RAPID: An End-System Aware Protocol for Intelligent Data Transfer over Lambda Grids

Amitabha Banerjee[1], Wu-chun Feng[2], Biswanath Mukherjee[1], and Dipak Ghosal[1]

[1]University of California Davis
Dept. of Computer Science
Davis, CA 95616, USA
abanerjee@ucdavis.edu
{mukherje, ghosal}@cs.ucdavis.edu

[2]Virginia Tech
Dept. of Computer Science
Blacksburg VA 24061, USA
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. To support such applications, lambda grid networks have been built to provide large, on-demand bandwidth between end-points that are interconnected via optical circuit-switched lambdas. It is extremely important to develop an efficient transport protocol over such high-capacity, dedicated circuits.

Because lambdas provide dedicated bandwidth between endpoints, they obviate the need for network congestion control. Consequently, past research has demonstrated that rate-based transport protocols, such as RBUDP, are more effective than TCP in transferring data over lambdas. However, while lambdas eliminate congestion in the network, they ultimately push the congestion to the endpoints — congestion that current rate-based transport protocols are ill-suited to handle. In this paper we introduce a "Rate-Adaptive Protocol for Intelligent Delivery (RAPID)" of data that is lightweight and end-system performance-aware, so as to maximize end-to-end throughput while minimizing packet loss. Based on self monitoring of the dynamic task-priority at the receiving end-system, our protocol enables the receiver to proactively deliver feedback to the sender, so that the sender may adapt its sending rate to avoid congestion at the receiving end-system. This avoids large bursts of packet losses typically observed in current rate-based transport protocols. Over a 10-Gigabit link emulation of an optical circuit, RAPID reduces file-transfer time, and hence improves end-to-end throughput by as much as 25%.

1. Introduction

Many e-Science applications, particularly in large-scale scientific computing, must transfer large volumes of data at very high data rates. For example, data may be aggregated from distributed information repositories that are physically located in different parts of the world, for subsequent analysis at a computing center. Alternatively, results generated at a supercomputer may be transferred to remote client sites for visualization. In each of these cases, the volume of data may be hundreds of terabytes, and in some cases, petabytes, transferred at very high speeds (e.g., OC-192: 10 Gbps) and oftentimes across long distances (e.g., intra- as well as inter-continental). OptIPuter [13] is one example of a project that aims to couple computational resources over lambda grid networks.

In order to connect distributed resources in a unified manner, middleware researchers have developed the concept of a Grid, where each link in a grid is a high-capacity bandwidth pipe. A Lambda Grid is a specific incarnation of a Grid, which offers dedicated, optical, circuit-switched, point-to-point connections (lambdas), which may be reserved exclusively for an application. Such dedicated connections avoid the problems of network congestion at intermediate routers. Examples of networks that enable Lambda Grids include National LambdaRail (NLR) [2] and DOE's UltraScience Net [4]

in the United States, CANARIE's CA*net in Canada [1], and NetherLight in the Netherlands [3].

The problem of how to transport data through such "fat" dedicated pipes is a challenging problem for grid researchers. The Transmission Control Protocol (TCP) is used in traditional packet-switched networks for adjusting the sending rate, depending on the state of network congestion. In a lambda grid, the network itself does not experience congestion, given that dedicated bandwidth is provided per connection. Optical circuits in lambda grids span large intra-continental or intercontinental distances, thus having a large bandwidth-delay product (BDP). TCP and its variants are not very efficient in such networks because of their aggressiveness in rate and congestion control. GridFTP[8] sets up multiple TCP connections between the source and the destination so as to achieve higher aggregate bandwidth relative to that achieved by a single TCP stream.

An alternate approach is a transport protocol based on the User Datagram Protocol (UDP) such as Reliable Blast UDP (RBUDP) [10]. In RBUDP, the sender transmits UDP packets at a fixed rate, specified by the user. After all the data has been transmitted, the receiver sends the error sequence numbers corresponding to the data packets it did not receive (due to network or end-system congestion) to the sender via a TCP connection. The sender then 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 of packet delivery is imposed on top of the unreliable connectionless UDP.

Although RBUDP has been demonstrated to perform well, its chief weakness is its inability to adapt the sending rate. This leads to unwanted packet losses, particularly when the receiving end-system is congested. Though dedicated optical connections as in lambda grids avoid network congestion at intermediate routers, the network throughput in such connections (namely 10 Gbps for an OC-192 connection) often exceed the capabilities of data processing at the end-system, as has been noted by researchers [14, 15]. In addition to receiving the data, the receiving end-system is expected to be running othercomputationally intensive processes such as visualization and analysis of the received data [15]. 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. Since the Network Interface Card (NIC) buffer size is usually bounded, packets may get dropped due to buffer overflow if the receiving data process is not scheduled by the operating system at the appropriate times to transfer the packets from the line card buffer to physical memory. When data is transmitted to such an end-system at a fixed rate relentlessly as RBUDP does, it only exacerbates the problem of end-system congestion. Thus, one of the major challenges of high-performance distributed computing is how to handle the network and computational processes simultaneously.

LambdaStream [15], the successor to RBUDP, adapts the sending rate to dynamic network conditions by monitoring the inter-packet receiving time interval at the end-system, and sending a feedback to the sender to adapt the sending rate accordingly. This is a clever scheme because it avoids polluting the network with probing packets.

In this work, we propose the idea of a lightweight end-system performance-monitoring-based protocol to improve the performance of data transport on a lambda grid. We emphasize on probing various end-system performance metrics such as the dynamic priority of various tasks at the receiving end-system, so that potential end-system congestion may be detected early, and an appropriate feedback may be sent to the sending end-system to take evasive action to avoid packet losses. One example of such an evasive action is to suspend transmissions at times when the receiving end-system is congested. Appropriately integrating the above features results in a prototype protocol that we call RAPID, short for Rate-Adaptive Protocol for Intelligent Delivery. In our experiments over a 10-Gigabit Ethernet network emulation of an optical circuit in a lambda grid, RAPID reduces file-transfer time, and hence, improves end-to-end throughput by as much as 25%.

The remainder of the paper is organized as follows. In Section 2, we provide an insight to the end-system behaviour when it is subjected to computational and network loads simultaneously. In Section 3, we describe our protocol. Section 4 discusses the results on a 10-Gigabit Ethernet network emulation. Section 5 presents our conclusions.

2. Analysis of the Receiving End-system

In order to emulate a lambda grid circuit, we deployed the following experimental setup. We connected two machines (configurations used are shown in Table 1) back-to-back with Chelsio 10-Gigabit Ethernet adapters [6]. The Iperf [7] measurements report the maximum bandwidth achieved by a TCP connection between two machines of the same configuration connected back-to-back with 10 Gigabit Ethernet adapters. The Maximum Transfer Unit (MTU) was

Table 1. System Configuration.

	Configuration I	Configuration II
Processor	Intel Pentium 4	AMD Opteron
Processor Speed	2.8 GHz	2.2 GHz
Cache size	512 KB	1024 KB
RAM	2 GB DDR RAM	1 GB DDR RAM
Iperf measurement	1.92 Gbps	3.65 Gbps

Table 2. Optimal sending data rate for RBUDP.

Computational Load	Sending rate for best transfer time
No load	1.6 Gbps
Single infinite loop	800 Mbps
Two infinite loops	650 Mbps
Three infinite loops	550 Mbps
Iso-surface extraction	800 Mbps

1500 bytes. TCP offloading was not enabled on the Chelsio adapters. Since hard-drive speeds are substantially slower to support a transfer rate of 10 Gbps, we emulated an end-system to end-system file transfer by transferring a file between two RAMdisks (which use the RAM for storing data). RAMDisks allow for RAM access times, and because the Linux OS considers the RAMDisk as a separate disk, the OS functionalities of copying data from memory to a disk are not by-passed in our experimental setup.

In order to analyze the end-system performance, we used MAGNET (Monitoring Apparatus for General kerNel-Event Tracing) [9]. MAGNET is a low-overhead tool that provides fine-grained monitoring of kernel- and user-space events by allowing any event to be monitored, and by time stamping each event with the CPU-cycle counter which is the highest-resolution time source available on most machines. Optionally, additional information can be exported to give a more detailed look at kernel operation.

We used MAGNET to monitor the context-switch times between the different processes at the receiving end-system. We transferred a 500-MB file via the RBUDP transport protocol, using the experimental setup described above. For this experimental setup, we used the Intel Pentium 4 based machines (Configuration I). For the RBUDP protocol, we measured that the end-system to end-system transfer time was the fastest when the sending rate was 1.6 Gpbs. The receiving end-system was under no additional computational load. We did not enable the TCP offload engine support on the Chelsio adapters. It is important to note that the above line rate is for a RAMDisk-to-RAMDisk transfer. A memory-to-memory transfer may be supported at a much higher line rate, as may be noted from the higher Iperf numbers in Table 1.

In order to emulate a computational load on the receiving end-system, we used two different processes. One process is an infinite "for" loop, which is expected to have constant computational overhead at all times. We hereafter refer to this as a *synthetic workload*. The other is an isosurface extraction visualization process of a knee joint image, which is expected to impose varying computational and I/O load, depending on the stage of computation of the process. We hereafter refer to this as a *visualization workload*.

When the receiving end-system is under some computational load, we observed that packets are dropped if the sending rate of 1.6 Gbps is maintained. The losses are lower if the sending rate is reduced. Table 2 shows the sending rate for which the fastest transfer time was measured, when the receiving end-system is under different computational loads. As may be inferred, as the computational load increases, the sending rate must be decreased to lower packet losses.

We shall now report our observations from the MAGNET traces. Figure 1(a) shows the duration of the time slices that the OS allocates to the synthetic workload, while the end-system is also receiving data. For simplicity, we show results for only the time duration of the first UDP blast (iteration of transmitting UDP packets) of RBUDP. We observe that a majority of the time slices allocated by the OS to the synthetic workload is of duration 10 ms. Most of the packets arriving at the receiving end-system in such a time interval when the synthetic workload is running would be lost, because the NIC does not support enough buffer to store the arriving packets. At a line rate of 1.6 Gbps, 2 MBytes of data would be lost in a duration of 10 ms, if not handled by the end-system.

We define the time intervals when no packet losses occur at the receiving end-system as a *lossless time interval*. We measure the duration of the lossless time interval by comparing losses reported by RBUDP with the MAGNET traces. Figure 1(b) shows the duration of the *lossless time intervals*. We observe that most of the *lossless time intervals* are of duration of 10ms. The very first *lossless time interval* is of duration 125 ms.

We explain the results as follows. Most OS schedulers differentiate between a I/O-bound process and a CPU-bound process. One of the goals of the OS scheduler is to improve the interactivity and response time of the system. To achieve this, the OS tries to favour
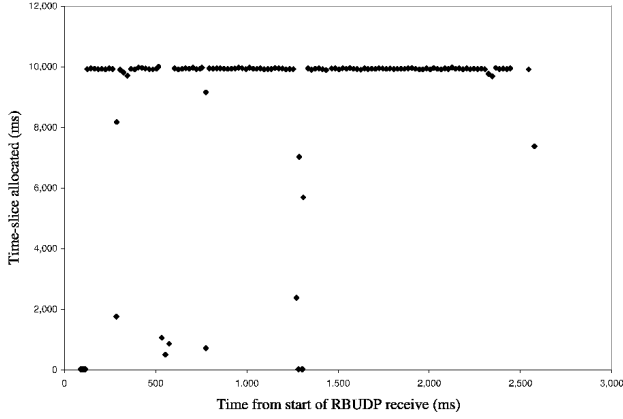
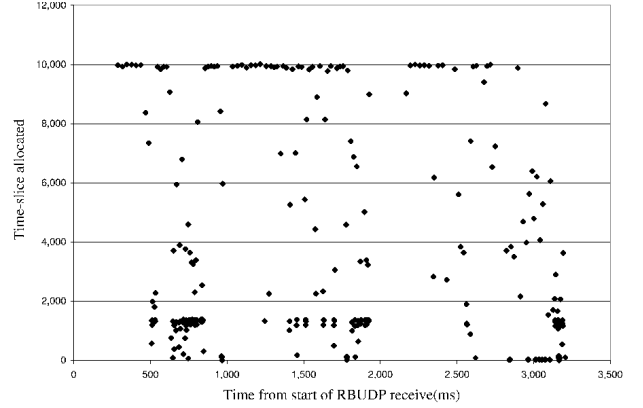(a) Time slice allocated by the OS
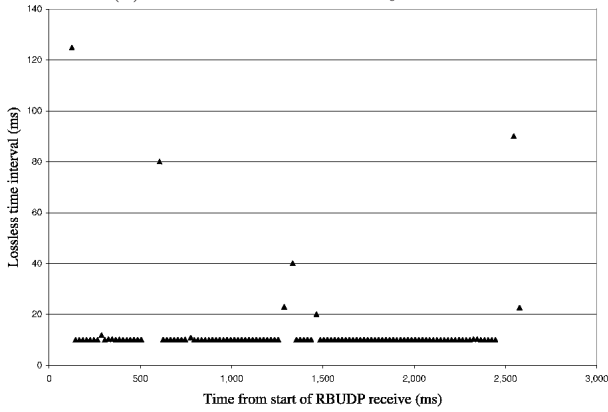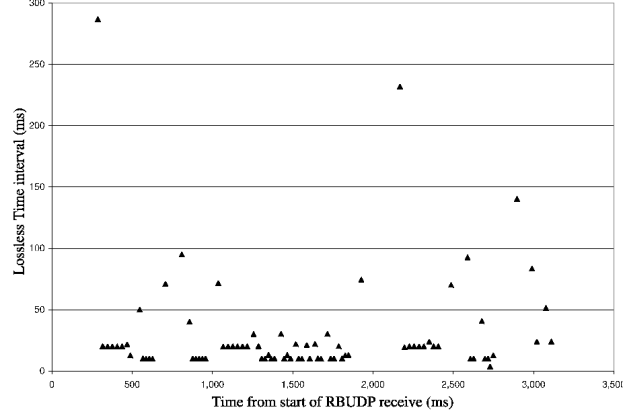

(b) *Lossless time interval* duration

Figure 1. Context-switch time analysis for the *synthetic* **workload.**

(a) Time slice allocated by the OS

(b) *Lossless time interval* duration

Figure 2. Context-switch time analysis for the *visualization* **workload.**

the I/O process while scheduling. Different OSs have different means of classifying I/O and CPU processes, and favoring the I/O-bound process. For example, the Linux 2.6 scheduler classifies between the above two, based on the average sleep time of a process, which is updated every time the process is context switched [5]. Moreover, a dynamic bonus proportional to the average sleep time is awarded to the task prority. I/O-bound processes which have higher average sleep time than CPU-bound processes are thus favored.

In the above example, when *RBUDP receive* starts, it is classified as an I/O process. Interrupts are frequently generated with the incoming packets arriving at 1.6 Gbps. While the OS is handling these interrupts, the CPU-bound synthetic workload is not handled. After some time (125 ms in this case), the OS begins to treat *RBUDP receive* as a CPU-bound process as it has run continously. Thereafter, both processes are handled equally. Comparing Figures 1(a) and 1(b), it is evident that after the first 125ms, the OS context switches between receiving data, and the synthetic workload every 10 ms.

In the above results, most time-slices are equal because the computational load is constant throughout. Figure 2 shows the equivalent results for the visualization workload. From Figure 2(a), we observe that the time slices allocated by the OS to the visualization workload are concentrated to be either less than 2 ms or approximately 10 ms. From Figure 2(b), the first lossless time interval is of duration 280 ms. Thereafter, it varies between 10 ms and 240 ms. Thus, for a general application whose computational load may vary between stages of computation and I/O, it is not trivial to predict the time slices allocated by the OS to the different processes, and the *lossless time interval* durations.

It is clear that the network performance may be substantially improved, if packet losses were avoided when the receiving end-system OS is handling an alternate process. We considered the following possible solutions:

1. A Real-Time OS (RTOS) may be employed. A Real-Time OS allows one to specify hard deadlines for tasks. The *RBUDP receive* process may be classified as a real-time task with periodicity specified to handle the packets at the incoming data rate. However, a RTOS is expensive to maintain, and may not be suitable for all applications. Device driver and hardware support is not commonplace for a RTOS. For example, we did not find it easy to build support for the Chelsio 10-Gigabit adapters on Real Time Linux OS.

2. The buffer capacity in the NIC may be increased, so that packets are not dropped at the NIC when the OS is not ready to handle them. But this is an expensive hardware solution. It is not common to have a NIC with a large capacity, and NIC manufacturers may not be willing to provide it. The Chelsio 10-Gigabit line cards that we used had 256 MB RAM, but that is shared between transmitting and receiving queues, as well as for other tasks.

3. Various parameters of the OS scheduler such as maximum allocated time slice, maximum dynamic bonus priority granted to an I/O process, etc., may be adjusted, so as to favor the *RBUDP receive* process, and thus reduce packet losses. We studied the effect of altering the maximum dynamic bonus priority granted to an I/O process by Linux 2.6. The file-transfer times improved significantly. We do not report the results here due to space constraints. We believe that this is not a good solution, as it would lead to custom kernel builds highly tuned to the application. Scientists dealing with high-performance computing or visualization applications would rather not deal with custom kernel configurations or kernel patches designed specifically to tune the network performance.

4. The times at when the *lossless time interval* ends, and when the OS allocates a much larger time slice to an alternate process, may be predicted in advance, so that timely feedback may be sent to the sending end-system to suspend or slow down the sending rate. We may thus achieve an efficient interleaving between the CPU-bound and I/O-bound process.

In this paper, we concentrate on solution (4), because it appears to be the most acceptable and generic out of all the solutions. We aim to build an end-system performance-monitoring tool, which may deliver the appropriate feedback to the sending end-system, so that the packet-transfer throughput may be improved. In the next section, we present our end-system performance aware rate adaptive protocol.

3. End-System Performance-Aware Rate-Adaptive Protocol

Our objective is to monitor the receiving end-system's performance, so as to identify durations of end-system congestion. We aim to predict the time at which the receiving end-system's OS may allocate a large time slice to an alternate process, as a result of which it may not handle interrupts and packets from the NIC. If the sending end-system does not transmit during such time durations, then packets will not be lost at the receiver end-system, thereby improving the data-transfer performance. For the purpose of monitoring, we considered the following system metrics: average CPU load, NIC card buffer occupancy, and task interactivity. Out of the above, we found task interactivity to be the most helpful indicator, because this may be easily monitored by inserting a simple probe in the kernel code as we illustrate below, and also because this knowledge may be easily applied in a prediction scheme.

As discussed earlier, most OSs distinguish between an interactive (I/O-bound) process and a CPU-bound process in the system in order to improve the system reponsiveness. We investigated the approaches taken by various OS schedulers to achieve the above. We present a summary for two OSs: Linux 2.6, and Free BSD 5.0.

3.1. Linux 2.6 Scheduler

The Linux 2.6 scheduler maintains a dynamic priority for all processes, which is computed as the static priority (related to task niceness) plus a dynamic bonus which is granted based on the task interactivity, as measured by the interactivity heuristics. In the Linux 2.6 scheduler [5], the static priority for user processes ranges from 0 to 40 (corresponding to niceness values between -20 and +20). A lower number corresponds to higher priority. The dynamic bonus is granted by the kernel to boost the priority of interactive tasks, and it ranges from -5 to 5. This is computed proportional to the average sleep time of the task. The value of the average sleep time for each task is updated whenever it wakes out of sleep or when it gives up the CPU voluntarily or involuntarily.

The scheduler allocates time slices to the task by scaling its static priority value into the possible time slice range. Once a time slice is awarded, the task is placed on an active priority array. While choosing tasks for execution from this array, the OS chooses the one with the highest dynamic priority. After the task has exhausted its time slice, it is awarded a new time slice but is placed on the expired priority array. However, those tasks which are deemed to be interactive are placed back on the active priority array even after a new time slice is generated. This step greatly improves the performance of interactive tasks in the Linux system. The active and expired priority arrays are switched only after the active priority array is empty.

3.2. Free BSD 5.0 Scheduler

The ULE scheduler in Free BSD 5.0 [12] has many commonalities with the Linux 2.6 scheduler. Each task is granted a time slice, and is assigned to the *current queue* or the *next queue*. Once all tasks in the current queue are executed for the duration of the allocated time slices, the next and current queues are switched. Interactive tasks are always inserted into the current queue. The interactivity score of a task is calculated using its voluntary sleep time and run time. The interactivity score may be computed as follows:

$$m = \frac{MaximumInteractiveScore}{2} \tag{1}$$

$$if \quad (sleep > run) \quad score = \frac{m}{\frac{sleeptime}{runtime}} \tag{2}$$

$$else \quad score = \frac{m}{\frac{runtime}{sleeptime}} + m$$

Tasks with interactivity score below a certain threshold are marked interactive. Since interactive tasks are always inserted into the current queue, they have a faster response time.

The end-system performance aware Rate-Adaptive Protocol for Intelligent Data-transfer (RAPID) is shown in Figure 3. We use RBUDP as an example of a rate-based transport protocol. Our objective is to monitor whether the *RBUDP receive* process is classified as being interactive or not by the OS scheduler. For the Linux 2.6 scheduler, probing the *dynamic task priority* would indicate the degree of interactiveness, while for the Free BSD 5.0 ULE scheduler, probing the *interactive score* would be a good measure.

The end-system <u>P</u>erformance <u>M</u>onitoring <u>P</u>rocess (PMP) monitors the interactiveness of *RBUDP receive* at fixed polling intervals. In addition, whenever
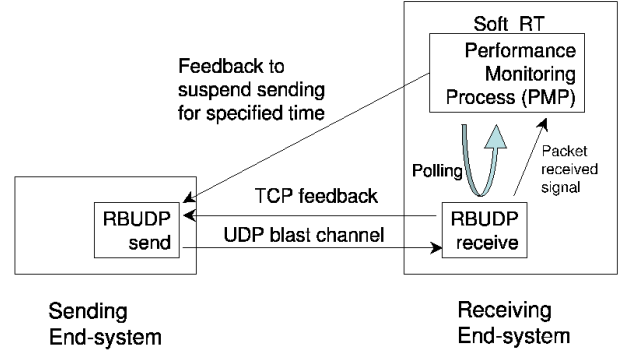
Figure 3. Design of our end-system performance aware Rate-Adpative protocol for Intellingent Data-transfer (RAPID).

RBUDP receive receives each packet, it signals to the PMP by setting a bit. When PMP does not receive any signals from *RBUDP receive*, it interpretes this as packets being dropped. The first time this happens, the PMP monitors the interactiveness of *RBUDP receive*. Subsequently, it tries to predict when *RBUDP receive* may reach such a state. At the predicted time, a feedback is sent from the PMP to *RBUDP send* to suspend sending for some duration of time, which we define as the *suspend interval*. The motivation is to allow *RBUDP receive* to be idle for the corresponding time, during which the OS may schedule the alternate (CPU-bound) process. Thus, the CPU-bound process is not starved, and when *RBUDP send* resumes, it regains its interactive status. Via this mechanism, we achieve an efficient interleaving between the two processes at the end-system. The *suspend interval* may be OS specific, and we demonstate in the next section how it be calculated for Linux 2.6.

We emphasize the following two important design criterion:

1. The end-system (PMP) must be scheduled as a Soft Real-Time (Soft RT) process so that it is scheduled by the OS at the polling times, and may thereby monitor the task parameters at fixedtime periods. This is important, else the monitoring process might itself not be scheduled.

2. The end-system PMP must be a very light-load process, particularly because it is scheduled soft real-time, so that it does not impose any additional CPU load. Hence, we are not using a heavier weight tool such as MAGNET for monitoring the task priorities. Our emphasis is on a lightweight design.

3.3. Implementation Specifics For Linux 2.6

In Figure 4, we describe our implementation specific to the Linux 2.6 scheduler. The *suspend interval* is calculated based on the current dynamic task priority and average sleep value for *RBUDP receive*, both of which are monitored. The priority at which packets are not handled by *RBUDP receive* is noted, which is in the variable $ERROR\_PRIORITY$. Thereafter, whenever the monitored dynamic task priority reaches one level higher than $ERROR\_PRIORITY$, a feedback signal is generated to request *RBUDP send* to suspend for the specified time interval. The dynamic task priority changes in increments of 1, and we observed that sending a feedback at a priority difference of one is sufficient. The *suspend interval* calculated in Step 2 (d) of the algorithm is Linux 2.6 specific, and is related to the algorithm by which Linux 2.6 computes dynamic task priorities. The idea is to have *RBUDP receive* idle long enough so that the OS assigns it the same dynamic priority at which it started thereafter.

The Linux OS does not provide a system call to monitor the task's dynamic priority from a user-level process; it only provides a system call to monitor the static priority (niceness). Hence, we designed a simple kernel patch to allow probing of the dynamic priority from a user-level process. We emphasize that we do not alter any properties of the kernel scheduler, and are not attempting to customize the kernel code in any way.

Since the sender suspends transmitting for brief intervals of time, this may increase the jitter of data made available to the application at the receiving end. This may be a concern for some visualization applications which are jitter sensitive. We intend to address this matter in future work.

3.4. Effect of Round-Trip Time (RTT)

Lambda Grids may span across large geographical distances, and hence optical circuits on Lambda Grids may have large RTTs. Thus, in the above protocol, any feedback sent by the task monitoring process is delayed by $RTT/2$ before it reaches the sender. Even if the sender suspends immediately, the receiving end-system receives packets in the pipeline for another $RTT/2$. Thus, it takes at least RTT amount of time after the feedback is transmitted for *RBUDP receive* to become idle.

Hence, any feedback that is sent from the receiving end-system must be sent at least RTT before the anticipated time of packets getting dropped. In the approach for the Linux 2.6 OS, we attempt to keep

---

Linux 2.6 Task Priority Monitoring Process (TPMP) Algorithm

1. Initialize
   $ERROR\_PRIO = -\infty$,
   $MAX\_PRIO = -\infty$,
   $MAX\_AVG\_SLEEP = -\infty$.

2. Begin polling loop:

   (a) Sleep for $POLLING\_TIME$.

   (b) Monitor current dynamic priority ($CURR\_DYN\_PRIO$) of the receive process. If it is greater than MAX_PRIO, set MAX_PRIO to $CURR\_DYN\_VALUE$. Set MAX_AVG_SLEEP value to the current value of average sleep value.

   (c) Monitor packet received signals from process. If no packet is received in the last polling interval, set $ERROR\_PRIO$ to the current task priority. Go to (e) to compute suspend interval, and send feedback.

   (d) If $CURR\_DYN\_PRIO - 1 \leq ERROR\_PRIO$, go to Step (e) to compute the *suspend interval* and send the feedback. Else go back to Step (a).

   (e) Monitor the current average sleep value, $CURR\_AVG\_SLEEP$ value. Calculate:

   $$SuspendInterval = MAX\_AVG\_SLEEP$$
 $$- \frac{CURR\_AVG\_SLEEP}{MAX\_PRIO - CURR\_DYN\_PRIO}$$

   Send feedback with suspend period interval.

Figure 4. Implementation of Task Priority Monitoring Process for the Linux 2.6 Scheduler.

---

the implementation simple and efficient, by transmitting the feedback one dynamic priority level before $ERROR\_PRIO$. We measured the interval between which the process shifts down by one priority level, and found it to be approximately 100 ms. Thus, for an RTT up to 100 ms, the above mechanism would be appropraite. For RTTs greater than 100 ms, possible solutions are to measure the times of change in priorities, and calculate the feedback sending time to be equal to RTT subtracted from the expected time at which the task priority reaches $ERROR\_PRIO$.
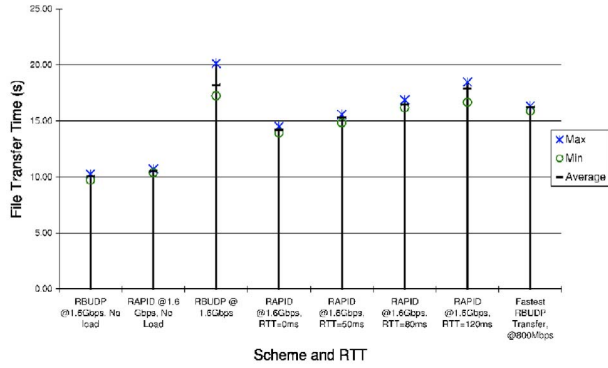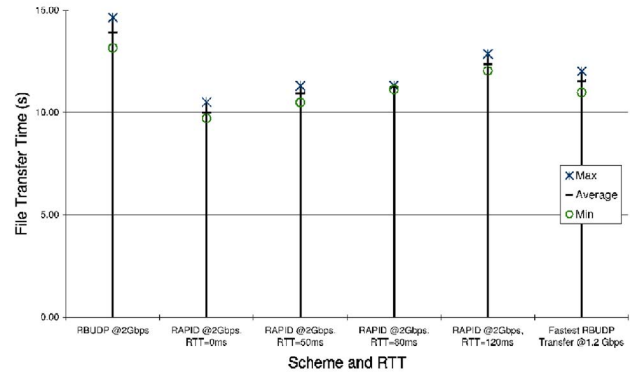
(a) MTU=1500 bytes
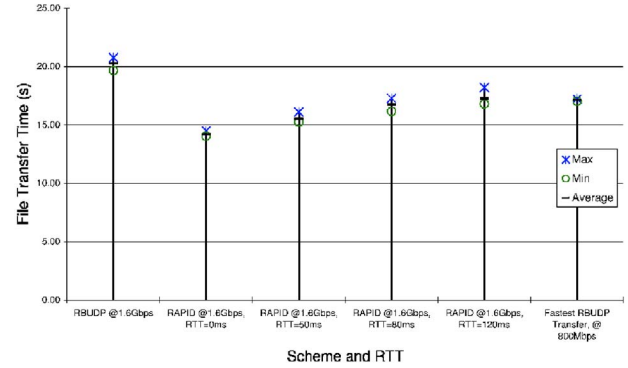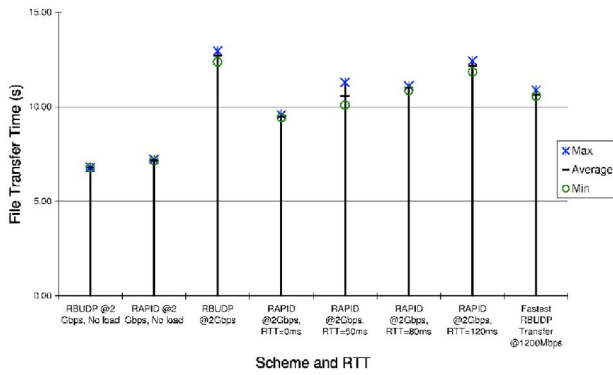
(b) MTU=9000 bytes

Figure 5. File-Transfer Time for synthetic workload for Configuration I (Pentium 4).

(a) MTU=1500 bytes

(b) MTU=9000 bytes

Figure 6. File-Transfer Time for the visualization workload for Configuration I (Pentium 4).

4. Results on a 10-Gbps Ethernet based network

Since we did not have direct access to a live Lambda Grid, we emulated a dedicated end-to-end optical circuit connection by connecting two machines back-to-back using the Chelsio 10-Gigabit Ethernet Adapter [6]. The OS on both machines was the Debian installation of Linux OS version 2.6.12. In order to visualize the effects of RTT, we delay the feedback signal from the receiving end-system by the corresponding RTT amount. We report results for two different Maximum Transfer Unit (MTU) values: 1500 bytes, and 9000 bytes. Increasing the MTU reduces the number of interrupts that will be processed at the end-system, and therefore improves the data transfer time.

Figures 5(a) and (b) show the file-transfer time for a 1.5-GB file transfer between two end-systems with Configuration I (Pentium 4) (shown in Table 1). We first report the results for the two schemes - RBUDP and RAPID with no additional load on the end-system. Thereafter, the receiving end-system is under the additional computation load of handling a synthetic workload. We show results for RAPID, using different RTTs. The RTT value of 50 ms corresponds to the typical coast-to-coast RTT in the United States. The RTT value of 120 ms would correspond to an intercontinental link. The final measurement reported is for the fastest RBUDP transfer at zero RTT that we observed by tuning the sending rate. The transfer time using different RTTs for RBUDP doesn't vary much. We use this result to compare RAPID with the best possible transfer time that RBUDP can give.

It must be noted that, at no load, the sending rate of 1.6 Gbps leads to the fastest transfer time using RBUDP at MTU of 1500 bytes. If the MTU is increased to 9000 bytes, the corresponding sending rate
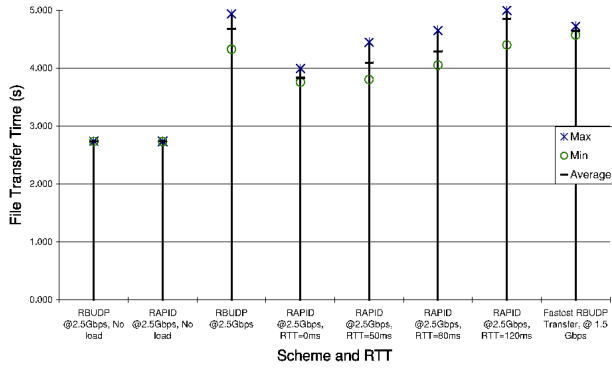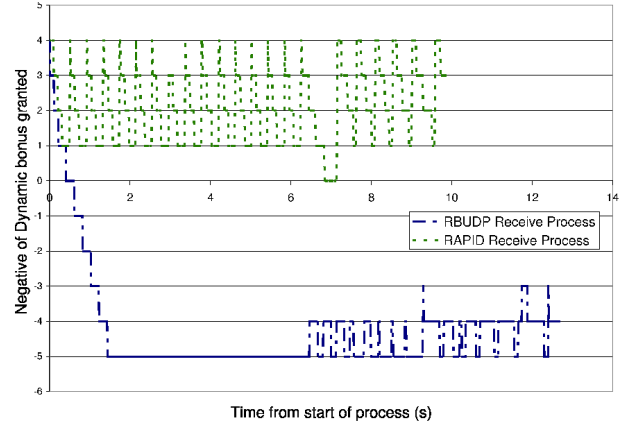
(a) MTU=1500 bytes
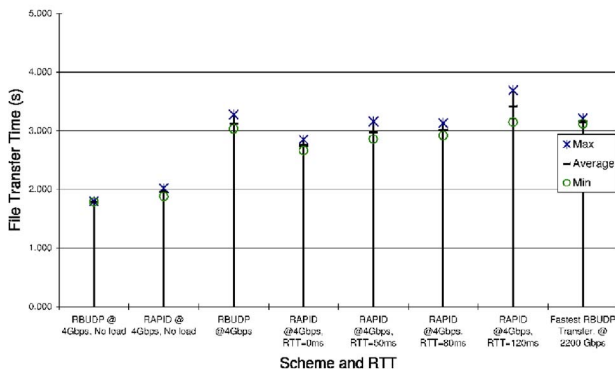
(b) MTU=9000 bytes

Figure 7. File-Transfer Time for the synthetic workload for Configuration II (AMD Opteron).

Figure 8. Dynamic bonus granted by OS to RBUDP receive and RAPID receive

would be 2 Gbps. When the synthetic workload is introduced, RBUDP takes substantially longer time for the file transfer, if the sending rate were maintained the same. Using RAPID, the file-transfer time is reduced by as much as 25% for RTT=0. The file-transfer time using RAPID increases marginally with higher RTT. This is because, with increase in RTT, the feedback signal takes a much longer time to reach the sender. If we compare with the fastest RBUDP transfer achieved by tuning the sending rate, RAPID compares well until an RTT of 80 ms. One may consider that tuning the sending rate for varying application loads would not be a very practical solution, and thus the feedback-based RAPID would be a favorable protocol.

If we compare RBUDP with RAPID at no load, RAPID leads to slightly higher transfer time. This is due to the additional overhead of end-system performance monitoring via a soft real-time process. One of our goals has been to design a low-overhead process.

Figure 6 shows the corresponding results for the visualization workload. The results are similar as before. Our scheme performs well with a synthetic process with constant CPU load, as well as a visualization application with varying CPU load.

We demonstrate the performance on a different architecture by running the same experiment on end-systems with Configuration II (AMD Opteron), shown in Table 1. A 800-MB file is transferred between the two end-systems. At no load, the fastest transfer time was observed at the sending rate of 2.5 Gbps when the MTU is 1500 bytes. The corresponding sending rate for MTU of 9000 bytes is 4 Gbps. Figure 7 shows the results.

Since our protocol is based on dynamic task priority monitoring and feedback, it is important to analyze how the dynamic bonus granted by the Linux 2.6 scheduler changes in the two protocols. Figure 8 compares the dynamic bonus granted to RBUDP and RAPID. The experimental setting uses Configuration I (Pentium 4) with the MTU of 9000 bytes, and the synthetic workload. As mentioned earlier, the dynamic bonus granted by Linux 2.6 scheduler ranges from -5 to +5. A lower number represents higher priority. In Figure 8, we show the negative of the numbers, for a better illustration. With RBUDP, the priority steadily decreases. In RAPID, the dynamic priority at which errors occur is recorded, and a feedback is sent whenever the priority reaches one level higher than $ERROR\_PRIO$. When the sender received the feedback, it suspends sending. Since the receiving task is then idle, the OS handles the alternate (CPU-bound) process. On resumption, the receiving task is granted a much higher

priority, because it has been idle for some time. The dynamic priority of the receiving task does not drop below a certain level. Thus, whenever the receiving task is running, it is handled as an I/O process by the OS, and interrupts are handled regularly. Packet losses are thus avoided to a large extent.

5. Conclusion

In this work, we demonstrated the impact of the end-system computational load on high-speed data transfers. To improve the performance of data transfer when the end-system is under additional computational load such as a visualization process, we have proposed a lightweight end-system performance monitoring based transport protocol – RAPID. By performing experiments on an emulated 10-Gigabit link experimental setup, we demonstrated that using RAPID leads to higher throughput by lowering packet losses than another rate-based transport protocol, RBUDP.

6. Acknowledgements

References

[1] CANARIE at http://www.canarie.ca/about/index.html

[2] National LambdaRail at http://www.nlr.net

[3] NetherLight at http://www.netherlight.net/info/home.jsp

[4] DoE UltraScience Net at http://www.csm.ornl.gov/ultranet/

[5] The Linux 2.6 scheduler, by Jos Aas, SGI Inc., at http://josh.trancesoftware.com/linux/linux_cpu_scheduler.pdf

[6] Chelsio T210 10 Gigabit Ethernet Adapter, at http://www.chelsio.com/products/T210.htm

[7] IPerf 2.0.2 - The TCP/UDP Bandwidth Management Tool at http://dast.nlanr.net/Projects/Iperf/

[8] W. Allcock *et al.*, "The Globus Striped GridFTP Framework and Server," *Proc., ACM Supercomputing 2005*, Seattle, Washington, November 2005.

[9] M. Gardner, W. Feng, M. Broxton, A Engelhart, and G. Hurwitz, "MAGNET: A Tool for Debugging, Analysis and Adaptation in Computing Systems," *Proc., CC-Grid 2003*, Tokyo, Japan, May 2003.

[10] E. He, J. Leigh, O.Yu, and T. A. DeFanti, "Reliable Blast UDP : Predictable High Performance Bulk Data Transfer," *Proc. IEEE Cluster Computing*, Chicago, Illinois, 2002.

[11] W. Lorensen and H. Cline, "Marching cubes: a high resolution 3D surface construction algorithm," *Proc., ACM SIGGRAPH 87*, 1987.

[12] J. Robinson, "ULE: A Modern Scheduler For FreeBSD," *BSD Con 2003*, San Mateo, California, September 2003.

[13] L. Smarr et al., "The optiputer, quartzite, and starlight projects: a campus to global-scale testbed for optical technologies enabling lambdagrid computing," *OFC/NFOEC Technical Digest*, Los Angeles, California, March 2005.

[14] R. Wu and A. Chien, "GTP: Group Transport Protocol for Lambda Grids," *Proc. CCGrid, 2004*, Chicago, Illinois, 2004.

[15] C. Xiong et al., "LambdaStream - a Data Transport Protocol for Streaming Network-intensive Applications over Photonic Netwotks," *Proc., PFLDNet 2005*, Lyon, France, 2005.