A Feedback Mechanism for Network Scheduling in LambdaGrids *

Pallab Datta, Sushant Sharma
Computer and Computational Sciences Division
Los Alamos National Laboratory
Los Alamos, NM 87545
E-mail:{pallab, sushant}@lanl.gov

Wu-Chun Feng
Department of Computer Science
Virginia Tech
Blacksburg, VA 24061
E-mail: feng@cs.vt.edu

Abstract

Next-generation e-Science applications will require the ability to transfer information at high data rates between distributed computing centers and data repositories. A LambdaGrid offers dedicated, optical, circuit-switched, point-to-point connections, which may be reserved exclusively for an application. Though such dedicated high-speed connections eliminate congestion in the network, they effectively push the network congestion out to the end systems, as processing speeds have not kept up with networking speeds. Therefore, developing an efficient transport protocol over such high-speed dedicated circuits is of critical importance.

In this work, we propose the idea of a lightweight end-system protocol, based on performance monitoring, to significantly improve the performance of data transport over a LambdaGrid. In particular, we focus on dynamically monitoring the OS task scheduling at the receiving end-system so that potential end-system congestion may be detected early and appropriate feedback can be transmitted back to the sending end-system to avoid packet losses. One example of such an evasive action is to suspend transmission for a certain duration of time during which the OS on the receiving end-system must handle other computational processes. With this in mind, we propose to extend the Reliable-Blast UDP (RBUDP) protocol to take such evasive action by using a simple feedback mechanism that is activated via performance monitoring. The new protocol, named RBUDP$^+$ *dramatically improves the performance of data transfer over LambdaGrids. We demonstrate the effectiveness of our proposed protocol and illustrate the performance gains achieved via network emulation.*

*This work was supported by the U.S Department of Energy through LANL contract W-7405-ENG-36. This manuscript is also available as Los Alamos Technical Report LA-UR-06-1514.

1 Introduction

The optIPuter project [10] observed that network speeds have been outstripping the ability of processor speeds to keep up. This technology inversion resulted in the emergence of LambdaGrids, which have fundamentally changed the way that we think about distributed computing.

Instead of a generic Grid, where distributed computational facilities are connected by some generic network; a LambdaGrid offers dedicated, optical, circuit-switched, point-to-point connections between such computational facilities. Such dedicated circuit-switched connections avoid the problems of network congestion at intermediate routers of a shared packet-switched network. Examples of networks that enable LambdaGrids include the National LambdaRail (NLR) [3] and DOE's UltraScience Net [2] in the United States, CANARIE's CA*net [5] in Canada, and NetherLight in the Netherlands.

In contrast to shared packet-switched Grid infrastructures, LambdaGrids have computational endpoints that are interconnected via dedicated high-speed links (e.g., OC-192 = 10 Gbps), thus providing an environment with no internal network congestion but significant endpoint congestion. In addition, LambdaGrids typically connect a small number of large computational resources (such as clusters) and might involve data-transfer models ranging from point-to-point communication to a collection of endpoints that engage in many-to-one or one-to-many communication. For example, a distributed scientific computation running on a LambdaGrid might engage in coordinated communication across a number of data servers in order to fetch large quantities of data (e.g., 100 GB) from distinct and distributed servers to feed a local computation or visualization. These and other similar scenarios pose a new set of research challenges for network communication in LambdaGrids.

Optical networks in LambdaGrids typically span over large intra-continental or inter-continental distances, thus resulting in networks with large bandwidth-delay products (BDPs). Delivering high throughput in large BDP networks is a long-standing research challenge, one that now has an en-

tire workshop devoted to it – *The International Workshop on Protocols for Fast Long-Distance Networks (PFLDnet)*. TCP and its variants [7, 4, 8] have been used in shared packet-switched networks for adjusting the sending rate depending on the inferred state of congestion *in* the network. Given that this type of congestion does not occur *in* a dedicated, circuit-switched, optical network; TCP and its variants have been shown to be inefficient in such networks. As a result, numerous UDP-based transport protocols such as Reliable-Blast UDP (RBUDP) [6] and Fixed-Rate Transport Protocol [13] have been proposed in the literature to overcome this limitation.

In RBUDP, the sender transmits UDP data packets at a fixed bit rate, specified by the user. After all the data has been transmitted, the receiver sends the error sequence numbers corresponding to the data packets that it did not receive (due to network congestion in a packet-switched network or end-system congestion in a circuit-switched network) to the sender via a TCP connection. The sender then re-transmits the error sequence data packets via UDP. The above continues until the receiver has received all data packets successfully. In this manner, a reliable mechanism for packet delivery is imposed on top of the unreliable connectionless UDP.

Although RBUDP has been demonstrated to perform fairly well in LambdaGrid environments, its main weakness is its inability to adapt the sending rate. This leads to unwanted packet losses, particularly when the receiving end-system is swamped with too many packets to process, i.e., the network outstrips the ability of the processor to keep up. Though dedicated optical connections such as those in LambdaGrids effectively eliminate congestion *in* the network, the network throughput of such connections (namely 10 Gbps for an OC-192 connection) often exceed the capabilities of data processing at the end system [11, 12], thus creating congestion at the end system.

In addition to receiving data, the receiving end-system is oftentimes expected to be running other processes such as visualization and analysis of the received data which may be computationally intensive. In such a case, the receiving end-system's operating system (OS) has to schedule a computationally (CPU) bound process (visualization and analysis) and an I/O-bound process (receiving data) simultaneously. Because the buffer size on the end-system's network interface card (NIC) is typically small, packets may be dropped due to buffer overflow, i.e., when the receiving data process is not scheduled by the OS at appropriate times to transfer the packets from the line-card buffer in the NIC to physical memory. Transmitting data to such an end system using RBUDP at a fixed rate, only exacerbates the problem of end-system congestion.

In this work, we propose the idea of a lightweight end-system protocol, based on performance monitoring, to significantly improve the performance of data transport over a LambdaGrid. In particular, we focus on dynamically monitoring the OS task scheduling at the receiving end-system so that potential end-system congestion may be detected early and appropriate feedback can be transmitted back to the sending end-system to avoid packet losses. One example of such an evasive action is to suspend transmission for a certain duration of time during which the OS on the receiving end-system must handle other computational processes. With this in mind, we propose to extend the Reliable-Blast UDP (RBUDP) protocol to take such evasive action by using a simple feedback mechanism that is activated via performance monitoring. The new protocol, named *RBUDP*$^+$ dramatically improves the performance of data transfer over LambdaGrids. We demonstrate the effectiveness of our proposed protocol and illustrate the performance gains achieved via network emulation.

The rest of this paper is organized as follows. First, we describe the problem in Section 2. We describe the end-system task monitoring in Section 3. The receiver's algorithm for the explicit feedback mechanism to avoid packet losses is described in Section 4. Section 5 presents the experimental set-up, followed by experimental results in Section 6. We finally conclude the paper in Section 7.

2 Problem Description

Dense wavelength division multiplexing (DWDM) allows optical fibers to carry hundreds of wavelengths of 2.5 to 10 Gbps each for a total of terabits per second capacity per fiber. A LambdaGrid is a set of distributed resources directly connected with DWDM links, in which network bandwidth is no longer the key performance limiter to communication. Compared to shared, packet-switched IP networks, the key distinguishing characteristics of LambdaGrid networks are as follows:

- Very high-speed (e.g., OC-192 or OC-768) dedicated links and long delays between interconnected sites.
- End-to-end network bandwidth that exceeds the data-processing (i.e., computing) capabilities of the attached end-systems.
- Optical links between end-system pairs that are viewed as virtual dedicated connections, which is in contrast to the commonly shared links in a traditional packet-switched network.

Clearly, network performance can be substantially improved in LambdaGrid environments if packet losses due to end-system congestion are avoided when the receiving end-system OS is context-switched to some process other than the networking process. The following are some possible solutions:

- A Real-Time OS (RTOS) can be employed. A Real-Time OS allows for specifying hard deadlines for tasks.

The RBUDP receive process may be classified as a real-time task with periodic hard deadlines specified so that packets can be handled at the incoming data rate. However, a RTOS is expensive to maintain and may not be suitable for all applications. Furthermore, device driver and hardware support is not commonplace for a RTOS. For example, no 10-Gigabit Ethernet NIC support currently exists in a RTOS.

- The buffer size in the network interface card (NIC) can be increased so that packets are not dropped when the OS is not ready to handle them. However, this is a very expensive hardware solution that NIC vendors are generally not willing to provide. The Chelsio 10-Gigabit NICs that we used in our experiments have 512-MB RAM, but it is shared between transmitting and receiving queues, as well as for other tasks.

- Various parameters of the OS scheduler such as maximum allocated time slice, maximum dynamic bonus priority granted to an I/O process, and so on, may be adjusted so as to favor the RBUDP receive process, and thus reduce packet losses. However, this is not a good solution, as it would lead to custom OS kernels for applications. Application scientists running in LambdaGrid environments would rather not deal with customized kernels or kernel patches to improve their network performance.

  Due to the limitations of the above approaches, we propose a rate-adaptive transport protocol that is lightweight and end-system performance aware, so as to maximize the end-to-end throughput while minimizing packet loss in a LambdaGrid environment. Based on the self-monitoring of the dynamic task scheduling at the receiving end-system, our protocol would enable the receiver to proactively deliver feedback to the sender to adapt its sending rate, thus avoiding congestion and packet losses at the receiving end-system. A proactive feedback mechanism that thwarts the sender from swamping the receiver with abundant data during process context-switch intervals could potentially deliver significantly better network performance.

3 End-System Task Monitoring

Our objective is to monitor the end-system performance, so as to identify forecasted periods of end-system congestion. By predicting the time at which the receiving end-system OS may allocate a large time-slice to a CPU-intensive process (and hence, does not respond to packet-handling interrupts from the NIC), we know that this predicted time is when end-system congestion may occur. If the sending end-system does not transmit during such times, packet loss can be averted at the receiving end-system, thus improving the data-transfer performance.

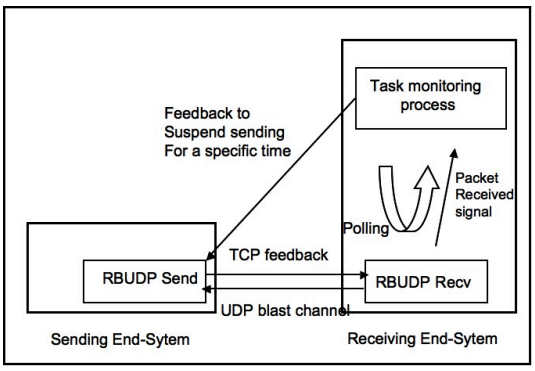

Figure 1. Soft real-time process at the receiving end-system that monitors the task scheduling at the end-system and sends explicit feedback to the sender

For the purpose of monitoring, the following system metrics may be considered: average CPU load, NIC buffer occupancy, task time-slice, and task priority. Out of all the above, we found the task time-slice, coupled with its priority, to be the most helpful indicator. Furthermore, these metrics are trivial to monitor in an OS. As shown in Figure 1, a soft real-time (SRT) process can be implemented at the receiving end-system that monitors the task scheduling at the end-system and sends an explicit feedback notification back to the sender to temporarily stop data transfer. This can significantly improve the end-to-end system throughput by reducing packet loss at the receiving host.

An OS typically distinguishes between an I/O-bound process and a CPU-bound process. For example, the Linux kernel 2.6.x scheduler maintains the dynamic priority of processes, which is the static priority (related to task niceness) plus a dynamic bonus granted based on process interactivity. In the Linux kernel 2.6.x scheduler [9], the static priority for user processes ranges from 0 to 40 (corresponding to niceness values between -20 and +19). A lower number corresponds to higher priority. The dynamic bonus granted by the kernel to boost the priority of interactive tasks ranges from -5 to +5. This is computed proportional to the average sleep time of the task. Since I/O-bound processes are more interactive than CPU-bound processes, they usually have a higher dynamic priority. While choosing a task amongst a set of tasks ready to run, the OS chooses the one with the highest priority.

In the experimental setup considered for our purposes, the RBUDP process starts off as an I/O-bound process with a very high priority. But as a large number of packets are received at the receiver, the I/O-bound receive process quickly uses all of its time-slice. This results in a reduction in the

avg_sleep_time of the I/O-bound process and translates into a very low *bonus*, thus causing the dynamic priority of the RBUDP process to decrease with time and then forcing the RBUDP process to be context-switched out.

4 RBUDP$^+$: Rate-Adaptive Protocol for LambdaGrids

The main objective of our RBUDP$^+$ protocol is to deliver high end-to-end throughput over LambdaGrids by monitoring end-system performance, so as to identify forecasted periods of end-system congestion. As noted in the previous section, by predicting the time at which the receiving end-system OS may allocate a large time-slice to a CPU-intensive process, we can predict when end-system congestion might occur because the receiving end-system OS will not be responding to packet-handling interrupts from the NIC. If the receiver can transmit explicit feedback to the sender so that the sender does not transmit during such times, packet loss can be averted at the receiving end-system, thus improving the data-transfer performance. Consequently, we propose a *RBUDP$^+$* protocol that enables the receiver to send explicit feedback to the sender.

We first note that the Linux kernel 2.6.x scheduler does not contain any algorithms that run in worse than O(1) time. That is, every part of the scheduler is guaranteed to execute within a certain constant amount of time regardless of how many tasks are on the system. The run-queue data structure is the most basic structure in the Linux kernel 2.6.x scheduler; there is one run-queue per processor. Essentially, a run-queue keeps track of all runnable tasks assigned to a particular CPU. In the Linux 2.6 kernel, we have two priority arrays, one is the *active array* and the other is the *expired array*. Each of these arrays consists of different queues of runnable processes each set at a different priority level. For example, in Figure 2 we have different processes in the active array varying between priority levels $1 \cdots m$. Each priority level can have a varying number of tasks, each having a particular allocated *time-slice* for execution, and a static priority (set relative to task niceness) and dynamic priority (set equal to the priority level).

Similarly, we have a set of processes between priority levels $1 \cdots m$, in the *expired array*. A task's static priority is stored in its *static_prio* variable, where *p* is a task, *p→static_prio* is its static priority. The Linux kernel 2.6.x scheduler rewards I/O-bound tasks and punishes CPU-bound tasks by adding or subtracting a task's static priority. The adjusted priority is called a task's dynamic priority and is accessible via the task's *prio* variable (e.g. *p→prio* where *p* is a task). If a task is interactive (the scheduler's term for I/O-bound), its priority is boosted.

The *effective_prio()* function calculates a task's dynamic priority. *effective_prio()* computes the bonus by the following formulae:

bonus = CURRENT_BONUS(p) - MAX_BONUS/2;

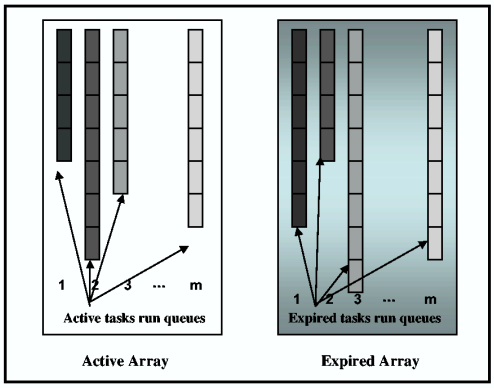

Figure 2. The active and expired priority-arrays with process-queues for different priority levels

prio = p→static_prio - bonus;
#define CURRENT_BONUS(p)
NS_TO_JIFFIES((p)→sleep_avg)× MAX_BONUS/ MAX_SLEEP_AVG);

For simplicity, let us assume that our system has only one I/O-intensive task and several memory-intensive tasks to be scheduled at the receiver end-system. There can be different scenarios in the kernel, which we illustrate here in this section, and propose a feedback mechanism for the most common case. We skip the details of the other cases in this paper since we conjecture that the receiver-bound I/O process would most likely be re-queued in the active array, due to its I/O-bound nature.

The I/O-intensive task can be in any of the priority levels in the active array, as shown in Figure 2. At the end of its time-slice, its dynamic priority is re-calculated, based on the bonus (which again depends on the average sleep time of the task). Depending on the value of the newly calculated dynamic priority and whether other tasks in the *expired array* have surpassed their *STARVATION_LIMIT* the task can get re-entered in any of the priority levels in the *active array* or may be migrated to the *expired array*.

For now, let us assume the simplest case, where we can have a soft real-time process (*SRT*) that monitors the state of the I/O-intensive process in the kernel. Assuming that the I/O process associated with the RBUDP data-receive is the k^{th} process in the active priority array, we can deterministically estimate the point of time when the *SRT* process can send a explicit feedback message back to the sender end-system so as to stop the transfer. This explicit feedback message contains two pieces of information: (1) when to stop transmission and (2) for how long to stop transmission. The former is

Input: All the tasks in the *active array* with their individual time-slices T_i. The time-slice value of the I/O intensive task is $T_{I/O}$.

Let the time instant at which the I/O intensive task is scheduled be given by *t*.

Let the *avg_sleep_time* of each task be denoted by $avg\_sleep\_time_i$.

Output: (a) The time instant when the data transfer needs to be stopped temporarily from the sender.

(b) The duration (D) for which the data transfer is stopped temporarily.

Algorithm: The time instance at which data transfer needs to be stopped temporarily is given by $t + T_{I/O}$ and the notification is sent at the time instant *t*, the I/O task is scheduled. (Assumption: all the tasks will use their individual time-slices and will not enter the sleep/wait queue).

The duration for which data transmission needs to be stopped temporarily $= \sum_i T_i \quad \forall\, T_i \gg avg\_sleep\_time_i$

Note: The above algorithm assumes that the new dynamic priority of the I/O task would be such that it gets scheduled after tasks in the *active array* whose *avg_sleep_time* are negligible compared to their time-slices.

Figure 3. Algorithm for Feedback Notification

calculated as the time period that the RBUDP receive process is scheduled for; the latter is the time period for which the RBUDP receive process is context-switched out.

The main challenge is to determine the time-period for which the *SRT* process asks the sending end-system to suspend data transmission. There are different possible scenarios at the end of the time-slice of the I/O-bound task: it can be enqueued into a queue in the *expired array*, or it might be re-queued into the *active array* after re-calculation of its dynamic priority. Because the RBUDP receive process is I/O-bound and generally has a very high priority, the probability of it getting re-enqueued into the *active array* is very high. Hence, we assume that the RBUDP receive process is re-entered in the *active array* at the end of its time-slice of execution.

Since we cannot be exactly sure at the beginning of the execution of the receiver task, as to what its next dynamic priority would be, we can conservatively estimate that its new dynamic priority would not be less than δ of its current priority. Assuming the I/O-bound RBUDP receive process has a new dynamic priority which is not less than δ of the current priority, we can estimate the time period for which the sender should be asked to suspend its transmission. Suppose the current priority of the receive task is $p \rightarrow prio$. We can then look ahead all the tasks that are in the queues of priority level higher than or equal to $p \rightarrow prio - \delta$. We take each task in each of these priority levels and add their time-slice values for all tasks with smaller *avg_sleep_time* values. This can be used as an estimate for which the I/O-bound process does not get scheduled, and hence, the sender should not be instantiating transmission from its end. One of the assumptions made here in the computation of this time period is that all the other remaining processes in the system finish execution, according to their time-slice values, and do not enter the wait/sleep queue during its execution.

The above calculation assumes that all the tasks that get scheduled after the I/O-bound process, do not re-enter at a priority level higher than $p \rightarrow prio - \delta$ if they are re-entered in the *active array*. This is based on the intuition that these processes initially had a lower priority as compared to the receive process. Hence, they get lower priority values in subsequent scheduling iterations. The detailed algorithm for feedback notification is presented in Figure 3.

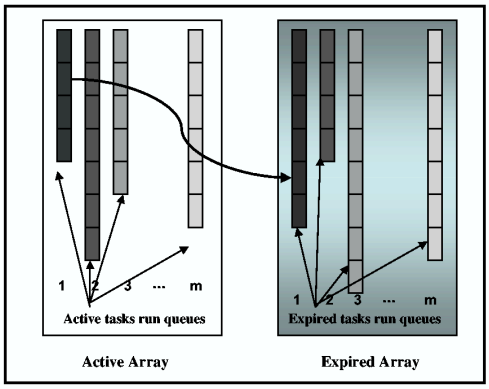

Figure 4. The receiver process moves to the *expired array* at the end of its time-slice.

Now, we analyze some of the other scenarios which can occur in the kernel and propose some insights about when and how feedback can be sent to the sending end-system to prohibit data transfer.

It can happen that the receiving I/O bound task is queued to the expired array after re-calculation of its dynamic priority (as shown in Figure 4). Feedback needs to be sent to the sending end-system at the beginning of the time-slice interval of the receiving process as described before to suspend data transfer for a certain interval of time. We need to estimate the time interval for which the data transfer needs to be stopped temporarily. **Assume** that all the other tasks in the *active array* are CPU-intensive tasks, and they get inserted into the run-queues of the *expired array* after execution and re-calculation of their dynamic priorities. Then the total time interval for which the data transfer needs to be suspended can be easily computed to be the summation of the time-slices of all the tasks in the *active run-queue* and the summation of the time-slices of the tasks ahead of the I/O-intensive task in

the *expired array*. This algorithm however explicitly assumes that each task gets enqueued to the expired run-queue after re-calculation of their dynamic priority at the end of each time-slice interval. It also assumes that each task executes to its completion at the end of their individual time-slices and does not enter the wait-queue in between. It has a computational complexity of O(n), where *n* is the total number of active tasks in the system.

5 Experimental Set-Up

In order to emulate a LambdaGrid network, we used the following experimental set-up. We connected two machines (both the machines were AMD Opteron's with a clock of 2.2 Ghz and a cache of 1024MB and 1GB DDR RAM) back-to-back with Chelsio 10-Gigabit Ethernet (10GigE) adapters [1]. The Maximum Transfer Unit (MTU) is 1500 bytes. We emulate an end-system to end-system file transfer by transferring a 700MB file between the two machines using the 10GigE Ethernet cards.

Though our set-up does not explicitly incorporate a variable network-delay element to emulate different round-trip times (RTTs), our *RBUDP*$^+$ protocol will work with any delay inserted between the end-systems. To emulate LambdaGrids with different RTT's, the receiver simply delays the feedback by an amount equivalent to RTT/2, since RTT/2 is the time that will be taken by feedback to reach the sender on an actual LambdaGrid network. As the sender also knows the RTT, instead of continuing to send the data for time equal to time-slice t allotted to the RBUDP receive process on receiver side, it will send data for a time-period equal to $t - (RTT/2)$.

We transferred a file of size 700MB via the RBUDP transport protocol [6] and our proposed *RBUDP*$^+$ transport protocol, using the experimental setup described above. For both protocols, we measured the end-system to end-system transfer time for sending rates between 0.8-3.4 Gbps. We performed emulation studies under two scenarios: (1) the receiving end-system was kept under no additional computational load, and (2) the receiving end-system was loaded with a synthetic load. In order to emulate a computational load, we created a *synthetic benchmark* that is both CPU- and memory-intensive. We did not enable any offload engine support that is available on the Chelsio network interface cards.

6 Experimental Results

This section presents our experimental studies for the RBUDP protocol [6][12] and our proposed protocol $RBUDP^+$. The experiments were conducted for a file of size 700MB. We also vary the sending rates of the RBUDP/RBUDP$^+$ at no load between 0.8-3.4 Gbps. The experimental studies for the above settings are repeated for

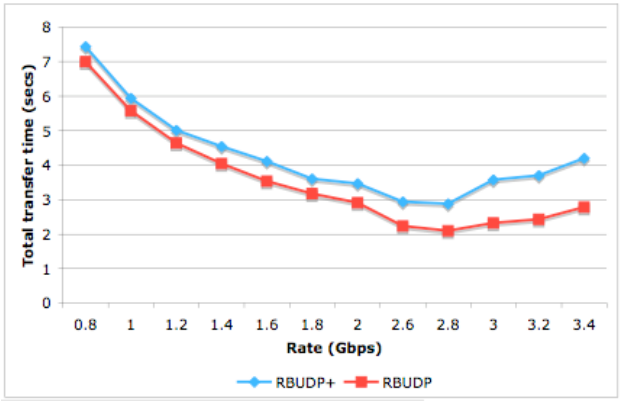

Figure 5. Comparison of data transfer times for $RBUDP^+$ and $RBUDP$ at no load conditions

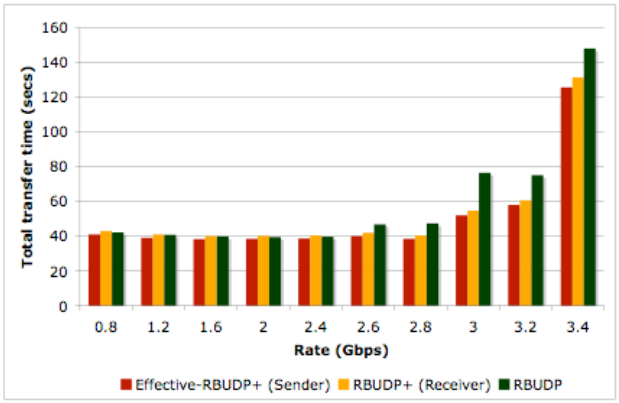

Figure 6. Comparison of data transfer times for $RBUDP^+$ and $RBUDP$ in loaded conditions

RBUDP and our proposed protocol $RBUDP^+$ in the presence of a synthetic load that is both CPU- and memory-intensive. The synthetic load can be varied to tune the load of the receiving end-system.

The experiments were repeated for varying synthetic load at the receiving end-system. Typically the RBUDP protocol should perform better than the $RBUDP^+$ protocol, under no load conditions, since the $RBUDP^+$ protocol interrupts the sending process from swamping data to the receiving end-system and it does that, based on a predictive estimate of how the tasks gets scheduled in the end-system.

As shown in Figure 5, the *RBUDP*$^+$ protocol performs slightly worse as compared to the traditional *RBUDP* protocol. This is because in the absence of any other load, the normal *RBUDP* scheme keeps pumping data from the sender to the receiver and the I/O-bound process never gets context-switched out. In comparison, the *RBUDP*$^+$ proto-

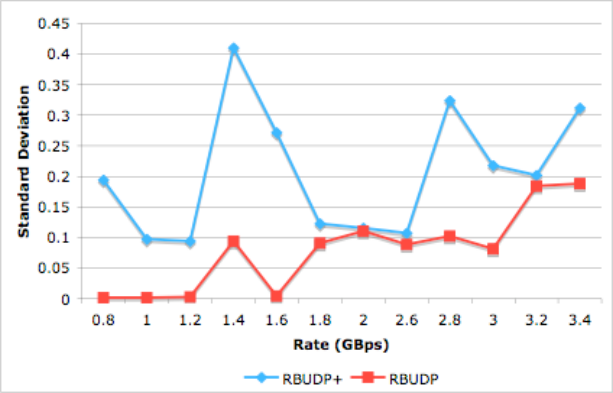

Figure 7. Comparison of standard deviation of the simulated data for *RBUDP*$^+$ and *RBUDP* at no-load conditions

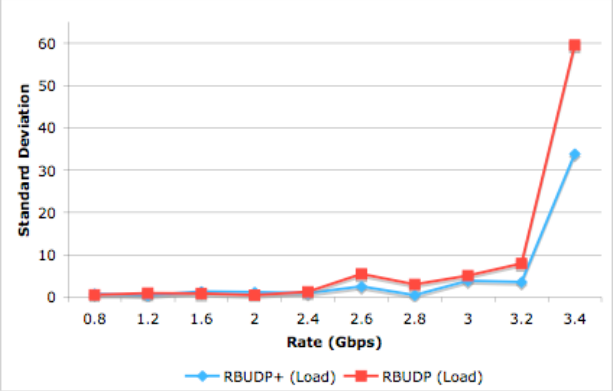

Figure 8. Comparison of standard deviation of the simulated data for *RBUDP*$^+$ and *RBUDP* in loaded conditions

col aggressively or proactively stops the sender from sending data at certain instances during its data transfer. This results in *RBUDP*$^+$ consuming slightly more time for the total data transfer as compared to the *RBUDP* scheme.

From Figure 5, we observe that the total data transfer time actually decreases steadily up to a transmission rate of 2.8 Gbps and then increases slowly for both the schemes.

Figure 6 shows the performance of the proposed *RBUDP*$^+$ protocol in comparison to the *RBUDP* protocol in the presence of the memory-intensive *synthetic benchmark*. It also shows the effective total transfer time of the *RBUDP*$^+$ protocol when we account for the time that is saved at the sender end (when we stop the sender from blasting data). As can be seen from Figure 6, there is a total time savings of the order of 3% during which other applications can be scheduled at the sending end. At the receiver end we see a savings of the order of 10-25% which can again used to schedule certain other compute intensive applications. It also shows how the proposed *RBUDP*$^+$ protocol outperforms the *RBUDP* protocol in the presence of computational loads at the receiving end. Hence we can infer that the *RBUDP* protocol only performs well in situations when the receiver is extremely lowly loaded. In the presence of other computational loads or memory-intensive applications, the I/O-bound process attached to the RBUDP transfer gets context-switched out more often, and hence suffers from the total time taken to transfer the data.

If we examine Figure 6 more closely we can see that the *RBUDP*$^+$ protocol without taking into account the time savings at the sender end, performs slightly worse than the standard *RBUDP* protocol at low sending rates. As the sending rates are increased beyond 2.4 Gbps, the *RBUDP*$^+$ protocol starts performing better than the *RBUDP* protocol. This might be because, at such high rates the receiver might get swamped with too much data and may not be able to handle all of it. The *RBUDP*$^+$ protocol prohibits such a scenario by proactively stopping the sender from sending any data by predictively estimating the time instances for which I/O-bound process will get context-switched out and also sending the feedback at the appropriate time.

Figure 7 and Figure 8 illustrate the standard deviation of the simulation results for both the *RBUDP*$^+$ and the *RBUDP* protocol under varying transmission rates, in no-load and loaded conditions. A standard deviation is a measure of how much spread variation there is in the data. As can be seen the standard deviation were much higher for the *RBUDP* transfers as compared to the *RBUDP*$^+$ protocol. Hence the *RBUDP*$^+$ protocol achieves more steady transfer time as compared to the *RBUDP* protocol.

Table 1 presents the 99% confidence intervals for all the emulation results for our proposed protocol *RBUDP*$^+$ and *RBUDP* under both loaded and no-loaded conditions. As can be observed, the confidence intervals are pretty close, except for *RBUDP* which produces fluctuating total transfer times at high transfer rates in the presence of system load.

Discussion

From the above simulation results, we can observe that the *RBUDP*$^+$ protocol outperforms the *RBUDP* protocol in the presence of high-computational and memory-intensive loads at the receiver end and also at high data transfer rates. This can be attributed to the proactive nature of *RBUDP*$^+$ to stop the sender from sending data during time-intervals when the I/O-bound is not scheduled at the receiving end.

We can hence derive some heuristic approach that fine tunes the selection of the traditional *RBUDP* approach and the proposed *RBUDP*$^+$ protocol, based on inputs on system load at the receiving end, and the bandwidth for transmission.

Table 1. 99% confidence intervals for all simulations

Rates	*RBUDP*$^+$	*RBUDP*	*RBUDP*$^+$-Load	*RBUDP-Load*
0.8Gbps	(7.14, 7.41)	(6.99, 6.995)	(41.83, 44.15)	(41.23, 42.90)
1.2Gbps	(4.87, 5.14)	(4.62, 4.63)	(40.34, 41.62)	(39.4, 42.14)
1.6Gbps	(3.71, 4.51)	(3.52, 3.53)	(38.14, 42.04)	(38.53, 41.06)
2.0Gbps	(3.29, 3.63)	(2.73, 3.06)	(38.60, 42.00)	(38.67, 40.15)
2.4Gbps	(2.77, 3.09)	(2.09, 2.35)	(39.08, 41.92)	(37.7, 41.5)
2.8Gbps	(2.39, 3.34)	(1.94, 2.24)	(39.75, 41.22)	(42.83, 51.75)
3.2Gbps	(3.39, 3.99)	(2.15, 2.7)	(55.41, 65.87)	(63.35, 86.73)
3.4Gbps	(3.73, 4.64)	(2.5, 3.05)	(81.46, 181.03)	(88.21, 207.46)

This might be one of the probable approaches to study in our future work.

As has been observed above the throughput of the *RBUDP*$^+$ protocol is significantly better than the *RBUDP* in the presence of heavy loads. This can be attributed to lower loss rates at the receiving end, thus preventing the sender from re-transmitting large amounts of lost packets. We observed some initial number on the loss rates at the receiver for both the protocols, and this confirms this assertion. Thus the *RBUDP*$^+$ protocol can be extremely efficient in multimedia transmission or real-video applications, where there may not be sufficient time to re-transmit the lost frames.

7 Conclusion

Recent advances in DWDM networks have fundamentally changed the communication requirements for future LambdaGrids, where there is sufficient network bandwidth but limited end-system capacity. This motivates our work of shifting the network transmission management from the network to the end-systems. We propose a receiving end-system feedback mechanism, which accounts for the dynamic prioritization of tasks and prevents the sender from swamping the receiver with more and more data. as well as a receiver-based rate allocation scheme

In this work, we demonstrated the impact of the end-system computational load on high-speed data transfer. To improve the performance of data transfer on a end-system under additional computational load, we have proposed a lightweight dynamic task priority monitoring protocol, named *RBUDP*$^+$.

We performed emulation studies and demonstrated the effectiveness of the protocol in terms of low data transfer time and also significantly lower loss rates at the receiver. Hence the proposed protocol can be applicable to high-speed data transfer over LambdaGrids for various high-performance network applications.

Acknowledgments

We express our sincere gratitude to Venkatram Vishwanath, who was an intern at LANL, and the Electronic Visualization Group at UIC, for providing the Quanta-RBUDP code for our simulation studies.

References

[1] Chelsio t210 10 gigabit ethernet adapter. *http://www.chelsio.com/products/T210.htm*, 2005.

[2] DoE ultrascience net. *http://www.csm.ornl.gov/ultranet*, 2005.

[3] The National Lambda Rail. *http://www.nlr.net*, 2005.

[4] L. Brakmo and L. Peterson. Tcp vegas:end to end congestion avoidance on a global internet. *IEEE Journal of Selected Areas in Communications*, 13(8):1465–1480, 2003.

[5] CANARIE. Canarie network. *http://www.canarie.ca*, 2005.

[6] E. He, J. Leigh, O. Yu, and T. A. DeFanti. Reliable blast udp:predictable high performance bulk data transfer. *Proceedings IEEE Cluster Computing, Chicago, Illinois*, 2002.

[7] V. Jacobson. Congestion avoidance and control. *Computer Communication Review*, 18(4), August 1998.

[8] M. Mathis, J. Mahdavi, S. Flyod, and A. Romanow. Tcp selective acknowledgement options. *RFC2018, Internet Engineering Task Force (IETF)*, October 1996.

[9] T. L. . Scheduler. *http://josh.trancesoftware.com/linux/linux_cpu_scheduler.pdf*, 2005.

[10] L. Smarr, A. Chien, T. DeFanti, J. Leigh, and P. Papadopoulos. The optiputer. *Communications of the Association for Computing Machinery*, 47(11), 2004.

[11] R. Wu and A. Chien. Gtp:group transfer protocol for lambda grids. *Proceedings of CCGrid, Chicago, Illinois*, 2004.

[12] C. Xiong and et.al. Lambdastream-a data transport protocol for streaming network intensive applications over photonic networks. In *Proceedings of PFLDNet, Lyon, France*, January 2005.

[13] X. Zheng, A. P. Mudambi, and M. Veeraraghavan. Frtp: Fixed rate transport protocol – a modified version of sabul for end-to-end circuits. *Proceedings IEEE BroadNets Workshop*, 2004.