Cabinet: Managing Data Efficiently in the Global Federated File System

Avinash Kalyanaraman, Andrew Grimshaw
Department of Computer Science
University of Virginia
Charlottesville, VA 22904
{ak3ka, grimshaw}@virginia.edu

***Abstract*—With ever expanding datasets, efficient data management in grids becomes important. This paper describes *Cabinet* which employs two techniques for efficiently managing data in grids- a caching system and a new file staging approach called *coordinated staging*. The caching system is designed based on the characteristics of grid applications. Coordinated staging is based on the BitTorrent Protocol model and is specifically designed for High Throughput Computing (HTC) applications, a common use-case for grids. In coordinated staging, each site that is assigned to execute an individual job of the HTC application treats other execution sites as potential replica-stores. In our evaluation, we show that coordinated staging lowered the download time of a file by 3.85x, and increased the throughput of the download by 2.86x over the conventional approach of file transfer from a single source.**

Keywords—grids; file staging; big data; distributed file systems

I. INTRODUCTION

Scientific collaborations and the increasing computational needs of applications have warranted the development of grids [1]. The ability of grids to provide a larger resource pool than what is available at a single site has helped solve problems previously believed to be intractable. On the other hand, the data-size used by applications has been increasing. Seidel [2] claims that the data generated each year is greater than the sum of the generated-data over all previous years. Thus, managing such large amounts of data in computational grids becomes important. Researchers are interested in both analyzing this data and harnessing the vast computational resources available via the grid. The consequence is a situation where the data and computation are not collocated, warranting the need for efficient data management techniques.

At the same time, storage is becoming cheaper and the amount of storage available for researchers at their campuses and supercomputing sites has been increasing. This storage space can be used to cache and replicate input files. Caching and replication are two traditional techniques for efficient data management in distributed systems. This work, *Cabinet* deals with caching in the context of a computational grid.

Commonly scientists' datasets are located at their institutions. Two techniques [3] exist for moving data to the compute site (referred henceforth as simply site): pre-staging input files [4] and on-demand input file access [3, 5, 6]. Pre-staging moves all the input files required by a job to the site before the job starts running. Typically users have allocations indicating the amount of wall-clock time they can use at a site. In this approach, no allocations are consumed during stage-in. On-demand access recreates the "home filesystem" of the user at the site by forwarding read requests to the institutions. However, this method suffers a serious limitation. The execution time of the application increases as it blocks during reads waiting for data to be fetched from "home". As users are often charged allocations from the time their job starts executing, this method consumes more allocations than pre-staging. Since allocations are valuable entities obtained via a strict peer-review process, it becomes natural for users to prefer pre-staging. Consequently, *Cabinet* deals with the pre-staging approach and employs a new technique called *coordinated staging* which makes pre-staging faster.

The work has been done in the context of the Global Federated File System (GFFS) [7]. The GFFS is a web-services based file system for grids. GFFS provides an ability for campuses, supercomputing centers, industrial centers etc to operate under a single, global path-based namespace without requiring the data owners and application developers to change the way they store and operate on the data.

In grids, the term *resource* denotes an implementation of a particular service. Since the GFFS is a standards-based filesystem, each resource can be understood as an implementation of a standard. For example, every file is a resource implementing the ByteIO standard [8] and every directory is a resource of the Resource Namespace Service [9] specification. Being a wide-area distributed filesystem, the GFFS suffers from the limitations of the network. The first part of this work: *coordinated staging* treats the GFFS from a computational grid's perspective. The second part of this work: *caching* treats the GFFS from both a computational grid and data grid's perspective.

The rest of this paper is organized as follows: Section II motivates the problem, Section III describes a new file staging technique called *coordinated staging*, Section IV describes the caching system and its working, Section V presents the implementation, Section VI shows the evaluation, Section VII discusses the related work and Section VIII summarizes the work.

II. MOTIVATION

In this section, we describe the need for caching and fast file staging. For the rest of this section, the following notations will be used. Let J be a job submitted by user U. Let $F = \{F_1, F_2, \ldots, F_n\}$ be the set of input files required by J, such that $F \neq \emptyset$. Let J' denote a reuse job. A *reuse* job is defined to be a job that is submitted after J finishes execution. Let F' be the set of

978-0-7695-5083-1/13 $25.00 © 2013 IEEE
DOI 10.1109/eScience.2013.36

input files required by *J'*, and $F \cap F' \neq \emptyset$. Let *U'* denote any user except *U*. *J'* can be understood to belong to *U* or *U'*, unless stated explicitly.

A. The Need for Caching

Caching is important in computational grids for two reasons: input files tend to be shared by jobs, and input files tend to be reused across jobs.

1) Sharing: Sharing occurs when a "single job submission" consisting of multiple individual tasks, uses a common input file(s). Such jobs form a sub-category of high throughput computing (HTC) applications [10], which are an important use-case of computational grids. HTC applications refer to a class of applications which involves users running multiple copies of their programs simultaneously with no communication between the individual tasks. In such cases, we would not want each individual task to stage-in the same input file. A single stage-in into the site must occur. An example where the same input files get used repeatedly is parameter sweep applications. For e.g., Garzon [11] showed the effectiveness of caching for a protein docking application.

2) Reuse: Reuse may occur for multiple reasons. In some cases, users learn input parameters from an initial set of runs. Alternatively, a user may modify his algorithm based on the initial results. The subsequent "modified" program will use the same set of input files and libraries. In other words, *trial-and-error* which is inevitable in scientific research results in reuse. At times, generating input files could be expensive, leading to unavoidable reuse. Reuse can also occur due to jobs failing because of programming errors, scripting errors or execution node failure. While these are use-cases of input file reuse from an individual user or his research group's perspective, some input files like public databases may be common across several research groups, creating a greater potential for reuse.

Storage is cheap enough to not hinder the caching and replication of input files that are shared and reused, instead of copying them across the network many times.

B. The Need for Fast File Staging

Despite caching, it is still necessary for files to be staged-in quickly (the first time), so that jobs can begin execution. Staging may be necessary even during reuse. There are many reasons why the reuse-job *J'* need not execute on the cached site(s) (sites of the initial run *J* by user *U*):

Primarily, the metascheduling algorithms need not always be data-aware. For instance, they may take queuing delay and the number of available processors into consideration. For example, GridWay [14] schedules jobs onto resources using a user-specified ranking model. Secondly, scheduling constraints may prevent *J'* from executing on the cached sites. For example, the memory or processor requirements of *J'* may not be satisfiable by the cached site. Finally, *J'* may belong to a different user *U'*, who may not have allocations or permissions to run on the cached site(s).

Hence, given that reuse need not necessarily occur on the cached site, a mechanism for fast staging is needed so that jobs can begin execution quickly. Moreover, we will also need a fast staging mechanism for those files which exist as a single copy (i.e those files which have not been cached or replicated). This need for fast file staging is addressed by *coordinated staging*.

III. COORDINATED STAGING

In this paper, we present coordinated staging, a new technique for fast staging, specifically designed for HTC applications. In coordinated staging, each site assigned to execute a sub-job of the HTC application treats the other sites as potential replica-stores.

To understand coordinated staging, let us first take a look at a typical HTC job-submission in grids. The user submits an HTC application-description file to a metascheduler [15]. The metascheduler parses this file and extracts the description of each individual task of the HTC job.

This description includes among other things, the list of files required to be staged into the site for successful execution of that (individual) job. The metascheduler assigns each (individual) job to a site based on a scheduling algorithm. Typically, these jobs run at multiple sites primarily because there is a limit on the number of jobs a user can have in the local queuing system of a site at any given instant of time. In other words, even though the execution-service running at the site may interface to a queuing system, the service cannot *qsub* an unbounded number of jobs on behalf of a user. For example, the maximum number of jobs that may be queued by a user on *TACC-Ranger* is 50.

As *Cabinet's* data-management strategies are employed in the context of the GFFS, *coordinated staging* is designed for the common case of the input file stage URI being a GFFS path. The common way of staging files which are not cached or replicated is for each site to download them from the lone copy. This lone copy which is explicitly created by the user (as opposed to an implicit creation by caching or replication) is called the *primary*. Such an approach has a big disadvantage: all sites fetch only from the primary making it a potential bottleneck and this primary may not be "close" to the downloading sites. The terms "close" and "far" refer to network proximity and bandwidth.

However, it may be possible that the sinks (sites) are closer to each other than to the source(s). For example, in DEISA [16] and XSEDE [17] the sites are connected to each other by high bandwidth links. In such cases, we would want sites to download as much content as possible from one another, as they are all downloading the same content. *Coordinated staging* achieves this by making each site that downloads a file *f* treat other sites as potential replica-stores for *f*.

To describe how each site treats other sites as potential replica-stores, we use the following notation. Let $J = \{J_1, J_2, ..., J_n\}$ be an HTC application, where J_i $(1 \leq i \leq n)$ denotes a sub-job of the HTC application. Let $S = \{S_1, S_2, ..., S_s\}$ be the set of sites available to the metascheduler to schedule jobs on. The metascheduler schedules each job J_i onto some site S_j $(1 \leq j \leq s)$. Each job J_i requires some set of files F_i to be staged into S_j for successful execution. While scheduling J_i onto S_j, the metascheduler also informs S_j about the set of sites who will stage-in each file *f*, where $f \in F_i$. The metascheduler ensures that each site which stages-in *f* sees this same set of sites in the same order. As we shall soon see, this order is critical to the working of *coordinated staging*.

Let $S_f = \{ S_1, S_2, \ldots, S_k \}$ be the set of sites that will be downloading *f*. The goal of *coordinated staging* is to maximize the traffic on faster peer links (links providing faster download rates). Clearly, this warrants some kind of ordering in the way the peers download a file. Consequently, each site $s \in S_f$ queries *f* and obtains its size *sz*. This *sz* amount of contents is implicitly partitioned among the S_f sites onto S_f disjoint chunks based on the order provided by the metascheduler. For example if *sz* = 400GB and $|S_f| = 4$, then a vector of the form $<(S_1, f, 0, 100GB), (S_2, f, 100GB, 100GB), (S_3, f, 200GB, 100GB), (S_4, f, 300GB, 100GB)>$ is constructed by all the sites. Each element of the vector is of the form (site, file, offset, size), meaning that the *site* is responsible for downloading *size* amount of data from *file* starting at *offset*. This chunk, which each site is said to be "responsible for", is referred to as the *primary chunk* (PC) of that site. The *PC* can be downloaded from the primary or replica(s) using the *download algorithm*, explained in Section III-B. The site responsible for the *PC* is called its *master*. All non-PCs of a *master* are referred to as *secondary chunks* (SCs). It is important to note that the download order of chunks does not matter, as long as they are arranged correctly at the sink.

The number of chunks for a file *f* equals the number of sites downloading *f*. Having obtained the *PC*, each site has to download the remaining $|S_f|$-*1 SCs*. Each site picks one *SC* at a time and downloads it completely before moving to the next one. As we are dealing with big datasets, and the duration of a chunk's download is non-trivial, each site following the same order of *SCs* could result in all of them attempting to read from that chunk's *master* (simultaneously). This will limit that site's share of bandwidth coming out of the *master*. This is an incarnation of the *First Mile Problem*. Since the order of the download of chunks do not matter, each site shuffles the order of the *SCs* by randomly picking a *SC* and downloading it to completion. Each *SC* is downloaded like the *PC* using the same download algorithm but by also including that *SC's master* as a potential replica (to explore) and the perceived bandwidth knowledge with the primary or replicas from any previous chunk's download associated with *f*. We refer to this approach as *unoptimistic coordinated staging* (*unopt-CS)*. The case for files of non HTC applications becomes a special case for coordinated staging with no peers.

A. Optimistic coordinated staging

We have also created an extension to *unopt-CS* called *optimistic coordinated staging (opt-CS)*. Since the order of chunk download has been randomized, it is possible for a site's peer who can provide a better download rate than the chunk's master or primary, or replicas to have already downloaded that chunk of interest. Consequently, *opt-CS* extends *unopt-CS* wherein each *SC* is downloaded using the download algorithm by (optimistically) including all the peers provided by the metascheduler as potential replicas. An attempt to read from a potential replica that has not yet downloaded the required content will result in a fault being returned. Such a site will no longer be considered as a potential replica for the download of that chunk (not the file).

B. The Download Algorithm

The goal of our download algorithm is to reduce the time taken by a sink to download a chunk that has been replicated. An advantage of this algorithm is that it does not use any explicit services for network monitoring or forecasting. Before explaining the algorithm's working, we first introduce the phases and the sub-phases of the algorithm: *Explore Phase(EP), Download Phase(DP)*, *Unchoke Phase(UP)*, *Ignore Sub-Phase*(*ISP)* and *Heed Sub-Phase(HSP).*

1) The Phases and Sub-phases of the algorithm:

Ignore Sub-Phase (ISP): It is the first step of *EP and UP* (explained below). In this sub-phase, the sink downloads D_i amount of data from a replica without noting its perceived bandwidth. The goal here is to warm-up the disk or filesystem caches at the source, so that comparisons can be made by reads in *EP* and *UP* with warmed-up reads of *DP*.

Heed Sub-Phase (HSP): It is a sub-phase of all three phases. Here, the sink downloads D_h amount of data from a replica and notes its perceived bandwidth. The connection and caches are warmed-up before this sub-phase is invoked.

Explore Phase (EP): This phase is run on those replicas about whom the sink wants to acquire knowledge on its perceived bandwidth. This phase starts with the *ISP* for warm-up purposes. *ISP* is followed by *HSP*, where the actual knowledge on perceived bandwidth is acquired.

Download Phase (DP): In this phase, the sink downloads from a replica identified to be the fastest via exploration or unchoke. This fastest replica is called the *chief replica*. This phase has two sub-parts. During the larger first sub-part, D_d amount of data is downloaded from the *chief replica* without noting the perceived bandwidth. It is different from the *ISP* in the following ways. While the goal of *ISP* is to warm-up, this sub-part is a consequence of that replica being the fastest. Secondly, D_d is much larger than D_i of *ISP*. The second sub-part is *HSP*. The bandwidth perceived in *HSP* is used in *UP*.

Unchoke Phase (UP): Since the perceived bandwidth with a replica is time-variant, each sink periodically selects a replica, and checks if it offers better bandwidth than the *chief replica*. This phase starts with the *ISP* to warm-up the chosen replica. The *ISP* is followed by the *HSP*, where the bandwidth offered by this chosen replica is noted. If this bandwidth is greater than the chief replica's bandwidth noted during its *DP*, then the unchoked replica becomes the new *chief replica*.

2) Methodology

The inputs to the algorithm are: the starting byte offset to download, the last byte offset to read, an exploration set (represented by E_n) denoting the set of sources about whom the (invoking) application has no knowledge about the perceived bandwidth, and an explored set (represented by E_d) denoting the set of sources about whom the (invoking) application has knowledge about the perceived bandwidth.

The algorithm begins by first running *EP* on each replica of E_n. The replica offering the best bandwidth among those in E_d and E_n becomes the *chief replica*. The remaining replicas are said to be *choked*. Next, the *DP* is run on the *chief replica*. After the *DP* is complete, the replica at the head of the *choked list* is removed. This replica is called the *unchoked replica*. *UP*

is run on this *unchoked replica* to see if it offers better bandwidth than the *chief replica*. If it does not, the *DP* is once again run on the *chief replica*. If it does, then the *chief replica* is appended to the *choked list*, and the *DP* is run on this new *chief replica*. This sequence of *DP-UP-DP* continues until the entire chunk is downloaded.

C. Improving Single File-Transfer Speed via Parallel-TCP

To improve the download speed from a single replica, we employ the proven technique of parallel TCP [18] in which multiple TCP streams are used to download from the source to mitigate the underlying slowness of TCP. On the other hand, a common way for users to access files on the GFFS is via FUSE [19]. Consequently, we also modified the GFFS-aware FUSE-driver to use parallel-TCP.

IV. The Caching System

In this section, we explain the working of the caching system. Before looking at the working, we shall explain how the execution-site is organized to perform caching.

A. Compute-site Architecture

A private working-directory is associated with each individual job. This directory is wiped out when the job completes. In existing implementations, the input files required by the job are staged into this directory. Besides the working-directory, there is a cache directory which stores the cached input files. This architecture is similar to the one used in the ARC grid middleware [20]. We next explain how re-stage-in is prevented during sharing and reuse.

B. Sharing

For this prototype implementation, we assume that input files are not modified by jobs during execution. This assumption is made because currently there is no standard way [15] of denoting such read-only input files. In order to detect sharing, the sites require a way to identify individual jobs of an HTC application. Consequently, the metascheduler generates a unique identifier for every HTC application. All individual jobs of a given HTC application have the same identifier. If multiple individual jobs are scheduled onto the same site, only one of the jobs will stage-in the shared input file to its working directory. The remaining jobs will create hard-links to this location. Creating links within the working directory of jobs of a single HTC application ensures that if any user incorrectly specifies his file to be read-only, then only his jobs get affected. If a job requires a certain file *f*, but other jobs of the same HTC application (at that site) requiring *f* have completed execution, then a link cannot be created. The staging is then treated like a reuse (explained below) by downloading *f* on a cache-miss.

C. Reuse

One of the goals of the caching sub-system is to prevent "big" input files from being staged-in during reuse. In order to achieve this, whenever a site stages in an input file, it caches that file. Only files with sizes greater than a certain *caching threshold* are cached. Small files are not cached for two reasons - owing to their small size they can be quickly re-staged during reuse. Moreover, even if individual jobs of an HTC application require the same file, a re-stage-in can be prevented (as explained in Section IV-B). In ARC, cached contents are visible only to jobs running on that site. However, *Cabinet* makes cached files globally visible. This results in both availability and performance benefits.

At the same time, the bigger input files generally tend to be read-only. This is because scientific datasets (representative of the "big" input files) are typically of the write-once-read-many type [21]. Data is collected using sensors, telescopes etc, released to the public, and then become effectively immutable. Typically, a write corresponds to a completely new version of the dataset being created. As a result, this globally visible cached file is treated as a *read-only replica* (*R-only replica*) which accepts reads from any client (subject to access control) but does not permit writes by anyone. Since a write generally corresponds to newer version of the file being created, we do not propagate changes on the primary or any non read-only copy (referred henceforth as *read-write replica* or *RW-replica*) of the file for three main reasons: a) There is no guarantee that the new version will be used before being evicted, b) To accommodate the update, other cached files might get evicted, which could result in unnecessary restaging (during reuse), and c) Simply destroying the cached-copy on an update notification results in simpler cache management. Consequently, the *RW-replica* notifies the *R-only replicas* of an update. The cached-copy destroys itself, when it receives this invalidation message. The details of the working are explained in Section IV-E. It is to be noted that the terms cached file and *R-only replica* mean the same.

D. Write vs Update

We first define the terms *write* and *update* with respect to a replica. A *write* on a *read-write* replica *R* means a user performed an explicit write on that replica. An *update* on a *read-write* replica *R* means a user performed a write on some replica *R′* (R' ≠ R), and *R′* propagated the write to *R*.

We next explain how the caching sub-system works to handle both reuse and global visibility of cached files.

E. Methodology

Once a decision has been made to cache an input file, the site first creates an empty cached-copy in the cache-directory. Since the cached copy will be globally visible, it is important to ensure that, if a replica is inaccessible to some user *U*, then the cached copy should also be inaccessible to *U*. Consequently, the (empty) cached-file subscribes to one of the *RW-replicas* to notify it on any access control change, and then sets its access control to that of the *RW-replica*. Next, the cached copy subscribes to all *RW-replicas* of the file for both *writes* and *updates*. It is necessary to subscribe to both *writes* and *updates* because the consistency model employed in GFFS is *eventual consistency* [22]. In our notification scheme, the publisher stores the notification message in persistent storage until it has been acknowledged (by the subscriber). After successful subscription, the *R-only* replica becomes globally visible for reads, and asynchronously downloads the contents (via coordinated staging). Any attempt to read a

section of the file that has not yet been downloaded, results in a fault being returned.

Since storage is a finite entity, a replacement algorithm is associated with caches. In our caching system, when the amount of storage used reaches a *High Water Mark*, files are evicted using a simple *Least Recently Used* algorithm until a *Low Water Mark* is reached.

V. IMPLEMENTATION

In this section, we explain how the caching sub-system and coordinated staging are implemented in GenesisII [23]. GenesisII, a web service standards-based grid middleware, is the first realization of the GFFS. Since GenesisII and GFFS are both standards-based, we first overview a few standards that helps understand the implementation better.

A. Standards

WS-Addressing Endpoint references (EPRs) [24] are a standard way to represent and address web service endpoints (namely resources). One of the elements of the EPR is the *Metadata.* The *Metadata* element contains extra information about the resource that can be used by clients as hints.

WS-Naming [25] extends WS-Addressing by providing means for transparent failover, replication (or caching) and migration. Since EPRs cannot be compared, they cannot be used as a cache key. WS-Naming addresses this limitation by providing means to uniquely "name" a resource via an *Endpoint Identifier* (EPI). This EPI is embedded within the *Metadata* of an EPR. For example, when a file (*ByteIO* resource) is replicated, all replicas have different EPRs but share the same EPI. WS-Naming also states that associated with each resource can be a *resolver*, which stores the EPR of the primary and replicas of that resource. When queried with an EPI, the resolver returns an EPR from this set, thus providing a way to achieve transparent failover.

The Basic Execution Service (BES) [12] describes a standard way for creating, monitoring and controlling jobs at execution sites. Depending on the implementation, a BES may execute the job on a single computer, a cluster interfaced by a queuing system, the cloud etc.

B. Caching and Coordinated Staging Implementations

Coordinated staging treats each site as a potential replica-store. Consequently, there needs to be a way for peers to address and read from replicas in this replica-store, Hence, each BES resource incorporates within the Metadata of its EPR, the EPR of another service called the *Fast Stager Service* (FSS). FSS behaves like a *resolver* by matching a given EPI with the set of EPRs in that replica-store, and returns the EPR corresponding to the replica. How the FSS builds this set of EPRs is explained later. It is to be noted that no access control check is performed by FSS. All authentications are performed by the replicas themselves.

We next explain how each site acquires the FSS EPR of other sites. Complying with the BES standard, to create a job on a BES, the metascheduler makes a *Create Activity* call on that BES passing an *Activity Document* describing the job. Inside the *xsd:any* element of the Activity Document, the metascheduler inserts the peer knowledge - i.e for each file *f* required by the activity, the metascheduler inserts the FSS EPRs of those sites who will also stage-in *f*. The HTC application identifier is also inserted within the *xsd:any* element. At the site, each individual job takes the help of the *Stager Manager* and *Download Manager* for staging-in.

The *Stager Manager*(SM) prevents re-stage-in during sharing by tracking the files staged-in (or being staged-in) by the sub-jobs of a HTC application, and their corresponding working directory paths. If a job requires a file *f* that has already been staged-in (or is being staged-in) by another job of the same HTC application, then *SM* creates a hard-link to that job's working-directory location of *f*. Files which cannot be linked are sent to the Download Manager for download.

The *Download Manager*(DM) interacts with the cache and performs coordinated staging. In our current implementation, only cacheable files undergo coordinated staging. In other words, this means the "*replica-store"* of a site is actually its cache. The DM first downloads those files for which that site has to behave like a peer in coordinated staging. This is done by looking at the *Activity Document*. Before downloading a file *f*, the DM checks the cache using the *f*'s EPI as the cache-key. On a cache-miss, the *DM* creates an empty ByteIO resource [8] (corresponding to *f* and having the same EPI as *f*), and sets up subscriptions as explained in Section IV-E.

The *DM* registers this ByteIO EPR with the local FSS, making *f* addressable by the peers. The *DM* partitions *f* into chunks equaling the number of FSS EPRs for *f* in the *Activity Document*. The *DM* identifies its *primary chunk* by comparing the EPI in the FSS EPR of the BES (associated with the job it is downloading) with the EPIs within the FSS EPRs associated with *f* in the *Activity Document*, and downloads it into the cache. Next, all the (randomized) *secondary chunks* of *f* are downloaded using coordinated staging. In *unopt-CS*, the *DM* asks the FSS EPR associated with the chunk's *master* (inferred from the Activity Document) for its local ByteIO EPR corresponding to the EPI of *f*. This EPR is treated as a replica while downloading that *secondary chunk*. In *opt-CS*, the *DM* asks the *masters* of all *secondary chunks* for their local ByteIO EPR corresponding to the EPI of *f*. These EPRs are treated as replicas for each secondary chunk. After downloading a cacheable file completely, its EPR is registered with the resolver (if one exists) making it visible beyond the peers. The file is then copied from the cache to the working directory of that job.

VI. EVALUATION

The experimental setup consisted of eleven geographically distributed sites. Table 1 describes the configuration of each site. Table 2 shows the average bandwidth perceived by each site with every other site.

In coordinated staging, each site "chunkifies" the file to be staged-in, and downloads chunks of the file in a random order. This results in seeks and writes to arbitrary offsets. We first evaluate the overhead of writing to random offsets of an empty file. Figure 1 shows that the overhead of seeking is negligible and the time taken to write at an arbitrary seek offset is nearly constant. This is because seeking generally

Execution site	Machine Type	Cache filesystem type
Texas Advanced Computing Center (TACC) - Alamo	CentOS 5.7	ext3 mounted via NFS
San Diego Supercomputing Center - Sierra	Red Hat Enterprise Server 5.8	ZFS mounted via NFS
Indiana University – India	Red Hat Enterprise Server 5.8	ext4 mounted via NFS
University of Virginia CS - Romulus	Ubuntu 10.04	ext3 mounted via NFS
University of Virginia CSE - UVACSE	Ubuntu 12.04	ext4
University of Virginia University Datacenter - UDC	Ubuntu 10.04	ext4
Amazon Small Instance UK Availability Zone 1a – UK1	Amazon Linux AMI	ext3
Amazon Small Instance UK Availability Zone 1b – UK2	Amazon Linux AMI	ext3
Amazon Small Instance UK Availability Zone 1c – UK3	Amazon Linux AMI	ext3
Amazon Small Instance Japan	Amazon Linux AMI	ext3
Amazon Small Instance Singapore – S'pore	Amazon Linux AMI	ext3

Table 1 Experiment Pool

Source \ Sink	Sierra	Alamo	India	Romulus	UDC	Uvacse	UK1	UK2	UK3	Japan	S'pore
Sierra	-	4.42	6.83	6.91	10.69	7.18	3.25	2.81	2.07	4.96	2.93
Alamo	8.53	-	8.10	6.93	8.26	6.82	4.19	3.79	3.26	3.30	2.24
India	9.01	5.20	-	7.49	17.10	7.56	4.93	4.82	4.47	3.15	2.40
Romulus	9.93	6.22	5.42	-	22.73	34.28	4.19	3.66	3.42	2.42	2.32
UDC	12.44	7.81	5.70	9.17	-	9.12	4.90	3.53	3.47	2.69	2.22
UVACSE	10.54	11.16	16.15	70.95	36.44	-	7.73	6.17	5.57	3.57	2.72
UK1	4.45	2.44	5.06	6.39	8.61	5.73	-	13.49	13.47	2.98	2.18
UK2	4.41	2.32	4.92	6.29	8.14	5.19	12.99	-	13.16	2.96	2.16
UK3	4.41	2.30	4.96	6.33	8.20	5.19	13.17	13.21	-	2.97	2.03
Japan	5.44	2.00	3.01	4.56	4.20	4.00	3.41	2.87	2.82	-	4.18
S'pore	2.80	1.88	2.38	3.45	3.22	3.25	2.80	2.27	2.25	4.99	-

Table 2 Average Perceived Bandwidth between sites in MBPS

Parameter	Value
Number of parallel TCP Streams	4
Ignore Sub-Phase D_i	32MB
Heed Sub-Phase D_h	64MB
Download Sub-Phase D_d	3GB - 64MB = 2.94GB

Table 3Parameters used in the experiment

involves modifying only an in-memory variable which tracks the byte-offset of the next read or write operation on the file. Typically, no storage is allocated for byte-offsets within the hole (which is created while seeking beyond end-of-file).

In order to evaluate coordinated staging, experiments with three different sources were conducted. The three sources were: *Singapore* (site providing the lowest upload rate to other sites), *Alamo* (site providing moderate upload rate to other sites) and *UVACSE* (site providing the best upload rate to other sites). For each experiment, the remaining ten end-points were used as execution-sites. An HTC application consisting of ten sub-jobs was submitted to a metascheduler placed on *Romulus*, a modest host at the *University of Virginia*. Each individual job of the HTC application required a single shared file to be staged-in. The source of this shared file was *Singapore, Alamo* or *UVACSE* depending on the experiment. The size of this shared file was 3GB or 15GB. The metascheduler was made to schedule exactly one individual job onto each execution site. Table 3 lists the parameters used for the experiments.

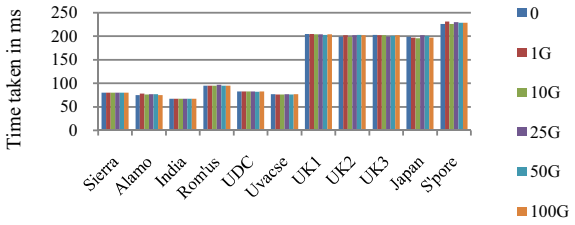
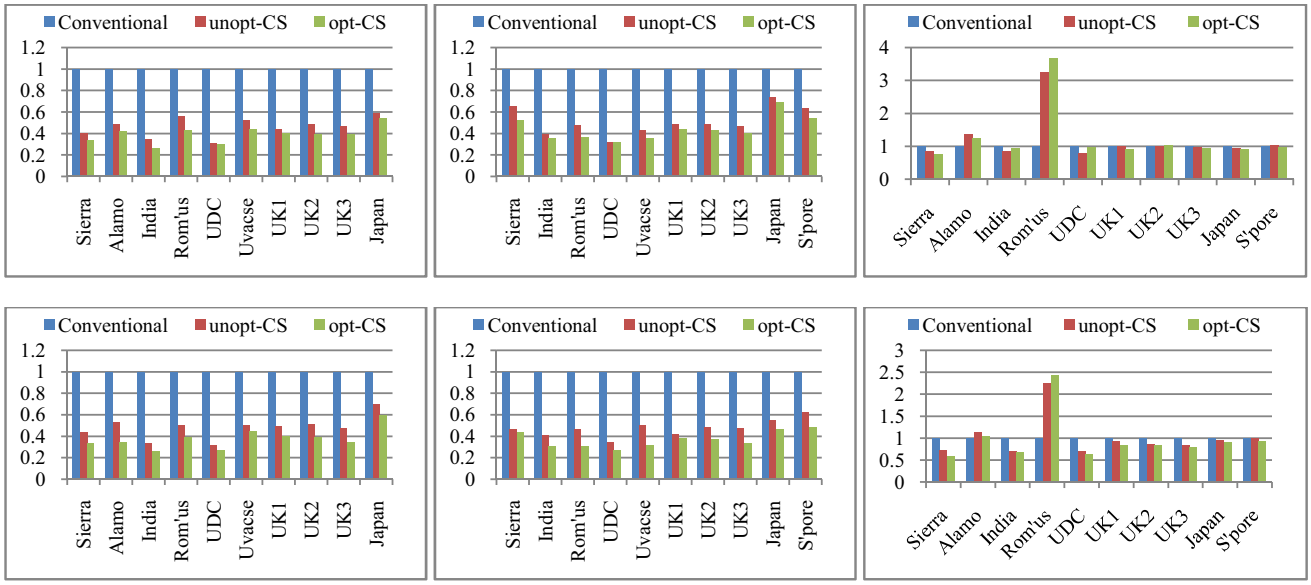


Fig. 1 Time taken to seek and write

Figure 2 shows the download time by the sites under all three studied approaches (normalized to the conventional approach of downloading from a single source). It is to be noted that all three approaches employed parallel-TCP. It was observed that with *Singapore* and *Alamo* as sources, every site benefitted under both *unopt-CS* and *opt-CS*. With *Singapore* as source the speed-up varied between 1.42 and 3.85, while with *Alamo* as source, the speed-up varied from 1.35 to 3.70. Speed-up was achieved not only because of the availability of faster peers for download, but also due to lesser bandwidth contention at the source. With a 3GB source at *UVACSE*, no site except *Romulus* suffered a significant slowdown. With a 15GB source at *UVACSE*, all sites except *Romulus* and *Alamo* obtained a speed-up. A speed-up of upto 1.56 was achieved. Typically, the 15GB downloads obtain greater speed-up than the 3GB ones. This is because a lesser percentage of the outcalls are spent on exploring, and the penalties of optimistically assuming a peer to have a file are less severe.



(a) Singapore as source (b) Alamo as source (c) UVACSE as source

Fig. 2 Comparison of the download time of three approaches normalized to the conventional approach for 3GB file (top) and 15GB file (bottom)

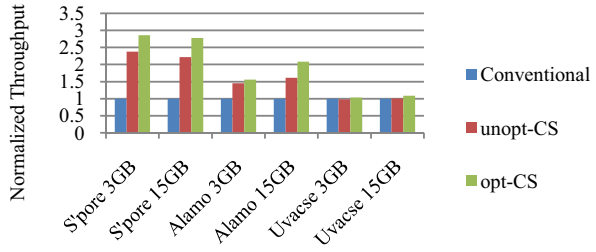


Fig. 3 Comparison of the throughput of the three approaches

Moreover, for the 3GB download under *opt-CS*, it is possible for a site to not download a chunk from a closer peer even though the peer may have already downloaded the chunk. This is because the chunk size is small enough to be covered by executing the *Explore Phase* on a small number of unexplored replicas. *Opt-CS* was on average 1.17x faster than the *unopt-CS* because the former tries to increase the traffic flowing on the faster links.

We define the *throughput* of a download to be the ratio of the total amount of data downloaded by all the sites to the time taken for the last site to finish downloading the given (shared) file. Figure 3 shows the throughput of a 3GB and 15GB file download under all three approaches, normalized to the conventional approach. Under *opt-CS*, the throughput of the system improved for all the cases. With the slower sources, both *opt-CS* and *unopt-CS* caused an increase in throughput. *Unopt-CS* on the 3GB source at *UVACSE* resulted in a negligible throughput decrease. It is to be noted that sites which finish downloading earlier can be simultaneously executing the job (using files from the working directory) and be providing shared files to peers (from the cache directory).

Site	Purge Policy
TACC Lonestar	10 days
TACC Ranger	Deletion on full filesystem
PSC Blacklight	7 days
Purdue Steele	90 days
IU Quarry	60 days
NCSA Forge	4 days for files >= 10GB, else 14 days
NICS Kraken	30 days

Table 4 Scratch space purge policy heterogeneity amongst a few XSEDE sites

VII. Related Work

The idea of using downloaders as uploaders has been used in peer-to-peer systems like in the BitTorrent protocol [26]. However, there exist differences between the two systems. Firstly, in our system, all the peers start downloading at almostthe same time. Secondly, the churn-rate is very high in peer-to-peer systems while the peers in our environment are more stable. The metascheduler performs the task of the *Tracker* in BitTorrent by helping downloaders find each other. In both systems, peers can become prospective *seeders*. BitTorrent uses a choking technique to prevent free-riding. The metascheduler prevents free-riding to an extent by preventing those BES' which do not incorporate the FSS EPR from obtaining peer knowledge. We assume that all peers are cooperating to achieve a common goal and assume no malicious peers.

The state-of-the-art production grids [16, 27, 28] typically advertise a manual approach for file transfer via *scp* or *GridFTP* [29]. Owing to storage quota constraints in the home directory at each site, users use a larger shared scratch space. The heterogeneity in the scratch space purge policy of sites is

the source of most problems in this manual approach. The user must not only remember if the correct version of his input files is in the scratch space of a particular site, but also remember the policies of each site. Table 4 shows the heterogeneity in scratch space purge policy among a few XSEDE sites. Such an approach defeats an important use-case of computational grids - "the user must not think" and should just "submit and forget" [14].

On the other hand, grid middlewares like GenesisII [23] and Globus [30] have components like BES [12] and WS-GRAM [13]. Some of these services support sharing of input files by jobs at a site. The shared files are deleted once the last job finishes. Consequently, reuse which is inevitable in scientific research is not addressed.

ARC [20] is a grid middleware with a cache for handling file sharing and reuse. However the cached files are readable only by jobs within the site. *Cabinet* has a globally visible cache resulting in both performance and availability benefits.

GridFTP [29] is a file transfer mechanism which also uses parallel-TCP for mitigating the inherent slowness of TCP. rftp [31] is a file-transfer tool implemented on GridFTP which downloads from multiple replicas simultaneously.

Typical staging techniques do not address the possibility of treating other execution sites as possible replicas. To the best of our knowledge, this is the first system to employ such a staging technique. In this paper, each site downloads from only one peer at a time. But our work can be extended to download from multiple peers simultaneously, and we leave that as a future work.

VIII. Conclusion

In this paper, we described two techniques for efficiently managing data in the GFFS- a caching system and a new file staging technique called *coordinated staging*. The caching system aims to achieve a "o*nce is enough"* file transfer goal by preventing re-staging during sharing and reuse of input files. The main contribution of this work is *coordinated staging* which treats execution sites as possible replica-stores. Two techniques to perform coordinated staging were discussed: an *unoptimistic* and an *optimistic* approach. Coordinated staging decreased the download time by upto 3.85x, and increased the throughput by upto 2.86x over the conventional approach of file-transfer from a single source. Also, the *optimistic approach* was on average 1.17x better than the *unoptimistic* approach.

Acknowledgment

This material is based upon work supported in part by the National Science Foundation under Grant No. 0910812. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number OCI-1053575.

References

[1] A. Grimshaw et al. "An open grid services architecture primer," Computer, vol. 42, pp. 27–34, Feb. 2009.

[2] E. Seidel, TeraGrid 2011 Keynote Address, 2011. https://www.xsede.org/wwwteragrid/archive/web/tg11/seidel-article.html.

[3] P.-C. Chen, J.-B. Chang, Y.-L. Su, and C.-K. Shieh, "Ondemand data co-allocation with user-level cache for grids," Concurr. Comput.: Pract. Exper., vol. 22, pp. 2488–2513, Dec. 2010.

[4] G. Wasson and M. Humphrey, "HPC File Staging Profile 1.0," tech. rep., Open Grid Forum, 2010

[5] J. Bent, A. Arpaci-Dusseau, R. Arpaci-Dusseau and M. Livny, "Migratory file services for scientific applications," tech. rep., Univ. of Wisconsin, Madison, 2002.

[6] J. Bester, I. Foster, C. Kesselman, J. Tedesco, S. Tuecke, "GASS: A data movement and access service for wide area computing systems," 1999.

[7] A. Grimshaw, M. Morgan and A. Kalyanaraman GFFS—The XSEDE Global Federated File System. *Parallel Processing Letters*,*23*(02).

[8] M. Morgan, "ByteIO specification 1.0," tech. rep., Open Grid Forum, 2006.

[9] M. Morgan, A. Grimshaw, and O. Tatebe, "RNS specification 1.1," tech. rep., Open Grid Forum, 2010.

[10] M. Morgan and A. Grimshaw, Methods in Enzymology, ch. 8. Elseiver, 2009.

[11] J. I. Garzon et al, "End-to-end cache system for grid computing: Design and efficiency analysis of a highthroughput bioinformatic docking application," Int. J. High Perform. Comput. Appl., vol. 24, pp. 243–264, Aug. 2010.

[12] I. Foster et al. "Basic execution service specification 1.0," tech. rep., Open Grid Forum, 2008.

[13] WS-GRAM: http://www.globus.org/toolkit/docs/4.0/execution/wsgram/.

[14] E. Huedo, R. S. Montero, and I. M. Llorente, "The GridWay framework for adaptive scheduling and execution on grids," Scalable Computing: Practice and Experience, vol. 6, no. 3, 2005.

[15] A. Anjomshoaa et al, "Job submission description language specification 1.0," tech. rep., Open Grid Forum, 2005.

[16] W. Gentzsch et al, "DEISA - distributed european infrastructure for supercomputing applications," J. Grid Comput., vol. 9, no. 2, pp. 259–277, 2011.

[17] XSEDE. https://www.xsede.org/.

[18] T. J. Hacker and B. D. Athey, "The end-to-end performance effects of parallel tcp sockets on a lossy wide-area network," 2001.

[19] Filesystem in Userspace. http://fuse.sourceforge.net/.

[20] M. Ellert et al. "Advanced resource connector middleware for lightweight computational grids," Future Gener. Comput. Syst., vol. 23, pp. 219–240, Feb. 2007.

[21] A. Chervenak et al. "Giggle: a framework for constructing scalable replica location services," in Proceedings of the 2002 ACM/IEEE conference on Supercomputing, pp. 1–17, IEEE Computer Society Press, 2002.

[22] S. Valente and A. Grimshaw, "Replicated grid resources," in Proceedings of the 2011 IEEE/ACM 12th International Conference on Grid Computing, pp. 198–206, IEEE Computer Society, 2011.

[23] GenesisII. http://www.genesis2.virginia.edu/.

[24] "Web Services Addressing". http://www.w3.org/Submission/ws-addressing/.

[25] A. Grimshaw, M. Morgan, and K. Sarnowska, "WS-naming: location migration, replication, and failure transparency support for web services," Concurr. Comput. : Pract. Exper., vol. 21, pp. 1013–1028, June 2009.

[26] B. Cohen, "Incentives build robustness in BitTorrent," 2003.

[27] TeraGrid. https://www.teragrid.org.

[28] UK National Grid Service. http://www.ngs.ac.uk/.

[29] W. Allcock et al. "GridFTP: Protocol extensions to FTP for the Grid," 2001.

[30] I. Foster and C. Kesselman, "Globus: A metacomputing infrastructure toolkit," Int. J. of Supercomputer Appl., vol. 11, pp. 115–128, 1996.

[31] J. Feng and M. Humphrey, "Eliminating replica selection - using multiple replicas to accelerate data transfer on grids," Parallel and Distributed Systems, International Conference on, vol. 0, p. 359, 2004