End-System Aware, Rate-Adaptive Protocol for Network Transport in LambdaGrid Environments *

Pallab Datta†
Computer and Computational Sciences Division
Los Alamos National Laboratory

Wu-chun Feng, Sushant Sharma‡
Department of Computer Science
Virginia Tech

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 that can be reserved exclusively for such applications. These dedicated high-speed connections eliminate network congestion as seen in traditional Internet, but they effectively push the network congestion to the end systems, as processing speeds cannot keep up with networking speeds. Thus, developing an efficient transport protocol over such high-speed dedicated circuits is of critical importance.

We propose the idea of a end-system aware, rate-adaptive protocol for network transport, based on end-system performance monitoring. Our proposed protocol significantly improves the performance of data transfer over LambdaGrids by intelligently adapting the sending rate based on end-system constraints. We demonstrate the effectiveness of our proposed protocol and illustrate the performance gains achieved via wide-area network emulation.

1 Introduction

The OptIPuter project [Smarr et al. 2004] 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 high-performance distributed computing.

LambdaGrids are a new paradigm in distributed computing, where dedicated high-bandwidth optical networks allow globally distributed compute, storage, and visualization systems to work together as a planetary-scale supercomputer. Such a distributed supercomputer will enable scientists to analyze, correlate, and visualize extremely large and remote datasets on-demand and in real time.

The networking aspect of a LambdaGrid consists of two interdependent parts. The first part requires an architectural infrastructure to enable a LambdaGrid, i.e., globally distributed nodes with different capabilities that are interconnected via high-bandwidth optical networks. Examples of such optical networks include National LambdaRail (NLR) [NLR 2006], DOE UltraScience Net [DoE 2006], CANARIE CA*net [Canarie 2005], and UKLight [UKLight 2006]. The second part consists of a collection of hardware-software interface tools that overlay the aforementioned architectural infrastructure to allow e-Science applications to harness and realize the potential of the LambdaGrid. This part has been the focus of significant research in the recent years.

In contrast to shared, packet-switched, Grid infrastructures, LambdaGrids have computational endpoints that are interconnected via dedicated high-speed links (e.g., OC-192 $\approx$ 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 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), i.e., they are characterized by both high bandwidth (e.g., 10 Gbps) as well as long round-trip time (*RTT*) delays (e.g., 100 ms). Delivering high throughput in large BDP networks is a long-standing research challenge, one that now has an

*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-06-5334.

†E-mail: pallab@lanl.gov
‡E-mail:{feng, sushant}@cs.vt.edu

SC2006 November 2006, Tampa, Florida, USA
0-7695-2700-0/06 $20.00 ©2006 IEEE

entire workshop devoted to it — *The International Workshop on Protocols for Fast Long-Distance Networks (PFLDnet)*. TCP and its variants [Jacobson 1988; Brakmo and Peterson 2003; Mathis et al. 1996] 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 [Feng and Tinnakornsrisuphap 2000]. Accordingly, researchers have pursued alternative solutions to overcome the the limitations of TCP/IP in large BDP networks and provide high-performance networking capabilities in such environments. In recent years, rate-controlled UDP/IP-based protocols [He et al. 2002; Xiong et al. 2005; Gu and Grossman 2004; Wu and Chien 2004; Dickens 2003; Zheng et al. 2004; Rao et al. 2004] have emerged as feasible alternatives.

For example, in RBUDP (Reliable Blast UDP), 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-sequenced data packets via UDP. The above cycle 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 performs reasonably well in LambdaGrid environments [He et al. 2002], its main weakness is its inability to adapt its 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 [Smarr et al. 2004].

The LambdaStream approach [Xiong et al. 2005] primarily supports visualization applications that can tolerate packet losses rather than applications that need reliable delivery (e.g., bulk data transfer). As such, our proposed end-system aware, rate-adaptive protocol, described in Section 6, arguably provides a complementary solution to LambdaStream.

The UDP-based data transfer (UDT) protocol [Gu and Grossman 2004] proposes rate-based congestion control that is implemented as application-level processes running atop UDP. Though UDT performs better than TCP over large BDP networks, UDT's potential is not fully realized as it does not model the end-system interactions between the operating system (OS) and network that contribute to congestion. The lack of such a model then forces UDT to rely on intuitive, but theoretically unfounded, heuristics.

The approach proposed in [Zheng et al. 2004] can achieve relatively high circuit utilization if the initial sending rate is set appropriately, but like RBUDP, it lacks rate adaptation at the sender side, which leads to unwanted packet losses if the initial sending rate is set too high.

Overall, we argue that the main problem with all the aforementioned protocols is that because they do *not* rigorously model the end-system dynamics between the OS scheduler, network, and other applications, they only perform well in isolated scenarios, e.g., when network transport is essentially the *only* task running on the sending and receiving end-systems. However, in addition to network transport, the receiving (or sending) end-system oftentimes runs other processes — from a seemingly innocuous desktop environment like GNOME to a more intrusive real-time visualization and analysis of the received data, which may be computationally intensive. With respect to the latter, the OS on the receiving end-system must schedule a 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 are routinely dropped due to buffer overflow, e.g., when the receiving-data process is not scheduled by the OS at appropriate times to transfer the packets from the line-card buffer on the NIC to physical memory. Transmitting data to such an end system (at a fixed rate relentlessly as in RBUDP or FRTP only exacerbates the problem of end-system congestion).

Therefore, we propose the notion of an end-system aware, rate-adaptive protocol, based on performance monitoring, to significantly improve the performance of data transport over a LambdaGrid. In particular, we focus on dynamically monitoring the packet losses at the receiving end-system so that it can be used as a trigger to modulate the sending rate, and hence, avoid further losses while still ensuring high circuit utilization in the presence of end-system constraints.

The rest of this paper is organized as follows. Section 2 describes the problem and potential approaches to the problem. Section 3 discusses the end-system task monitoring that is needed to support our end-system modeling, and hence, end-system aware, rate-adaptive protocol. Section 4 provides an overview of the internals of an operating system (OS) that are relevant to network scheduling over LambdaGrids, including the life cycle of a process and the structure of processor run-queues and task migration. With an understanding of the key OS internals for LambdaGrid networking in place, Section 5 presents detailed performance models for receiver-driven feedback in support of preemptive data transfer and theoretically proves the impossibility of accurately estimating the process context-switch intervals in a general-purpose OS (such as Linux) at the receiving end-system. Consequently, current rate-adaptive protocols that are based

on such estimations are flawed, and Section 5 closes with an illustration of this via a network-emulation study. Section 6 presents our end-system aware, rate-adaptive protocol, followed by experimental results in Section 7. Finally, we conclude the paper in Section 8.

2 Problem Depiction & Approaches

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 (Tbps) capacity per fiber. A LambdaGrid is a set of distributed resources directly connected with such DWDM links, in which network bandwidth is no longer the key performance limiter to communication.

Network performance can be substantially improved in LambdaGrid environments if packet losses (due to end-system congestion) are avoided, e.g., when the receiving end-system OS is context-switched to another process other than the networking process. The following are some possible approaches:

- A real-time OS (RTOS) can be employed. A RTOS allows hard deadlines to be specified for tasks. However, a RTOS is generally expensive to maintain and unlikely to be adopted by the general scientific community. 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 on 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 will not provide.
- Various parameters of an OS scheduler, such as maximum allocated timeslice and maximum dynamic-bonus priority granted to an I/O process, may be adjusted to reduce packet loss. However, this leads to custom OS kernels for applications, and application scientists running in LambdaGrid environments would rather not deal with customized kernels (or kernel patches) to improve their network performance.
- A feedback-based, network-scheduling protocol can allow the receiver to proactively deliver feedback to the sender, e.g., to suspend transmission of data for a specified interval of time, based on the monitoring of the dynamic priority and scheduling of tasks at the receiving end-system. These approaches have been studied by [Banerjee et al. 2006; Datta et al. 2006]; however, they cannot accurately estimate the context-switch intervals, as will be proven in Section 5, and can also lead to poor circuit utilization, due to their *"stop-and-go"* approach.

Given that a feedback-based, network-scheduling protocol is the most feasible approach of the above alternatives, we demonstrate in this paper the problems with a feedback-based network-scheduling algorithm that is based on end-system monitoring using a *"stop-and-go"* approach. More specifically, we rigorously expose the problems with the latest, and arguably, one of the best-performing algorithms over LambdaGrids called RBUDP$^+$ [Datta et al. 2006] and propose a new end-system aware, rate-adaptive protocol called RAPID$^+$ that addresses these problems.

3 End-System Task Monitoring

As part of our initial study in [Datta et al. 2006], we monitored 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 timeslice to a CPU-intensive process (and hence, not respond to packet-handling interrupts from the NIC), we can approximately estimate when end-system congestion might occur.

In [Datta et al. 2006], a soft real-time (SRT) process was implemented at the receiving end-system in order to predict periods of end-system congestion and to send explicit feedback notification back to the sender to stop data transfer for a specified duration of time. This was then used to implement a feedback-based, network-scheduling protocol on the Linux 2.6 kernel. That is, we modified the *Reliable Blast UDP* (RBUDP) protocol and studied the performance of this modified protocol (named RBUDP$^+$) under varying transmission rates. A similar approach was studied in [Banerjee et al. 2006], but its feedback mechanism was based on monitoring the priority levels of tasks using MAGNET (Monitoring Apparatus for General KerNel-Event Tracing) [Feng et al. 2002; Gardner et al. 2003] and had the limitation that it worked only for round-trip time (*RTT*) values on the order of 100 ms or less. Since a typical LambdaGrid environment could experience much higher RTT values, the work in [Banerjee et al. 2006] was not general enough and invalidated the use of this approach under such conditions. Ultimately, however, the fundamental problem with both the approaches studied in [Banerjee et al. 2006; Datta et al. 2006] is that they attempt to decide exactly when to suspend data transmission at intermittent intervals, i.e., the *"stop-and-go"* approach, thus resulting in low circuit utilization.

The following sections illustrate the difficulties and limitations that we encountered during our implementation of RBUDP$^+$ proposed in [Datta et al. 2006]. The ensuing sections also describe why this approach is not accurate due to the dynamics of process handling in a Linux kernel.

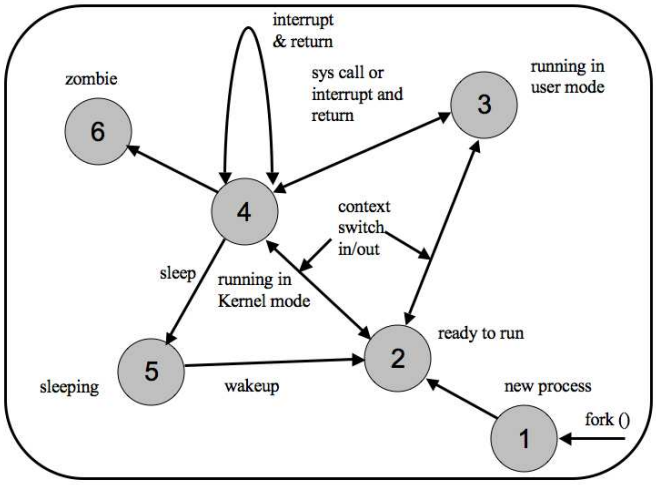

Figure 1: Different states of a process during its life cycle

4 OS Internals for Networking

Here we provide an overview of the internals of an operating system (OS) that are relevant to network scheduling over LambdaGrids, specifically the life cycle of a process and the structure of processor run-queues and task migration. For the sake of convenience, we focus on the Linux OS, particularly given its ubiquity in LambdaGrid environments.

4.1 Life Cycle of a Linux Process

Below we outline the different states that a process migrates through from its invocation until it exits from the *process table*. These changes can occur, for example, when the process makes a system call, it is someone else's turn to run, an interrupt occurs, or the process asks for a resource that is currently not available.

A newly created process enters the system in State 1 as shown in Figure 1. If the process is simply a copy of the original process (i.e., a fork but not an exec), it then begins to run in the state that the original process was in (State 3 or State 4). If an exec() is made, then the process will end up in kernel mode (State 4). It is possible that the fork()-exec() was done in system mode, and the process goes into State 3. However, this is highly unlikely.

When a process is running, an interrupt may be generated (more often than not, this is the system clock), and the currently running process is preempted (State 2). This is the same state as State 2 because it is still ready to run and in main memory. The only difference is that the process was just kicked off the processor.

When the process makes a system call while in user mode (State 3), it moves into State 4 where it begins to run in kernel mode. Assume at this point that the system call made was to read a file on the hard disk. Because the read is not carried out immediately, the process goes to sleep, waiting on the event that the system has read the disk and the data is ready. It is now in State 5. When the data is ready, the process is awakened. This does not mean it runs immediately, but rather it is once again ready to run in main memory (State 2).

If a process that was asleep is awakened (perhaps when the data is ready), it moves from State 5 (sleeping) to State 2 (ready to run). This can be in either user mode (State 3) or kernel mode (State 4).

A process can end its life by either explicitly calling the exit() system call or having it called for them. The exit() system call releases all the data structures that the process was using. If the exiting process has any children, they are "inherited" by init. [1] One value stored in the process structure is the PID of that process' parent process. This value is (logically) referred to as the parent process ID or PPID. When a process is inherited by init, the value of its PPID is changed to 1 (the PID of init).

A process state change can cause a context switch in several different cases. One case is when the process voluntarily goes to sleep, which can happen when the process needs a resource that is not immediately available. When a process puts itself to sleep, it sleeps on a particular wait channel (WCHAN). When the event that is associated with that wait channel occurs, every process waiting on that wait channel is awakened.

When a process puts itself to sleep, it voluntarily relinquishes the CPU. A process that puts itself to sleep can then set the priority at which it will run when it awakens. Normally, the kernel process-scheduling algorithm calculates the priorities of all the processes. However, in exchange for voluntarily giving up the CPU, the process is allowed to choose its own priority.

4.2 Process Run-Queues & Task Migration

The run-queue data structure is the most basic structure in the Linux 2.6 scheduler; there is one run-queue per processor. Essentially, a run-queue keeps track of all runnable tasks assigned to a particular CPU. In Linux 2.6, there are two priority arrays, one is the *active array* and the other is the *expired array*. These are queues of runnable processes per priority level.

Each of these arrays consists of different queues of runnable processes with each set at a different priority level. For ex-

[1] init is the parent of all processes. Its primary role is to create processes from a script stored in the file **/etc/inittab**. This file usually has entries which cause init to spawn **gettys** on each line that users can log in. It also controls autonomous processes required by any particular system.

ample, 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, with each having a particular allocated *timeslice* for execution and a static priority (set relative to task niceness) and dynamic priority (set equal to the priority level).

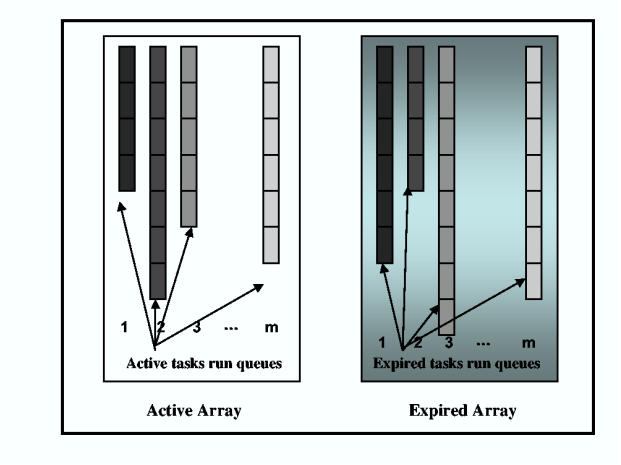

Figure 2: Active and expired priority arrays at different priority levels.

Similarly, we have a set of processes between priority levels $1 \cdots m$ in the *expired array*. All tasks have a static priority, often called a nice value. In Linux, nice values range from -20 to +19, where higher values correspond to lower priority (tasks with high nice values are nicer to other tasks). By default, tasks start with a static priority of 0, but that priority can be changed via the nice() system call. A task's static priority is stored in its *static_prio* variable, where p is a task, $p \rightarrow$*static_prio* is its static priority.

The Linux 2.6 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. $p \rightarrow$*prio* where p is a task). If a task is interactive (the scheduler's term for I/O-bound), its priority is boosted. For more details about how the dynamic priorities are calculated, interested readers should refer to [LinuxScheduler 2005]. At the end of its timeslice, each task's dynamic priority is recalculated, 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 *active array* have surpassed their *STARVATION_LIMIT*, a task can get re-entered into any of the priority levels in the *active array* or may be migrated to the *expired array*.

5 End-System Modeling and Evaluation

Building on our understanding of the key OS internals for LambdaGrid networking, this section first presents detailed performance models for receiver-driven feedback in support of preemptive data transfer. These performance models then form the basis of our proof on the impossibility of accurately estimating the process context-switch intervals in a general-purpose OS at the receiving end-system, estimations that current state-of-the-art rate-adapting protocols depend upon. Finally, the section concludes with a network emulation that empirically demonstrates the above problem.

There are several scenarios that can occur in the kernel, and we illustrate why it is impossible to accurately estimate the time bounds for which a process will be context-switched out, thus effectively invalidating the "stop-and-go" approach of proactive feedback from the receiver, as proposed in RAPID [Banerjee et al. 2006] and in RBUDP$^+$ [Datta et al. 2006]. We provide an in-depth derivation of different possibilities of task-migration scenarios in the kernel and analyze how easy or difficult it is to estimate accurate time bounds for which to suspend data transfer.

Here we propose two end-system models: the ***constant model*** and the ***varying model***. The constant model assumes that the fraction of tasks from each priority level that gets re-inserted to the active array is a constant across all priority levels (when such a scenario occurs). In the varying model, we assume that the fraction of tasks from each priority level that gets re-inserted to either the active array or in the expired array ahead of the I/O-task varies with the priority levels.

5.1 Notation

The following are the notations that we will be used in this section:

- α_i: No of tasks in the i^{th} level of active array.

  $i = 1, 2 \cdots m$ (Figure 2).

- β_i: No of tasks in the i^{th} level of expired array.

  $i = 1, 2 \cdots m$ (Figure 2).

- t_k^j: k^{th} task in the j^{th} level

  $j = 1, 2 \cdots m$

  $k = 1, 2 \cdots \alpha_i$ or

  $k = 1, 2 \cdots \beta_i$.

- T_k^j: value of timeslice of the k^{th} task in the j^{th} level

  $j = 1, 2 \cdots m$

$k = 1, 2 \cdots \alpha_i$ or

$k = 1, 2 \cdots \beta_i$.

- $D_k^{j,current}$: Current dynamic priority of k^{th} task in the j^{th} level.
- $D_k^{j,new}$: New dynamic priority value at the end of the timeslice of k^{th} task in the j^{th} level..
- δ: maximum deviation of dynamic priority for an I/O-bound task.
- ρ: is the constant fraction of tasks from each level in the active array, that gets re-inserted into the active array (used in constant model).
- ρ_i: is the fraction of tasks from each level that gets re-inserted into the active array or in the expired array ahead of the I/O-task (used in varying model).
- Δ: time period for which tasks might enter the wait-queue waiting for other resources.
- $I_{t_k^j}$: Task indicator, indicates whether the task is in the active or expired array.

$$I_{t_k^j} = \begin{cases} 1 & if\ t_k^j \in ActiveArray \\ 0 & if\ t_k^j \in ExpiredArray \end{cases} \quad (1)$$

- T: Total time period to stop the data transfer.

5.2 Performance Models

The following equations illustrate the constant and the varying models for the different scenarios of task migration in the kernel, based on its newly calculated dynamic priority.

Suppose the current dynamic priority of the I/O-bound task, which is the k^{th} task at the j^{th} level is $D_k^{j,current}$. Let the new dynamic priority of the I/O-bound task at the end of its timeslice be $D_k^{j,new} = D_k^{j,current} - \delta = D_0$, irrespective of whether it is re-entered in the active array or dispatched to the expired array.

Case I: Let $I_{t_k^j} = 1$ for the I/O-bound task after re-calculation of the dynamic priority with $\rho = \rho_i = 0$ and for all other tasks in the active array, $t_k^j\ (\forall k,\ j = 1, 2 \cdots m)\ I_{t_k^j} = 0$.

Varying Model: The total time for which the transmission needs to be suspended in order to avoid data losses can be **accurately** calculated in this scenario and is given by:

$$T = \sum_{j=1}^{m} \sum_{k=1}^{\alpha_i} T_k^j + \Delta$$

The Δ factor accounts for tasks which migrate to the wait-queue waiting for some resources or results.

Case II: Let $I_{t_k^j} = 1$ for the I/O-bound task after re-calculation of the dynamic priority but with $\rho \neq 0$. In this case, some tasks $\rho \cdot t_k^j\ (\forall k,\ j = 1, 2 \cdots m)$ from the active array are re-entered into the active array, hence $I_{t_k^j} = 1$, and for these tasks, $D_k^{j,new} > D_0$.

Varying Model: This case is similar to Case I. Here, the total time that the data transfer needs to be suspended is equal to the summation of the timeslices of all the tasks that are in the active array (before calculation of their new dynamic priority) and the time period for which the tasks are in the wait queue.

$$T = \sum_{j=1}^{m} \sum_{k=1}^{\alpha_i} T_k^j + \Delta$$

Case III: Let $I_{t_k^j} = 1$ for the I/O-bound task after re-calculation of the dynamic priority. As in Case II, we consider the case where $\rho \neq 0$ and some tasks $\rho \cdot t_k^j\ (\forall k,\ j = 1, 2 \cdots m)$ from the active array are re-entered into the active array, i.e. $I_{t_k^j} = 1$; but unlike Case II, $D_k^{j,new} \leq D_0$ for these tasks.

- ***Constant Model***: In this case, the total time to suspend the data transfer is equal to the summation of the timeslices of all the tasks in the active array (before re-computation of their dynamic priorities), the summation of the timeslices of the fraction of tasks that keeps re-entering the active array at priority levels greater than D_0. Hence, the total time is given as:

$$T = \sum_{j=1}^{m} \sum_{k=1}^{\alpha_i} T_k^j + \sum_{j=1}^{D_0} \sum_{k=1}^{\rho\alpha_i} T_k^j + \sum_{j=1}^{D_0} \sum_{k=1}^{\rho^2\alpha_i} T_k^j + \sum_{j=1}^{D_0} \sum_{k=1}^{\rho^3\alpha_i} T_k^j + \cdots + \Delta$$

  The Δ factor accounts for time intervals if all the processes enter the wait queue and are waiting for certain resources or results.

- ***Varying Model***: In this case, we assume that the fraction of tasks from each priority level *i* that re-enters the active array at a priority level higher than D_0 is given by ρ_i, instead of a constant ratio ρ as considered in the varying model. The total time is calculated as follows:

$$T = \sum_{j=1}^{m} \sum_{k=1}^{\alpha_i} T_k^j + (\sum_{j=1}^{D_0} \sum_{k=1}^{\rho_1\alpha_1} T_k^j + \sum_{j=1}^{D_0} \sum_{k=1}^{\rho_2\alpha_2} T_k^j + \cdots + \sum_{j=1}^{D_0} \sum_{k=1}^{\rho_{m_1}\alpha_m} T_k^j) + (\sum_{j=1}^{D_0} \sum_{k=1}^{\rho_1^2\alpha_1} T_k^j + \sum_{j=1}^{D_0} \sum_{k=1}^{\rho_2^2\alpha_2} T_k^j + \cdots + \sum_{j=1}^{D_0} \sum_{k=1}^{\rho_{m_1}^2\alpha_m} T_k^j) +$$

$$(\sum_{j=1}^{D_0}\sum_{k=1}^{\rho_1^3\alpha_1} T_k^j + \sum_{j=1}^{D_0}\sum_{k=1}^{\rho_2^3\alpha_2} T_k^j + \cdot + \sum_{j=1}^{D_0}\sum_{k=1}^{\rho_{m1}^3\alpha_m} T_k^j) + \cdot \cdot + \Delta$$

As can be observed from the equations, the calculation of either the approximate or exact total time involves a recursive addition of the timeslices of a *fraction* of tasks that re-enter the active array. However, it is impossible to estimate (1) the exact fraction of tasks that re-enter the active array, (2) the timeslices of tasks that re-enter the active array (which is based on the average sleep time of the task), (3) whether all the tasks would finish their timeslices before re-entering the active array, and (4) the value of Δ as all tasks migrate from State 5 to State 2. Consequently, it is impossible to predict the time period for stopping the sender from sending data since no accurate or approximate estimation can be made. This "corner case" ultimately results in a partial failure of the feedback-based *"stop-and-go"* approaches proposed in [Banerjee et al. 2006; Datta et al. 2006].

Case IV: Let $I_{t_k^j} = 0$ for the I/O-bound task after re-calculation of the dynamic priority. Consider that $\rho = 0$ and all tasks t_k^j ($\forall k,\ j = 1,2\cdots m$) from the active array are dispatched to the expired array, i.e. $I_{t_k^j} = 0$, and their priorities are $D_k^{j,new} > D_0$.

Varying Model: In this scenario, all the tasks in the active array migrate to the expired array, but at a lower priority as compared to the I/O task. Therefore, the total time can be calculated as the summation of the timeslices of all the tasks in the active array along with the wait period (if all tasks enter the wait queue). So, the total time is given as follows:

$$T = \sum_{j=1}^{m}\sum_{k=1}^{\alpha_i} T_k^j + \Delta$$

Case V: Let $I_{t_k^j} = 0$ for the I/O-bound task after re-calculation of the dynamic priority. Consider that $\rho = 0$ and all tasks t_k^j ($\forall k,\ j = 1,2\cdots m$) from the active array are dispatched to the expired array, i.e., $I_{t_k^j} = 0$, but that their newly calculated priorities are $D_k^{j,new} \leq D_0$.

Varying Model: In this scenario, the total time can be calculated as the summation of the timeslices of all tasks in the active array and the timeslices of these tasks when they are entered in the expired array at a priority level higher than D_0. Thus, the total time becomes

$$T = \sum_{j=1}^{m}\sum_{k=1}^{\alpha_i} T_k^j + \sum_{j=1}^{D_0}\sum_{k=1}^{\beta_i} T_k^j + \Delta$$

Case VI: Let $I_{t_k^j} = 0$ for the I/O-bound task after re-calculation of the dynamic priority. Consider the case where $\rho \neq 0$, and hence, some tasks, $\rho \cdot t_k^j$ ($\forall k,\ j = 1,2\cdots m$) from the active array are re-entered in the active array, i.e. $I_{t_k^j} = 1$.

This scenario is similar to **Case III**, and the total time for the varying and the constant model can be given by the equations derived above for Case III. Since it is impossible to accurately predict the fraction of tasks that are re-entered in the active array as well as their future timeslices, prediction of the time period is impossible.

As can be observed in all the six cases described above, there are many dynamics in the kernel that cannot be accurately estimated by the soft real-time process in order to schedule feedback at the correct instant in time — particularly, calculating the feedback to stop data transmission for an exact duration of time. The kernel cannot be certain how the task timeslices get utilized, nor can it be sure about the time interval (Δ) for which all processes might enter the sleep queue (State 5 $\rightarrow$ 2 in Figure 1).

In the next subsection, we will illustrate the performance drawbacks that we observed as part of our feedback-based *"stop-and-go"* approach in [Datta et al. 2006] due to the inaccuracy of prediction of the time intervals for which a process gets context-switched out. (These drawbacks also apply to [Banerjee et al. 2006].)

5.3 Performance Evaluation: "Stop-And-Go" Scheduling

To emulate a very fast LambdaGrid network, we connected two machines back-to-back with Chelsio 10-Gigabit Ethernet (10GigE) adapters. The details of the experimental setup can be found in [Datta et al. 2006].

We transferred a file of size 700MB via RBUDP [He et al. 2002] and RBUDP$^+$ [Datta et al. 2006]. For both protocols, we measured the end-system to end-system transfer time for sending rates between 0.8-3.4 Gbps.[2] We performed emulation studies under two scenarios: (1) the receiving end-system was under no additional computational load, and (2) the receiving end-system was loaded with a synthetic load.

Figure 3 shows that in the case of no additional computational load, the RBUDP$^+$ protocol actually performs worse than the traditional RBUDP protocol. In addition, it shows that the total data transfer time actually decreases steadily up to a transmission rate of 2.6 Gbps and then increases slowly for both the schemes. In the absence of any other load, the normal RBUDP scheme keeps sending data from the sender to the receiver and the I/O-bound process never gets context-switched out. In comparison, RBUDP$^+$ aggressively stops the sender from sending data at certain instances

[2]Note: We did not enable any of the offload engine support that was available on the Chelsio NIC's as they only directly support TCP, not UDP.

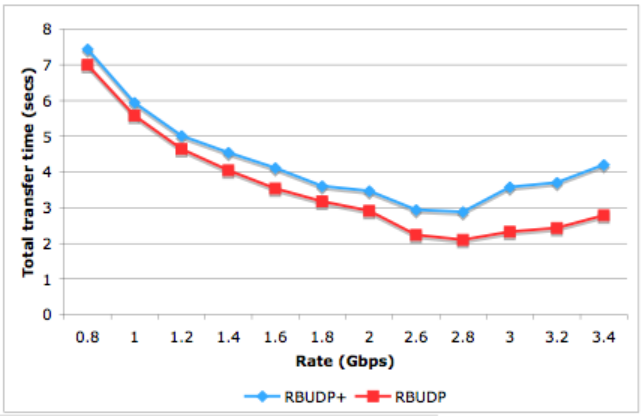

Figure 3: Comparison of data transfer times for RBUDP$^+$ and RBUDP at no load

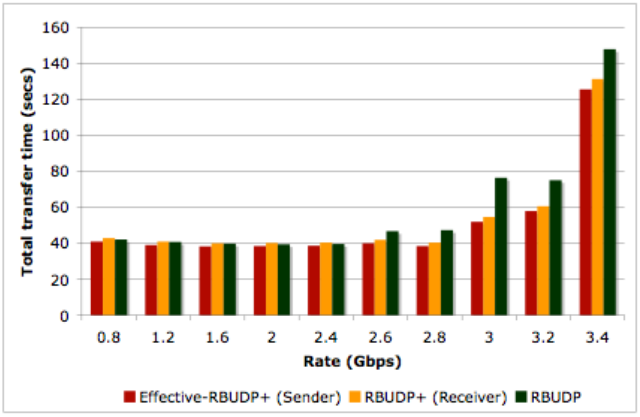

Figure 4: Comparison of data transfer times for RBUDP$^+$ and RBUDP under load

during its data transfer. This results in RBUDP$^+$ consuming slightly more time for the total data transfer as compared to the RBUDP scheme. As can be seen this *"stop-and-go"* approach definitely leads to poor circuit utilization. The RBUDP$^+$ protocol required 6.6% -50.5% more time in data transfer as compared to RBUDP protocol.

On the other hand, in the presence of a synthetic load, Figure 4 shows that RBUDP$^+$ generally performs as well as or better than RBUDP, particularly at rates greater than 2.4 Gbps. The primary reason for this is that at such high data rates, the receiver simply gets swamped with too much data. The RBUDP$^+$ protocol prohibits such a scenario by proactively stopping the sender from sending any data by predictively estimating the time instances for which the I/O-bound process will get context-switched out and by transmitting feedback to the sender at the appropriate time. However, due to the inaccuracies in the prediction (as outlined in Section 5), we see that RBUDP$^+$ does not perform as well at lower sending rates.

6 RAPID$^+$: Rate-Adaptive Protocol for Information Delivery

Because TCP has been shown to be inefficient in networks with large bandwidth-delay products (BDPs) [Borman et al. 1992; Feng and Tinnakornsrisuphap 2000], a number of TCP enhancements have been proposed to upgrade TCP's congestion control and/or flow control. Examples include High-Speed TCP [Floyd 2003], FAST TCP [Jin et al. 2005], and ScalableTCP [Kelly 2003]. These protocols are proposed to be implemented in the kernel space and require modifications to the OS. To avoid the complexity of kernel changes, other groups of researchers have proposed new transport protocols which are implemented as application-level processes running atop UDP. Examples include SABUL [Gu et al. 2003], UDT [Gu and Grossman 2003], Tsunami [Tsunami 2006], and RBUDP [He et al. 2002]. These protocols are rate-based, rather than window-based like TCP, because they are regarded as a more efficient solution for high-speed networks [Walrand and Varaiya 2000].

Our proposed RAPID$^+$ is an end-system aware, rate-adaptive protocol that fundamentally differs from the aforementioned protocols in that it models the dynamics of end-system interactions between the OS and the network and intelligently adapts its rate based on information from this model in order to achieve high circuit utilization while simultaneously allowing multiple applications to run on the end-systems.

In RAPID$^+$, we propose to use a UDP blast channel for data transfer from the sender to a receiver and a TCP control channel from the receiver to the sender for acknowledging receipt of data and for the notification of lost packet sequences at the receiver, which can be re-transmitted from the sender (shown in Figure 5). A new session starts with a TCP connection establishment between the sender and the receiver. The sender opens a TCP listening port and waits for an incoming connection attempt. A TCP connection is established upon receipt of a request from the receiver. The sender and receiver then exchange a set of parameters via the TCP connection, such as the user-specified sending rate and UDP data channel's port number. The end-to-end RTT and the NIC buffer capacity can also be conveyed as part of the initial connection setup.

After successful control-channel establishment and parameter exchange over TCP, the data transfer on the end-to-end circuit starts over the UDP channel. During the data transfer, the sender is responsible for data transmission and retransmission based on feedback from the receiver. Prior to the data transfer, the sender and receiver negotiate an *initial sending rate*. The sender starts blasting data to the receiver at this initial rate. At the receiving end, the rate at which the receiving application reads the data is calculated based on the packets received as compared to the packets sent by the UDP blast,

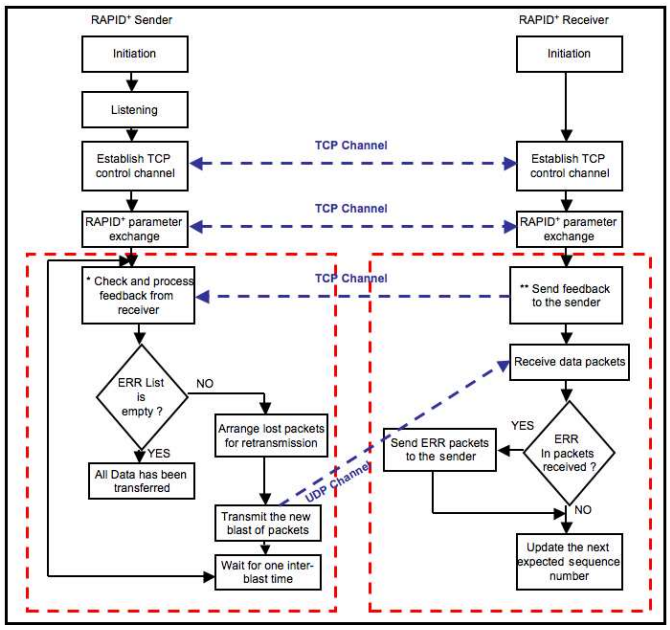

Figure 5: RAPID$^+$: Rate-Adaptive Protocol for Information Delivery

and the sequence numbers (SN's) of the packets that are *not* received are marked to be re-sent by the sender in subsequent intervals. All this information is then used to tune the sending rate appropriately.

Based on the measurements made by the receiving end-system, a feedback message is sent back to the sender. This message notifies the sender of the receiver's reading rate and the error-sequence numbers for the packets that have not been received. After the receipt of the feedback, the sender modulates its sending rate (based on the algorithm described below) until the next feedback message arrives. The end result is that RAPID$^+$ supports both rate adaptation and maximal circuit utilization under end-system constraints.

The details of the rate-adaptation algorithm in RAPID$^+$ are described below as well as shown in Figure 6. Figure 7 illustrate the feedback checking at the sender and the feedback sending at the receiver.

6.1 Notation

Below is the notation that we will use in this section.

- B_{NIC}: The buffer capacity of the NIC at the receiver.
- RTT: The round trip time delay between the sender and the receiver.
- δ_n: The number of incoming packets read by the receiving application at the receiver (measured using MAGNET) during the current iteration.[3]
- α_n: The number of packets lost due to buffer overruns, at the receiver during the n^{th} iteration.
- α_{n-1}: The number of packets lost at the receiver during the last (n-1^{th}) iteration of data sent.
- R_n: The sending rate at the current (n^{th}) iteration.
- R_{n+1}: The sending rate at the next (n+1^{th}) iteration.
- γ : The average rate of decrement of packet losses over $\boldsymbol{k}$ successive iterations.

Input: The rate at which the application drains the NIC buffer space (δ_n), the current packet loss rate (α_n), packet loss rate at the preceding iteration (α_{n-1}) and the user-defined parameter $\boldsymbol{k}$.

Output: (a) The new rate (R_{n+1}) for data transfer from sender to receiver.

Initialization Parameters: R_1 (user-defined), n=2, count =0; $\gamma = 0$;

Algorithm:

```
While (Data to be sent from the sender ≠ 0)
    {
        Measure δ_{n-1};
        If (α_n ≤ α_{n-1})
          {
            count++;
            If (count ≤ k)
              {
                R_{n+1} = R_n
                γ = (γ + (α_{n-1} / α_n))/count;
                n++;
              }
            Else
              {
                R_{n+1} = R_n(1 + γ);
                /* Scaling the rate according to */
                /* the measured decrement rate */
                count = 0; γ = 0;
                n++; continue;
              }
          }
        Else,
              {
                R_{n+1} = δ_{n-1}; γ = 0; n++; count =0;
              }
    }
```

Note: The above algorithm attempts to minimize data losses at the receiver and to maximize end-to-end circuit utilization under receiver end-system constraints.

Figure 6: Rate-Adaptation Algorithm

[3]Note that each iteration here indicates instances when data is sent from the sender, after receiving a feedback from the receiver

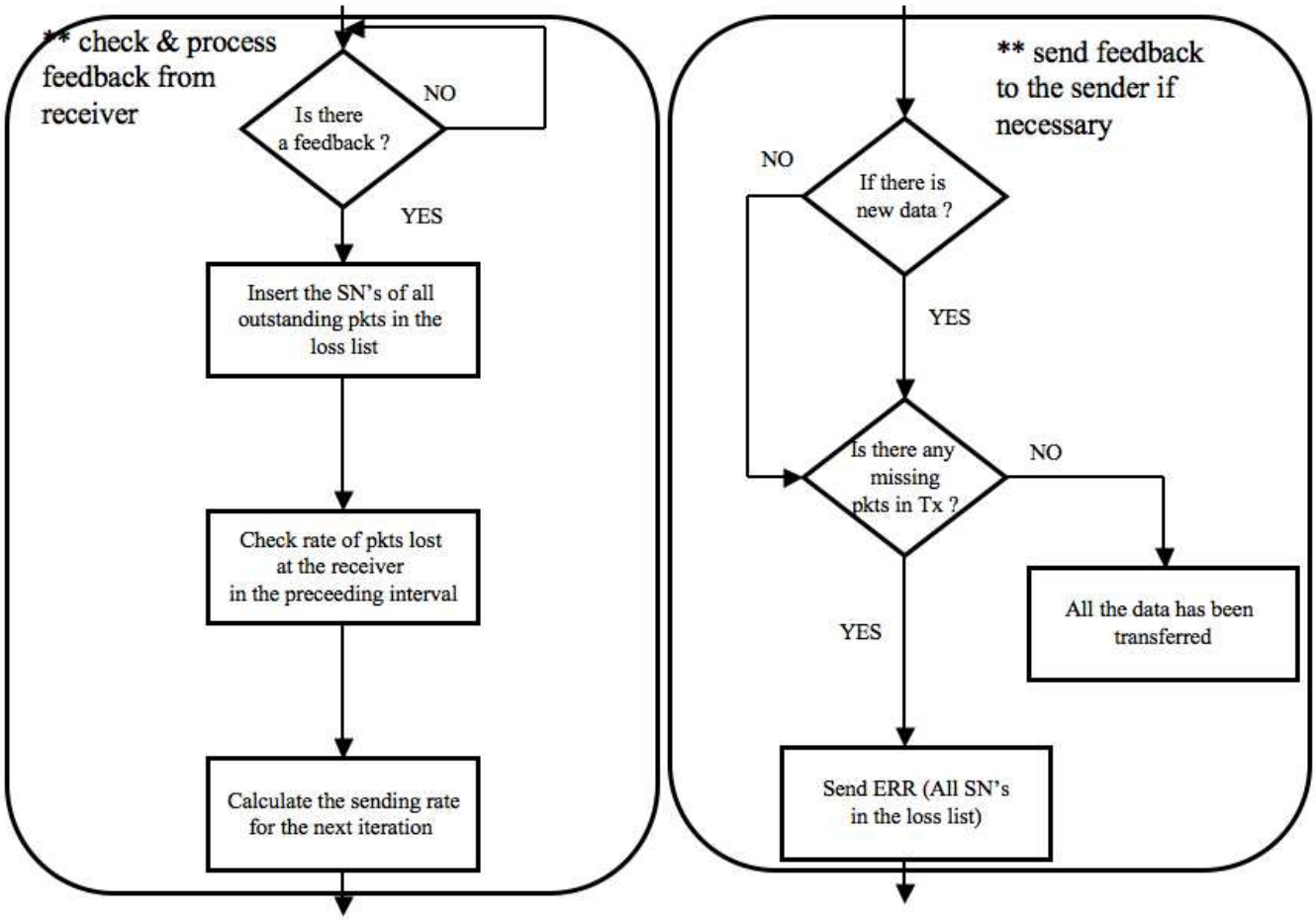

Figure 7: (a) Feedback checking and processing at the sender. (b) Feedback sending at the receiver.

We know the following:

$$\alpha_n = R_n \times RTT/2 - \delta_n - B_{NIC} \tag{2}$$

At the initiation of the algorithm, $n = 2$ and $\gamma = 0$. The initial data transfer rate is set to the user-defined R_1. The initial rate is set assuming that the NIC buffer space can be completely utilized to hold data that is transferred from the sender.

As data transmission proceeds, the rate-control mechanism tunes the sending rate at the sender according to the set of functions given below: If the packet-loss rate at the current iteration of measurement (α_n) is identical to or lower than the packet loss at the preceding iteration (α_{n-1}), then the sending rate for the next iteration of packet send, is held equal to the current sending rate. Otherwise, our attempt would be to make the packet loss minimal, i.e., zero. Hence, using Equation 2 and assuming that the buffer space (B_{NIC}) would already be full with packets from the preceding transmission, our new sending rate for the next iteration is set equal to the rate at which the application is draining the buffer at the receiver at the preceding iteration. Hence the new sending rate would be

$$R_{n+1} = \delta_{n-1} \tag{3}$$

If the packet-loss rates at the receiver continue to decrease in successive iterations, we proportionately ramp up the sending rate based on the rate at which the packet loss decreases over successive iterations. Let us assume that the rate at which the packet loss decreases over $\boldsymbol{k}$ successive iterations is given by γ.[4]

Then, we use the following equation to update the value of γ for $\boldsymbol{k}$ successive iterations.

$$\gamma = \frac{1}{\boldsymbol{k}} \cdot (\gamma + \frac{\alpha_{n-i}}{\alpha_n}) \tag{4}$$

where α_n is the loss rate at the current snapshot of measurement and α_{n-1} is the loss rate at the preceding iteration. Equation 4 averages the proportionate increase in the number

[4]The value of $\boldsymbol{k}$ is empirical and can be varied in the experiments to study the aggressiveness in rate adaptation of the protocol.

of packets received at the receiver between two consecutive iterations, over ***k*** successive iterations. This factor is utilized to increase the sending rate by a factor of $(1 + \gamma)$ as shown in Figure 6.

7 Experimental Results

To test our proposed RAPID$^+$, we connected two machines (2-GHz Pentium 4s with 512-KB cache and 1-GB DDR RAM) via a 3-Com Gigabit Ethernet switch. We emulated an end-system to end-system file transfer by transferring two files of size 1GB and 2GB between these two machines. The experimental setup is an emulation of a wide-area network with any amount of variation in round-trip times. It is to be noted that the decision points for the sending rate-adaptation in our initial implementation modifying the RBUDP protocol, depends on the feedback from the receiver and hence is dependent on the round-trip time (RTT). However the variation or the modulation in the sending rate is independent of the value of RTT.

The dependence of the rate modulation instances on the RTT, is inherently because of the way RBUDP works, it can modulate the sending rates only after receiving a feedback from the receiver, before sending off a new blast of data. The proposed RAPID$^+$ protocol does not necessarily impose any constraints for rate-adaptation at the end of each RTT. The rate-adaptation can happen independent of the round-trip time for data transfer.

In our initial implementation of RAPID$^+$, we modified the RBUDP protocol to make it rate-adaptive at the sender, based on feedback from the receiver. The traditional RBUDP protocol transmits the entire file as a UDP blast and keeps resending the sequence of packets that is not received by the receiver (based on the error map it receives from the receiver) until the entire file is transmitted. In RAPID$^+$, we modulate the rate at which the packets are sent, each time, based on status notification from the receiver. Once the packet loss rate at the receiver reaches zero, the entire file is transmitted.

We compared the data-transfer times for transferring the files of sizes 1GB and 2GB using the RBUDP protocol [He et al. 2002] and an initial implementation of our proposed RAPID$^+$ transport protocol. For both protocols, we measured the end-system to end-system transfer time for starting sending rates varying between 500-900 Mbps.

Comparisons of the data-transfer times between RBUDP and RAPID$^+$ for the 1-GB and 2-GB files are shown in Figures 8 and 9, respectively. These figures show that RAPID$^+$ improves the transfer times by 6.03% and 18.22% for the 1-GB and 2-GB files, respectively.

We also note the significant improvements that we observed

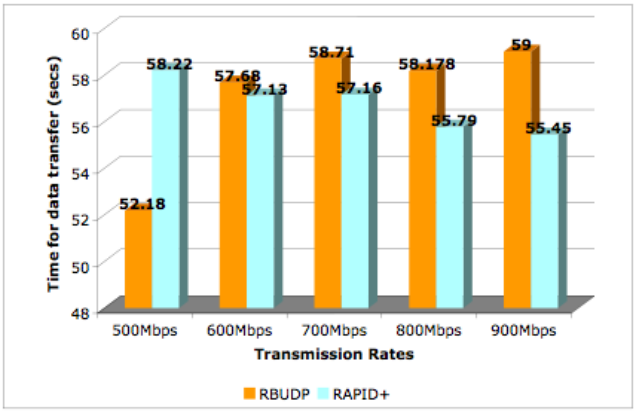

Figure 8: Comparison of transmission times for 1GB file

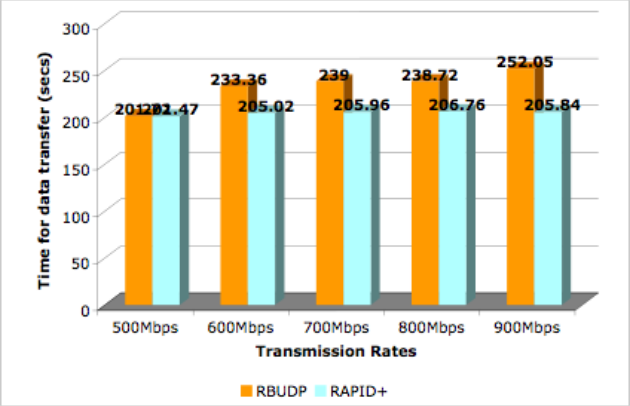

Figure 9: Comparison of transmission times for 2GB file

in the actual amount of data that was transmitted by the sender while sending the 1-GB and 2-GB files for RBUDP and RAPID$^+$, respectively. In Figure 10, we see a 41.47%-111.86% reduction in the amount of actual packets that are transmitted for a 1-GB file. In Figure 11, we see an even more dramatic 139.32%-387.91% reduction when transmitting a 2-GB file. From these figures, the RAPID$^+$ protocol needs significantly less packet re-transmission to transfer large files.

Finally, Figures 12 and 13 show how the transmission rates at the sender get modulated in the initial implementation of RAPID$^+$ when the initial sending rate starts between 500-900 Mbps.

8 Conclusion

In this paper, we presented the design and initial implementation and evaluation of a next-generation Rate-Adaptive Protocol for Information Delivery (RAPID$^+$) that is end-system aware and designed specifically to transport data over

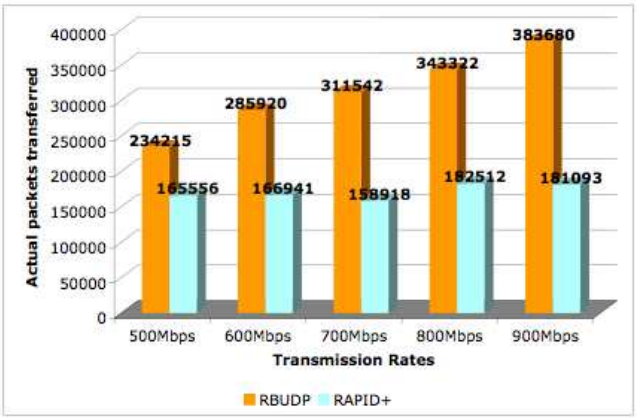

Figure 10: Comparison of actual packets transmitted for 1GB file

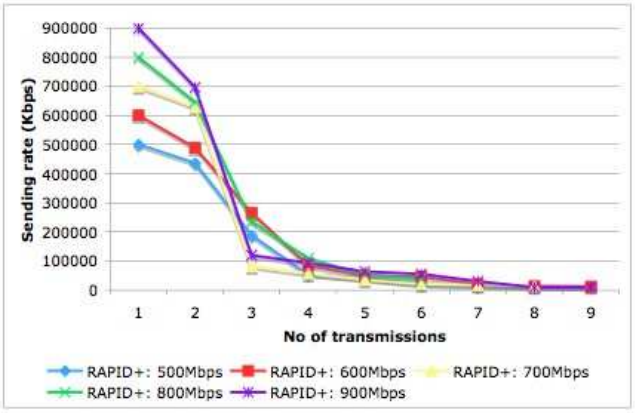

Figure 12: Rate adaptation at the sender while transferring a 1GB file

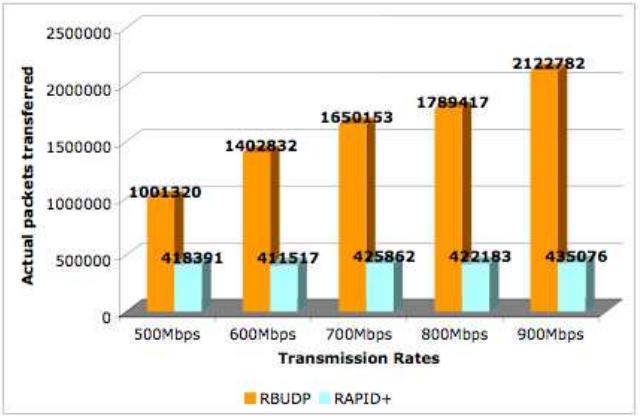

Figure 11: Comparison of actual packets transmitted for 2GB file

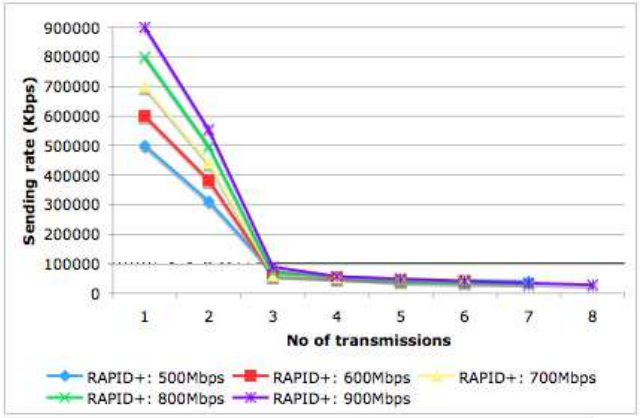

Figure 13: Rate adaptation at the sender while transferring a 2GB file

dedicated end-to-end circuits in support of LambdaGrids.

RAPID$^+$ has two features that distinguish it from other transport solutions: (1) data is transmitted at a rate that is adapted to the end-system (receiver) limitations, moreover attempting to keep the circuit fully utilized under such constraints; (2) it uses dual communication paths — a unidirectional dedicated end-to-end circuit for data transfer and the Internet for end-system congestion notification and rate adaptation. We implemented RAPID$^+$ as an extension of the standard RBUDP protocol and carried out a series of experiments in our local testbed to quantify its performance. The experimental results show that the RAPID$^+$ implementation is effective in significantly improving the data-transfer times and the actual number of packets that needs to be transmitted for transferring a file.

Future work includes explicitly demonstrating the effectiveness of RAPID$^+$ over a real wide-area network rather than an emulated one as well as varying the load scenarios at the sender and receiver. In addition, we intend to more rigorously analyze and characterize our proposed feedback control loop in RAPID$^+$, relative to issues such as scalability, throughput, and stability.

References

BANERJEE, A., FENG, W., MUKHERJEE, B., AND GHOSAL, D. 2006. RAPID: An end-system aware protocol for intelligent data transfer over lambda-grids. *Proceedings of 20th International Parallel and Distributed Processing Symposium (IPDPS 2006), Rhode Island, Greece*.

BORMAN, D., BRADEN, R., AND JACOBSON, V. 1992. TCP extensions for high performance. *RFC1323, Internet Engineering Task Force (IETF)*.

BRAKMO, L., AND PETERSON, L. 2003. TCP Vegas: End to end congestion avoidance on a global internet. *IEEE Journal of Selected Areas in Communications 13*, 8, 1465–1480.

DATTA, P., SHARMA, S., AND FENG, W. 2006. A feedback mechanism for network scheduling in lambdagrids. *Proceedings of the* 6^{th} *International Symposium on Cluster Computing and the Grid (CCGrid), Singapore*, 584–591.

DICKENS, P. 2003. FOBS: A lightweight communication protocol for grid computing. 9^{th} *International Euro-Par Conference, Austria*, 938–946.

FENG, W., AND TINNAKORNSRISUPHAP, P. 2000. The failure of TCP in high-performance computational grids. *SC00: High-Performance Networking and Computing Conference*, 37–48.

FENG, W., GARDNER, M., AND HAY, J. 2005. The MAGNeT toolkit: Design, evaluation, and implementation. *Journal of Supercomputing 23*, 1, 67–79.

FLOYD, S. 2003. Highspeed TCP for large congestion windows. *RFC3649, Internet Engineering Task Force (IETF)*.

GARDNER, M., FENG, W., AND HAY, J. 2002. Monitoring protocol traffic with a MAGNeT. *Proceedings of the* 3^{rd} *Passive and Active Measurement Workshop (PAM2002), Fort Collins, Colorado*.

GARDNER, M., FENG, W., BROXTON, M., ENGELHART, A., AND HURWITZ, G. 2003. MAGNET: A tool for debugging, analysis and adaptation in computing systems. *Proceedings of the* 3^{rd} *International Symposium on Cluster Computing and the Grid (CCGrid), Tokyo, Japan*.

GU, Y., AND GROSSMAN, R. 2003. End-to-end congestion control for high performance data transfer. *IEEE/ACM Transactions on Networking*.

GU, Y., AND GROSSMAN, R. 2004. Experiences in the design and implementation of a high performance transport protocol. *SC04: High-Performance Computing, Networking and Storage Conference*, 22–35.

GU, Y., HONG, X., MAZZUCCO, M., AND GROSSMAN, R. 2003. SABUL: A high performance data transfer protocol. *IEEE Communication Letters*.

HE, E., LEIGH, J., YU, O., AND DEFANTI, T. 2002. Reliable Blast UDP: Predictable high performance bulk data transfer. *Proceedings of the* 4^{th} *International Conference on Cluster Computing, Chicago, Illinois*, 317–324.

HTTP://JOSH.TRANCESOFTWARE.COM/LINUX/LINUX_CPU_SCEDULER.PDF. 2005. *The Linux 2.6 Scheduler*, December.

HTTP://WWW.BITCONJURER.ORG/BITTORRENT. 2005. *BitTorrent*, November.

HTTP://WWW.CANARIE.CA/CANET4/. 2005. *CANARIE CA*network 4*, December.

HTTP://WWW.KAZAA.COM. 2005. *Kazaa*, December.

HTTP://WWW.CHELSIO.COM/PRODUCTS/T210.HTM. 2006. *Chelsio T210 10 Gigabit Ethernet Adapter*, April.

HTTP://WWW.CSM.ORNL.GOV/ULTRANET. 2006. *DoE UltraScience Net*, May.

HTTP://WWW.ICAIR.ORG/OMNINET/. 2006. *OMNInet*, January.

HTTP://WWW.INDIANA.EDU/∼ANML/ANMLRESEARCH.HTML. 2006. *Tsunami*, March.

HTTP://WWW.JA.NET/DEVELOPMENT/UKLIGHT/. 2006. *UKlight*, February.

HTTP://WWW.NLR.NET. 2006. *National Lambda Rail*, July.

HTTP://WWW.SURFNET.NL/EN/. 2006. *SURFnet*, June.

JACOBSON, V. 1988. Congestion avoidance and control. *Computer Communication Review 18*, 4, 314–329.

JIN, C., WEI, D., LOW, S., BUHRMASTER, G., BUNN, J., CHOE, D., COTTRELL, R., DOYLE, J., FENG, W., MARTIN, O., NEWMAN, H., PAGANINI, F., RAVOT, S., AND SINGH, S. 2005. FAST TCP: From background theory to experiments. *IEEE Network 19*, 1, 4–11.

KELLY, T. 2003. Scalable TCP: Improving performance in highspeed wide area networks. 1^{st} *International Workshop on Protocols for Fast Long-Distance Networks (PFLDNet) 33*, 2, 83–91.

MATHIS, M., MADHAVI, J., FLYOD, S., AND ROMANOW, A. 1996. TCP selective acknowledgement options. *RFC2018, Internet Engineering Task Force (IETF)*.

RAO, N., WU, Q., CARTER, S., AND WANG, W. 2004. Experimental results on data transfers over dedicated channels. *Proceedings of the* 1^{st} *International Workshop on Provisioning and Transport for Hybrid Networks (PATHNETS), in conjunction with the* 1^{st} *International Conference on Broadband Networks*.

SMARR, L., CHIEN, A., DEFANTI, T., LEIGH, J., AND PAPADOPOULOS, P. 2004. The OptIPuter. *Communications of the ACM 47*, 11.

WALRAND, J., AND VARAIYA, P. 2000. *High-Performance Communication Networks*. Morgan Kaufmann.

WU, X., AND CHIEN, A. 2004. GTP: Group transport protocol for lambda-grids. *Proceedings of the* 4^{th} *International*

Symposium on Cluster Computing and the Grid (CCGrid), Chicago, Illinois, 228–238.

XIONG, C., LEIGH, J., HE, E., VISHWANATH, V., MURATA, T., RENAMBOT, L., AND DEFANTI, T. 2005. Lambdastream-a data transport protocol for streaming network intensive applications over photonic networks. *Proceedings of 3rd International Workshop on Protocols for Fast Long-Distance Networks (PFLDNet), Lyon, France*.

ZHENG, X., MUDAMBI, A., AND VEERARAGHAVAN, M. 2004. FRTP: Fixed rate transport protocol - a modified version of sabul for end-to-end circuits. *Proceedings of the 1st International Workshop on Provisioning and Transport for Hybrid Networks (PATHNETS), in conjunction with the 1st International Conference on Broadband Networks*.