Abstract—Many science workflows rely on networks to move data between geographically distributed locations for analysis, sharing, and storage. The Science DMZ model has proven to be effective at improving data transfer performance by eliminating bottlenecks for science flows in campus networks. However, it has two key limitations: lack of support for real-time/streaming analysis and underutilized, dedicated/static data transfer nodes (DTNs). We present here the design and implementation of an elastic data transfer infrastructure (DTI) for a dynamic Science DMZ architecture that leverages elastic resources for large data transfers. Our results show that our elastic DTI can save up to 98% of resources compared with a typical DTN deployment, while experiencing only minimal overhead (∼1%).

I. INTRODUCTION

Many science workflows are bursty, and thus their computing and network demands fluctuate significantly over time [1–4]. To handle these fluctuating demands from different departments efficiently, university campuses are consolidating compute resources into a private cloud [5–7].

General-purpose networks, also referred to as enterprise networks, can transporting basic data such as emails and web content, but they face numerous challenges when transferring terabyte- and petabyte-scale data. At best, transfers of science data on enterprise networks may last days or even weeks. In response to this challenge, the Science DMZ model [8] for high-speed wide area file transfers was proposed. The Science DMZ is a network or a portion of a network designed to facilitate the transfer of big science data across wide area networks (WANs), typically at rates of 10 Gbps and above. The model has been highly successful and has been deployed at many universities, research institutions, and science facilities. Nevertheless, the Science DMZ model has several limitations.

Real-time analysis of data streaming from instruments such as light sources, fusion devices, and telescopes is emerging as an important use case in science. Real-time analysis will help adjust the data collection dynamically during an experiment and help guide the selection of parameters for the next experiment. With the current Science DMZ model, however, this use case is not possible. Clearly needed is a dynamic Science DMZ that extends to the high-performance computing (HPC) resources in an institution.

Another limitation of the current Science DMZ model is its use of dedicated transfer nodes, or DTNs. These DTNs are provisioned to meet peak data transfer demands. With the exponential growth in the data produced by science experiments, however, the number of DTNs required to meet peak demands is increasing. Some institutions have deployed ten or more DTNs, and in one instance 28 DTNs have been deployed [9]. Yet as we observed in [10], DTN utilization is surprisingly low. On average, DTNs were completely idle (i.e., no transfers) for 94.3% of the time in 2017, and 80% of the endpoints were active less than 6% of the online time. Since the hardware requirement is high for high-throughput DTNs, it is desirable that either a DTN resource be properly shared for other purposes or that we could provision resource from other HPC or cloud computing resource pools to work as DTNs.

We present here the design and implementation of an elastic Data Transfer Infrastructure (DTI) for a dynamic Science DMZ that leverages elastic resources for large data transfers. We evaluated our elastic DTI implementation in the Chameleon cloud using Docker containers, GridFTP, and globus-url-copy. Our results show that the elastic DTI can save up to 98% of resources compared with a typical DTN deployment, while experiencing only minimal overhead (approx. 1%). Moreover, half of the transfers finish faster in the elastic DTI than in a typical DTN deployment we used as baseline.

The remainder of this paper is organized as follows. Section II provides background and motivation, and Sections III and IV describe the architecture and design, respectively. Section V presents the evaluation results. Section VI presents related work. Section VII summarizes our approach and its benefits and briefly outlines future work.

II. BACKGROUND AND MOTIVATION

In this section, we provide background on Science DMZ and discuss the limitations of the current Science DMZ model, which serve as the motivation for our work.

A. Science DMZ

A Science DMZ [8] provides a network design pattern for research institutions to support high-speed wide-area data transfers. It was developed based on the observation that local-area networks (LANs) at most institutions are not designed to support large science data flows. Configuration, policies, procedures, and security measures used to support organization’s email, web browsing, procurement and other administrative operations are not suitable for supporting high-speed transfer of large science data. The design pattern prescribes deploying the Science DMZ at the network perimeter of the institution (in order to minimize the number of network devices in the data...
path between the Science DMZ and the wide area network) with the following resources [11]:

- Dedicated data transfer nodes with network capabilities that match that of the wide area network (WAN) and that run high-performance data transfer tools such as GridFTP [12]
- Performance monitoring hosts with network-monitoring software such as perfSONAR [13] to conduct both active and passive network measurements
- Security policies and tools that can be applied specifically to the science-only traffic

B. Limitations in the Science DMZ model

The Science DMZ model has been extensively tested and is being used at many supercomputer centers, national laboratories, and universities. Although it has helped improve the performance of wide area file transfers at many institutions, it suffers from two limitations.

1) Lack of support for real-time/streaming analysis: In the current science DMZ model [8], incoming data from the WAN go into the DTN(s) deployed at or near the network perimeter of the institution and are written either to the DTN’s local disk or to the parallel file system that is shared by the DTN and compute nodes. This model does not work well for science workflows that require real-time data analysis and transfer. Figure 1 shows the streaming data transfer performance of two DTN nodes (1 km apart). The configuration memory-memory is the performance when data is moved between DTN memories, whereas local disk-memory shows the performance when data is streamed from one DTN’s local disk to another’s memory, that is, with an additional disk read operation. We observe more than an order of magnitude performance decrease due to the disk accesses in the local disk-memory configuration. Some compute facilities may operate a high-performance parallel file system; but since this is a shared resource, contention can result in significant overhead. For real-time analysis, it is desirable to stream data directly into the memory of the compute nodes—a use case that is not addressed in the current science DMZ model.

2) Dedicated and static DTNs: The Science DMZ model recommends using dedicated systems as DTNs. AMost DTNs have Globus GridFTP servers deployed on them, which they use as the primary means of data transfer. In our previous work [14], we analyzed the usage logs of GridFTP servers from about 1,800 DTNs to study their utilization. These logs provide the following information: transfer type (store or retrieve), size in bytes, start time of the transfer, transfer duration, IP address of the GridFTP server, number of parallel TCP streams, TCP buffer size, and block size. Additionally, we used port scanning to check whether those DTNs had other high-performance data transfer tools, such as BCP [15], FDT [16], XDD [17], or Aspera [18], that may add extra utilization. We found that less than 1% of the endpoints had other tools installed. Thus, the utilization reported in [14] is fairly accurate for 99% of those 1,800 DTNs.

Working with data from these 1,800 DTNs for the year 2017, we marked a given DTN as active if there was at least one transfer over the DTN; otherwise we marked it as idle. We did this for each minute. We found that, on average, DTNs were completely idle (i.e., there is no transfers) 94.3% of the time. Figure 2 shows the cumulative distribution of the time that DTNs are active. Clearly, the percentage of active time is low. For example, 80% of the DTNs are active less than 6% of the time.

Figure 3a shows the percentage of time that at least one transfer was happening over the 100 mostly heavily used endpoints. To investigate how busy an endpoint is when there is at least one transfer, we assume that an endpoint’s resource utilization is 100% when it gets the maximum aggregate throughput (incoming and outgoing), and we compute the utilization at a given instant as the ratio of the aggregate throughput at the instant to the maximum aggregate throughput observed at the endpoint in the entire year.

Figure 3b shows the different percentile values of the utilization of the top 100 most heavily used endpoints. Clearly,
even when an endpoint is busy, utilization is typically low. These grossly underutilized DTN resources can be put into good use by pooling them with the compute resources and using cloud technologies to acquire them on demand.

III. Architecture

To address the limitations of the current Science DMZ, we propose a dynamic model (see Figure 4) that securely extends the DMZ all the way to the compute nodes and uses an elastic data transfer infrastructure that expands and shrinks dynamically to conserve resources.

The Science DMZ was initially conceived as an isolated resource pool in the perimeter of the network. What is needed now is a way to dynamically connect HPC resources with Science DMZ resources and the WAN. To this end, we propose software-defined networking (SDN) [19] for provisioning these paths on demand. To realize a dynamic Science DMZ in a nondisruptive fashion, we propose to reserve minimal resources that will work as a dedicated DTN; we call this a thin dedicated DTN. Whenever CPU or network utilization increases, additional resources will be dynamically allocated from a pool of compute nodes. We call this an elastic data transfer infrastructure, or DTI.

This paper focuses on the elastic DTI component of our dynamic Science DMZ model. The architecture, shown in Figure 5, is composed of an orchestrator that oversees a pool of compute nodes and agents that reside in each compute node of the pool. These agents maintain communication with the orchestrator over a LAN. The main function of the orchestrator is to maintain a global view of resource utilization across the infrastructure and make decisions about how to allocate resources efficiently. Agents can be seen as an extension of the orchestrator on the physical infrastructure. They collect usage information and report it back to the orchestrator, as well as execute configuration changes on the behalf of the orchestrator. In the following subsections we explain each component in detail.

A. Orchestrator

The orchestrator is composed of four main building blocks: a statistics collector, a resource manager, a decision engine, and a load balancer. The statistics collector aggregates utilization measurements taken by all agents in the physical infrastructure. The resource manager keeps an inventory of all the dynamic resources deployed in the physical infrastructure. The decision engine uses the utilization statistics and the resource inventory to make decisions on whether to provision or remove resources. Similarly, the load balancer uses statistics and inventory information to balance the load of incoming transfer requests.

B. Agents

An agent comprises a monitoring module that collects resource utilization statistics and a provisioning module that provisions or removes resource. The communication between the monitoring module and the statistics collector can be implemented via either a polling model or a push model. In the former, the orchestrator periodically requests statistics from each agent; in the latter, agents send measurement updates continuously to the orchestrator. The provisioning module executes actions only when directed by the orchestrator’s decision engine.

IV. Design

The architectural changes proposed in Section III raise many questions: How do we provision/remove compute nodes from a pool of existing virtualized resources? When do we provision/remove these compute nodes? How do we determine the optimal time to provision/remove them? Basically, our Dynamic Science DMZ has two design parameters:

1) Virtualized infrastructure (how to provision/remove)
2) Provisioning and removal schemes (when to provision/remove)

A. Virtualized Infrastructure

Two scenarios exist in which underutilized resources can be reallocated dynamically. In the first scenario, we assume...
that scientific facilities already have a private cloud platform, such as OpenStack [20] or CloudStack [21], to serve the needs of their users. Here, underutilized resources can be assigned dynamically to the Science DMZ and connected on demand by using SDN. In the second scenario, an existing, underutilized DTN pool can be made available to the rest of the campus as a compute resource through a dynamic Science DMZ. In this paper we focus on the first scenario. We propose to add a container [22] orchestration layer on top of an existing private cloud infrastructure. We recommend containers because of their short provisioning and removal times.

B. Provisioning and Removal Schemes

The question of when to provision or remove resource can be answered by applying different schemes on the decision engine. However, since the focus of the paper is on the elastic DTI architecture rather than the logic of the decision engine, we propose to use thresholds as our decision logic. When the average resource utilization goes above a high threshold, we provision a new resource, whereas we will remove an active resource after the utilization goes below a low threshold. For instance, let us consider an elastic DTI with high and low thresholds of 50% and 10% CPU utilization, respectively. At time \( t_1 \), the infrastructure has two active compute nodes, \( \text{node1} \) and \( \text{node2} \), with CPU utilizations of 65% and 40%, respectively. Since the average CPU utilization of the whole infrastructure is 52.5%, we provision a new compute node, \( \text{node3} \). At time \( t_2 \), \( \text{node1} \) and \( \text{node2} \) remain the same, and the newly provisioned \( \text{node3} \) has 0% CPU utilization. The average CPU utilization is 35%, so no action is required.

When to remove resources is a more interesting question and influence even the provisioning process. Removing a compute node (or container) will interrupt the active transfers running on it. We consider two schemes for removing resources. In the first, we immediately remove resources after the decision engine determines that they are no longer needed. Similarly, for an add decision we immediately add resources. In the second scheme, a compute node drains (i.e., finishes) its active transfers before it is removed. The orchestrator’s load balancer does not assign new transfers to a draining compute node, which will be removed once all transfers on that node are completed. This second scheme gives the orchestrator more options for adding new resources: it can either bring draining containers back online or add fresh new containers. For the rest of the paper, we call the first scheme dynamic (or dyn for short), and the second scheme dynamic with graceful removal (or dynG for short).

For the dynG scheme, we present the pseudocode for provisioning (see Algorithm 1) and removing resources (see Algorithms 2 and 3). The provisioning procedure in Algorithm 1 attempts to bring back draining containers before adding a new container to the infrastructure. The procedure starts by sorting a list of draining containers in ascending order and setting an AddNew flag to True. Then, the orchestrator adds the least-used container on the list of draining and recomputes the average resource utilization of the infrastructure. If the new average is less than the high threshold, the container is added...
to the “bring back” list, the AddNew flag is set to False, and all the containers on that list are brought back online. If the new average is greater than the previous average, we break the loop. Otherwise, we add the container to the “bring back” list and continue iterating over the draining list. If we finish the loop and the AddNew flag is still True, we provision a new container.

Algorithm 1 Resource Provisioning

1: procedure PROVISION( void)
2:   AddNew ← True
3:   BringBackList ← []
4:   drain ← SortedByResourceUsage()
5:   for c in drain do
6:     ActiveList ← c
7:     NewAvg ← CompNewAvg()
8:     if NewAvg < Hi then
9:       BringBackList ← c
10:      undrain(BringBackList)
11:     AddNew ← False
12:     break
13:   else if NewAvg ≥ OldAvg then
14:     break
15:   else
16:     BringBackList ← c
17:   end if
18: end for
19: if AddNew then
20:   AddResource(LeastUsedBareMetal)
21: end if
22: return
23: end procedure

Algorithm 2 Garbage Collector

1: procedure GARBAGECOLLECTOR( void)
2:   for ∀ c in drain do
3:     if Active transfers in c == 0 then
4:       RemoveResource(c)
5:     end if
6:   end for
7:   return
8: end procedure

Algorithm 3 Resource Removal

1: procedure REMOVE( void)
2:   GarbageCollector()
3:   c ← GetLeastUsedContainer
4:   if Active transfers in c == 0 then
5:     RemoveResource(c)
6:   else
7:     c.drain ← True
8:   end if
9:   return
10: end procedure

V. EVALUATION

We evaluated our architecture in the Chameleon cloud [23]. Our testbed is composed of eight bare metal nodes with 48 cores of an Intel Xeon CPU E5-2670 v3 with 2.30 GHz and 128 GB of RAM running Ubuntu 16.04.5 LTS. All nodes are connected to Chameleon’s shared LAN using 10 Gigabit Ethernet interfaces. For our experiments, four nodes work as server DTNs and the remaining four as client DTNs. This configuration provides 40 Gbps of aggregate bandwidth between the server and client sites. All bare metal nodes run Docker version 18.09.0 to manage containers, Globus GridFTP server version 12.12 as DTN software, and globus-url-copy version 9.29 to manage data transfers. These packages are the software stack for our elastic DTI implementation. This globus-url-copy tool lacks many of the features of the Globus transfer service (hosted GridFTP client) and is less user-friendly, but it is more convenient for experimental purposes.

A. Implementation

We implemented the orchestrator and agents for our elastic DTI in Python and the communication between them using gRPC [24]. The agents collect CPU and network utilization statistics with a sampling frequency of one second. The CPU utilization is collected per container, while the network throughput is collected per bare metal server. The orchestrator polls agents every second, computes thresholds, and decides whether a container needs to be provisioned or removed.

Figure 6 shows a schematic of our implementation. For the baseline, each bare metal node on the server side works as a DTN. For elastic DTI, we deployed an agent on each bare metal server and a single orchestrator for the server-side infrastructure.

For our evaluations, we created a trace in which the transfer arrival time follows a Poisson distribution with \( \lambda = 3 \) seconds. Each transfer on our trace uses a dataset that follows the characteristic of real datasets (both file size and dataset size) according to the distribution shown in Figure 7 (see 10 for more details on the distribution). For all transfers, the source is always on the server side, and the destination is on the client side. We use globus-url-copy to execute a file-to-memory transfer because we were not able to obtain a parallel
file system on the Chameleon testbed. Using file-to-memory transfers allows us to have an identical copy of the dataset on each bare metal server, while avoiding overwrites on the client side. Upon a transfer failure, we resend the entire dataset, an action that impacts the dyn scheme most. We plan to address this limitation in future work.

B. Evaluation Scenarios

Considering the two schemes defined in Section IV (i.e., dyn and dynG) and a combination of high and low thresholds, we defined six evaluation scenarios as shown in Table I. Thresholds are defined as percentages of average resource utilization (e.g., CPU usage or network throughput), so measurement metrics can be interchanged without affecting the orchestrator’s logic. We use CPU utilization as the decision metric. As mentioned before, each bare metal node on the server side works as a DTN. For the baseline, we limited their resources to 12 cores and 96 GB of RAM using Docker containers to match the specifications of a typical DTN deployment [25].

![Fig. 6: Elastic DTI implementation on Chameleon](image)

![Fig. 7: Cumulative distribution of real transfer size to/from the BigSite’s DTNs](image)

![Fig. 8: CPU resources saved with respect to a typical DTN deployment](image)

<table>
<thead>
<tr>
<th>Name</th>
<th>Scheme</th>
<th>High (%)</th>
<th>Low (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>dyn-30/10</td>
<td>dyn</td>
<td>30</td>
<td>10</td>
</tr>
<tr>
<td>dyn-50/10</td>
<td>dyn</td>
<td>50</td>
<td>10</td>
</tr>
<tr>
<td>dyn-70/20</td>
<td>dyn</td>
<td>70</td>
<td>20</td>
</tr>
<tr>
<td>dynG-30/10</td>
<td>dynG</td>
<td>30</td>
<td>10</td>
</tr>
<tr>
<td>dynG-50/10</td>
<td>dynG</td>
<td>50</td>
<td>10</td>
</tr>
<tr>
<td>dynG-70/20</td>
<td>dynG</td>
<td>70</td>
<td>20</td>
</tr>
</tbody>
</table>

C. Results

In this section we present the results of evaluating the six elastic DTI scenarios.

1) CPU Resources Saved: Figure 8 shows the percentage of CPU resources saved by dyn and dynG schemes compared with a typical DTN deployment. For the baseline we have 48 cores in total (12 cores \(\times\) 4 bare metal nodes). In our implementation, we limit one container to one core in the bare metal node. We assume that Docker enforces a hard limit on the CPU utilization, so we consider that a container deployed is equal to one core used. We obtain the resource utilization over the makespan of the experiment by integrating the container count over time. Then, we compute the percentage of CPU resources saved. We observe that this is directly proportional to the high threshold (i.e., scenarios with a higher threshold saved more CPU resources). For instance, the dynG-70/20 scenario saved 98% of the CPU resources, while the dynG-30/10 scenario saved approximately 96.75%. No significant difference is found between dyn and dynG in the amount of CPU resources saved for both the 50/10 and 70/20 scenarios. On the contrary, dynG is slightly better than dyn for the 30/10 scenario.

Figure 9 shows the CPU usage and container count over time for both the dyn-70/20 and dynG-70/20 scenarios. We observe that container provisioning and removal operations follow the CPU usage pattern. Furthermore, the dynG scheme presents fewer provision/remove operations than does the dyn scheme, since containers pass through a draining phase before being completely removed and may be brought back if needed.

2) Slowdown: This metric, commonly used in compute scheduling, is the factor by which a compute job is slowed relative to the time it would take on an unloaded system. Here, we define slowdown as the factor by which a transfer is slowed relative to the time it would take with static dedicated DTNs (a.k.a. baseline). A commonly used variant of slowdown, called bounded slowdown, limits the influence of extremely short jobs on slowdown by measuring the slowdown of such jobs relative to an interactive threshold or bound, rather than
the actual runtime. We adapt that measure here and define bounded slowdown \( (SD_b) \) as follows:

\[
SD_b = \frac{\max (T_d, \text{bound})}{\max (T_b, \text{bound})},
\]

where \( T_b \) is the baseline transfer time and \( T_d \) is the transfer time in the elastic DTI. We use a bound of 5 seconds. Figure 10 shows the distribution of bounded slowdown for \( dyn \) and \( dynG \). Surprisingly, \( dyn \) performs better than \( dynG \). One would expect \( dynG \) to perform better than \( dyn \) because \( dynG \) removes the containers in a graceful manner. Later in this section we present additional analysis to explain this counterintuitive results. From Figure 10, we observe that the median slowdown of transfers in all \( dyn \) scenarios is close to 1, which implies that roughly half the number of transfers has better performance than does the baseline and that the performance of the rest of the transfers is worse than baseline. Among the 50% of the transfers that performs poorly in \( dyn \), half of the transfers have a slowdown less than 2, and almost all have a slowdown less than or around 4.

To understand why the performance of some of the transfers in \( dynG \) is worse than that of \( dyn \), we look at the network utilization of each (bare metal) server. Figures IIa and IIb show the baseline, \( dyn \), and \( dynG \) results, respectively, for the 30% and 10% thresholds. We see that both the baseline and \( dyn \) finish in approximately 2,000 seconds (they both use more network bandwidth than \( dynG \)). On the other hand, \( dynG \) takes close to 3,000 seconds to finish. Moreover in \( dynG \), after 1,000 seconds, only two servers (server-3 and server-4) drive most transfers, and after 2,200 seconds server-3 is the only active bare metal. These observations tell us that letting containers “drain” may have undesired effects in terms of slowdown, as (possibly large) transfers and containers may get stuck on a saturated bare metal. Furthermore, the action of immediately removing a container in the \( dyn \) scheme may help stuck transfers to restart on a container in a less-congested bare metal node.

3) Overhead: Figure 12 shows the overhead as the number of extra bytes transferred for different \( dyn \) scenarios with respect to the baseline. We note that only \( dyn \) incurs overhead and that \( dynG \) does not because it removes a container only when it does not have any active transfers. We also observe that the \( dyn-70/20 \) scenario has a high overhead compared with the other scenarios. Figure 9a shows a drop in CPU usage below the low threshold in the middle of the experiment, followed by a spike close to 100%. This event may have caused the interruption of several transfers, besides the saturation of the last active container with new transfers. However, more instrumentation on the testbed and more investigation are needed in order to provide a definitive answer. Also, as noted in §V-A, upon a failure of a transfer, the entire dataset is retransmitted, incurring high overhead for the \( dyn \) scheme.
VI. RELATED WORK

Wide-area file transfer performance has been studied extensively. Studies focus mostly on explaining performance [14] and on optimizing performance by tuning application-level parameters [26–30]. For example, Liu et al. [27] proposed a reinforcement machine learning-based method to maximize aggregated DTN throughput. Jung et al. [31] proposed a serverless data movement architecture in which they bypass DTNs, the file system stack, and the host system stack and directly move data from one disk array controller to another. Experimental results show that our proposed architecture is feasible and outperforms the traditional architecture significantly.

A new trend emerging across university campuses involves deploying Science DMZs to support science drivers that involve, for example, data-intensive applications needing access to remote instrumentation or public cloud resources [32]. Several use cases from universities, supercomputing centers, and research laboratories highlight the effectiveness of the Science DMZ model in diverse operational settings. Dart et al. [33] claim that the Science DMZ model vastly improves collaboration, accelerating scientific discovery.

Crichigno et al. [8] reviewed fundamental network concepts that have a large impact on Science DMZs, such as router architecture, TCP attributes, and operational security. They examined protocols and devices, from the physical cyber infrastructure to application layer tools and security appliances, that must be carefully considered for the optimal operation of Science DMZs. The importance and criticality of DTNs in the Science DMZs were clarified in their work.

Saha et al. [34] evaluated the performance of containers for scientific workflows in cloud environments. The authors provided guidelines that application developers can follow to use containers in scientific work flows. Our decision of implementing our elastic data transfer infrastructure over Docker is supported by their results.

Calyam et al. [32] presented a “campus Science DMZ reference architecture” for adaptively managing host-to-host accelerated flows of multiple researchers over wide-area overlay networks with shared underlay infrastructure components. The proposed novel approaches can handle challenges of policy specification, security enforcement, and performance engineering within Science DMZs in order to support diverse accelerated flows on a scalable/extensible basis. A multi-disciplinary case study of a bioinformatics science driver application in a double-ended campus Science DMZ testbed was also shown to prove the effectiveness of the model.

VII. CONCLUSION

The Science DMZ as a network design pattern has had a notable impact on the science community by improving the performance of wide-area file transfers significantly. Yet, it falls short in efficient utilization of data transfer nodes and is not suitable for some emerging scientific data transfer use cases. We proposed here a dynamic Science DMZ architecture that extends all the way to the HPC resources and can be created and removed on demand. We also presented a design and implementation of an elastic data transfer infrastructure that resides in the dynamic Science DMZ and grows and shrinks based on demand. We realized an instantiation of this elastic DTI in the National Science Foundation’s Chameleon testbed and showed that up to 98% of the CPU resources can be saved when compared with a typical DTN deployment. The overhead in terms of extra bytes transferred is as low as 1%, with the slowdown incurred by around 50% of the transfers being compensated by speedups for an equal proportion of transfers.

ACKNOWLEDGMENTS

This material is based upon work supported by the U.S. Department of Energy, Office of Science, under contract number DE-AC02-06CH11357.

REFERENCES


[5] “Clemson University continues data center innovation with next generation network built on big switch technology.”

http://vapor.gatech.edu/

[7] “State University of New York selects Oracle cloud at customer to modernize the learning experience for students and staff.”


[9] “Blue Waters - Data Transfer.”
https://bluewaters.ncsa.illinois.edu/data
