | 826 | 857 | 848 | 848 | 842 | 832 | 12.296 |
| Tsunami | 669 | 662 | 679 | 687 | 667 | 667 | 664 | 666 | 696 | 680 | 11.275 |

as $k \to \infty$. Alternatively, the required stability property can be derived for this algorithm using the monotonic property of $G_R(.)$ and $X(.)$ to show this convergence result as in [4]. Thus, this step ensures that PAPTC probabilistically stabilizes at the bottleneck processing rate $\lambda^*$ of the connection while ensuring the low loss rate. Informally, by maintaining $r_S(t) = \overline{\mu}$, we would achieve an average goodput of $g^*$, and an increase (decrease) in $r^*$ results in an increase (decrease) in $M(r^*)$.

## 5 Implementation and Experimental Results

The proposed transport protocol is implemented according to the architecture shown by Fig. 1, written mostly in C++ on Linux operating system.

### 5.1 Types of Acknowledgment

The proposed PAPTC protocol is implemented in C++ in Linux. We consider four different types of acknowledgment at the receiver: NXT (Next), RXM (Retransmission), TNT (Timeout Next), and TMO (Timeout Retransmission). For every normal ACK control period, if all datagrams received so far are in continuity, an "NXT" ACK is generated and sent to the sender; otherwise if there are lost datagrams (i.e. "holes" in the datagram checklist), the receiver compiles a list of lost datagram sequence numbers and sends them with a "RXM" ACK. If no datagram is received within a certain period of time, a timeout event is triggered where the receiver sends either a "TNT" ACK if all datagrams received so far are in continuity, or a "TMO" ACK enclosing the lost datagram sequence number list if there are "holes" in the datagram checklist. For all ACK types, the receiver measures the current bottleneck processing rate and sends it to the sender as part of the acknowledgment. On the sender side, for each incoming acknowledgment, we apply rate control as described in Section 4 using the bottleneck processing rate measurements enclosed in the acknowledgment.

### 5.2 Experimental Results

For performance comparison, we run PAPTC, UDT (version 4.4) and Tsunami on a local dedicated connection, which is provisioned by a back-to-back link between two Linux boxes with kernel 2.6.27. Each Linux box is equipped with a 1 Gigabit NIC, AMD Athlon(tm) 64X2 Dual Core Processor 5000+, 2 GBytes of RAM, and 900 GBytes of SCSI hard drive. A CPU-bound program named cpuburn is specifically designed and executed to emulate concurrent host background

**Table 2.** Goodput performance comparison (Mbps) with 2 concurrent cpuburn processes

| 10 cases with 2 cpuburn proc. | #1 | #2 | #3 | #4 | #5 | #6 | #7 | #8 | #9 | #10 | Std. Dev. |
|---|---|---|---|---|---|---|---|---|---|---|---|
| PAPTC | 816 | 802 | 813 | 801 | 807 | 818 | 808 | 815 | 807 | 806 | 5.889 |
| UDT | 716 | 718 | 734 | 717 | 716 | 717 | 718 | 718 | 726 | 741 | 10.601 |
| Tsunami | 670 | 676 | 676 | 669 | 679 | 675 | 661 | 668 | 639 | 671 | 11.549 |

**Table 3.** Goodput performance comparison (Mbps) with 4 concurrent cpuburn processes

| 10 cases with 4 cpuburn proc. | #1 | #2 | #3 | #4 | #5 | #6 | #7 | #8 | #9 | #10 | Std. Dev. |
|---|---|---|---|---|---|---|---|---|---|---|---|
| PAPTC | 655 | 654 | 643 | 646 | 644 | 682 | 635 | 656 | 665 | 633 | 14.622 |
| UDT | 613 | 623 | 615 | 626 | 610 | 623 | 629 | 620 | 618 | 622 | 5.934 |
| Tsunami | 622 | 625 | 621 | 622 | 628 | 626 | 623 | 625 | 626 | 621 | 2.424 |

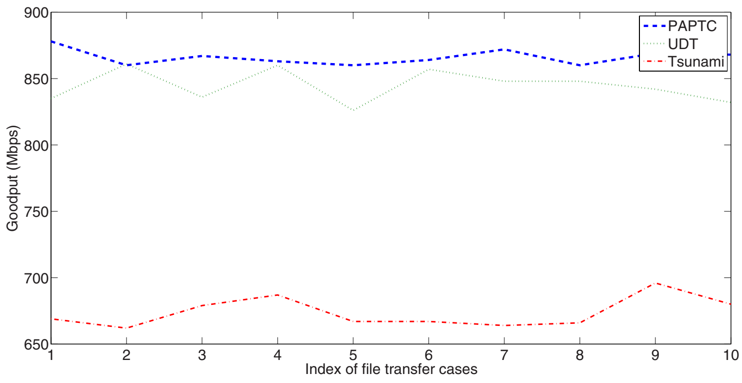

**Fig. 5.** Goodput performance comparison without concurrent load

workloads. We conduct three sets of transport experiments, in each of which, 10 files are transferred using three transport methods. In the first set of experiments, no cpuburn process is executed while in the other two sets of experiments, 2 and 4 concurrent cpuburn processes are launched, respectively. The goodput performance measurements and standard deviations for three transport methods are tabulated in Tables 1, 2, and 3, and their corresponding performance curves are plotted in Figs. 5, 6, and 7 for a better visual comparison. From these measurements, we observe that the amount of concurrent background workloads has a significant effect on the performance of each transport method. Tsunami is relatively insensitive to the change of concurrent workloads at a sacrifice of its goodput performance. Similar to PAPTC, UDT also adapts to the workload changes but adopts a somewhat more conservative rate control than PAPTC.

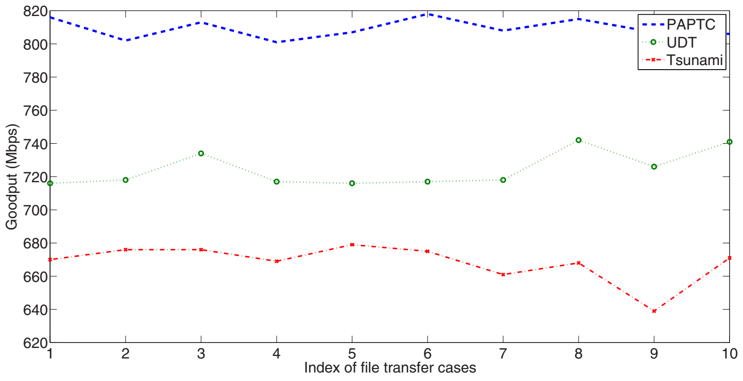

**Fig. 6.** Goodput performance comparison with 2 concurrent cpuburn processes

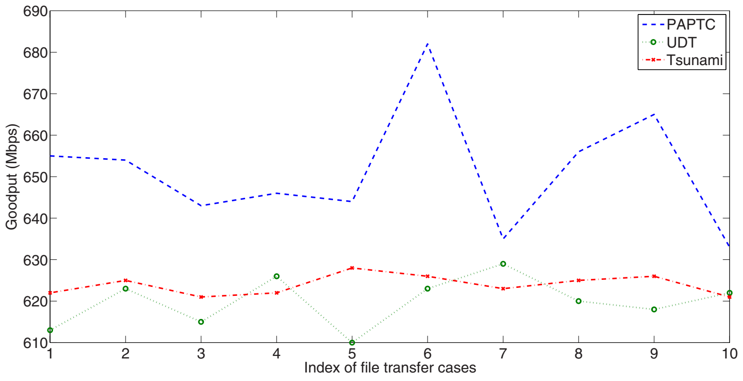

**Fig. 7.** Goodput performance comparison with 4 concurrent cpuburn processes

In all the cases we studied, the proposed PAPTC protocol consistently achieves higher goodputs than the other two methods in comparison.

## 6 Conclusion

We developed PAPTC to support high-speed data transfers over dedicated channels. To account for the host dynamics and the random components in transport performance measurements, we designed control strategies based on performance modeling and stochastic approximations to achieve sustained high goodputs at a low packet loss. We implemented and tested PAPTC over a back-to-back connection and the experimental results illustrated its superior performance over existing methods.

## Acknowledgments

This research is sponsored by National Science Foundation under Grant No. CNS-0721980 and Oak Ridge National Laboratory, U.S. Department of Energy, under Contract No. PO 4000056349 with University of Memphis.

## References

1. DRAGON: Dynamic Resource Allocation via GMPLS Optical Networks, `http://dragon.maxgigapop.net`
2. Tsunami, `http://newsinfo.iu.edu/news/page/normal/588.html`
3. Beltran, M., Guzman, A.: A new cpu availability prediction model for time-shared systems. IEEE Transaction 2009 57, 865–875 (2009)
4. Benveniste, A., Metivier, M.: Adaptive Algorithms and Stochastic Approximation. Springer, New York (1990)
5. Brakmo, L., O'Malley, S., Peterson, L.: Tcp vegas: new techniques for congestion detection and avoidance. In: SIGCOMM 1994 Conf. on Communications Architectures and Protocols, London, United Kingdom, October 1994, pp. 24–35 (1994)
6. Dinda, P., OHallaron, D.: Host load prediction using linear models. Cluster Computing 3(4), 265–280 (2000)
7. Rio, M., et al.: A map of the networking code in linux kernel 2.4.20. Technical Report DataTAG-2004-1 (March 2004)
8. Floyd, S.: Highspeed tcp for large congestion windows, Internet Draft (February 2003)
9. Gu, Y., Hong, X., Mazzucco, M., Grossman, R.L.: SABUL: A high performance data transfer protocol. Submitted to IEEE Communications Letters (2004)
10. He, E., Leigh, J., Yu, O., DeFanti, T.: Reliable blast udp: predictable high performance bulk data transfer. In: IEEE Int. Conf. on Cluster Computing, Chicago, Illinois, September 23-26 (2002)
11. Katabi, D., Handley, M., Rohrs, C.: Internet congestion control for future high-bandwidth-delay product environments. In: Proc. of ACM SIGCOMM 2002, Pittsburgh, PA, August 19-21 (2002), `http://www.acm.org/sigcomm/sigcomm2002/papers/xcp.pdf`
12. Kelly, T.: Scalable tcp: Improving performance in highspeed wide area networks. In: Workshop on Protocols for Fast Long-Distance Networks (Februrary 2003)
13. Kushner, H.J., Yin, C.G.: Stochastic Approximation Algorithms and Applications. Springer, New York (1997)
14. Kuzmanovic, A., Knightly, E., Cottrell, R.L.: Hstcp-lp: A protocol for low-priority bulk data transfer in high-speed high-rtt networks. In: The Second Int. Workshop on Protocols for Fast Long-Distance Networks (February 2004)
15. Love, R.: CPU Scheduler. Sams (2003)
16. Low, S., Peterson, L., Wang, L.: Understanding vegas: a duality model. J. of the ACM 49(2), 207–235 (2002)
17. Rao, N., Wing, W., Carter, S., Wu, Q.: Ultrascience net: Network testbed for large-scale science applications. IEEE Communications Magazine 43(11), s12–s17 (2005), `http://www.csm.ornl.gov/ultranet`

18. Wu, Q., Rao, N.: Protocol for high-speed data transport over dedicated channels. In: Proc. of the 3rd Int. Workshop on Protocols for Fast Long-Distance Networks, February 3-4, pp. 155–162 (2005)
19. Zheng, X., Mudambi, A., Veeraraghavan, M.: Frtp: Fixed rate transport protocol – a modified version of sabul for end-to-end circuits. In: Proc. of Broadnets (2004)
20. Zheng, X., Veeraraghavan, M., Rao, N., Wu, Q., Zhu, M.: Cheetah: Circuit-switched high-speed end-to-end transport architecture testbed. IEEE Communications Magazine 43(11), s11–s17 (2005)