Summer 7-24-2015

Enabling Distributed Scientific Computing on the Campus

Derek J. Weitzel
University of Nebraska-Lincoln, dweitzel@cse.unl.edu

ENABLING DISTRIBUTED SCIENTIFIC COMPUTING ON THE CAMPUS

by

Derek Weitzel

A DISSERTATION

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Doctor of Philosophy

Major: Computer Science

Under the Supervision of Professor David Swanson

Lincoln, Nebraska

July, 2015

ENABLING DISTRIBUTED SCIENTIFIC COMPUTING ON THE CAMPUS

Derek Weitzel, Ph.D.

University of Nebraska, 2015

Adviser: David Swanson

Campus research computing has evolved from many small decentralized resources, such as individual desktops, to fewer, larger centralized resources, such as clusters. This change has been necessitated by the increasing size of researcher's workloads, but this change has harmed the researcher's user experience. We propose to improve the user experience on the computational resources by creating an overlay cluster they are able to control. This overlay should transparently scale to national cyberinfrastructure as the user's demands increase.

We explore methods for improving the user experience when submitting jobs on a campus grid. To this end, we created a remote submission and overlay computational framework called Bosco. This framework can remotely submit processing from the user's laptop to clusters on the campus or on national cyberinfrastructure. To illustrate the possibilities of improving the user experience of remote submission, we created BoscoR, an interface to Bosco in the popular statistics and data processing programming language, R. Bosco improves the user experience of submitting to campus clusters, while also being an efficient method for job management.

In order to solve some of the issues with data distribution on opportunistic resources, we created the CacheD, a data management framework for managing and provisioning storage resources on the campus. The CacheD additionally optimizes transfers to multiple resources by using the peer-to-peer transfer protocol, BitTorrent. Further, the CacheD optimizes shared data between multiple jobs by caching

the input data directly on the execution resources. The CacheD decreases the stage-in time over current transfer methods and significantly decreases stage-in time when the data is already cached.

Finally, we explain how to control data distribution on a campus through a comprehensive policy framework. This framework is implemented in the CacheD. We present the policy language, its currently available attributes, and how to extend the policy language beyond the default behavior. Multiple examples are given for different data distribution scenarios observed on campus resources.

Combining easy-to-use campus job submission with Bosco, efficient data distribution with the CacheD, and a policy language to manage the data distribution, we have created a unified framework for campus computing.

ACKNOWLEDGMENTS

I am thankful to have worked with many fine people throughout graduate school.

Dr. David Swanson, my dissertation advisor, has provided a fertile environment of both distributed computing users and the resources to test and provide solutions. In addition, I am forever thankful for the opportunity he provided for me to attend graduate school.

Dr. Brian Bockelman has provided immense guidance for my dissertation. I thank Brian for mentoring me throughout my collegiate career.

I thank my colleagues at the Holland Computing Center. I have broken their systems, asked for advice, and caused an untold amount of extra work for them. I could not have completed this dissertation without the skilled experts at HCC who have helped me throughout the process.

I would also like to thank my colleagues in the Open Science Grid. They have provided me with a fellowship and internships in support of my Ph.D. I would like to especially thank Ruth Pordes and Dan Fraser for providing valuable advice and mentorship in my academic career.

My family deserves thanks for many intangible gifts. I thank them for encouraging me to complete my goals.

Last but not least, I thank my lovely fiancée Katie for providing immeasurable support.

Contents

Contents **v**

List of Figures **vii**

List of Tables **x**

1 Introduction **1**

1.1 Campus Grid Computing 1

1.2 Data Management on Campus 5

1.3 Data Distribution Policy Language 6

1.4 Overview of Dissertation 8

1.5 A Note on Terms 9

2 Related Work **10**

2.1 Batch Systems and Grids 10

2.2 Data Management 18

2.3 State of Practice in Campus Computing 25

3 Campus Job Distribution **29**

3.1 Introduction 29

3.2 Bosco Architecture 31

3.3 Load Balanced Access to Computational Resources 40
3.4 BoscoR: Extending R from the desktop to the Grid 40
3.5 Conclusion 55

4 Campus Data Distribution 57

4.1 Introduction 57
4.2 Background and Related Work 59
4.3 Implementation 63
4.4 Results For Campus Cluster 76
4.5 Results for Open Science Grid 84
4.6 Conclusions 92

5 Campus Storage Policy Language 94

5.1 Introduction 94
5.2 Policy Language 96
5.3 Uses of the Policy Language in the CacheD 98
5.4 Example Policies 102
5.5 User Scenario 107
5.6 Conclusions 109

6 Conclusion 110

6.1 Future Work 112

Bibliography 116

List of Figures

1.1 Bosco Architecture . . . 3
1.2 Bosco's Growing Reach as Demand Increases . . . 4
2.1 Architecture of Local HTCondor . . . 12
2.2 ClassAd Example Showing Usage of Requirements . . . 13
2.3 Architecture of GlideinWMS . . . 15
2.4 Grid Laboratory of Wisconsin Campus Grid . . . 16
2.5 Storage Technology Categorization . . . 20
2.6 Blast Workflow . . . 26
3.1 Bosco Architecture . . . 33
3.2 Bosco SSH Network Connections to the Cluster Login Nodes . . . 34
3.3 Bosco Job Submission Flow From Submission Host to Remote Cluster . . . 36
3.4 Bosco Glidein Launch . . . 37
3.5 Session Log From Bosco Traceroute . . . 39
3.6 Function Definition of `lapply` . . . 42
3.7 GridR Input Generation . . . 45
3.8 GridR Function with Remote Package Installation Before Execution . . . 47
3.9 Direct Submission . . . 50
3.10 Comparison of Submission Methods . . . 52

3.11 Number of Simultaneously Running Jobs 53

4.1 Blast Workflow 58
4.2 HTTP Cache Architecture 60
4.3 Daemon Locations 64
4.4 Flow of Replication Requests 68
4.5 Cache Replication Showing Bottleneck 70
4.6 Illustration of Cache Parenting Inside a Node 73
4.7 Illustration of Cache Parenting 74
4.8 Example Code to Create a Cache 75
4.9 Function to Request a Replica of the Cache 75
4.10 Illustration of Direct and BitTorrent Transfer Methods Experimental Setup 78
4.11 Comparison of Direct and BitTorrent Transfer Methods with Increasing Distinct Node Counts 79
4.12 Histogram of Transfer Modes vs. Download Times for a BitTorrent Workflow 80
4.13 Timeline of Blast Runs 80
4.14 Transfer Method vs Number of Jobs 81
4.15 Graph of Transfer Nodes in the BitTorrent Transfer 83
4.16 OSG Daemon Locations 84
4.17 Violin Plot of the Transfer Speed Comparing Transfer Protocols 87
4.18 Campus Aggregate Bandwidth Showing the Source Node's 1 Gbps Connection 88
4.19 OSG Aggregate Bandwidth Showing the Source Node's 1 Gbps Connection 88
4.20 Download Time by Mode for a Single OSG BitTorrent 92

5.1 CacheD ClassAd Example 96
5.2 CacheD Cron Configuration 98

5.3 Example Output from CacheD Cron: `test.sh` 98

5.4 Example Replication Method for a Cache 102

6.1 Bosco Job Submission Flow From Submission Host to Remote Cluster . . 113

List of Tables

2.1 HTCondor Daemon Functions 11

4.1 Table of Capabilities for Grid Data Caching Software 62

4.2 Table of Capabilities for Direct and BitTorrent Transfer Methods 67

4.3 Transfer Times for Different Transfer Type 86

4.4 HTTP Transfer Speeds by Site 90

4.5 BitTorrent Transfer Speeds by Site 91

5.1 Sample CacheD Attributes 101

5.2 Example Data Types 103

Chapter 1

Introduction

In this dissertation, we optimize distributed computing workflows on a campus grid. We are interested in optimizing a researcher's use of the computational and storage resources on the campus to increase the reliability of, and decrease the time to solution for, scientific results. We first extend prior work to enhance the computational capabilities of researchers on a campus. We then expand our work to the data needs of modern workflows.

1.1 Campus Grid Computing

The increase of performance of computer hardware following Moore's law [64] has allowed scientists to tackle larger problems. As they increase their use of research computing, their applications far exceed the locally available resources. Such applications often turn to distributed computing to aggregate more computational, memory, or storage resources than locally available resources can provide.

Batch computing can combine the computational, memory, and storage resources of multiple computers in a single cluster through concurrent scheduling of applications. A computational grid is an extension of batch computing, where resources may be combined from multiple pools of resources to be used for an application.

A computational grid is a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities [28]. A campus grid is a specialized grid where multiple resources are owned by the same organization, although it may be in multiple administrative domains.

A campus grid has become necessary to spread demand across multiple clusters. This is important when demand for a single cluster is large, due to improved performance or increased storage, and demand is low on other available clusters. One aim for a campus grid is to move computation from the in-demand cluster to other clusters, which can result in a shorter time to completion for the users' jobs.

To succeed, a campus grid requires a framework to distribute jobs to multiple clusters in a campus. In [84], I proposed a solution based on HTCondor [47]. The solution required installation of a campus factory [83] on each cluster's login node. An on-demand overlay was created that could efficiently run high throughput jobs on multiple campus resources.

Although my solution was efficient and fault tolerant, it was deficient in several ways. Installation and setup of the campus factory was difficult since it was not automated. The communication inside the overlay was insecure. We set out to correct these deficiencies.

We have enhanced my Masters thesis' solution to include:

- Easier installation through automation
- Increased security through secure key exchange
- More supported cluster types and configurations

- Improved access to computing through language frameworks such as R [71]

We created a framework for job submission to remote resources that the user does not control. Typical grid submission uses custom interfaces such as the Globus Resource Allocation Manager (GRAM) [32], which is previously installed by an administrator. We assume resources do not have such dedicated grid software installed. This framework does not require administrator intervention for remote submission to opportunistic resources, since it uses interfaces that are installed on nearly all clusters that are typically used for interactive access. It automates the submission and error handling of jobs submitted to remote resources, while providing the user a consistent interface over multiple, load-balanced clusters.

The new framework is named Bosco [82]. It uses secure protocols to connect to remote clusters in order to transfer files and submit, and monitor, jobs. Installation of Bosco on remote clusters and the submit host has been automated with simple tools. Clusters with restrictive firewalls are supported by multiplexing operations through a single secure connection. Furthermore, many cluster schedulers are supported by the underlying technology. A diagram of the architecture of Bosco is shown in Figure 1.1.

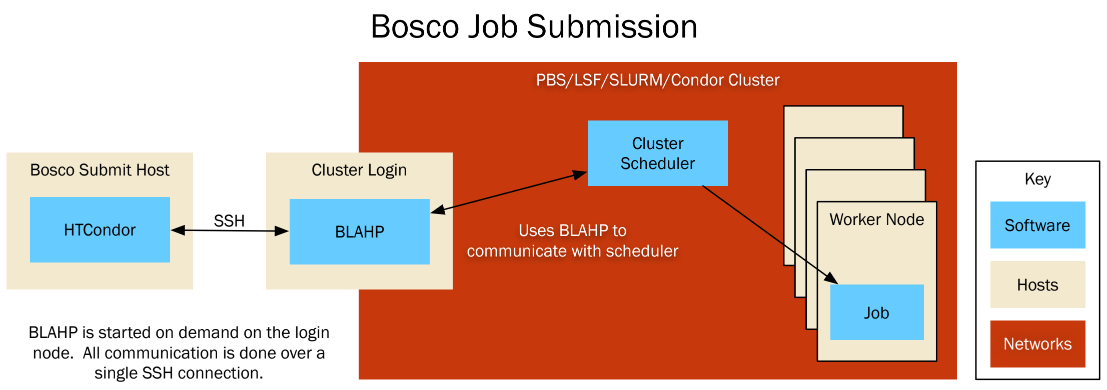

Bosco Job Submission
PBS/LSF/SLURM/Condor Cluster
Cluster Scheduler
Bosco Submit Host
HTCondor
SSH
Cluster Login
BLAHP
Uses BLAHP to communicate with scheduler
Worker Node
Job
Key
Software
Hosts
Networks
BLAHP is started on demand on the login node. All communication is done over a single SSH connection.

Figure 1.1: Bosco Architecture

Bosco, in coordination with technologies in HTCondor, enables a job distribution method which is provisioned based on demand. A default Bosco installation is able to submit to one local cluster. If that cluster does not meet the user's computational needs, then Bosco can be configured to submit to multiple clusters with load balancing between them. If the user's computational needs are still not met with multiple clusters, they can configure Bosco to submit resource requests to national cyberinfrastructure such as the Open Science Grid (OSG) [55]. The provisioning capabilities of Bosco create an ever expanding network of available resources. The goal is to provide an expanding network of resources as shown in Figure 1.2.

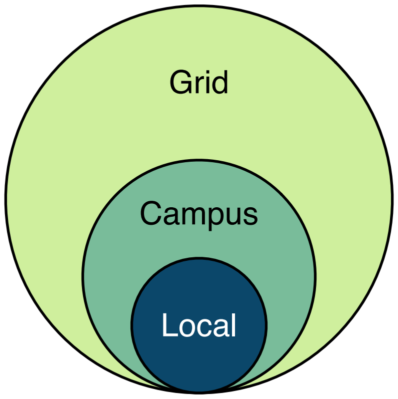


Figure 1.2: Bosco's Growing Reach as Demand Increases

In order to ease access to Bosco for data processing, an interface has been developed in the most widely used data processing language, R. This BoscoR framework enables users to never leave their R environment in order to start remote data processing.

Nonetheless, Bosco is not designed to be enough for researchers that have large data requirements. Input and output data are explicitly listed by the user. The data is transferred over the secure, but slow, connection between the submitter and resource for every job. Therefore, we must consider data and storage management on the campus grid.

1.2 Data Management on Campus

There are many challenges in data management and distribution in scientific computing [25]. For batch computing, one challenge is transferring the data from the user's computer to the execution resources. Large data workflows can strain the network near the data's source, which can result in unreasonable amounts of time used solely for data transfer.

Data management is the framework and policies controlling data through the research cycle. In this dissertation, we are concerned with optimizing data management when using campus computational resources.

As users spread their computation across multiple clusters either on the campus or across campuses, data distribution and collection becomes more difficult. Before using the campus grid, a user would select a cluster to do their processing. The user then could host all of their data on that cluster by copying the data onto that cluster's shared filesystem. The jobs access the data from the shared filesystem just as it would on the user's desktop, available for all executions at the same directory.

These assumptions do not hold for a campus grid. A grid is made up of multiple computational clusters, with potentially many separate filesystems; no single filesystem is accessible from every computational resource. Further, the shared filesystem could become a bottleneck if many jobs are requesting the same data simultaneously. Therefore, data management techniques must evolve along with computation.

Most distributed batch schedulers are able to transfer the input data for each job execution. Each job starts with an empty execution area and the scheduler will transfer the files into it. When the user is not using the scheduler to transfer data, the input data must still reach the execution host. Data will be transferred from the source (usually the user's computer) to the execution resources for processing. The network connection between the source and the execution resources may be a bottleneck for the computation. Frequent re-transfers of the same input data will further congest the network between the source and the execution resources.

In this dissertation, we optimized two attributes of distributed data management: efficient transfer methods and reduction of duplicate transfers. We introduce the CacheD [85], a caching and data transfer daemon for input data in distributed computing. The CacheD uses novel data management methods based on technology developed for large peer-to-peer data transfers on the Internet, BitTorrent. It also caches input data on the execution resources to enable quick transfers on subsequent requests for the same input data.

Similar to the work with Bosco, the CacheD does not require privileged access in order to provision storage resources. It can use the storage on worker nodes spread across multiple clusters as a data input caching system.

1.3 Data Distribution Policy Language

Users of grid submission software currently have to describe how their files will be transferred from their submission host to the remote execution resource where the data will be processed. They have to coordinate the storage and computational resources without help. We propose a policy language that allows an agent to decide an appropriate method for data transfer. It determines the transfer method by negotiating between the following three sources: a user-given policy language for the data, the remote execution resource's capabilities and preferences, and the submitting resource's capabilities and preferences. In addition, the policy language should determine if the cache should be replicated to multiple resources. A modern flexible policy language for describing data distribution for campus users is needed.

This policy language must help the CacheD make decisions when interacting with other agents, such as other CacheDs or the local node. We discuss extending this policy language to include custom attributes that users can include to improve choices on data distribution. The policy language utilizes the ClassAds [59] language. These ClassAds were originally developed in the context of matchmaking between computational resources and potential jobs. ClassAds are a schema-free language for describing heterogeneous resources. We demonstrate usage with new attributes that pertain to storage and expressions that can be evaluated to make decisions.

The user must specify preferences for the cache to consider. Examples include: where this cache should be distributed, how the cache should be distributed, and how long the cache should be stored. Each of these preferences must be negotiated with the preferences of the CacheD that may store or is storing the cache. The user's preferences will affect how fast the cache is transferred (different transfer methods are more efficient than others), and also on which, and on how many, nodes that cache should be replicated.

Further, each CacheD must coordinate with one another in order to distribute the caches in an efficient method. Replication of caches between CacheDs must be negotiated. Each CacheD may decide, through evaluating its own policies, whether or not to accept a cache to be stored. These policies are again expressed in the ClassAd policy language.

1.4 Overview of Dissertation

This dissertation describes how data intensive applications can be run in a distributed campus environment.

Chapter 2: There are many distributed computing platforms available publicly. In this chapter, we will discuss these schedulers and differentiate them from Bosco. Also, we will discuss other available data management, distributed storage, and caching systems.

Chapter 3: We will discuss how computing can be managed on the campus using the Bosco framework. We will also discuss a case study of integrating Bosco with the programming language, R, in order to provide an easy-to-use interface to campus distributed computing.

Chapter 4: We will discuss the CacheD, a campus data distribution service. The CacheD is able to combine novel transfer methods with data caching to improve the stage-in time for large data sets. Through evaluation, we show that the CacheD demonstrates a significantly shorter stage-in time for large data sets over existing solutions that have been deployed.

Chapter 5: Simply caching and transferring data does not provide the flexibility that the CacheD requires to operate in a distributed environment. In this chapter, we will discuss the user configured policy framework and language that enable the CacheD to interact with the user and other daemons in order to make decisions.

1.5 A Note on Terms

High performance and distributed computing often use terms inconsistently. Below is a definition list of such terms, and how we will define them for this dissertation:

Job: A packaged unit of work with input and output. A job may consume computational, memory, network, and/or storage resources in a batch system.

Workflow: A logical grouping of jobs executed on resources. The jobs may have some ordering.

Campus: An organization that may have multiple administrative domains which may vary access policies to resources.

Execution Resource: A resource which fulfills the requirements of a job and may also run it. This may be a worker node in a cluster.

Cluster: A set of execution resources that have high interconnection bandwidth and are managed by a single scheduler.

Batch System: A scheduler for the resources of a cluster.

Agent: An independent entity that can make decisions on its own without the control of another entity. In this dissertation, we will use the word agent to describe a daemon which can make independent decisions without the explicit control of other daemons.

Pilot: Pilot jobs are containers that once started, will request work from the user's job queue.

Chapter 2

Related Work

2.1 Batch Systems and Grids

Several batch systems and grid schedulers are able to schedule tasks on execution resources. Examples of cluster schedulers that are frequently used are PBS [42], HTCondor [47], LSF [23], and Slurm [88]. Each scheduler is capable of resource management within a single administrative domain. Each of these resource managers has a limited ability to send processing to remote resources, which are typically under a separate administrative domain. PBS and Slurm can send jobs between clusters that run the same schedulers. HTCondor also has the ability to send processing to other clusters running HTCondor, and it can also transform jobs to the language of other schedulers such as PBS and Slurm.

2.1.1 HTCondor

HTCondor was developed at the University of Wisconsin–Madison. An overview from [72]:

> Condor is a high-throughput distributed batch computing system. Like other batch systems, Condor provides a job management mechanism, scheduling policy, priority scheme, resource monitoring, and resource management. Users submit their jobs to Condor, and Condor subsequently chooses when and where to run them based upon a policy, monitors their progress, and ultimately informs the user upon completion.

Commonly used HTCondor daemons and their functions are listed in Table 2.1.

Daemon	Function
`condor_master`	Maintains HTCondor daemons
`condor_collector`	Information Provider
`condor_schedd`	User Job Queue
`condor_negotiator`	Scheduler: Matches jobs with resources
`condor_startd`	Execution manager. Runs on the resource

Table 2.1: HTCondor Daemon Functions

Figure 2.1 shows the architecture of a local installation of HTCondor. Users interact with the SchedD to submit jobs. The Negotiator schedules the jobs on worker nodes. The StartD and the Starter runs on the worker node. The Starter starts a user's jobs.

An important technology used in HTCondor is the Classified Advertisement (ClassAd) mechanism. ClassAds are the language that HTCondor uses to communicate between daemons and for matchmaking [59]. A ClassAd is a list of keys and values, where the values can be strings, numbers, or expressions. All resources and jobs are described by ClassAds. Job ClassAds have attributes such as log file, output, input, and job requirements. Resource ClassAds have attributes such as requirements to run on the resource, ownership, and policies. ClassAds are used for matching jobs to resources by evaluating requirements of both the jobs and the resources.

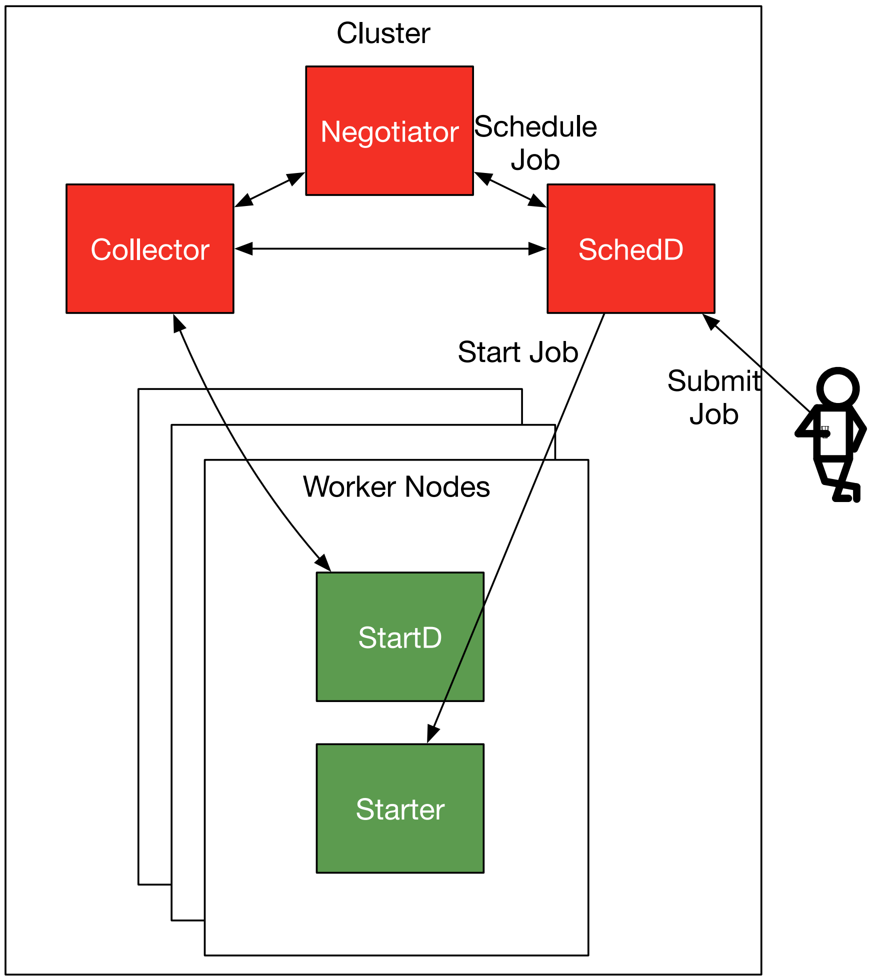
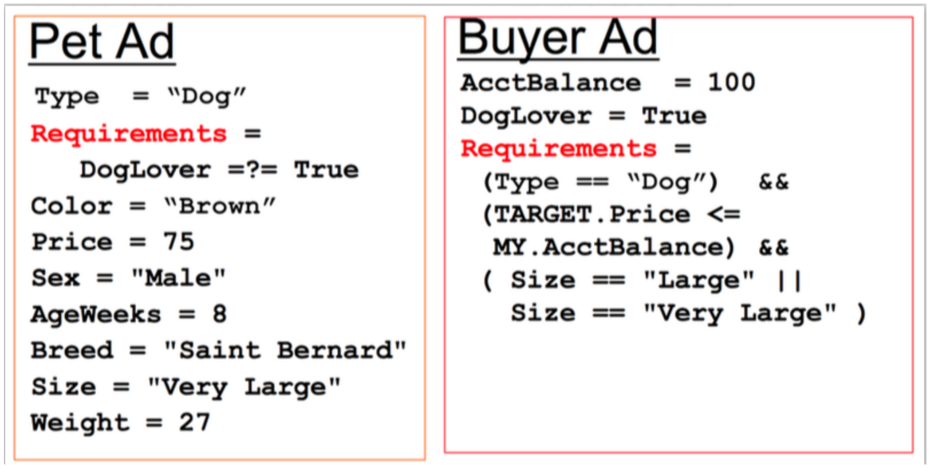


Figure 2.1: Architecture of Local HTCondor

An example of using the `Requirements` attribute is shown in Figure 2.2. In this example, we have two ads. A pet which describes the attributes of a dog and a buyer ad which describes the attributes of a potential buyer. The pet ad has the requirements that the buyer is a `DogLover`. The buyer has the requirements:

- The pet must be a dog.
- The pet must cost less than the money the buyer has.
- The pet must be “Large” or “Very Large”.



```
Pet Ad
Type    = "Dog"
Requirements =
   DogLover =?= True
Color = "Brown"
Price = 75
Sex = "Male"
AgeWeeks = 8
Breed = "Saint Bernard"
Size = "Very Large"
Weight = 27
```

```
Buyer Ad
AcctBalance   = 100
DogLover = True
Requirements =
 (Type == "Dog")   &&
 (TARGET.Price <=
  MY.AcctBalance)  &&
 ( Size == "Large" ||
   Size == "Very Large" )
```

Figure 2.2: ClassAd Example Showing Usage of Requirements

In this example, the pet ad would match the buyer ad.

Another component of HTCondor is the grid computing agent HTCondor-G [34]. HTCondor-G communicates with Globus [31] sites. HTCondor provides job submission, error recovery, and creation of a minimal execution environment. Along with HTCondor-G, HTCondor can also submit jobs to other systems including Amazon EC2 [10], PBS [42], Slurm. But, HTCondor is not capable of submitting to remote clusters without one of these specialized interfaces to the cluster.

HTCondor categorizes jobs by `universe`. A `universe` in HTCondor specifies how the job should be handled. The simplest example is the `vanilla universe`. In this `universe`, the job is handled as a single executable, with input and output, that will exit when the job has completed. When the `universe` is `grid`, this means that the job will be translated to another submission method. The other submission methods could be a Globus submission, or as used in this dissertation, a PBS or Slurm job submission.

2.1.2 GlideinWMS

The Glidein Workflow Management System (GlideinWMS) [66] is a job submission method that abstracts the grid interface, leaving only a batch system interface. GlideinWMS accomplishes this abstraction by using pilot jobs. Pilot jobs are containers that once started, will request work from the user's queue. User jobs will not wait in remote queues; therefore, user jobs do not waste time in remote queues when idle resources are available. GlideinWMS separates the system into two general pieces, the frontend and the factory. The frontend monitors the local user queue and requests glideins from the factory. The factory serves requests from multiple frontends and submits pilots to the grid resources on their behalf. The factory only submits jobs to grid interfaces on multiple resources and can be optimized for that purpose. The frontend only deals with the local batch system, and can be optimized for the user's jobs.

The GlideinWMS system uses HTCondor throughout. It uses HTCondor-G [34] to submit to the grid resources, as well as manage user jobs on the frontend. The frontend and factory are daemons that run on the user submit host and a central machine, respectively.

The architecture of GlideinWMS is shown in Figure 2.3. Just as in the HTCondor architecture, the user interacts with the SchedD. In the GlideinWMS architecture, the StartD and the Starter run on worker nodes on clusters not managed by the same organization as the GlideinWMS Frontend.

GlideinWMS is heavily used on the the Open Science Grid. A major user and developer of the software is high energy physics, specifically the Compact Muon Solenoid (CMS) experiment [18] and the Collider Detector at Fermilab (CDF) [89]. They have demonstrated recently that GlideinWMS can scale beyond 200,000 running jobs.

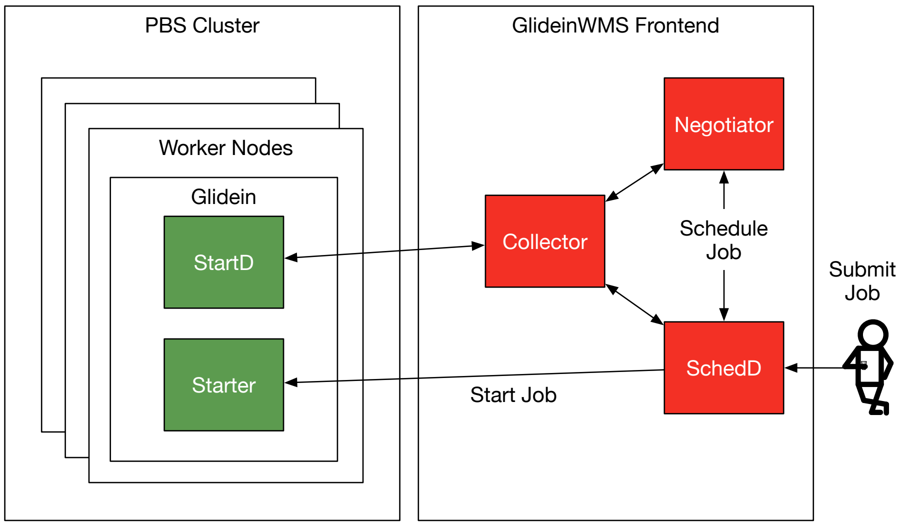


Figure 2.3: Architecture of GlideinWMS

GlideinWMS does have a few drawbacks. GlideinWMS uses an external factory that acts as a single point of failure. If the factory quits submitting jobs to grid resources, then users cannot run jobs. Also, GlideinWMS is designed to only submit to Globus Security Infrastructure (GSI) secured sites. GSI is typically used only on production grids, and is rarely used inside a campus where the trust relationship is implicitly stronger.

GlideinWMS requires many components to work together. The GlideinWMS Frontend requests pilots from a GlideinWMS Factory. The factory is run centrally and can server many frontends. Debugging of failed GlideinWMS pilots is a difficult since the pilot submission originates from a central factory. Debugging information is returned to the central factory rather than the researcher's frontend. GlideinWMS requires significant to personnel time to install and maintain.

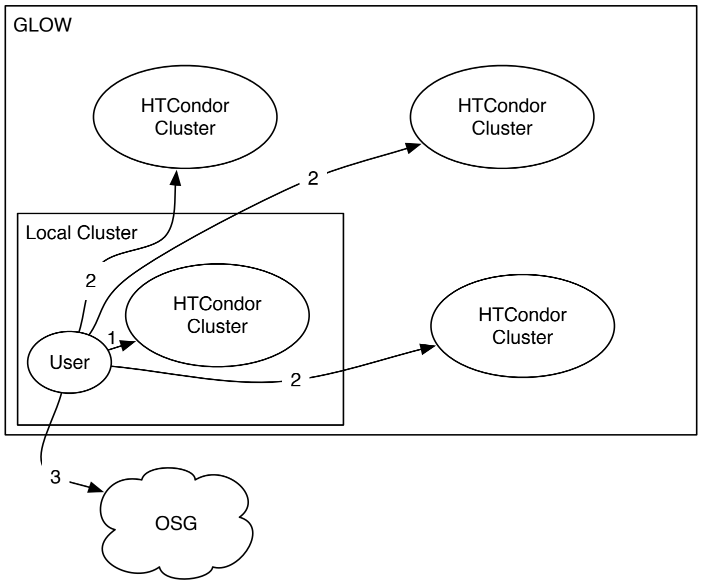


Figure 2.4: Grid Laboratory of Wisconsin Campus Grid

2.1.3 Existing Campus Grids

An example of an existing campus grid is the Grid Laboratory of Wisconsin's campus grid. The Grid Laboratory of Wisconsin (GLOW) [33, 52] is a grid at the University of Wisconsin at Madison. GLOW uses Condor to distribute jobs on their campus grid. A diagram of the campus grid is shown in Figure 2.4. When the user submits a job, it first attempts to run at the local HTCondor cluster. If the local cluster is unable to meet the demands of the user's jobs, then it attempts to run at HTCondor clusters at the University of Wisconsin. If they do not meet the demands of the user, the jobs may then run on the Open Science Grid (OSG), a national grid of clusters.

Security at the GLOW campus Grid is based on IP whitelisting. While there is a central team available to assist with management, each resource is free to define its own policies and priorities for local and remote usage. Cluster ownership is distributed, although there's also a general-purpose cluster available. Software and data are managed by an Andrew File System (AFS) [50] installation and Condor file transfer. AFS is a global file system that every worker node will mount, therefore providing a global space to store data and applications. This simplifies data distribution by providing a staging area.

Users of the GLOW campus grid have difficulty expanding to outside resources due to their reliance on a shared file system. Further, they rely on HTCondor to provide an interface to other clusters. They are unable to submit to clusters which have other schedulers such as PBS or Slurm. HTCondor does not excel at multi-node parallel scheduling, typically used in HPC. Therefore, campuses will not use HTCondor to managed their HPC resources. Instead, the HPC clusters will be managed by schedulers such as PBS or Slurm. HTCondor cannot submit jobs to remote PBS or Slurm schedulers, and therefore cannot use these HPC resources.

2.1.4 Open Science Grid

One example of a grid is the Open Science Grid (OSG) [55]. The OSG is a national cyber infrastructure consortium that provides dedicated and opportunistic use of computational and storage resources located at nationally distributed universities and national laboratories. The OSG provides infrastructure, services, software, and engagement to the users.

The OSG Production Grid provides a common interface to computing and storage resources. Globus or HTCondor provide an interface to the computing resources, while the Storage Resource Manager (SRM)/GridFTP provide an interface to the storage. Access to resources is provided by an OSG client. Globus, HTCondor, and SRM/GridFTP were developed by consortium members and are included in the OSG packaging.

A typical OSG site has a compute element that can run jobs and a storage element that can store data. The storage element is both physically close and tightly coupled to the compute element in order to minimize latency for data. Since most sites have their own storage element, they are used to stage data into and out of the site.

A user will install the OSG Client tools and submit to the sites. The OSG tools will also include applications that can be used to stage data to the storage elements. A user only needs a valid OSG certificate to access the grid resources.

The OSG is organized into groups of users called Virtual Organizations (VO's). These VO's help users to run on the grid, as well as provide organization to many thousands of researchers. Additionally, the VO can sign members' certificates to help sites identify users as belonging to the VO.

While users can directly submit jobs with a valid certificate, this method requires considerable amounts of detailed information for the user to be successful. New users are required to learn considerable technical information in order to exploit OSG to any significant extent, and must belong to a well-established VO to gain access to many OSG resources.

As illustrated by the GLOW campus grid above, the OSG can serve as an overflow for the campus' computational resources. When the campus' resources do not meet the demands of the users, the users' jobs can overflow to the OSG using technologies such as GlideinWMS.

2.2 Data Management

The previous discussion primarily concerned access to computational resources. Data management is a further challenge for distributed computing. For this dissertation, data management can be broken down into three categories: storage systems, transfer protocols, and high level services. Storage systems provide access to the underlying data storage. Transfer protocols provide a method to exchange data between storage systems. High level services provide data management services on top of the file systems, such as:

Data Life Cycle Management: Managing data through the life cycle of creation, processing, storing, and possibly deletion.

Data Placement: Determine where the data should be placed depending on requirements created by the system or a user. Additionally, able to respond to queries on the location of data.

Meta-Data Storage: Storage of information that describes the data.

Search Capabilities: Capability to search data that meet a set of requirements.

Rate Limit: Limit the number of transfers occurring simultaneously depending on a configuration or a policy.

Retries: Retry transfers that have failed.

Batching: Grouping transfers together logically. For example, group all files in a directory into a single logical transfers.

Figure 2.5 shows an overview of the different protocols and services, and how they fit into the three categories: Data Management, Data Services, and Transfer Protocols.

The OSG has has used and maintained many Storage Elements (SE). An SE is an interface to a storage system that my be accessed by many research organizations. An SE provides services such as data movement and data discovery. But, the SE model has been a failure for opportunistic organizations [17]. DQ2 [19] and Phedex [61] are able to manage data movement for the Atlas and CMS experiments, but the cost of

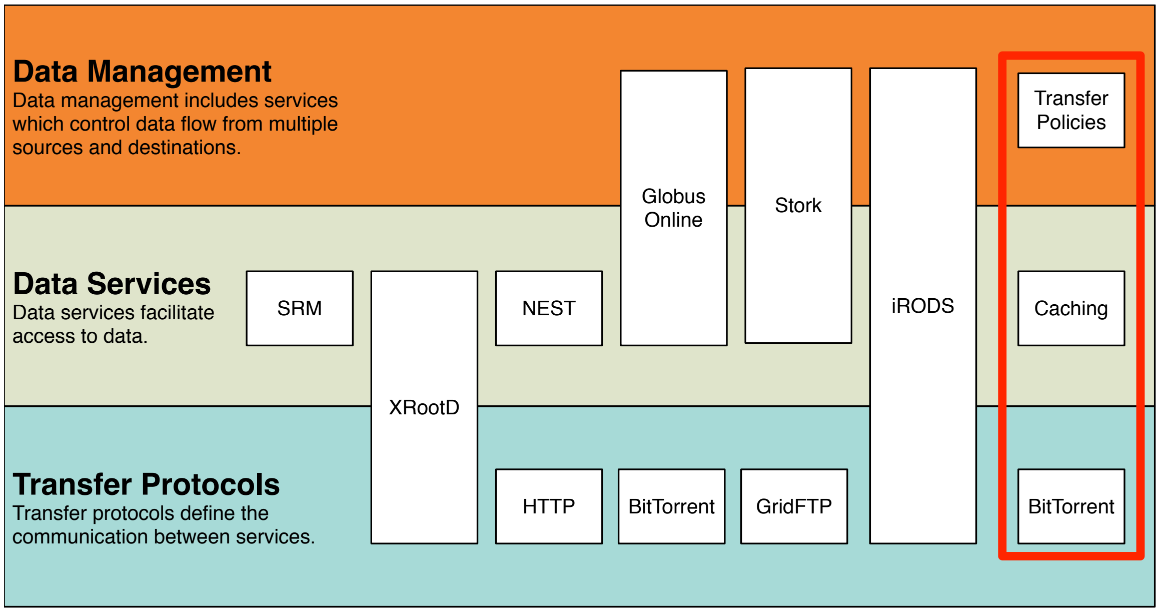

Data Management
Data management includes services which control data flow from multiple sources and destinations.
Data Services
Data services facilitate access to data.
Transfer Protocols
Transfer protocols define the communication between services.
SRM
XRootD
NEST
Globus Online
Stork
iRODS
Transfer Policies
Caching
HTTP
BitTorrent
GridFTP
BitTorrent

Figure 2.5: Storage Technology Categorization

management of these systems is too high for smaller research organizations. The key to this failure is that file loss is considered an “exceptional” event. For opportunistic research organizations, file loss must be treated as an expected outcome. Additionally, the interface with a storage element is too low level for many users. They must make decisions on their own about what directory the files should be stored.

An example of a data management service is the integrated Rule-Oriented Data System iRODS [58]. It provides meta-data storage, querying, and rule-based placements. It can perform transfers to storage resources. When given input, iRODS also has the capability to create rules and take action on data. It creates a small policy framework that, upon certain actions, can execute micro-services. The rules for iRODS can be large and cumbersome for simple data replication. Further, to do anything substantial with the rules, custom code must be written.

iRODS is inflexible in the presence of unreliable underlying storage. iRODS assumes that any data stored in it is available at any time, and does not periodically check the status of the data it is storing. This does not make iRODS a good fit for opportunistic storage, which can be unreliable. Further, the iRODS rules can be cumbersome to write. They are written in micro-services, which are external programs that must interface with iRODS. In contrast, ClassAds expressions are written as simple boolean expressions that are understandable to most programmers. iRODS is an excellent example of data management, but it doesn't map will to the use case we consider because of the limitations outlined above.

Stork [44] is a data placement scheduler. It can schedule data placement and transfers to and from remote storage systems. Stork is innovative in that it treats data transfers similar to jobs. Just like jobs in a batch system, Stork will queue transfers and check for proper completion of the transfers. If the transfers fail, it is able to retry the transfers according to policy. These policies are written in HTCondor ClassAds. Stork is limited to managing data transfers between storage elements. It is unable to utilize opportunistic storage resources directly.

Kangaroo [73] is another storage scheduler multi-level file access system. It allows for multiple levels of staging in order to send job output back to a storage device. It can do this by asynchronously staging data through multiple storage devices on its path to the destination filesystem. The goal of the Kangaroo system is to utilize unused network and storage resources in order to provide a resilient method for output data transfers. The Kangaroo system does not attempt to optimize input data transfers.

DQ2 [19] and Phedex [61] are production transfer services for the Atlas and CMS physics experiments, respectively. They are used to manage distributed transfers to and from sites inside the collaborations. Additionally, they have had databases built on top of them that provide features such as combining files into datasets for easier bulk transfer management. Both were designed for their respective physics experiments, and therefore, would be very difficult to generalize for outside users. Further, both were designed for reliable storage rather than opportunistic storage.

Distributed filesystems such as Hadoop [86] can provide data placement policy. For example, Hadoop can be configured to replicate the contents of a directory at least X times. Further, a script can be given to Hadoop which it can query to create a topology of the data center, further providing control of how the data replicas are sent. This topology script has been used by us to create a data center aware Hadoop replication policy [38]. These Hadoop policies are created and maintained by administrators rather than users. Therefore, a user's preferences will not be considered when making replication decisions.

2.2.1 Storage Systems and Access Protocols

Distributed file systems have long had their own storage access methods. An example of this is Hadoop, a popular distributed file and processing system. The only method to access Hadoop storage is through the Hadoop protocol. On the Open Science Grid, the primary access methods are through file system independent middleware such as the Storage Resource Manager (SRM) [67] and XRootd [27]. They provide a translation layer from system independent grid protocols and security mechanisms to the underlying storage system, such as Hadoop. The Storage Resource Manager (SRM) is a previously popular protocol to manage remote distributed filesystems. It is a standardized protocol that allows remote, distributed access to large storage with APIs to balance transfers among many data servers.

Beyond storage access methods are storage schedulers. These schedulers do not define a protocol to access the storage, rather they coordinate the access. NeST [14] is a software-only grid aware storage scheduler. It supports multiple transfer protocols into a storage device, including GridFTP [7] and NFS [77]. Further, it provides features such as resource discovery, storage guarantees, quality of service, and user authentication. It is layered over a distributed filesystem to provide access to it. NeST functions as the interface and access scheduler for a storage device. Features such as the storage guarantees and quality of service require NeST to be the only interface into the storage device, a very rare feature in today's grid storage. Today's storage elements, such as the four petabyte Hadoop Distributed Filesystem (HDFS) instance at University of Nebraska–Lincoln, include multiple protocols and interfaces to access the storage element. Nebraska runs four methods of accessing and modifying storage [11], SRM [67], GridFTP, XrootD, and FUSE [69] mounted Hadoop. All of these methods are required for compatibility with different access patterns and clients. NeST could implement each of these protocols, but it would be extremely difficult to manage the storage centrally. For example, FUSE is mounted on all 300 worker nodes. The GridFTP and XrootD servers run on 10s of servers, with an aggregate bandwidth of 10 Gbps. Scaling quality of service and storage allocation/enforcement across all of these access methods would likely prove impossible.

None of the storage management technologies discussed work for limited-duration data. This is data that is tied to a particular workflow, and only needs to be available during that workflow. The storage technology would need to have information about the workflow in order to implement this feature, or provide a clear eviction policy that the user could manage.

2.2.2 Data Services and Management

Storage systems may exchange data via a number of transfer mechanisms, such as BitTorrent. BitTorrent has been used for data transfer in computational grids by

Wei, Fedak, & Capello [79, 80, 81]. It has been shown to improve data transfer speeds when compared to traditional source and sink transfer methods, such as FTP [56], the base protocol to GridFTP, a grid enabled FTP protocol. The researchers did not compare performance of the BitTorrent protocol when compared to modern grid transfer techniques, such as using HTTP caching. Further, the authors did not test BitTorrent transfers across diverse network topologies that are common on the grid. A worker node from one cluster may not be able to communicate directly with a worker node from another cluster. Therefore, BitTorrent may not work between clusters but will work inside clusters.

GridFTP is a grid enabled version of FTP. It uses grid security in order to authenticate transfers between storage resources. GridFTP transfers files in multiple TCP streams in order to optimize transfer services. But, it does not have data services features such as automatic retry of failed transfers and transferring entire directories.

Globus Online [30] is a web interface for transferring files between sites and sharing data with other users. It offers an intuitive web interface for bulk transfers between endpoints. It expands on the capabilities of GridFTP to provide automatic retry of failed transfers and the ability to transfer entire directories. It only supports the GridFTP [7] transfer protocol and requires GridFTP implementations at all endpoints. Further, it does not offer either the ability to find data or to find storage resources that fulfill a set of requirements. Since Globus Online has these limitations, it fits into the "Data Services" category, and only provides limited "Data Management" capabilities.

There are also popular data transfer tools used on clusters such as secure copy (SCP) from OpenSSH [53] and rsync [6]. SCP is a simple copy tool that uses the Secure Shell (SSH) protocol to transfer files from a source to a client. Rsync is also able to copy files from a source to a client, but it can also do differential copies, where only the changed portion of a directory will be copied at a time. Both of these methods are used heavily when the data size is small. But they both use single stream TCP in order to transfer data, which in practice has been shown to be slower than multiple TCP streams as used in GridFTP [7] or BitTorrent.

2.3 State of Practice in Campus Computing

In order to illustrate the available technologies on the grid, we will begin with a common user application. We will then describe the technologies that could enable this computing on the grid.

2.3.1 Use Case

We will focus on the use of a commonly used biology application, BLAST [9]. BLAST workflows typically include the following files:

- **Executable:** The BLAST application is distributed with many executables, such as `blastp` and `blastx`. They are run on the remote execution resources and produce output. In this dissertation, we will be primarily using `blastp`, which searches protein sequences.
- **Database:** A large listing of sequences that will be scanned for similarities to the sequences in the query files.
- **Query Files:** A small list of sequences that will be compared to the sequences in the database to find similarities.

Each of these files have different properties. The executable is relatively small, often 10s of megabytes, but identical copies are used by all executions of BLAST.

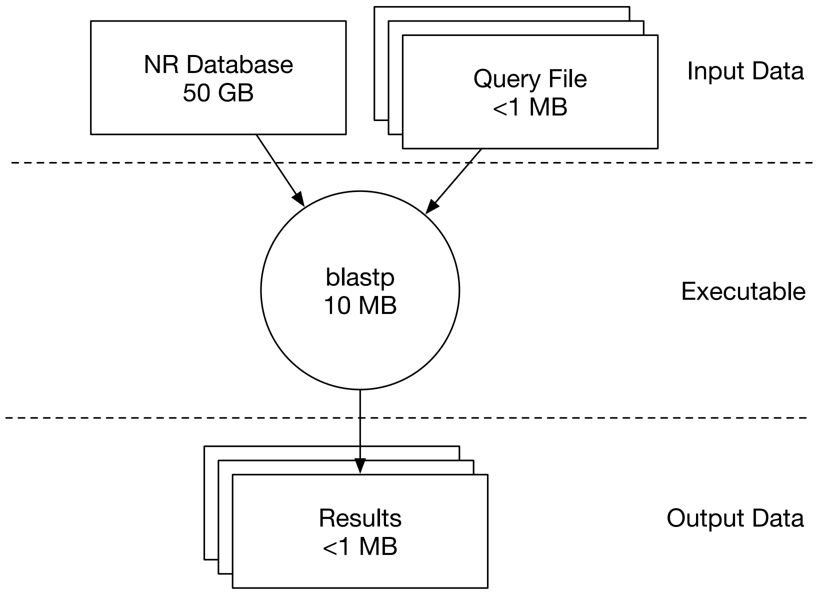


Figure 2.6: Blast Workflow

The query files are unique to each job but are typically very small, not exceeding 1 MB. The database is a large collection of proteins that is searched for each protein in the query file. Many databases are publicly available for use in BLAST. The most common is the non-redundant (NR) database, which is currently 50 GB and updated weekly. A diagram of the workflow is shown in Figure 2.6. The workflow can be broken into hundreds or thousands of jobs by splitting the queries into subsets of the larger query.

2.3.2 Current Approach

If the user has a BLAST application and wishes to run a large number of jobs, they must first gain access to computational resources. They may have access to a campus cluster. In that case, they will login to the campus cluster. The first step for the user is to learn the scheduler language. There are many different languages, such as PBS, Slurm, or HTCondor; all have slightly different syntax.

Once the scheduler language has been learned enough to write a submission file, the data must be transferred to the cluster from their laptop. This is usually accomplished with a tool such as SCP from the OpenSSH [53] package. This will be transferred slowly as SCP only uses a single encrypted stream to send data. The 50 GB NR database, transferred from the user's wireless connected laptop to the cluster could take two hours (at 54 Mbps), if nothing goes wrong with the transfer. The NR database is updated frequently, therefore frequent re-transfers of the database will be required when future BLAST jobs are submitted.

The data is managed on the cluster's shared filesystem. The filesystem is available on every worker node. The BLAST executables will read the database directly from the shared filesystem.

Once the data is on the cluster, the user will submit the jobs to the scheduler to process the data. The BLAST database will be copied for each and every execution to the execution resources from the cluster's shared filesystem.

The user must wait for the BLAST jobs to complete before checking the results. If the user has more BLAST jobs to process, they may submit yet more jobs to the cluster, or to another cluster on the campus. If the user submits to another cluster, they must re-transfer the database to that cluster, and submit the BLAST jobs.

Once the computation has completed, the user will copy the output data back to their laptop for further analysis. If the user submitted to multiple cluster, they will need to copy results from each of the clusters.

2.3.3 Issues with Current Approach

There are many issues with the current approach:

- Users must learn one or more scheduler languages. If users want to submit to only one cluster, then they only need to learn one submission language. But if their demands grow, and they need more resources, they will need to learn another programming language.

- Data copies are very expensive and should be minimized. The NR database is updated frequently; therefore, it must be updated on the cluster frequently.

- Once on the cluster, each job will need to copy the NR database to the worker node in order to process it, even if was already copied by a previous job. This copy will happen every time, as there is no caching in the vast majority of distributed filesystems.

- No storage management is used for the input data, executables, or output. This data may not be cleaned up after the jobs completion, therefore depleting the storage resources available for future workflows.

Chapter 3

Campus Job Distribution

3.1 Introduction

Campus grids are usually defined as a set of clusters available to researchers. These clusters are divided either by purpose, i.e. owned by a certain group, or by hardware generations. Users submit to a single cluster, and their jobs are eligible to run on that cluster. We propose a framework, Bosco, to distribute jobs to multiple campus clusters transparently to the user.

There are many challenges for researchers on campuses with multiple, distinct clusters. For example, a researcher may not know which cluster may run their jobs the quickest. The researcher may not be able to determine which campus cluster will start their jobs first. Each of these challenges can force the researcher to make decisions with insufficient information that may be suboptimal and delay their workflow's time-to-completion causing costly research delays.

Traditional cluster computing requires the user to log into the cluster and submit their processing and data there. However, researchers are most comfortable on their own laptop computer. A submission method to enable job processing to originate from the user's laptop to be run on a remote cluster would provide a more convenient user interface.

Another challenge for researchers attempting to use these clusters is the inability to install applications. In order to protect the clusters' integrity, campus clusters only give researchers limited capabilities on the resources.

A number of different methods have been used to distribute jobs across multiple resources. Inside a cluster, schedulers such as HTCondor [47] and PBS [39] have been used. Neither of these schedulers are typically used to submit jobs from users' laptops, a key requirement in improving the user experience. Compute Element based remote submission is heavily used in computational grids, and they use technology such as Globus [29] and UNICORE [63]. This remote submission requires compute element software installation on a server inside the cluster, which requires an administrator.

Bosco [82] is used to effortlessly create a remote submission endpoint on a cluster without requiring the administrator to install any software. Bosco is a remote submission framework based upon HTCondor. It uses the SSH [87] protocol to submit and monitor remotely submitted jobs. Additionally, it performs file transfers using the same SSH connection as the submission.

Improving the user experience was a primary goal of Bosco. We addressed the user experience by improving the interaction with the user during installation and configuration. Another problem area we found is when a user must debug issues with distributed software. In order to address this, we created a traceroute [48] like utility. The traceroute utility tests every step of the job submission process, from network access to a properly configured remote scheduler. If an error is found at any step of the traceroute, a useful message is given to the user, including possible steps to fix the problem.

In this chapter, we will discuss the methods developed to aid in distributed scientific computing on a research campus using Bosco.

3.2 Bosco Architecture

The Bosco user experience can be described in two sections: installation/configuration and running jobs. Each of these areas was approached with the goal of improving the typical user experience for installing and running distributed computing software. The architecture of Bosco is shown in Figure 3.1.

The Bosco architecture is divided into the submit host and the cluster login node. The submit host is where the user submits their jobs and where the user interacts with the Bosco system. The login node is the submit host for a cluster. The login node is assumed to not be maintained by the user submitting to Bosco. The login node has access to the local scheduler with commands such as `qsub` and `qstat` (for PBS).

3.2.1 Installation & Configuration

Though typically separate, Bosco combines the installation and configuration steps in order to improve the user experience. Both are done by a single script, the `bosco_quickstart` script that follows several steps:

1. Determine the platform and download the appropriate version of Bosco (supports Mac, EL5/6, Debian 6)
2. Install Bosco into the user's home directory.
3. Prompt the user for details of connecting their first cluster to Bosco.

The script downloads the Bosco binaries from a central server to the submit host. Bosco is installed into the user's home directory by default in order to enable non-root installations. Connecting a cluster to the Bosco submit host requires configuring the secure connection to the cluster, and installing a small amount of software on the submit node that will be used for job submission and job status checks.

The installation on the user's laptop and on the remote cluster do not require administrator access. All files and applications are installed in the user's directories.

We have also created a native installer, a PKG, for the Mac version of Bosco. It is distributed in an Apple Disk Image (DMG) for consistency with other Mac software. Unlike the Linux installer, it does not automatically configure a cluster at first installation.

3.2.2 Running Jobs

The image shown in Figure 3.1 shows the architecture of job submissions of a Bosco submit node. Job submissions are done from the Bosco submit host, which in turn submit to the connected cluster login node.

First, the Bosco submit node connects to the login node over an SSH connection and creates the forwarded SSH tunnel back to the submit node, which is used for file transfer. Figure 3.2 shows the network usage for Bosco. SSH was chosen as the protocol, since it is used nearly universally for cluster access. It creates the forwarded SSH tunnel in order to minimize the number of connections between the Bosco submit host and the login node. By reusing the same SSH connection, the number of SSH logins is reduced to one. Further, the number of connections is not dependent on the number of jobs, as Bosco will reuse the same connection for all jobs submitted to a cluster. Minimizing the number of connections to a login node is important since

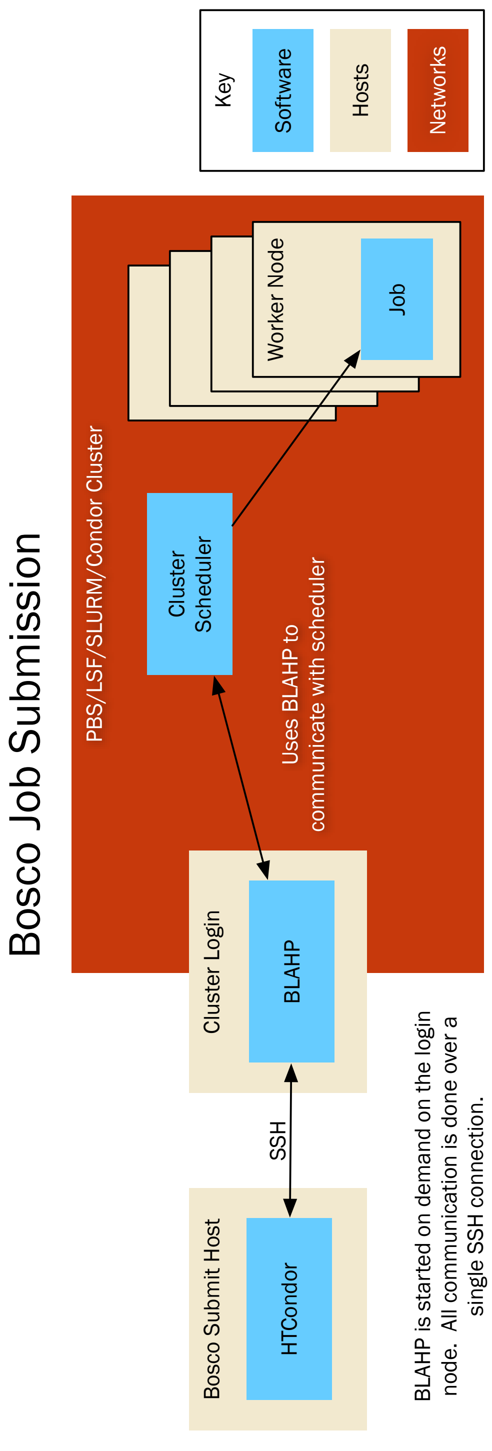

Bosco Job Submission
Key
Software
Hosts
Networks
PBS/LSF/SLURM/Condor Cluster
Worker Node
Job
Cluster Scheduler
Uses BLAHP to communicate with scheduler
Cluster Login
BLAHP
SSH
Bosco Submit Host
HTCondor
BLAHP is started on demand on the login node. All communication is done over a single SSH connection.

Figure 3.1: Bosco Architecture

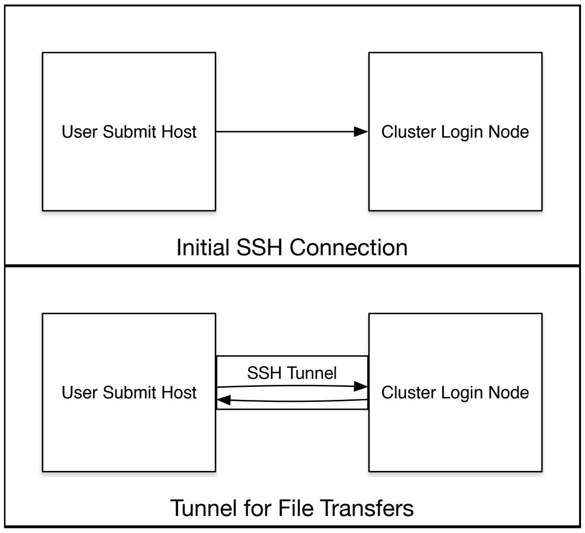


Figure 3.2: Bosco SSH Network Connections to the Cluster Login Nodes

many login nodes include firewall rules to slow down brute force SSH logins that block frequent successive SSH connections. The Bosco submit host does not require any open ports in a firewall, only outgoing connections to a remote login node's SSH port.

Next, Bosco checks for the necessary installed software on the login node, and starts the BLAHP [60] daemon that will communicate with the scheduler on the login node. The BLAHP daemon on the login node starts the file transfer daemon to connect back to the submit host through a forwarded SSH tunnel that Bosco created. The transfer daemon creates and transfers the job sandbox to the login node. The transfer to the login node is performed by HTCondor's fault tolerant file transfer mechanisms and is entirely encrypted over the SSH connection. Authentication between the login node and the Bosco submit host is performed by a shared symmetric key that was previously sent out-of-band over the SSH connection to the BLAHP daemon. Integrity of the files is ensured by using HTCondor's built in integrity checking of all communication between daemons.

After the files have been transferred, Bosco sends the job's submission ClassAd [59] to the BLAHP daemon to translate to the local site's scheduler language. The BLAHP supports PBS [22], LSF [23], SGE [36], Slurm [88], and HTCondor schedulers. Since the BLAHP will translate the HTCondor job into any of the above schedulers, there is no need to write submission scripts in anything other than the HTCondor language. The BLAHP creates the submission file, and submits it to the cluster's scheduler. Bosco periodically polls the BLAHP over the SSH connection for the status of the job. Once the job is detected to have been completed, the BLAHP starts HTCondor's transfer daemon to transfer the output sandbox back to the submitter machine.

Figure 3.3 shows the job flow from the user to the Slurm scheduler on the remote cluster. As you can see, it goes through six daemons before reaching the cluster scheduler. The HTCondor job manager agent (SchedD) accepts the user's jobs, acting as an agent which will attempt to execute the job. HTCondor's grid interface daemon (GridManager) is spawned by the SchedD in order to service the Bosco job. The RemoteGahp (Grid Ascii Helper Protocol) starts the SSH tunnel between the user submit host and the cluster. The BLAHP (C portion) interprets the job requirements, and executes the BLAHP (shell portion) which translates the requirements into the local scheduler language. Finally, the BLAHP submits the jobs to the Slurm Scheduler.

Bosco has two modes of job submission:

1. **Direct** – A single job on the Bosco submit host corresponds to a single job on the remote cluster. Each job is submitted individually to the remote cluster's scheduling system. This method is the simplest to run and imposes no special requirements on the submit machine.

2. **Glidein** – Bosco submits many pilot jobs to the remote cluster using the direct method. But each of the pilots can service multiple user-submitted jobs from

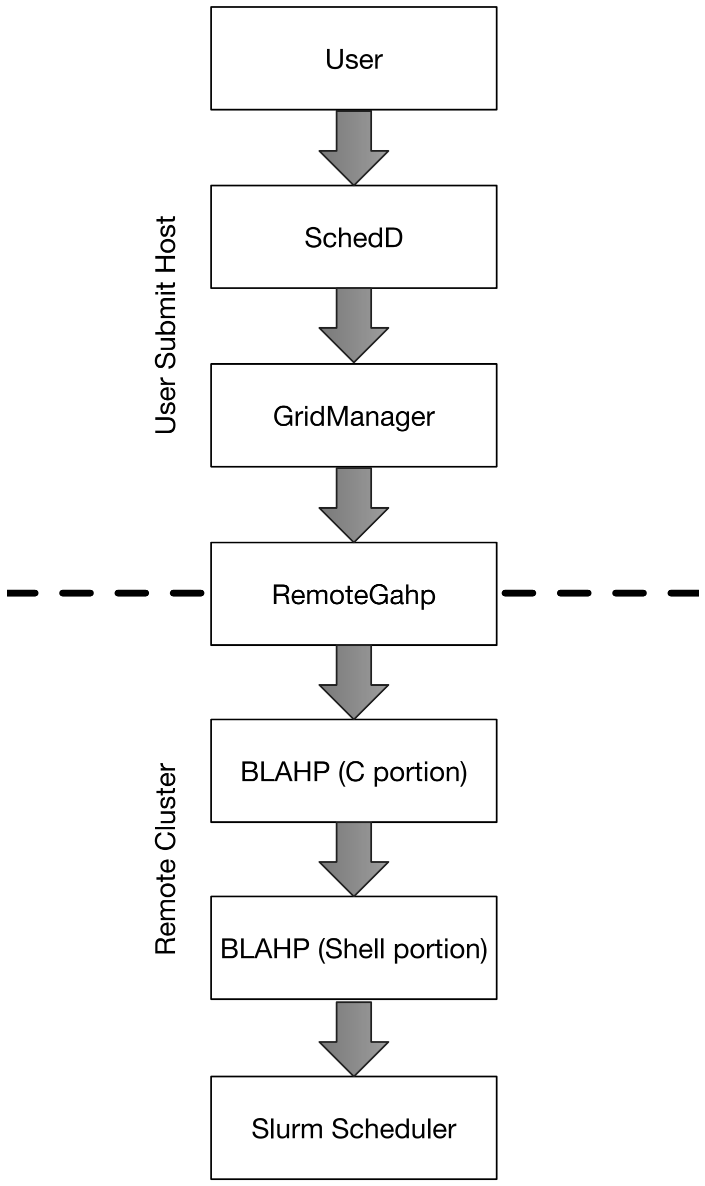


Figure 3.3: Bosco Job Submission Flow From Submission Host to Remote Cluster

the Bosco submit host. This method minimizes the overhead on the remote cluster since Bosco is not submitting many jobs through the cluster scheduler. The glidein method requires that the Bosco submit host can be contacted from the remote cluster worker nodes. The glidein submission method is based off of previously written software, the Campus Factory [84].

The two modes of job submission allow users to optimize for their environment. If they are running many short identical jobs (which is frequent in high throughput computing), then the glidein method is ideal for them. The glidein pilots are much faster at starting jobs than the direct method (as will be seen in Section 3.4.7).

If the users are running fewer, longer, and possibly unique jobs, then the direct submission method would work best. When submitting using the direct method, different requirements may be specified for each job. In contrast, all glideins request the same resources, and are therefore not ideal for heterogeneous jobs. We expect most users will start with the direct method then graduate to glidein once they become accustomed to submitting batch jobs.

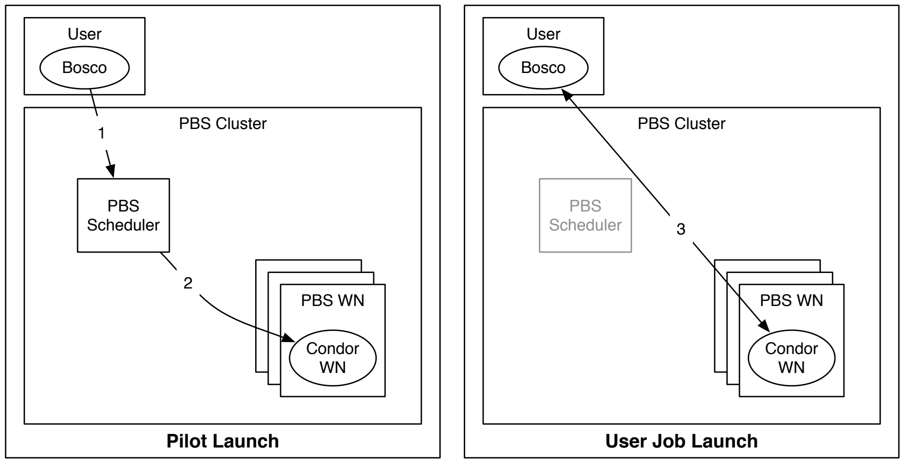


Figure 3.4: Bosco Glidein Launch

Bosco glidein pilot submission is depicted in Figure 3.4. (1) Bosco first submits a glidein pilot to the remote PBS cluster. (2) Once the pilot starts on a worker node, it will connect to the user's Bosco instance to request work. (3) Jobs will then be run on the PBS cluster's worker node under the Bosco pilot.

Bosco's fault tolerance is in its handling of the remote cluster. For example, if the user's computer loses connection with the remote cluster due to network issues, or even if the user suspends their laptop, Bosco will place the jobs on hold. Although no new jobs will be submitted to the remote cluster, jobs that were already submitted will continue to run. Further, when Bosco re-establishes a connection to the remote cluster, Bosco will check the status of already submitted jobs. It will then bring back any output data from any jobs that have completed and continue to submit jobs to the remote scheduler.

3.2.3 Improving User Experience

In order to improve the user experience at each step of the job process, extra effort has been given to provide useful error messages in case of failures. For example, HTCondor was modified to relay the standard error for any commands sent to the BLAHP daemon, as useful debugging information is available there.

Also, an additional traceroute was created to test each step of the job submission process, including:

- **SSH connection to the remote login host:** Tests network connection, login host availability, and passwordless SSH setup.
- **Job submission to the Bosco submit host:** Tests Bosco daemon availability and Bosco submit host file system availability.
- **Job submission to the remote login host:** Tests the remote scheduler availability, remote cluster software setup, input file transfer, and cluster file system availability.
- **Job completion and status update from login host:** Tests Bosco status check process and output file transfer.

The traceroute is useful for debugging issues with a Bosco installation. It is designed to test each step in the job execution life cycle and give meaningful error messages and possible solutions.

```
$ bosco_cluster -t glidein.unl.edu
Testing ssh to glidein.unl.edu...Passed!
Testing bosco submission...Passed!
Submission and log files for this job are in
    /home/swanson/dweitzel/bosco/local.bosco/bosco-test/boscotest.ibhJN
Waiting for jobmanager to accept job...Passed
Checking for submission to remote condor cluster (could take ~30
    seconds)...Passed!
Waiting for job to exit... this could take a while (waiting 60
    seconds)...Failed
The job did not end in 60 seconds.  This is not always a bad thing...
Maybe condor is waiting longer to poll for job completion?
Here is the current status of the job:"


-- Submitter: red-foreman.unl.edu :
    <129.93.239.170:11000?noUDP&sock=15147_7985_2> : red-foreman.unl.edu
ID      OWNER            SUBMITTED     RUN_TIME ST PRI SIZE CMD
38529.0   dweitzel        7/1  00:17   0+00:00:00 I  0   0.0  echo Hello
```

Figure 3.5: Session Log From Bosco Traceroute

Figure 3.5 shows the session log from a successful Bosco traceroute. It first tests the SSH connection to the remote host, `glidein.unl.edu`. Then, it creates a test job, which it submits to the local Bosco instance. It checks that the local Bosco instance accepts the job, and that Bosco submits the job to the remote cluster. Then, the traceroute waits 60 seconds for the job to exit. In this case, the job did not exit in 60 seconds because there was no available resource to run this test job. Traceroute then tells you the current status of the job before it exits.

3.3 Load Balanced Access to Computational Resources

Bosco is used in conjunction with the Campus Factory [83], which is described in full in my Master's thesis [84]. The Campus Factory submits pilot jobs to remote clusters to create an overlay that provides a consistent interface to the resources. The campus factory submits to multiple Bosco endpoints simultaneously, load balancing between them by keeping idle jobs (constant pressure) on all clusters. PBS then provisions resources for the Bosco pilots. The pilots call back to the Bosco submit host in order to request jobs.

Submission via the Bosco framework is done using the HTCondor submit file language. When an idle user job is detected by the Campus Factory, it begins to submit glideins to all Bosco endpoints simultaneously. The Campus Factory maintains idle glideins on each of the endpoints until the user's jobs have completed. The jobs run on any resources that become available.

3.4 BoscoR: Extending R from the desktop to the Grid

As a case study of improving the user experience when using Bosco and distributed computing, we created BoscoR, an interface to Bosco from the R statistical programming language.

3.4.1 BoscoR: Introduction

Usage of the R language [70] by data miners has grown much faster than any other programming language [62, 43]. Data mining requires computational resources, sometimes more computational resources than can be provided by their desktop computer. In a recent study of data scientists [62], "Available computing power" was the second most common problem for big data analysis. In addition, the respondents stated that distributed or parallel processing was the least common solution to their big data needs. This could be attributed to the difficulty of processing data with the R language on distributed resources, a challenge we tackled with BoscoR.

A reason that distributed computing is not seen as a popular solution to big data processing is that scientists are more familiar processing on their desktop than in a cluster environment. R is typically used by researchers that have not used distributed computing before and do most of their analysis on their local systems with integrated development environments (IDE) such as RStudio [57]. Users are unaccustomed to the traditional distributed computing model of batch processing in which there is no interactive access to the running processes.

Though researchers may not have experience with distributed computing, most have extensive computational resources available to them, either locally provided by their institution or university, or through national cyberinfrastructure such as the OSG [55] or XSEDE [2].

In this section, we will describe GridR [78], an R library used to interact with the Bosco framework. In section 3.4.4, we describe how we combined these two components to create a fault-tolerant framework that provides a positive user experience. Next, in Section 3.4.7, we discuss preferred submission methods based on the length of the R function, and show results from numerous test runs against a production cluster. Finally, some conclusions and future work are offered.

3.4.2 Background

BoscoR is primarily made up of two components, Bosco and GridR. Bosco provides a simple setup interface to the remote batch system, and it provides fault tolerance for job submission and file transfers. GridR provides a user interface to create and initiate remote processing.

3.4.3 GridR

Many parallel libraries are available for R. Most focus on managing the R processing on a single server such as the `parallel` package [5]. The `parallel` package comes bundled with R and provides simple, single machine, multi-core parallelism. Parallelism is done by using variations of the R function `lapply`. A simplified definition of `lapply` is shown in Figure 3.6. `lapply` is the basis for nearly all parallel libraries in R.

`lapply`(**X**, **FUN**, **. . .**):

X a vector (atomic or list) or an expression object.

FUN the function to be applied to each element of X

. . . optional arguments to FUN.

Returns list of the same length as X, each element of which is the result of applying FUN to the corresponding element of X.

Figure 3.6: Function Definition of `lapply`

The `lapply` function is ideally suited for high throughput computing. There is no communication between executions of the function **FUN** on the input array.

The input vector can be partitioned in order to split the execution across multiple resources. Accordingly, most parallel applications use the `lapply` model to provide parallelism. Examples are the parallel package which defines the functions `mcapply` (multi-core apply) and `parapply` (parallel apply). In the `parallel` package, calling `mcapply` causes R to fork a process that will execute the function **FUN** on each element in the input vector. A similar process happens when calling `parapply`.

Another built-in parallelization package is Simple Network of Workstations (snow) [4]. Snow allows for multiple computers to organize and execute parallel processing of data. The computers communicate over regular network sockets or using MPI [37]. This allows for multiple computers inside the same cluster to process data. Snow also has built-in `lapply` style functionality.

GridR also follows this `lapply` model for parallelization. It uses a function called `grid.apply`, which will apply a function to every element of an input vector, similar to `lapply`. Instead of forking a process like `mcapply`, it compiles the input data and function, and submits the execution to a grid endpoint.

GridR was originally written for use with data analysis in ACGT clinico-genomics trials [78]. It was written with the capability to submit with a limited set of grid protocols, some of which are no longer supported. Further, GridR made assumptions of the remote resources. These assumptions were:

- R is installed on all of the worker nodes.
- The R binaries are in the same location on all of the remote resources.
- The GridR package is installed on all of the remote resources.

All of these assumptions cannot be met on modern grid resources. Applications cannot assume that a (non-standard) processing tool, such as R, is installed on every computer or that it is installed at exactly the same location on all clusters in the grid. Modifications were made to GridR to erase these assumptions, as well as to adapt it to submit to Bosco.

3.4.4 Implementation

Bosco is designed to run on resources that are not controlled by the submitting user. Further, it is designed to run on resources without any conditions as to what is installed. In order to operate under these assumptions, Bosco must bootstrap itself by bringing in all the libraries and dependencies required to operate. BoscoR must run under these same assumptions.

GridR was modified to submit to Bosco. The input generation of GridR is shown in Figure 3.7. When a user or script calls `grid.apply`, GridR compiles the input function and input data into a R data file, which can be read later by another R process. GridR handles function dependencies by using R's dependency detection and compiles any functions that may be required into the input.

GridR was modified to first create a submit file which will be submitted to the local system where Bosco is installed. The submit file explicitly lists the input files and the expected output file, all of which will be transferred by Bosco. The input files, as shown in Figure 3.7, are the compiled function and input data and a bootstrap executable. The output file contains the return value from the executed function.

The submit and polling script is executed by GridR after forking a new R process. This is a lightweight process that submits the Bosco submission script and watches for any errors. If the input function is executed many times by many separate jobs, the polling script will aggregate the results as they are returned to the submit node into a vector that will be returned to the user.

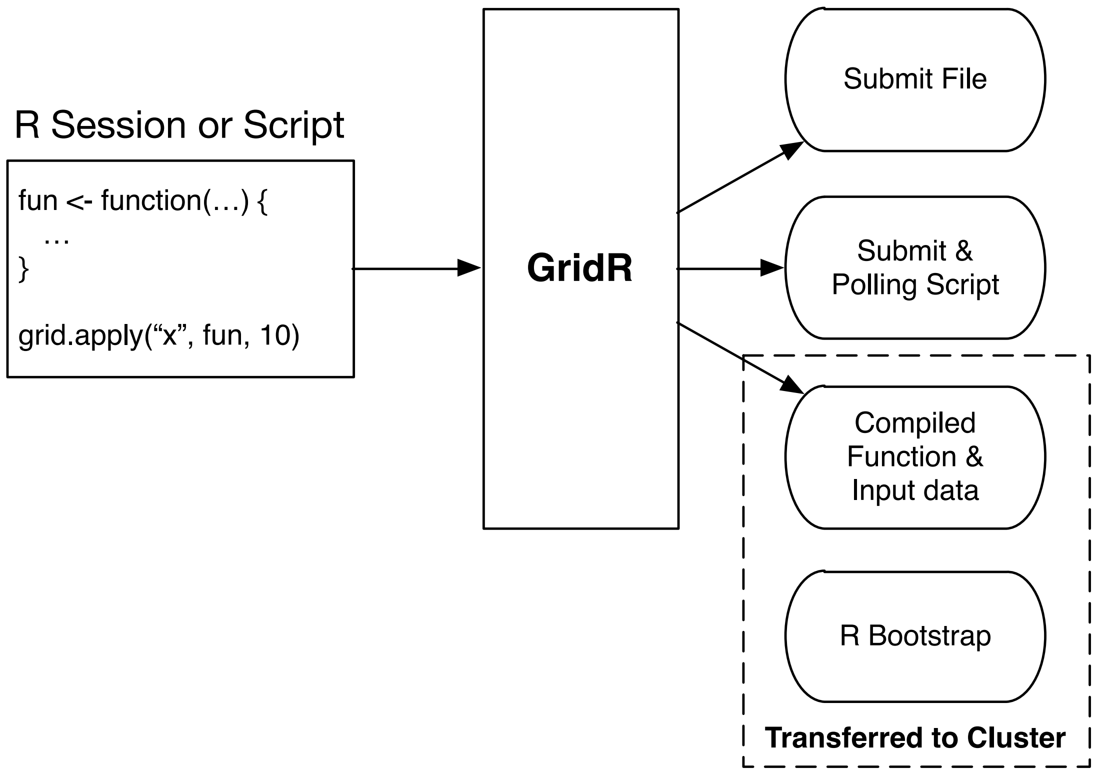


Figure 3.7: GridR Input Generation

3.4.5 Bootstrap

Since Bosco cannot make assumptions as to what is installed on the remote cluster, neither can BoscoR. Therefore, GridR was modified to detect, and if necessary install, R on the remote system. This was performed by a bootstrap process.

In the GridR generated submit file, the listed executable to run on the remote system is not R, but the bootstrap executable. The bootstrap executable detects if R is installed on the remote system. If it is installed, it simply executes the user defined function against the input data. If R is not installed, the bootstrap downloads the appropriate version of R for the remote operating system. The supported platforms are identical to Bosco's: CentOS 5/6 and Debian 6/7. R is downloaded from a central server operated by the OSG's Grid Operations Center [3].

The bootstrap executable installs R in a shared directory. By utilizing a shared directory, subsequent GridR jobs may use the same R installation. Several bootstrap jobs could start at once on a remote cluster, so a simple transactional file-locking mechanism was devised so only a single bootstrap executable on a cluster will download and install for the entire cluster if a shared filesystem is available. If a shared filesystem is not available for installation, R is installed in a temporary directory that is removed upon job completion.

3.4.6 Running on the Open Science Grid

Submitting to the Open Science Grid (OSG) is done by direct submission. The OSG hosts access nodes which can be used to submit to resources on the grid.

Most OSG sites do not have shared directories for grid users. Therefore, the bootstrap script must install R on every node in a temporary directory unique to each job. In order to optimize the R installation, the bootstrap script utilizes the HTTP forward proxy infrastructure [35] available on the OSG to minimize requests to the central hosting server.

3.4.7 Results

The results section is broken into two categories: results from user feedback and results from experimental runs on a production cluster, as well as the OSG.

A primary goal with BoscoR was to improve the user experience of using R on distributed resources. After acquiring a few users, we received feedback on how BoscoR could be improved. The improvements to GridR and the integration with Bosco made working with campus or institutional resources much better. In this section we will describe the improvements.

3.4.7.1 User-Provided Packages

Many users require additional packages to be installed before their function can execute. It was assumed that most of these packages would be in the major R package repositories, such as the Comprehensive R Archive Network (CRAN) [1]. If the package is in CRAN, the user-provided function can install the package. After receiving user feedback, it was found that not all desired packages are available in CRAN. A modification to both the submit file and the bootstrap executable was designed to install such packages.

In order to install a user-provided package, it first needs to be transferred to the remote cluster. This is done by including the package in the list of input files to be transferred by Bosco for each job. Additionally, the packages should be installed before the user function is executed on the remote resources. This required modification to the bootstrap executable in order to install the packages after installing R, but before executing the user's function on the input data. The packages are specified on the `grid.init` function as the optional argument `remotePackages` argument. An example function call would be:

```
#define a function that will be executed remotely
a<-function(s){return(2*s)}
# For the bosco.direct service, adding the remotePackages
grid.init(service="bosco.direct",
   localTmpDir="GridRTmp/",
   remotePackages=c("GridR_0.9.8.tar.gz"))
# Again, apply the function
grid.apply("x", a, 10, wait=TRUE)
```

Figure 3.8: GridR Function with Remote Package Installation Before Execution

Figure 3.8 shows the function call `grid.init` with the remotePackages attribute added to it. In this case, it is only installing the GridR package before executing the function `a`.

3.4.7.2 Quick Jobs

Since the GridR interface only provides for a single function to execute against the input data, it was assumed that the function would be a time-consuming data-processing function. Therefore, the overhead Bosco introduces would not significantly affect the performance of the executions. After receiving feedback from users, it was determined that the more common use case is to use smaller functions that could execute in seconds or minutes. In order to accommodate shorter jobs, a modification to the GridR generated submit script was required.

In this case, we modified the submit file so that Bosco would use the glidein method of job submission, described in Section 3.2.2. Using the glidein job submission method, the shorter jobs can be executed much more quickly, one after another, reusing the same resources. Additionally, this saves the remote cluster scheduler from scheduling many small jobs which can cause issues in many HPC schedulers.

3.4.8 Performance Results

3.4.8.1 Experimental Setup

To test BoscoR, we simulate a R workload with varying lengths of the executed functions. The length distribution is based on a variety of user workloads. As noted before, we assumed that the functions would be long-running data processing. But, we learned that users were instead submitting short functions to be executed quickly. We varied the length of function from one second to 30 minutes. To verify our solution to quick jobs, we tested different job lengths using both the direct and glidein submission method.

To execute the test jobs, we used the production cluster, Tusker, at the University of Nebraska – Lincoln Holland Computing Center. This cluster is composed of 106 nodes, each with 64 cores, for a total of 6,784 cores. The cluster has numerous users that are submitting to the central Slurm [88] scheduler. The cluster traditionally runs at >90% utilization, with dozens of users' jobs fair sharing the resources. The Slurm scheduler is a HPC oriented scheduler that matches submitted jobs to resources. Fair share scheduling is used on Tusker; each group has equal priority with all others, therefore allowing the maximum number of users to run on the resources.

Tusker is utilized enough that the jobs would be competing against other users' jobs for available resources, and therefore, not all submitted GridR functions would be able to execute simultaneously. We believe this best represents most clusters, which are typically highly utilized by many researchers. It is plausible that a cluster could be so highly utilized that no user jobs could be executed. The other extreme could also be true: enough resources are available for all submitted jobs to be executed immediately. We found that Tusker utilization is somewhere between these two extremes. It is capable of running many, but not all, jobs submitted to it immediately. The rest will execute as resources become available.

For testing, we submitted 1,000 GridR jobs per test run. This level of jobs was chosen as a reasonable representation of workflows we have seen when helping users of GridR. They typically submit many jobs, sometimes reaching into the thousands. The goal was to submit more jobs than could be run instantly by the remote cluster.

3.4.8.2 Direct submission

First we look at Bosco's direct submission method. In GridR, the library is initialized with the argument `service="bosco.direct"`. When this setting is used, GridR generates submission scripts that use Bosco's default routing mechanism to submit to a single cluster. The GridR functions are submitted as jobs directly to Tusker's Slurm scheduler.

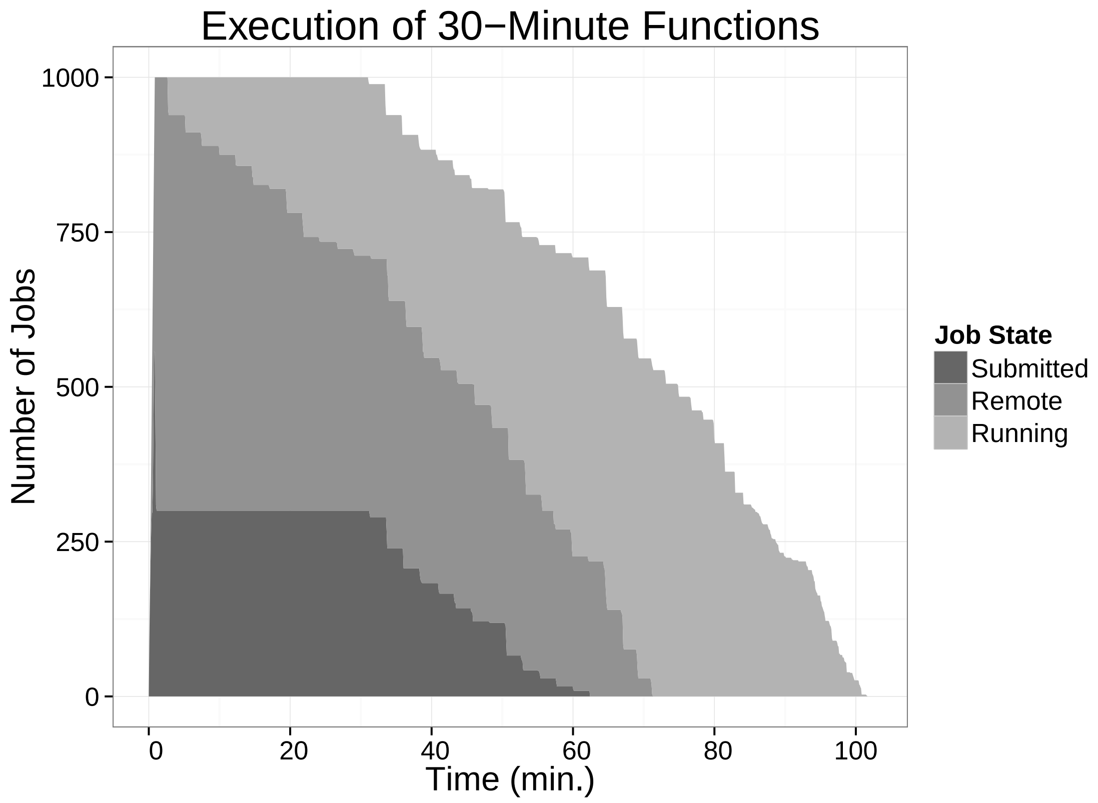


Figure 3.9: Direct Submission

A timeline of the GridR submissions to Bosco is shown in Figure 3.9. The submitted jobs are jobs which are submitted locally, but not yet submitted to the remote scheduler. Remote represents the jobs which are submitted to the remote Slurm scheduler. The submission to the remote scheduler is very rapid. Bosco is able to submit 700 jobs within a minute. Slurm is able to rapidly begin executing many, but not all, of the submitted jobs. Bosco maintains constant pressure in the form of idle jobs in case resources become available on the cluster. You will notice the straight line in the number of submitted jobs which dips after 30 minutes. At 30 minutes the first jobs begin to complete and Bosco begins to submit more jobs to the cluster, attempting to always keep the maximum of 700 jobs either idle or running on the remote cluster. In this workflow, all 1,000 30-minute jobs finish in just over 100 minutes.

The Open Science Grid runs use the direct submission method as well. But since the OSG access nodes run HTCondor, the jobs are capable of starting much quicker after being submitted to the remote resource. In this way, the OSG provides the best of both the direct and glidein approaches. It is simple to setup like any direct submission method. And as with the glidein submission method, the jobs start quickly on the remote resource.

The OSG direct submission presents different failure modes than a traditional HPC cluster. For example, in one of the experimental runs with one-second long jobs, a single job took over 30 minutes to complete. The issue with this particular job was that the job was matched to a single node that was misconfigured. In this case, the job eventually finished after being matched to a different node. The OSG may not be an ideal solution for short executions of GridR functions.

3.4.8.3 Glidein

Glidein submissions use a pilot job that is submitted directly to the Slurm scheduler. Once the pilot starts, it calls back to the submission node to request work. This method allows multiple jobs to run within the same Slurm job, independent of the cluster's scheduler.

A comparison of the direct submission to the glidein submission is shown in Figure 3.10. For longer jobs, direct and glidein submission methods have approximately similar workflow runtimes. But for short jobs, glidein has significantly shorter workflow runtimes. This can be attributed to the advantages of using a high throughput scheduler over a high performance scheduler.

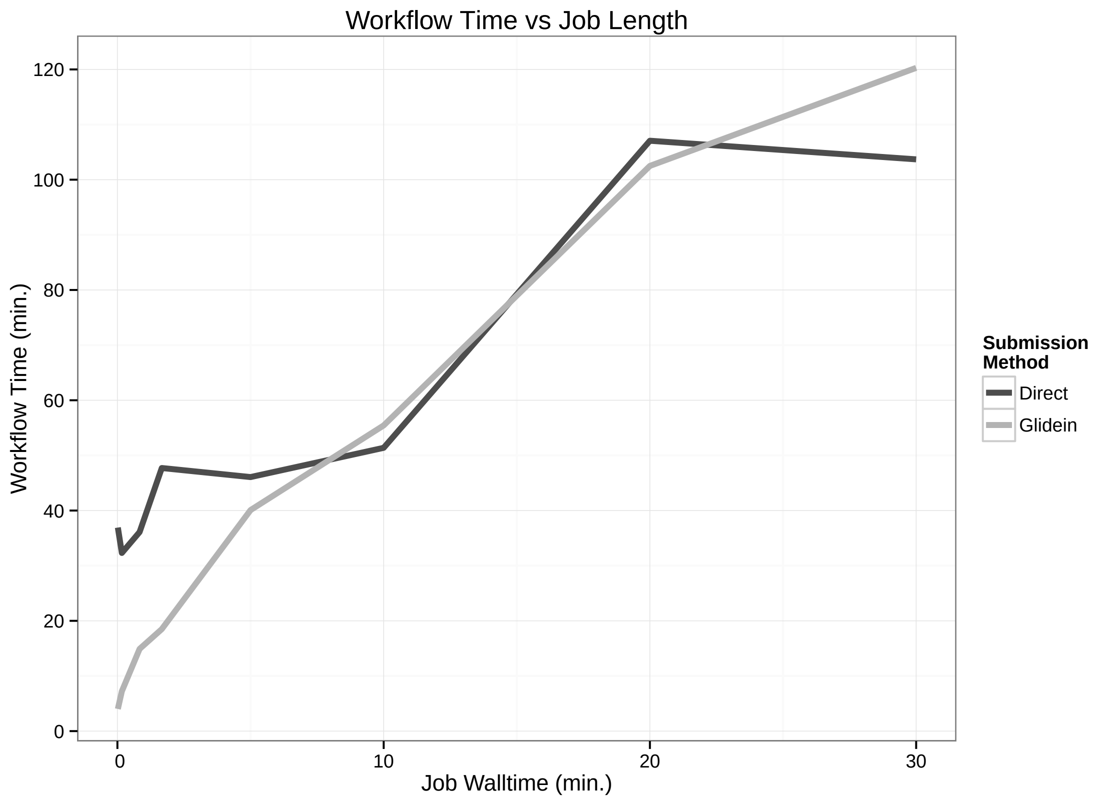


Figure 3.10: Comparison of Submission Methods

Bosco is built on top of HTCondor. HTCondor is a very efficient high throughput scheduler that can quickly start the execution of jobs on available resources. Since many R workflows are designed to run a short function upon a large amount of data, HTCondor is a good fit. By submitting HTCondor pilots to the remote scheduler, Bosco is able to utilize its strength of running many small jobs quickly, resulting in a shorter workflow completion time for shorter jobs.

Figure 3.11 illustrates how the glidein submission method is superior to the direct submission method for short jobs, and why both methods are roughly equivalent for longer jobs. For longer jobs, you can see that both glidein and direct submission methods start jobs at nearly equal rates. The variation is relatively small and could be explained by variations in the available resources at the time of running the

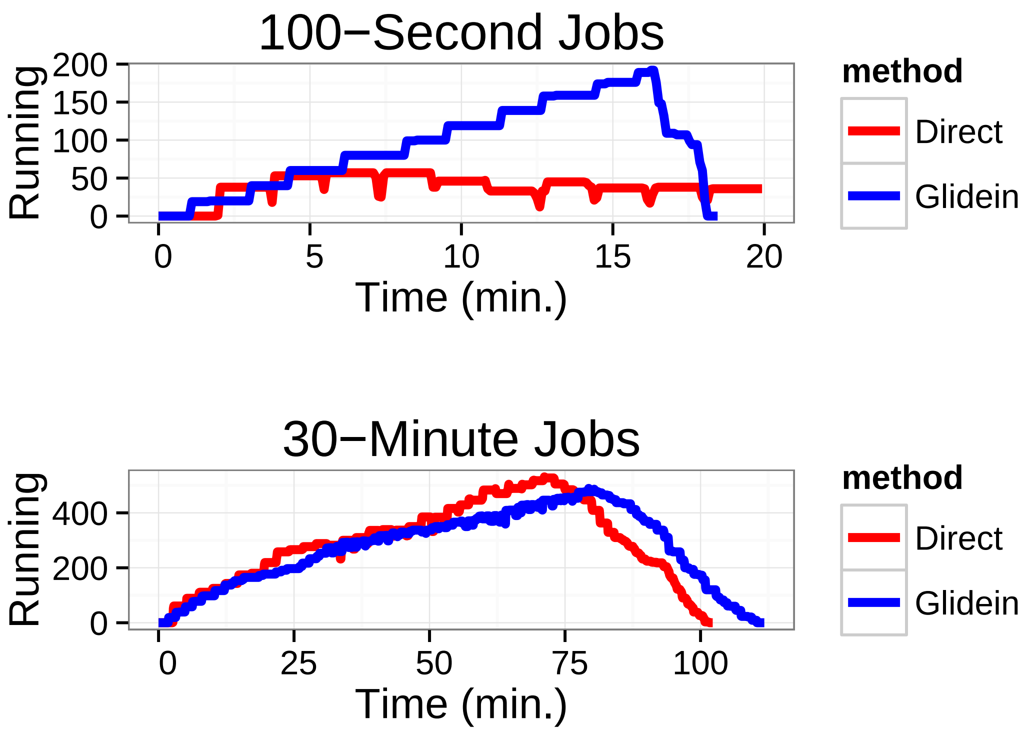


Figure 3.11: Number of Simultaneously Running Jobs

experiments.

For the shorter 100-second jobs, the start rate begins nearly the same, but then Bosco and Slurm are unable to sustain the job start rate. Since the jobs only run for 100 seconds, the overhead from Bosco submission and Slurm starting the jobs becomes a bottleneck. Bosco is only able to sustain roughly 50 jobs running on the cluster. Note that the direct submission method running jobs is truncated at the end of the graph to better illustrate the difference between the two submission methods. On the other hand, the glidein submission method continues to grow in the number of jobs running. This is due to eliminating the Bosco submission overhead, as well as the Slurm scheduling overhead. Instead, the glidein method is utilizing the much more efficient HTCondor scheduler, which is able to start jobs much faster than the

Slurm scheduler. We can conclude that the glidein submission method results in a shorter total workflow execution time for shorter R functions.

3.4.9 BoscoR Conclusion

BoscoR is a framework to execute R functions on distributed resources. It is a simple method for users to distribute processing to remote resources. BoscoR incorporated user feedback in order to improve the framework.

As with any complicated system, many parameters can be varied in order to obtain different results. For example:

- Resource contention may be high, which could cause the cluster not to start any GridR jobs.

- Resource contention may be low, which would cause Slurm to start all submitted jobs immediately.

- The number of glideins submitted in a batch could be varied in order to optimize the start rate for a particular cluster. Any lower and it would slow job starts, increasing the workflow run time for both short and long jobs. Any higher, and it could overwhelm the remote scheduler.

We chose reasonable values for these parameters that an end user may use. In the future we will tune these parameters automatically using the feedback provided by Bosco. Although Bosco and the bootstrap process significantly improved the fault tolerance of GridR, further fault tolerance testing and development is needed to provide a positive user experience when running on national infrastructure such as the OSG.

During follow up interviews with users after using BoscoR, we received many positive reviews of the framework. Improving the user experience of using R on distributed resources was a primary goal of BoscoR. One example of a positive review was from a Micro-Biology researcher from the University of Wisconsin:

> I have a huge set of data, which I have to split into pieces to be handled by each node. This is something I can do with the "grid.apply" function. This reduces the submit time from several hours, to several seconds... it is a phenomenal improvement. This will greatly increase my use of grid computing, as right now, I only use grid computing when I have no other choice.

The experimental testing we ran showed that the glidein submission method is significantly better at running short R functions than the direct submission method. At longer job runtimes, the difference between direct and glidein submission to remote resources is negligible.

3.5 Conclusion

In this section, we showed that Bosco transparently and effectively distributes computational jobs across multiple clusters on a campus, while maintaining simple usage for users. Using Bosco as a foundation, we were able to create an interface from the R programming language called BoscoR. This interface proved straight forward for users to use.

During the evaluation of BoscoR, it was demonstrated that the direct submission method is slower for quick jobs than the glidein submission method. For longer jobs, there is almost no difference between direct and glidein jobs.

Bosco's usage has increased since we originally published the Bosco paper. For example, it is heavily used by the University of Chicago in order to submit OSG processing to opportunistic resources around the country. They find Bosco useful since it does not require the installation of any software on the remote cluster. Additionally, the CMS experiment has used Bosco to access opportunistic resources around the world, and they have published several papers on the subject [41, 54, 76, 45].

If the user's workflow requires significant data, it may be inefficient to use Bosco's transfer mechanisms which are bottlenecked by the transfer speed of the Bosco submit host. It is expected that large datasets will not be an efficient use of Bosco. Therefore, the framework to efficiently transfer data, discussed in Chapter 4, is needed to complement Bosco's overlay network.

Chapter 4

Campus Data Distribution

4.1 Introduction

Large input datasets are becoming common in scientific computing. Unfortunately for campus researchers, the staging time of the datasets to computational resources has increased along with dataset sizes. The typical large dataset workflow may consist of thousands of individual jobs, each sharing input files.

In the Background, Section 2.3, we described a common application, BLAST. Figure 4.1 shows the input and output of the BLAST application. Each BLAST query requires an entire reference database, which can range in size from a few kilobytes to many gigabytes. The workflow to run a BLAST query requires a large stage-in time in order to make the reference database available. Additionally, the databases are frequently updated with new entries.

Users in the past have copied the database using various methods. The naïve method includes copying the database for each job. Storing the database on a shared filesystem has the same effect as copying the database for each job, since the database must be transferred to the execution node for each job. This may reduce the effective

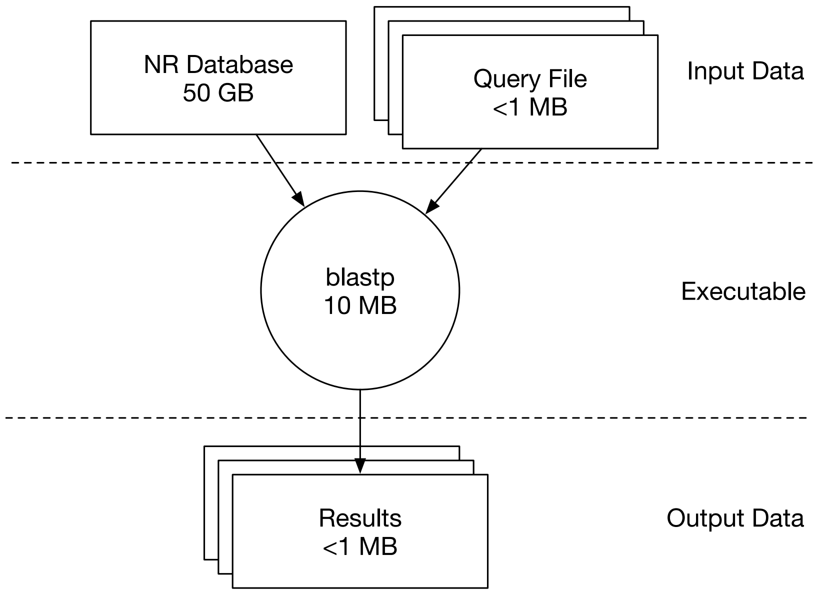


Figure 4.1: Blast Workflow

parallelism of the users' jobs, because only as many as have managed to squeeze their input data through the network bottleneck can run at one time. We find that the BLAST workflow described above is common among large data researchers.

Our goal is to minimize the number of times the input data is transferred from source to the execution resource by introducing the concept of shared data and prestaging this data to nodes. The primary concern for our use case is the cost of delivering input data over the network. Although reducing network usage alone is not guaranteed to increase computational throughput, it is part of the solution.

Additionally, the campus resources made available to researchers are shared; therefore, the researchers may have the limitation of not having access to install programs on the clusters. In Chapter 3, Bosco [82] built an overlay on top of campus resources to create a virtual, on-demand pool of resources for task execution. We expand the capabilities of this virtual pool to include data caching and novel transfer methods to enable big data processing.

We limit the design and analysis to a campus cluster computing environment. Our solution is unique in that it is designed to run opportunistically on the campus computing and storage resources. Additionally, it does not require administrator intervention in order to create a virtual, on-demand pool of resources.

4.2 Background and Related Work

Data caching on distributed systems has been used many times and at many levels. Caching can be done on the storage systems and on the execution hosts, as well as within the infrastructure separating the two.

Some distributed filesystems use local caches on the worker nodes. GPFS [65] has a read-only cache on each worker node that can cache frequently accessed files. It is designed for a fast, shared filesystem and is recommended when file access latency is a concern. It is not recommended for large files since internal bandwidth to the local disk is assumed to be less than the bandwidth available to the GPFS shared filesystem. GPFS file transfers are typically done over high speed interconnects which can provide high bandwidth for large files. These interconnects are not typically available to users' jobs for transferring data from a remote source.

HTTP caching is used throughout the web to decrease latency for page loads and to distribute requests among servers. In high throughput computing, a forward proxy is commonly used to cache frequent requests to external servers. The forward proxy caches files requested through it, and it will respond to subsequent requests for the same file by reading it from memory or its own disk cache.

The HTTP forward proxy caching does have limitations. The HTTP protocol was designed and is used primarily for websites. Websites have very different requirements than high throughput computing. The data sizes are much smaller [8]. Software designed as forward proxies, such as Squid [68], are optimized for web HTTP traffic, and therefore, do not handle large data file sizes optimally. Further, the Open Science Grid (OSG) [55] sites typically only have one or possibly a few squid caches available to user jobs. They are not designed to scale to large transfers for hundreds of jobs, the target use case. Figure 4.2 shows the caching architecture of a typical OSG site.

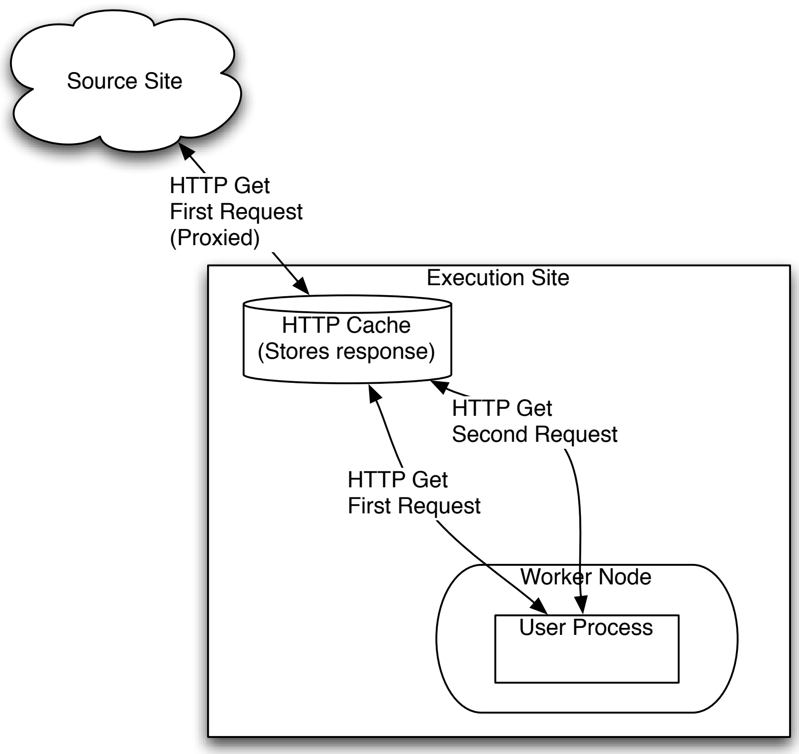


Figure 4.2: HTTP Cache Architecture

The working set of a job is the amount of data required for a job to run successfully. Each job in a workflow must have this working set available while it is executing on the resources. For workflows such as BLAST, the working set is the database and the query files.

A typical Squid server at an OSG site has between 32 and 64 GB of RAM, 100 GB of disk space, and 1-2 Gbps connection to the cluster's execution resources. Assuming a BLAST working set size of 50 GB, each Squid server could hold the working set of two BLAST workflows. But, other jobs are also using the Squid server, which will lead to cache thrashing of both the RAM and disk caches. There is no method to pin a working set to a Squid cache to keep it available during the lifetime of a workflow.

Parrot [75] is another application that will cache remote files when using certain protocols. Parrot uses interposition [74] to capture and interpret IO operations by an unmodified binary application. The interposition allows Parrot to provide a transparent interface to remote data sources. Parrot caches some of those sources such as HTTP with GROW-FS, a filesystem using HTTP. Parrot caches an entire file to the local storage. Parrot must download directly from the source the first time it is requested, exhausting WAN bandwidth quickly for large files. For example, in the case of the BLAST working set, 50 GB would need to be transferred the first time the working set is requested on a node. If 20 nodes are requesting the working set simultaneously, that equates to 1 TB, which will take two hours to transfer on a 1 Gbps connection.

CernVM-FS [16] provides a filesystem over the HTTP protocol. It integrates into the worker node system using the FUSE [69] interface. The CernVM-FS local node client caches files on the node, as well as using Squid to cache files at the site. CernVM-FS has optimized large file caching by chunking the files into smaller files. Neither the Squid caches nor the web servers optimally transfer large files, but they can transfer small files efficiently. Since CernVM-FS relies on Squid, it suffers from the same cache thrashing issues when working with large working sets. Further, CernVM-FS requires administrator access to install and configure, a privilege that campus users do not have.

XrootD [27] is designed for large data access, and it has even been used for WAN data transfers [12] using a federated data access topology. There has been some work in creating a caching proxy for the XrootD [13]. The caching proxy is designed to cache datasets on filesystems near the execution resources. The caching proxy requires installation of software and the running of services on the cluster. Unprivileged campus users will be unable to run or install these services.

Caching Software	**Local Caching**	**Site Caching**	**Dataset Pinning**	**Administrator Required**
GPFS	Yes	No	No	Yes
HTTP Caching	No	Yes	No	Yes
Parrot	Yes	Yes	No	No
CernVM-FS	Yes	Yes	No	Yes
XrootD	No	Yes	No	Yes

Table 4.1: Table of Capabilities for Grid Data Caching Software

Table 4.1 compares the caching technologies discussed here. We define local caching as saving the input files on the worker node and making them available to local jobs. Local caching is different from site caching, which is done in the OSG by Squid caches. We define site caching as when data files are stored and available to jobs from a closer source than the original. In most cases on the OSG, the site cache is a node inside the cluster that has both low latency and high bandwidth connections to all of the execution hosts. We define dataset pinning as saving the dataset for a configurable amount of time, despite any other cache eviction policies.

We use distributed transfer to mean transfers that are not from a single source. In this dissertation, we will be using BitTorrent [21], in which a client may download parts of files from multiple sources. Additionally, the client may make available to other clients parts of the files that have already been downloaded.

BitTorrent is a transfer protocol that is designed for peer-to-peer transfers of data over a network. It is optimized to share large datasets between peers. The authors of [80] and [81] discuss scheduling tasks efficiently in peer-to-peer grids and desktop grids. Their discussion does not take into account the network bottlenecks that are prevalent in campus cluster computing, such as Network Address Translation (NAT) configurations which tunnel all outgoing traffic through a few nodes.

In [20], the authors use scheduling, caching, and BitTorrent in order to optimize the response time for a set of tasks on a peer-to-peer environment. They build the BitTorrent and caching mechanisms into the middleware which is installed and constantly running on all of the peers. They do not consider the scenario of opportunistic and limited access to resources. Their cluster size is statically set, and therefore, may not see the variances that users of campus clusters may see.

4.3 Implementation

The HTCondor CacheD is a daemon that runs on both the execution host and the submitter. For my purposes, a cache is defined as an immutable set of files that has metadata associated with it. The metadata can include a cache expiration time, as well as ownership and acceptable transfer methods.

The CacheD is designed to interface with other daemons as well as the user. The CacheD communicates only over BSD sockets. The transport application protocol is over HTCondor's secure communication library, CEDAR [49], and heavily uses ClassAds [59].

4.3.1 Architecture

The CacheD follows the HTCondor design paradigm of a system of independent agents cooperating. Each CacheD makes decisions independently of each other. Coordina-

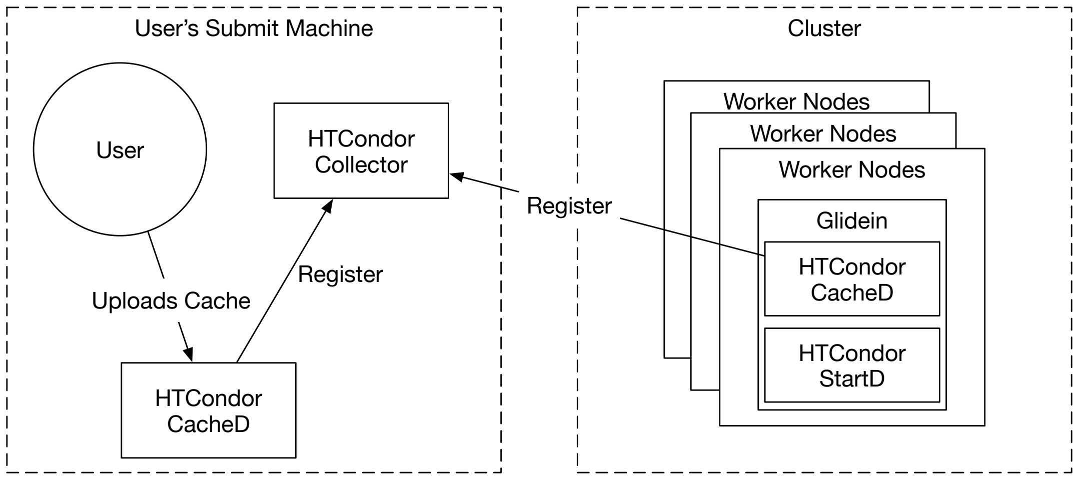


Figure 4.3: Daemon Locations

tion is done by CacheDs communicating and negotiating with each other. Figure 4.3 shows the location of daemons both on the submission host and the worker nodes. The CacheD on the user's submit machine acts as the cache originator, discussed below. The CacheDs on the worker nodes download the cache when requested. Each caching daemon registers with the HTCondor Collector. The collector serves as a catalog of available cache daemons that can be used for replication.

The policy expression language uses the matchmaking language in the HTCondor system [59]. The caching daemon is matching the cache contents to a set of resources; therefore, it is natural to use HTCondor's same matchmaking language that is used to match jobs to resources. Once a resource is determined to match the cache's policy expression, the caching daemon will contact the resource's caching daemon in order to initiate a cache replication. The caching daemon on the remote resource is an independent agent that has the ability to deny a caching replication even after matchmaking is successful. A full discussion of the policy language, as well as possible configurations, is discussed in Chapter 5.

Libtorrent [51] is built into the CacheD to provide native BitTorrent functionality.

The CacheD is capable of creating torrents from sets of files in a cache, as well as downloading cache files using the BitTorrent protocol. Since this is a distributed set of caches, we will not use a static torrent tracker. Rather, we will use a Distributed Hash Table [26] and local peer discovery [46] features of the BitTorrent protocol.

The CacheD supports multiple different forms of transferring data. Using HTCondor's file transfer plugin interface, it can support pluggable file transfers. Only the BitTorrent and Direct transfer methods will be covered here. The BitTorrent method uses the libtorrent library to manage BitTorrent transfers and torrent creation. The Direct method uses an encrypted and authenticated stream to transfer data from the source to the client.

An important concept of the caching framework is a cache originator. The original daemon that the user uploaded their input files to is the cache originator. The cache originator is in charge of distributing replication requests to potential nodes, as well as providing the cached files when requested.

In addition to the CacheD, a transfer plugin is used to perform the cache transfers in the job's sandbox. The plugin uses an API to communicate with the local CacheD to send local replication requests to the local host. After the cache is transferred locally, the plugin then downloads the cache to the job's working directory.

Expiration time is used for simple cache eviction. A user creates a cache with a specific expiration time. After a cache has expired, a caching server may delete it to free space for other caches. The expiration may be requested to be extended by the user.

The caching daemons interact with each other during replication requests. A cache originator sends replication requests to remote caching daemons that match the replication policy that is set by the user. The remote caching daemon then confirms that the cache data can be hosted on the server. The remote cache then initiates a file transfer in order to transfer the cached data from the origin to the remote CacheD.

The receiving CacheD can deny a replication request for many reasons, including:

- The resource does not have the space to accommodate the cache.
- The resource may not have the necessary bandwidth available in order to expediently transfer the cache files.
- The resource does not expect to be able to run the user's jobs and thus the cached files will not be used.

The ability of the receiving CacheD to deny a replication request follows HTCondor's independent agent model. Discussion of these policies is covered in Chapter 5.

4.3.2 Creation and Uploading Caches

The user begins using the caching system by uploading a cache to their own CacheD, which then becomes the cache originator. This is very similar to a user submitting a job to their own HTCondor SchedD. Using the cache's metadata, the CacheD decides whether to accept or reject the cache. If the CacheD accepts the cache, it stores the metadata into resilient storage. The user then proceeds to upload the cache files to the CacheD. Every cache is immutable; no partial caches are allowed.

The CacheD stores the cache files into its own storage area on the origin server. Once uploaded, the CacheD takes action to prepare the cache to be downloaded by clients. This includes creating a BitTorrent torrent file for the cached files.

Numerous protections are used in order to ensure proper usage of the CacheD. The upload size is enforced to the size advertised in the metadata. The client cannot upload more data to the CacheD than was originally agreed upon during cache creation. Further, the ownership of the cache is stored in the metadata, and is acquired by authenticating with the client upon cache creation. Only the owner may upload and download files from the cache directly.

When uploading the cache, a two-phase commit is implemented. First, the client asks the CacheD if it is allowed to send files that make up a cache. The CacheD can deny or accept the files based on attributes such as the size of the requested cache. If the CacheD accepts the cache, the cache is marked as `UPLOADING` state, then the client can send the files. Upon completion of the transfer, the cache is marked as `COMMITTED` state, and cannot be further modified.

A client may mark a cache as only allowing certain replication methods. This can be useful if a user wishes to keep data private. BitTorrent doesn't offer an authorization framework to ensure privacy of caches. Users may mark the cache as only allowing Direct replications, which are encrypted and authenticated. These properties are summarized in Table 4.2.

Capability	**Direct**	**BitTorrent**
Encrypted	Yes	No
Authenticated	Yes	No
Peer-to-Peer	No	Yes

Table 4.2: Table of Capabilities for Direct and BitTorrent Transfer Methods

4.3.3 Downloading Caches

When a job starts, the CacheD begins to download the cache file using a file transfer plugin. The cache is identified by a unique string that includes the cache's name and the cache's originator host. The flow of replication requests is illustrated in Figure

4.4. The replication requests originate from the file transfer plugin, which sends the replication request to the node local CacheD. The node local CacheD then sends the replication to its parent or the origin cache. The propagation of replication requests are modeled after well-known caching mechanisms such as DNS.

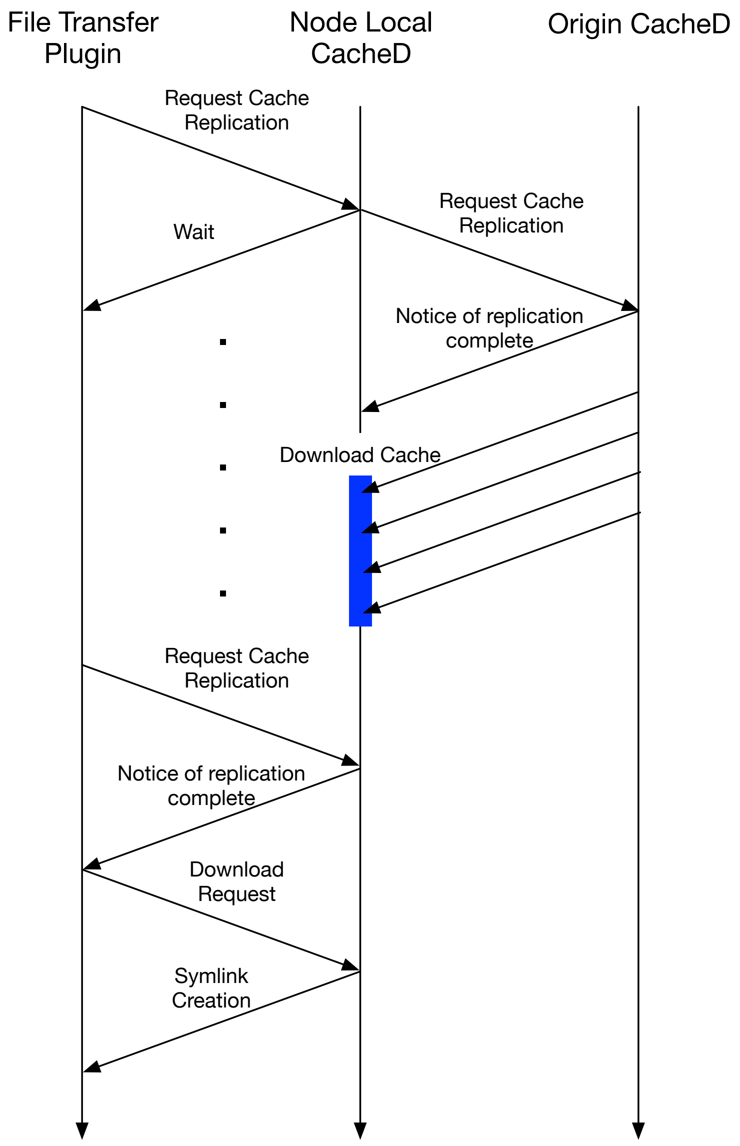

File Transfer Plugin
Node Local CacheD
Origin CacheD
Request Cache Replication
Wait
Request Cache Replication
Notice of replication complete
Download Cache
Request Cache Replication
Notice of replication complete
Download Request
Symlink Creation

Figure 4.4: Flow of Replication Requests

Figure 4.4's flow can be shown in the following steps:

1. The plugin contacts the node local CacheD daemon on the worker node. It requests that the cache is replicated locally in order to perform a local transfer.

2. The node local CacheD responds to the file transfer plugin with a “wait” signal. The file transfer plugin polls the node local CacheD periodically to check on the replication request.

3. The local CacheD daemon propagates the cache replication request to its parent, if it exists. If the CacheD does not have a parent, it contacts the cache originator in order to initiate a cache replication.

4. If the cache is detected to be transferable with BitTorrent, the download begins immediately after receiving the cache’s metadata from the parent or origin.

5. Once the cache is replicated locally, the plugin downloads the files from the local CacheD.

All communication between CacheDs are authenticated using the regular HTCondor methods. ClassAds are used for communication between the CacheDs so that the protocol can be expanded if needed.

Each download is negotiated for the appropriate transfer method between the source, client, and the cache. Each entity has its own preferences on the method of transfer. Further discussion of this negotiation is discussed in Chapter 5. By default, the CacheD is capable of two transfer methods between CacheDs: the BitTorrent and the Direct transfer methods.

If the transfer plugin successfully authenticates with a local CacheD, transfer methods are negotiated. If supported, another transfer method is possible: the symbolic link (symlink) method. The symlink method is preferred to directly downloading the cache for two reasons:

1. Downloading the cache will create yet another copy of it, filling disk space on the local node.

2. A symlink can create a nearly instantaneous transfer of the data from the cache directory to the execution directory.

A symlink does not actually copy the data. Instead, it creates a pointer to the data which is in another directory. This symlink creates the possibility that the cache may be altered by the job, but this issue is largely ignored for now. BitTorrent will not allow a modified cache to be replicated; therefore, there is no chance that the altered cache will propagate to other nodes in the system. BitTorrent provides this guarantee by checksumming all files in the cache before and after transfers. If the checksum does not match, the file is re-downloaded.

A symlink is created by the CacheD in the job's working directory pointing to the cache directory. This symlink method eliminates transferring the cache to each job.

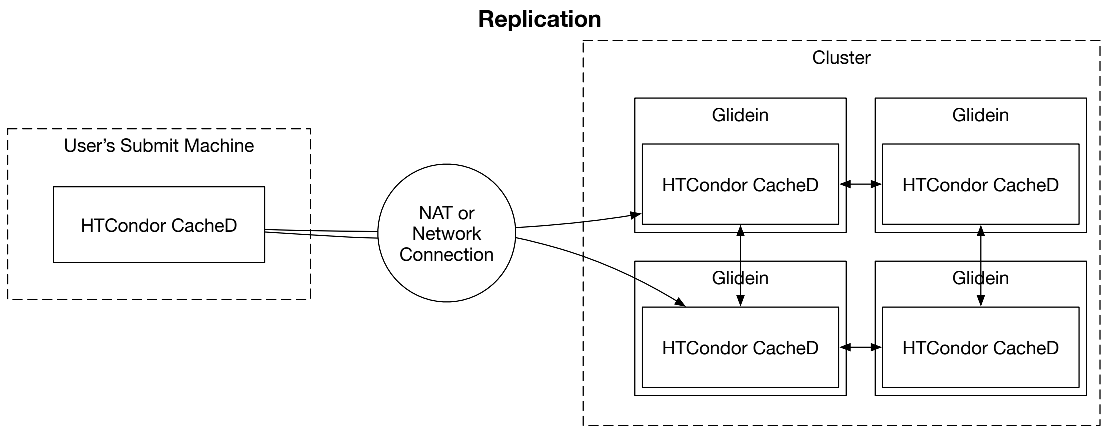


Figure 4.5: Cache Replication Showing Bottleneck

In Figure 4.5, you can see a traditional configuration of a cluster. The configuration shows that there is a Network Address Translation bottleneck or a network bottleneck between the submit machine and the execution nodes. The bottleneck limits the bandwidth between the submit machine and the execution nodes. For data distribution, total bandwidth is much improved when peer CacheD instances are utilized.

This is just one of many possible network topologies. Some clusters have all of their worker nodes on the public internet. Some clusters do not allow any internet traffic to reach the worker nodes. On the OSG, the network topology shown in Figure 4.5 is the most common. It it also the configuration of Nebraska's clusters.

4.3.4 Proactive Replication

The CacheD is capable of performing proactive replication. In the same way that Kangaroo performed proactive replication for output files back to the final destination, the CacheD can perform proactive replication of caches to available CacheDs. The replication is completed in the background, while jobs are being executed on the resource. By interleaving the computing and replication on the same node, we more fully utilize the node's resources. Further, if a subsequent job requires the cache, it will already be available locally.

Proactive replication also serves the purpose of creating more peers for the BitTorrent transfers. More peers in a BitTorrent transfer can increase the speed of transfers.

4.3.5 Reporting of Replicas

Once the cache has replicated to a CacheD, it will periodically report the replica to the origin CacheD. The replica locations are stored in a hash table keyed by the cache name and hostname. Finally, the value is the time of the last report. The updates are transferred in with ClassAds from the replica CacheDs.

The design of the data structure is to make lookups of cache locations very fast. In the future, we hope to use this data structure to assist in scheduling jobs where the cache is located. In Chapter 5, we discuss possible uses of this data structure when determining where the replica should be placed.

4.3.6 Parenting of CacheDs

The use of BitTorrent increases the IO activity on the host server significantly. If multiple CacheD instances are writing to disk concurrently, it degrades the IO performance for all jobs on the server. This increased IO activity leads to competition between BitTorrent-enabled CacheDs on the same host. In order to address the increased IO activity, each CacheD will designate a single daemon on the host that downloads the files through BitTorrent. All other CacheDs will then download the cache from the parent using Direct file transfer mechanisms. Figure 4.6 shows the parenting inside a node.

A CacheD will discover a node local parent by querying the HTCondor Collector that holds a catalog of all CacheDs known to the system. Figure 4.3 illustrates the layout of the HTCondor Collector with respect to the worker nodes. If it discovers a CacheD on the same node as itself, the CacheD started first will be chosen. By choosing a parent that is older, it improves the chance that the parent CacheD may already have the cache downloaded. If multiple CacheDs have the same starting time, they are alphabetically ordered by their unique name, and the first CacheD alphabetically is chosen as parent.

In addition to preserving local IO performance, parenting can also be used to create a hierarchy of caching. This is especially useful for direct transfer mechanisms. Figure 4.7 shows an example of caching parenting on different clusters.

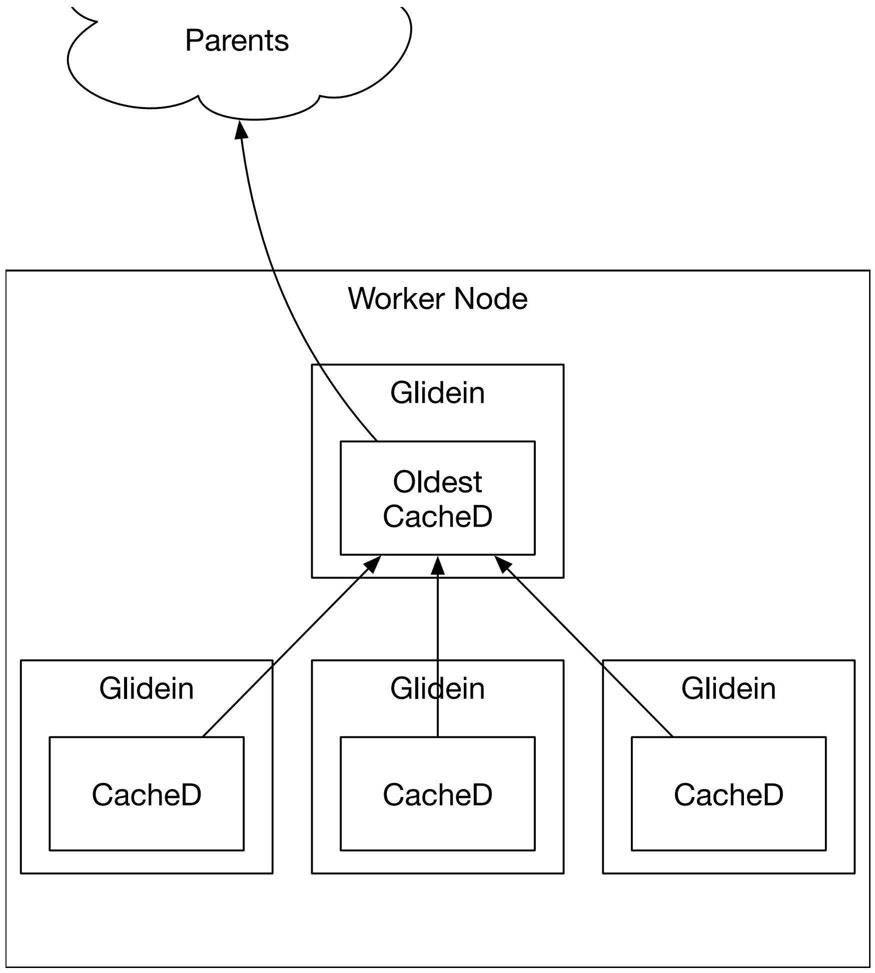


Figure 4.6: Illustration of Cache Parenting Inside a Node

A CacheD's parent can also be discovered through the configuration. When a CacheD starts, it checks the configuration for a special configuration attribute, the `CACHED_PARENT`. It then attempts to connect to the parent to verify that the parent is functional. If the parent is found to be functional and responding to queries, the child then forwards all requests it receives to the parent, just as the node local CacheD will forward all requests to the origin in Figure 4.4.

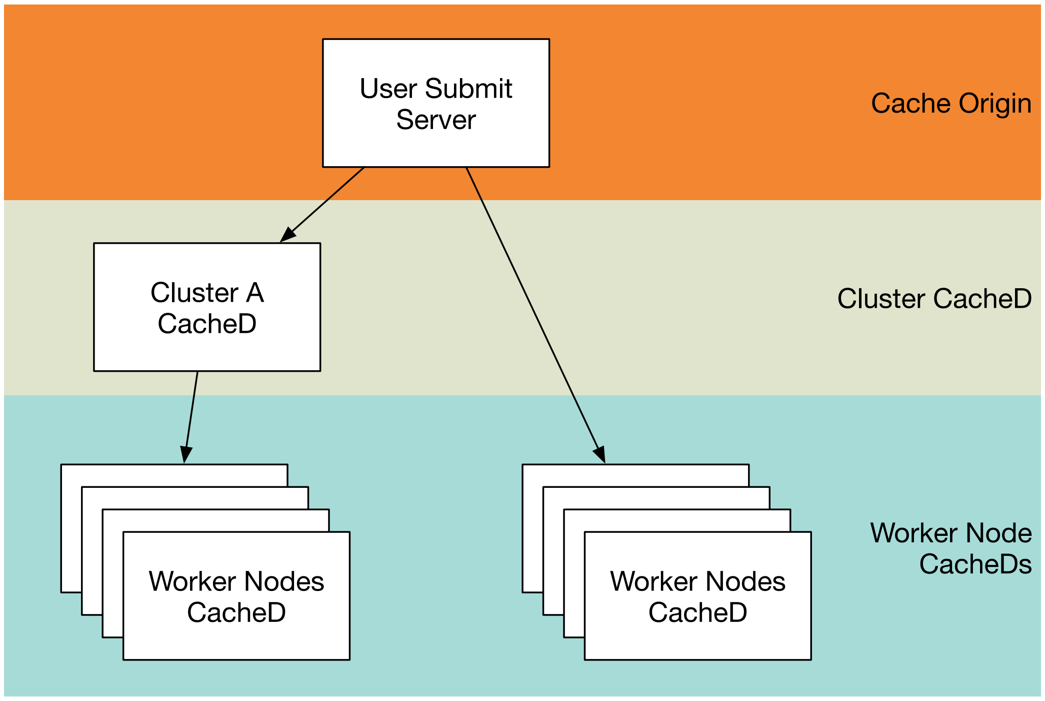


Figure 4.7: Illustration of Cache Parenting

4.3.7 Interface and API

To simplify communication with the CacheD, clients may use an API that will communicate with the CacheD. Further, Python bindings to the CacheD C++ API were written to simplify interacting with the daemon. Figure 4.8 shows the example code necessary to create a cache.

Creating a simple API to interact with the CacheD improves the user experience of using the CacheD. The CacheD also uses this same API in order to communicate with other CacheDs. For example, the call to replicate a cache is shown in Figure 4.9. All CacheDs and clients use this same function.

The most important function, `requestLocalCache`, asks the CacheD you are communicating with to replicate the cache to itself. It may respond with several options:

`OK`: The cache will be replicated. The current status of the replication is in the

```
#!/bin/env python

import htcondor
import glob
import time
import sys

cached = htcondor.Cached()
cacheName = sys.argv[1]

try:
    cached.createCacheDir(cacheName,
       int(time.time())+1000)
except RuntimeError:
    print "Create cache failed"
    sys.exit(1)

input_glob = glob.glob(sys.argv[2])
print input_glob
try:
    cached.uploadFiles(cacheName, input_glob)
except:
    print "Upload files Fail"
    sys.exit(1)
```

Figure 4.8: Example Code to Create a Cache

```
int requestLocalCache(const std::string &cached_server,
   const std::string &cached_name,
   compat_classad::ClassAd& response, CondorError& err)
```

Figure 4.9: Function to Request a Replica of the Cache

`CacheState` attribute of the returned ClassAd `Reponse`.

`WAIT`: If the CacheD has not decided whether or not to accept the cache, it can send a `WAIT` signal to the requester to ask again after some delay. The client may determine the appropriate delay.

`REJECTED`: If the CacheD has decided, through the policy language described in Chapter 5, to not replicate the cache, the CacheD will respond with `REJECTED`

A CacheD may call this function in order to replicate a cache it stores to other CacheDs. A transfer plugin may call this function in order to replicate the cache locally before downloading the cache.

Users may also use this API in order to modify the replication policies of the job, to extend the lease time of a cache, or to download the cache. All interactions with the CacheD are available through the API.

4.4 Results For Campus Cluster

4.4.1 Experimental Design

To evaluate the solution, we will run a BLAST benchmark from UC Davis [24]. We chose a BLAST benchmark due to many factors. BLAST is used frequently on campuses, but used infrequently on clusters due to the size of the database. BLAST has very large databases that are required by each job. This makes it difficult to use on distributed resources since each job requires significant data. BLAST databases are frequently updated, making them poor candidates for static caching, but good candidates for short-term caching, for which the CacheD specializes.

The BLAST database distributed with the benchmark is a 3 GB subset of the Nucleotide NR database. In the tests, we will use a larger subset (15 GB) of the NR database in order to demonstrate the efficiency of the solution.

For researchers, the time to results is likely the most important metric. The stage-in time of data can be a large component of the entire workflow time. We will measure the time for stage-ins as well as the average stage-in time.

We designed two experiments that represent our experience on campus infrastructure. In the first experiment, we will allow 100 simultaneous jobs to start at the same time and measure the average download time versus the number of distinct nodes. This experiment also includes the download time for child caches. We chose 100 jobs to completely fill all of the nodes we were allocated on the cluster.

In the second experiment, we compared the total stage-in time for a variable number of jobs while the number of distinct nodes remains constant at 50. This will show that the cache is working to eliminate transfer times when the files are already on the node. Further, it will compare HTCondor's File Transfer method versus the CacheD's two transfer methods: BitTorrent and Direct.

When the number of jobs is fewer than 50, each job must download the cache since there are 50 nodes available for execution. When the number of jobs is more than 50, all jobs that run after the initial download use a cached version of the data.

In the experiments, each job uses the CacheD to stage-in data to the worker nodes. The jobs are submitted with glideins created by Bosco [82] and the Campus Factory [84], as covered in Chapter 3. Bosco allows for remote submission to campus resources while the Campus Factory allows for on-demand glidein overlay of remote resources. The Campus Factory was used in order to create and run glideins which, in turn, run the CacheD daemon. Bosco was used in order to submit to multiple campus resources simultaneously. Figure 4.10 shows the experimental setup. The submit node is external to the cluster Crane.

These two experiments were conducted on a production cluster at the Holland Computing Center at the University of Nebraska–Lincoln (UNL).

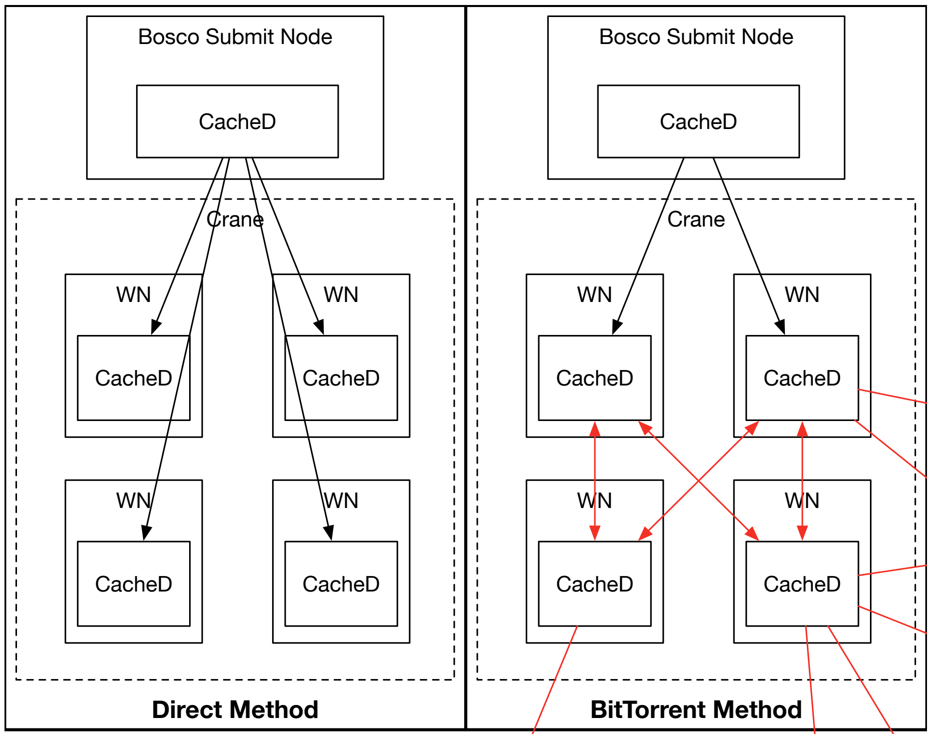


Figure 4.10: Illustration of Direct and BitTorrent Transfer Methods Experimental Setup

4.4.2 Results

We completed 41 runs of the BitTorrent versus Direct transfer experiments on the UNL production cluster. The Direct transfer method results in a linear increase in the average stage-in time to transfer the cache as we increased the number of distinct nodes. Conversely, we found that the BitTorrent transfer method did not markedly increase the average stage-in time as we increased the number of distinct nodes. The BitTorrent transfer method was faster than the Direct in all experiments.

Figure 4.11 shows that the BitTorrent transfer method is superior to Direct for all experiments that were run. Since multiple CacheDs on the same node will parent to a single CacheD, the number of distinct nodes is the controlled variable. After the parent cache downloads the cache for the node, then each child cache will download

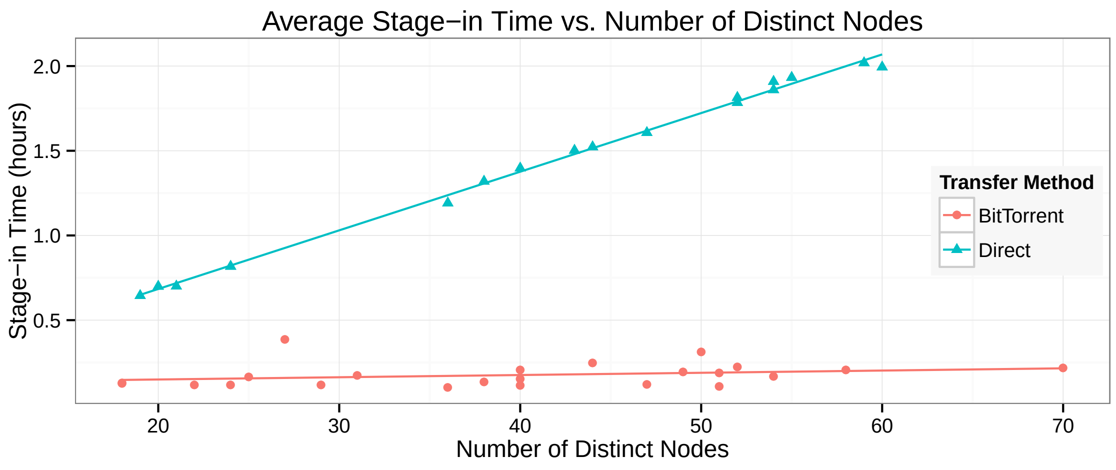


Figure 4.11: Comparison of Direct and BitTorrent Transfer Methods with Increasing Distinct Node Counts

from the parent using the Direct transfer method.

The Direct method of transfer follows a linearly increasing time to download the cache files. This can be explained by bottlenecks of the transfers between the host machine and the execution nodes as shown in Figure 4.10. The increase in number of distinct nodes increases the stage-in time for any individual node.

The average download times for BitTorrent stage-ins are also shown in Figure 4.11. The stage-in time does not significantly increase as the distinct nodes increases. This trend continues as the number of distinct nodes increases since BitTorrent can use peers to speed up download time.

To better illustrate how parenting affects the download time of a cache, we show a histogram of the different download types in Figure 4.12. The figure shows that while the parents download first, and nearly at all the same time, the children take a variable amount of time to download. This variability can be attributed to the number of children on a node. The more children downloading the cache simultaneously, the slower each download will take.

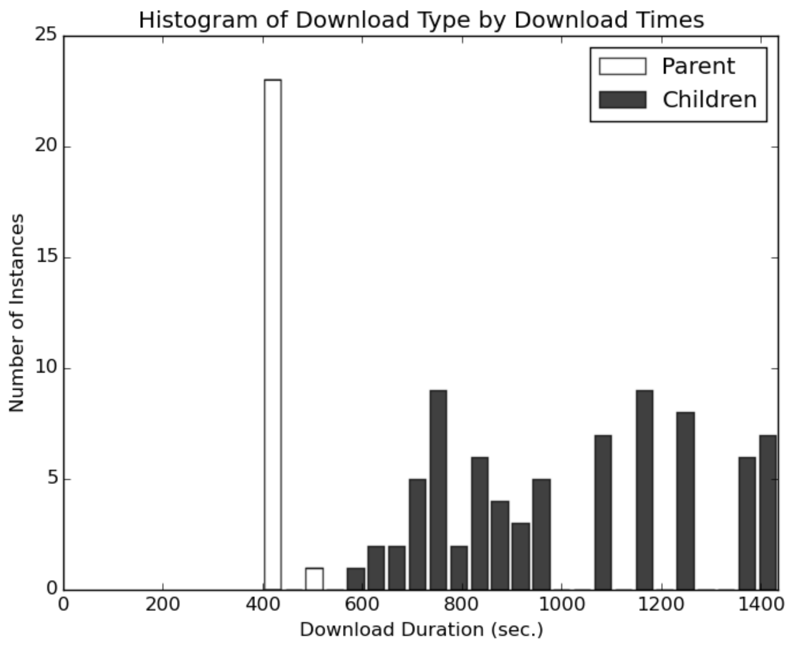


Figure 4.12: Histogram of Transfer Modes vs. Download Times for a BitTorrent Workflow

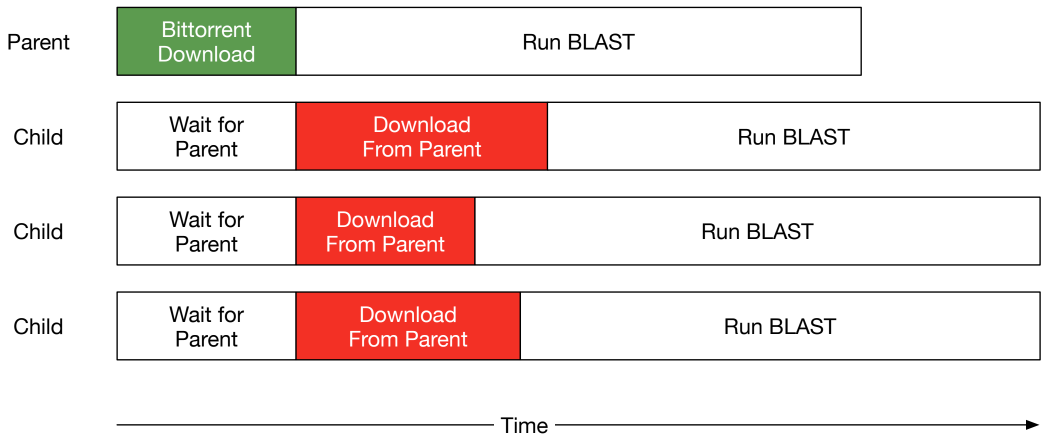


Figure 4.13: Timeline of Blast Runs

The observed behavior of the CacheD timeline is shown in Figure 4.13. If there are several children on a node, then the CacheD will wait for the parent to download the cache, then each child will download from the parent. The parent will begin the

BLAST job immediately after downloading the cache.

Disk contention was observed on the nodes while the children were downloading the cache and the parent was running BLAST. This disk contention warrants further investigation. Multiple copies of the cache will reside on the same node, but this is necessary since the CacheD is running on opportunistic resources. At any time, a parent or child may be preempted, and their copy of the cache will be removed. An independent copy of a cache for each CacheD will guarantee that the cache will survive as long as the CacheD, and the cache will be available for subsequent executions.

For the second experiment, we calculated the total stage-in time for a variable number of jobs. When we limit the number of nodes to 50, one can clearly see the effect of the caching by varying the number of jobs.

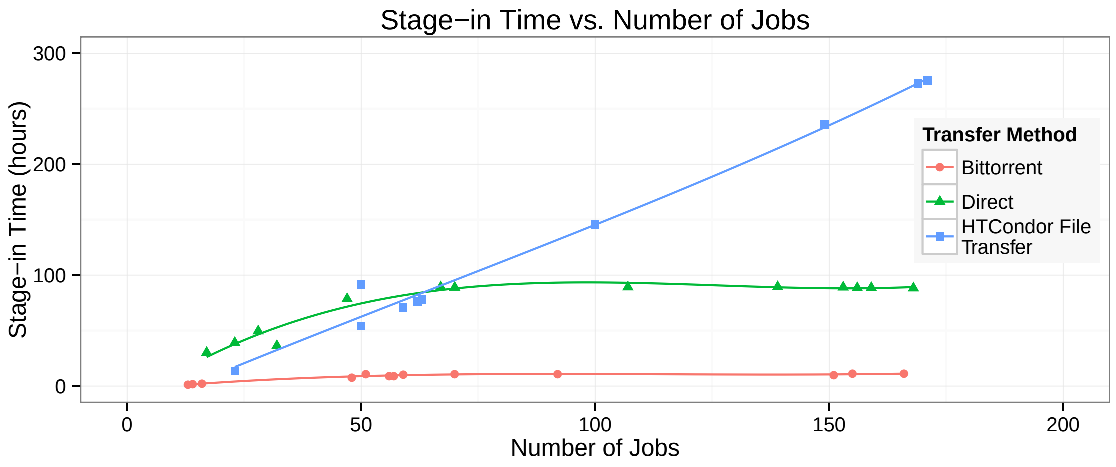


Figure 4.14: Transfer Method vs Number of Jobs

In Figure 4.14, both the Direct and BitTorrent transfer methods have a natural bend at about 50 jobs. This correlates to when the CacheD has on-disk caches of the datasets, and the transfer to the job's sandbox is nearly instantaneous.

The HTCondor file transfer method has a shorter stage-in time for low numbers of distinct nodes than the Direct method. This can be explained by the increased overhead that the CacheD introduces when transferring datasets. After all 50 nodes have the dataset cached locally, the Direct transfer method becomes more efficient than the HTCondor file transfers.

4.4.2.1 BitTorrent Behavior

The topology of BitTorrent downloads are harder to visualize, as the transfer topology is more complex. We captured each block (the smallest unit of transfer in the BitTorrent protocol) from the source to the destination. We then graphed the resulting transfer links. For this experiment, we only submitted five jobs in order to limit the size of the graph, and to improve readability. We used the same 15 GB NR database as before.

Figure 4.15 shows the transfers between the nodes. The libtorrent library that is used by the CacheD to implement the BitTorrent protocol allows for alerts to be propagated each time a block is transfered. The CacheD periodically polls the library for alerts and prints any block movement activity. Note, not all blocks transferred between the nodes are captured, since the alert buffer may overflow and further alerts will be lost until the alert buffer is cleared by the periodic check.

Scanning software was written to scan the debug output from the CacheD for the block movement data and plot it as in Figure 4.15. The nodes are identified by their unique BitTorrent assigned identifiers, except the origin, which is signified by *Cache Origin*. The *Cache Origin* is the original server with the cache before the BitTorrent transfers replicate it to other nodes.

From Figure 4.15, one can notice several interesting points. First, even though the cache must be transferred to all five nodes of the cluster, the *Cache Origin* is shown as only transferring the cache approximately one time, and nearly equally to two different nodes. Once the cache is in the cluster, the *Cache Origin* does not

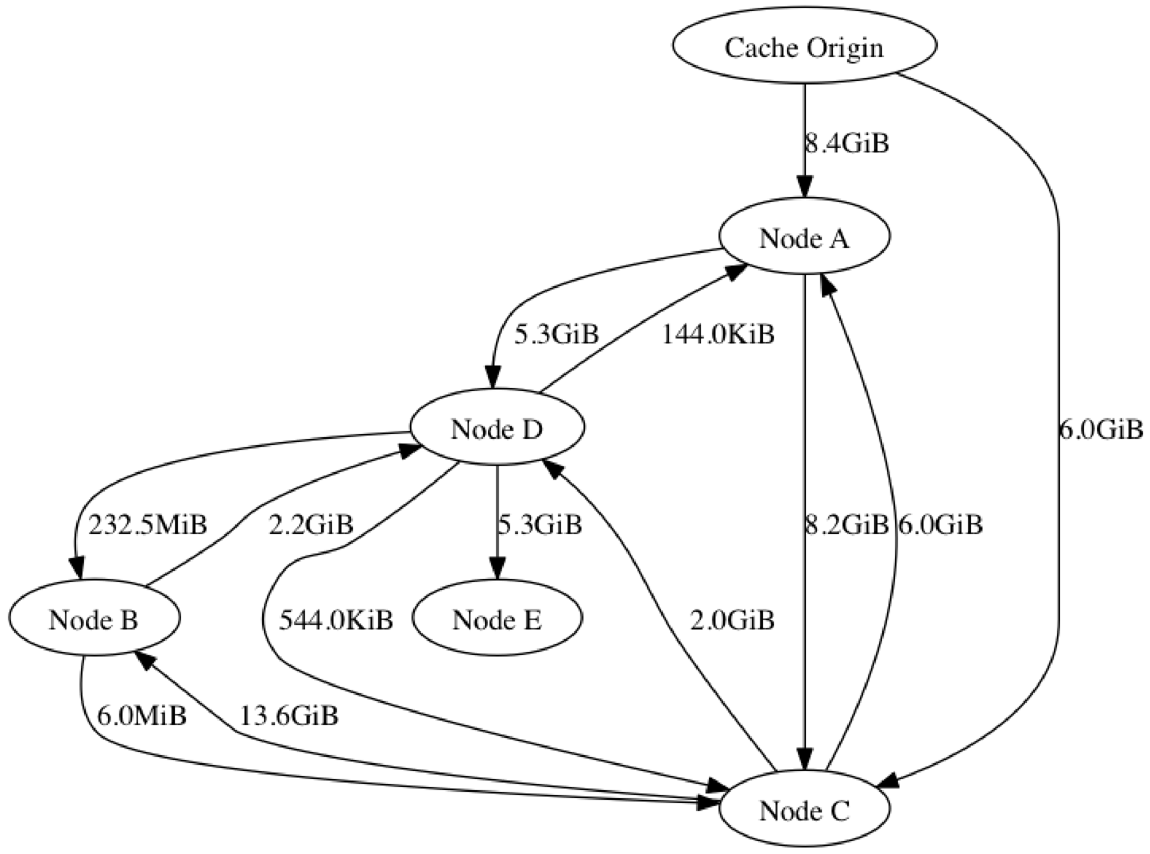


Figure 4.15: Graph of Transfer Nodes in the BitTorrent Transfer

transfer the data to any of the other nodes.

Once the cache is fully inside the cluster, the CacheDs transfer data only between each other inside the cluster. The libtorrent library detects “fast” nodes and preferentially transfers with them. Since nodes inside the cluster are near each other on the network and are on an HPC system with a high performance network interconnect, the library highly prefers transferring from only the cluster nodes. You can see that the two original nodes, nodes A and C, transfer the cache to the other nodes in the system, including each other.

4.5 Results for Open Science Grid

4.5.1 Experimental Design

To test on the Open Science Grid, Bosco was connected to the Holland Computing Center's (HCC) GlideinWMS frontend which will run the Bosco glideins on worker nodes on the OSG. The GlideinWMS system for HCC submits to about 20 sites out of 100 on the OSG.

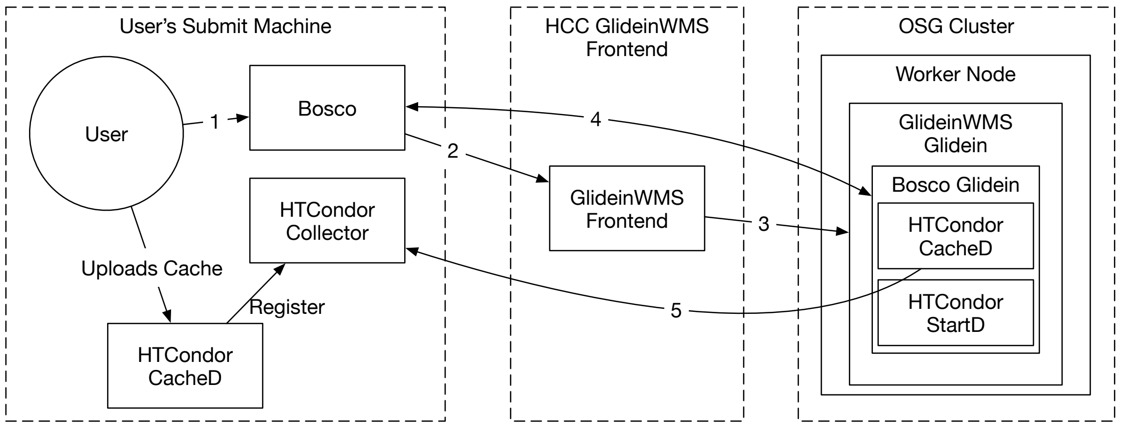


Figure 4.16: OSG Daemon Locations

Figure 4.16 shows the daemon layout when running on the OSG. The flow of the experimental BLAST jobs are:

1. The user submits jobs to Bosco using standard HTCondor commands.

2. Bosco detects idle jobs on the Bosco system, and deploys Bosco glideins to the connected HCC GlideinWMS Frontend in order to service the idle jobs. Bosco transfers the worker node binaries to the HCC GlideinWMS Frontend to be run as the jobs.

3. The GlideinWMS glidein starts on the remote OSG cluster. It communicates back with the GlideinWMS Frontend in order to retrieve the job, which is a Bosco glidein.

4. The Bosco glidein starts and reports back to the Bosco system on the user's submit machine. The submit machine then will send the BLAST job to the remote Bosco glidein running on the OSG cluster.

5. At the same time that the Bosco glidein reports back to the Bosco system, the CacheD is reporting to the HTCondor Collector on the user's submit machine. This will add the CacheD to the list of known CacheDs in the system.

For each experiment, we submitted 100 BLAST jobs at a time. The BitTorrent transfer method is compared to that of a common transfer method on the OSG, HTTP-based caching.

Unlike the single campus experiments, the HTTP requests have no method to parent to one another. Instead, the average time to completion of the transfer of the input data files are compared.

We again used the same data source node as in the campus experiments. It has a 1 Gbps connection to the Internet.

Since the CacheD is able to cache the input data on the node, and we have shown in the campus experiments that the CacheD is nearly instantaneous in transferring the cached data to the job directory, we will not compare subsequent total stagein times as we did in the campus experiments.

Transfer Type	**Average Time (minutes)**	**Average MB/s**
BitTorrent	17.77	13.66
HTTP	52.17	4.65

Table 4.3: Transfer Times for Different Transfer Type

4.5.2 Results

As you can see in Table 4.3, the Bittorrent method is significantly faster than the HTTP method.

While processing the results from the experimental runs, we noticed that the vast majority of CacheDs where running in parent mode. In the results, there are between 71 and 96 parents out of 100 jobs. Indeed, we experienced so many parents, that we were unable to complete any Direct transfer experiments. The transfers took too long to complete and the jobs were evicted from the remote OSG clusters before the CacheD could complete the transfer of the BLAST databases. Therefore, the Direct transfer method was not tested on the OSG.

We can also compare all of the transfer methods across both the campus and OSG experiments. This comparison is shown in a Violin Plot [40] in Figure 4.17. The violin plot shows a probability distribution of transfer speeds for the different transfer methods. We ran each experiment between 25 and 43 times (in the case of OSG BitTorrent runs).

Figure 4.17 shows that the BitTorrent methods are faster at transferring the experimental data than the Direct or HTTP methods. Even though the HTTP and Direct methods look similar in speed, they are comparing different aspects of the transfers. The Direct method has sometimes as few as half as many downloaders (parents) as the HTTP method, in which all of the downloaders are parents. Therefore, given the same number of downloaders, HTTP should be faster than Direct.

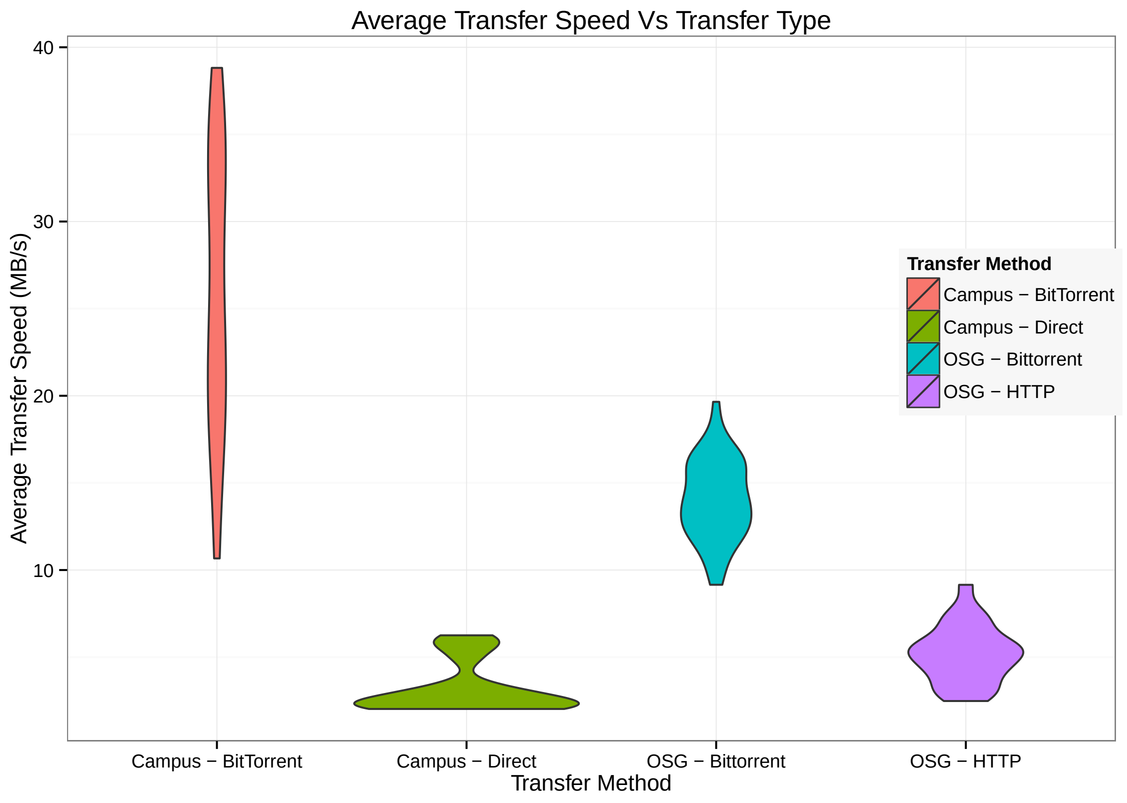


Figure 4.17: Violin Plot of the Transfer Speed Comparing Transfer Protocols

The BitTorrent method for the campus had a much wider variance of transfer speeds than any other transfer method. This can be explained by the large variance in the number of parents available to download, compared to the other transfer methods. The BitTorrent method for the OSG was almost always faster than the HTTP method on the OSG.

4.5.3 Aggregate Bandwidth

While running the CacheD experiments, the average aggregate bandwidth for transfers across all jobs was much higher than the source node's capable bandwidth.

The aggregate bandwidth shown in Figures 4.18 and 4.19 is calculated from the

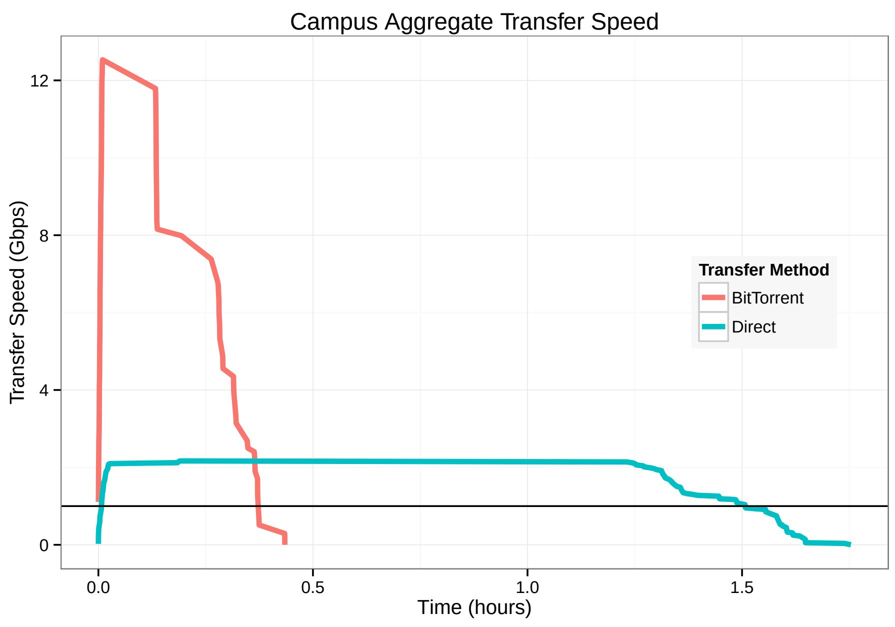


Figure 4.18: Campus Aggregate Bandwidth Showing the Source Node's 1 Gbps Connection

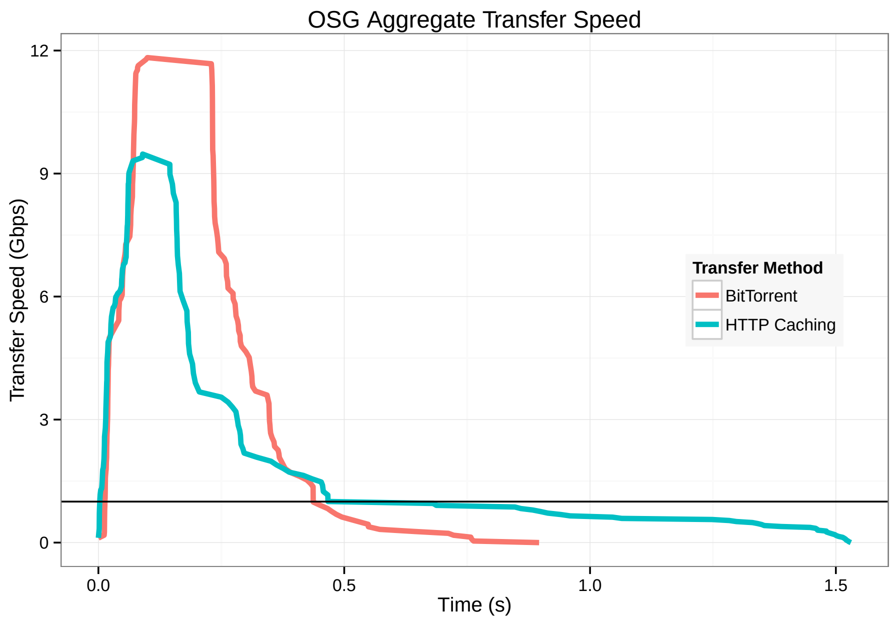


Figure 4.19: OSG Aggregate Bandwidth Showing the Source Node's 1 Gbps Connection

historical logs of the experimental runs. For each transfer, the average bandwidth is assumed to start at the transfer start time and end at the transfer end time. Then, each of these transfers is overlaid on top of each other. Each of the graphs in the Figure are a single experimental run, but are typical for the transfers. The origin server's available bandwidth is also shown on the graphs as a horizontal line at 1 Gbps.

From Figure 4.18, one can see the difference between the different transfer methods. The Campus Direct method has the lowest aggregate bandwidth of all of the transfer methods, while BitTorrent on the campus has the highest. The campus BitTorrent method provides 12 times the available bandwidth of the origin server. This can be attributed to servers in the cluster distributing data between themselves with the BitTorrent protocol without transferring it from the origin server. Using the Direct method, the CacheD must transfer the data from either the origin or a parent on the local node.

For the two OSG transfer methods, Figure 4.19, one can see that the HTTP method almost reaches the same aggregate bandwidth as the BitTorrent method, nearly 10Gbps. Both transfer methods have long tails after a large peak in transfer speed. But the BitTorrent method maintains the peak of transfer speed until roughly 1200 seconds into the transfer. The HTTP method's peak drops significantly before 800 seconds into the transfer. The HTTP also has significant transfers yet to be completed, hence not only the long tail, but high transfer speed of the tail. Upon further investigation, the HTTP performance differed significantly between sites. Table 4.4 shows the transfer speeds of sites in the same run.

Table 4.4 illustrates the disparity in transfer speeds between sites. University of California, San Diego (UCSD) is twice as fast downloading input HTTP data than the next highest, Nebraska. But MIT was 10 times as slow as UCSD. This can be

OSG Site	**Average Transfer Speed**
UCSD	236 Mbps
Nebraska	116.8 Mbps
AGLT2	104 Mbps
Wisconsin	91.2 Mbps
UChicago	90.4 Mbps
SPRACE	80.8 Mbps
MIT	26.4 Mbps

Table 4.4: HTTP Transfer Speeds by Site

attributed to the speed and configuration of the HTTP caching servers at the different sites. UCSD has a HTTP caching server with a 10 Gigabit connection and large, fast SAS disks. This enables very fast transfers from the caching server to the worker nodes.

Table 4.5 shows the BitTorrent transfer speeds from the same experiment shown in Figure 4.18. There are more sites involved downloading the cache than the HTTP transfer method. But you can see that the transfers are usually faster than the HTTP method. In addition, there are more sites that are faster than the HTTP method.

It is not unusual to run on very different sites in separate experiments on the OSG. The BitTorrent and the HTTP experiments were run around 20 days apart.

When comparing the campus and OSG transfers, observe the long tails as the transfer speeds slowly decline. This can happen for many reasons. For example, for the OSG HTTP method, it occured because a single cluster had a very slow transfer speed, which slowed the download for all jobs running at that cluster.

The long tail of the OSG BitTorrent method are very slow transfers. After the initial very fast BitTorrent transfers, then children CacheDs begin their download

OSG Site	**Average Transfer Speed**	**Percent Change**
UCSD	169.6 Mbps	-28.14 %
NU Crane	134.4 Mbps	NA
Northwestern	133.6 Mbps	NA
Nebraska	129.6 Mbps	+10.96 %
Purdue	124 Mbps	NA
MIT	113.6 Mbps	+330.3 %
Michigan	104.8 Mbps	+0.77 %
Wisconsin	94.4 Mbps	+3.5 %
UChicago	88 Mbps	-2.69 %
Brookhaven	67.2 Mbps	NA
Connecticut	54.4 Mbps	NA

Table 4.5: BitTorrent Transfer Speeds by Site

of the cache. These downloads of the cache from the parent are slower than the BitTorrent downloads because they are coming from a single source, and multiple children may be downloading at the same time. To further look at the OSG BitTorrent method, we graphed the download time by the mode (either Parent or Child) in Figure 4.20.

Figure 4.20 shows the download method time by mode. As you can see in the graph, the vast majority of downloads are from parents. But some children are also interleaved with the parent downloads. The final download is a singular child download. This child download may have gone slowly because of inadequate disk bandwidth on the local node. This inadequate disk bandwidth could be exacerbated by the parent CacheD running BLAST while the child is still downloading the cache.

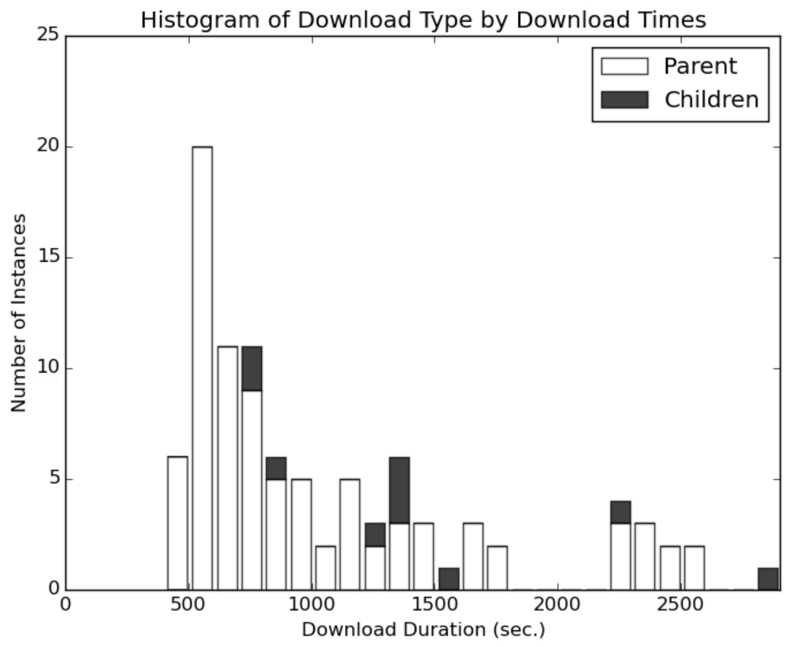


Figure 4.20: Download Time by Mode for a Single OSG BitTorrent

4.6 Conclusions

We have presented the HTCondor CacheD, a technique to decrease the stage-in time for large shared input datasets. The experiments proved that the CacheD decreases stage-in time for these datasets. Additionally, the transfer method that the CacheD used can significantly affect the stage-in time of the jobs.

The BitTorrent transfer method proved to be an efficient method to transfer caches from the originator to the execution hosts. In fact, the transfer time for jobs did not increase as the number of distinct nodes requesting the data increased. Any bottlenecks that surround the cluster are therefore irrelevant using the BitTorrent transfer method.

We investigated OSG transfers for both HTTP and BitTorrent. We found that not all sites have equivalent HTTP caching setups. For example, UCSD was three times faster than the second fastest site when using HTTP caching for transfers.

In addition, we found that the CacheD using the BitTorrent transfer method outperformed the popular HTTP-based transfer method on the OSG. Further investigation of slow transfers must be completed in order to further optimize the BitTorrent transfers on the OSG. A possible solution could be to give up on the transfer after some timeout or if the transfer speed is too slow, although timeout and transfer speed thresholds would be difficult to set accurately.

Chapter 5

Campus Storage Policy Language

5.1 Introduction

In the previous chapter, the HTCondor CacheD was introduced and benchmarked. In this section, we will discuss the policy framework that allows the CacheD to represent heterogeneous resources on campus or cyberinfrastructure resources. The CacheD's policy framework is used whenever it interacts with another CacheD. This policy framework allows the CacheD to act as an independent agent within the distributed system.

In a distributed computing system, independent agents are designed to act on behalf of entities such as users, hosts, or entire clusters. The agents attempt to fulfill the goals of the entities that they represent, even in the chaotic environment characteristic of a distributed computing system. In order to fulfill these goals, the agents must know and understand them. Therefore, a policy language exists to express the goals of the entity.

The policy language must be flexible enough in order to express and follow the goals and instructions of the users. The goals of the policy language are:

1. To express attributes of the entities such as the cache, CacheDs, and the host.
2. To write policies, taking into account the attributes of the entities.
3. To be easy to read and write expressions.
4. To allow users to define their semantics and attributes in a schema-free manner.

We implement this policy language in the CacheD using HTCondor ClassAds [59]. ClassAds are an independent library developed by the HTCondor project and are used for communication between HTCondor components. They provide all of the attributes described above.

1. The Key-Value structure of ClassAds allow for attributes of the entities to be expressed in strings, values, lists of string, or expressions.
2. Expressions can reference attributes in the current and the matching ClassAds.
3. Expressions are written as expressions with semantics familiar to most programmers.
4. ClassAds are scheme-free and me be extended.

The CacheD has a few interaction points when it must interact with other agents, such as those representing other CacheDs or users. Those interaction points are:

Choosing a Replication Target - A CacheD that is the origin to a cache may choose to proactively replicate to other CacheDs. Choosing a replication target requires matching the cache's requirements with that of the target CacheD's.

Accepting Cache Replication - A CacheD must decide if it can accept a cache when it receives a replication request. This decision is based on its own policy, as well as attributes of the incoming cache.

```
CachedServer = true
Machine = "red-foreman.unl.edu"
LastHeardFrom = 1433790880
UpdatesTotal = 8660
Name = "cached-22815@red-foreman.unl.edu"
CondorPlatform = "$CondorPlatform:
   X86_64-ScientificLinux_6.5 $"
UpdatesHistory = "0x00000000000000000000000000000000"
UpdatesLost = 0
TotalDisk = 6769920
UpdateSequenceNumber = 32307
UpdatesSequenced = 8659
MyAddress = "<129.93.239.170:11000?noUDP&sock=22815_fb39>"
AuthenticatedIdentity = "dweitzel@unl.edu"
DetectedMemory = 7807
Requirements = MY.TotalDisk > TARGET.DiskUsage
CondorVersion = "$CondorVersion: 8.3.1 Dec 22 2014
   BuildID: UW_development PRE-RELEASE-UWCS $"
DetectedCpus = 2
DaemonStartTime = 1431839398
CurrentTime = time()
MyCurrentTime = 1433790880
```

Figure 5.1: CacheD ClassAd Example

Transfer Method - The transfer method for a cache to be replicated is chosen after a cache has been accepted. This is a prioritized list of acceptable transfer methods for the cache.

5.2 Policy Language

The policy language used by the CacheD is the HTCondor ClassAds [59]. ClassAds offer the flexibility to describe resources with attributes. Figure 5.1 shows an example of a CacheD's ClassAd. The attributes describe the CacheD daemon and the host it runs on. For example, the `DaemonStartTime` is a representation of when the daemon started. `TotalDisk` describes how much disk is available on the host where the CacheD is running.

Matching of ClassAds is done by comparing attributes between two sets of ClassAds. The attribute `Requirements` takes a special meaning when matching two ClassAds. The `Requirements` attribute is a boolean expression that is evaluated in the context of both the current ClassAd and the matching ClassAd. In the example in Figure 5.1, the CacheD's ClassAd would only match another ClassAd if the expression is `MY.TotalDisk > TARGET.DiskUsage`. This means that the CacheD will only accept caches that are smaller in size than the available disk on the host. `MY` and `TARGET` refer to the current ClassAd and the matching ClassAd, respectively.

The `Requirements` attribute in Figure 5.1 references other attributes in both the current ClassAd and the matching ClassAd. Attributes can reference other attributes in order to form strings, lists, or boolean expressions. In this example, the `Requirements` attribute references other attributes in order to create a boolean expression.

An example of using the requirement was shown in the Background in Figure 2.2 on page 13.

5.2.1 Extending CacheD Attributes

The ClassAd describing the CacheD can be extended by using the CacheD Cron mechanism. The CacheD Cron executes an external program in order to collect statistics and report the results in the CacheD's ClassAd. These statistics can then be used to better describe either the daemon or the host machine. The CacheD Cron is configured by specifying the job's attributes in the HTCondor configuration.

Figure 5.2 shows the configuration in order for the CacheD to periodically run a

```
CACHED_CRON_CONFIG_VAL =
   $(RELEASE_DIR)/bin/condor_config_val
CACHED_CRON_JOBLIST = $(STARTD_CRON_JOBLIST) test
CACHED_CRON_TEST_MODE = Periodic
CACHED_CRON_TEST_EXECUTABLE = $(RELEASE_DIR)/test.sh
CACHED_CRON_TEST_PERIOD = 15s
```

Figure 5.2: CacheD Cron Configuration

```
TestResult = 100
TestRan = TRUE
TestHost = "hostname.unl.edu"
```

Figure 5.3: Example Output from CacheD Cron: `test.sh`

program named "test." The executable, `test.sh`, will run tests and output a ClassAd that will be merged into the CacheD's ClassAd. It will be run at a period of every 15 seconds.

The output of the test executable is ClassAds that will be injected into the daemon. Figure 5.3 shows the example output from running the test program. In this output, it sets three attributes, an integer, a boolean value, and a string.

A example of using the CacheD Cron is to measure the IO operations per second that a host is able to complete. This information can be used to better match caches with machines which can run the applications.

5.3 Uses of the Policy Language in the CacheD

The CacheD uses the ClassAd policy language when communicating with other daemons. For each interaction, the CacheD must make a decision, and therefore relies on the ClassAd policies in order to decide whether to perform an action. Each of these actions are described briefly in the introduction to this chapter. We will now discuss the details of those interactions and choices the CacheD may make in the next few sections.

5.3.1 Choosing a Replication Target

Each cache has a single "origin CacheD." This CacheD is the CacheD where the user initially uploaded the cache. This origin CacheD has the option to proactively replicate the cache without it being requested by jobs. The data transfer can occur while another job is currently being run.

The origin CacheD will periodically query the HTCondor Collector to receive a list of CacheD ClassAds. Then, the origin will iterate through each of these ClassAds, attempting to match the cache with a CacheD.

For each ClassAd from the cache and a remote CacheD, the origin will attempt a mutual match. Therefore, the cache must accept the CacheD, and the CacheD must accept the cache. The `Requirements` expression is evaluated for both of the ClassAds. The default basic `Requirements` expression is to require that the CacheD has enough disk space for the cache.

For the CacheD, the default `Requirements` are:

```
Requirements = MY.TotalDisk > TARGET.DiskUsage
```

And for an uploaded cache, the default `Requirements` are:

```
Requirements = MY.DiskUsage < TARGET.TotalDisk
```

`MY` refers to attributes in the current ClassAd, while `TARGET` refers to attributes in the matching ClassAd. `TotalDisk` is the amount of disk available to a CacheD. `DiskUsage` is the total file size of the cache. An example value of the `TotalDisk` can be seen in the example ClassAd of a CacheD shown above in Figure 5.1 on page 96.

If the cache and remote CacheD match, the origin CacheD will send a cache replication request to the remote CacheD. The remote CacheD will then decide if it will accept the replication request. If the two do not match, then the origin server will not send a replication request to the remote CacheD.

Additionally, there are special attributes available during this matching. One special attribute used during some of the experiments is the `CacheRequested` attribute. This attribute is set to the boolean `TRUE` when the cache is requested by a job. When the cache is requested by a origin CacheD replication request, it is set to `FALSE`. This attribute can be used in a cache's `Requirements` expression to limit cache replication to only those nodes that have jobs that have requested the cache. An example expression would be:

```
Requirements = (MY.DiskUsage < TARGET.TotalDisk) &&
   (TARGET.CacheRequested =?= true)
```

Further special attributes are planned, such as an attribute whose value is the number of replications of the cache already completed. This can be used to limit the number of CacheDs that have the cache.

5.3.2 Accepting Cache Replication

A CacheD can receive cache replication requests from three sources:

1. An origin CacheD sending out proactive replication requests.
2. A job requesting a cache.
3. A child CacheD (as described in section 4.3.6).

In each of these requests, the receiving CacheD has the choice to accept the replication or deny it. When it receives a cache replication request, it looks up the

Attribute	**Use**
`CacheRequested`	Boolean cache property set to `TRUE` when the cache has been requested by a job.
`CacheReplicas`	Cache property available on the cache origin. A numerical value representing the number of complete replicas stored by CacheDs.
`DiskUsage`	Size, in kilobytes, of the cache.
`TotalDisk`	Available disk, in kilobytes, to store caches.
`BandwidthUsed`	Bandwidth used on the CacheD host, in Gbps.
`CacheDIops`	CacheD property of current measured IOPS on the host.
`ActiveTransfers`	Number of active transfers on the CacheD.

Table 5.1: Sample CacheD Attributes

cache's ClassAd and does mutual matching with its own ClassAd. This is similar to the mutual matching done when an origin CacheD is issuing proactive replication requests. It is important to re-run this mutual matching in case the CacheD's state has changed. The CacheD's state could change if it has downloaded a large cache, therefore, altering the available disk for additional caches.

If the CacheD's mutual matching with the cache's ClassAd is successful, then the CacheD will accept the cache and begin negotiating transfer methods. If the CacheD and the cache's ClassAd do not match, then the CacheD will reject the cache.

The approval or rejection of the cache is done asynchronously from the request for replication. Therefore, the CacheD keeps a data structure of rejected caches (accepted caches are kept in the local cache database). When the client next asks for the replication status of the cache, the CacheD will respond with the accept or reject status. The cache rejection request expires after 15 minutes.

```
ReplicationMethods = "BITTORRENT, DIRECT"
```

Figure 5.4: Example Replication Method for a Cache

5.3.3 Transfer Method

Each cache has a list of acceptable transfer methods. A user may set this list of acceptable transfer methods when uploading the cache. This list is priority ordered, with the preferred transfer method listed first.

In the example shown in Figure 5.4, the cache has a preference for transferring the files over BitTorrent, but will accept the Direct transfer method if needed. Transfer methods are described in full in Section 4.3.1. These are the only possible methods now, but may expand to other methods in the future.

After accepting a cache to be downloaded, the CacheD will negotiate the transfer method with the cache's ClassAd that was downloaded during the acceptance testing stage. The cache's ClassAd includes the `ReplicationMethods` attribute, which is a priority list of acceptable transfer methods. The CacheD has its own `ReplicationMethods` that is set in its configuration. The CacheD iterates through its own methods until it finds a matching transfer method in the cache's methods.

As an independent agent, the CacheD prefers its own transfer priority list rather than the cache's priority list. Psuedo code for the transfer negotiation is shown in Algorithm 1.

5.4 Example Policies

In order to better describe how data should be moved, we must categorize the data as shared or unique, private or non-private. This creates a 2x2 matrix of possibilities of data. Below, we define each of these categorizations. Each of these categories comes

Algorithm 1 Negotiating Transfer Method Function

```
$cacheMethods \leftarrow cacheClassAd[ReplicationMethods]$
$cachedMethods \leftarrow config(ReplicationMethods)$
for all $cachedMethod \in cachedMethods$ do
    for all $cacheMethod \in cacheMethods$ do
        if $cachedMethod = cacheMethod$ then
            return $cachedMethod$
        end if
    end for
end for
```

with its own restrictions on how the data may be moved and how it is presented to the user.

	Public	**Private**
Shared	Executables and Libraries	Personal Identifying Information
Unique	Input Parameters	BLAST Query Files

Table 5.2: Example Data Types

Shared data is data that is the same for multiple jobs in a job set. In many cases, the majority of the files in the job sandbox can be considered shared data. Examples of shared data are job executables and libraries.

Frequently the job executables are the same for a large number of jobs. Since the executables are the same, contextualization of the job is done through other methods, such as arguments or parameter files. An example application that would use the same executables and libraries are Monte Carlo [15] simulations. In these applications, the executables are the same for every job. Each job is given a unique identifier which is used for the starting condition for the random generation.

Experimental data could also be shared between multiple jobs in a job set. This can include common input data such as databases. For example, BLAST [9] jobs require a database of sequences of proteins which are then matched with specific queries. The database is typically the same for a large number of queries.

We define unique data as data which is different for each job. The unique data may be small, such as parameter files. Or they may be large, such as sections of a database to search. By definition, unique data is defined as data which would not benefit from shared transfer; no other job needs the same data. For our consideration, data which is not the same for every job in a job set, but is shared between jobs in a subset of the jobs, will be considered shared data.

We define private data as data which the user wants to prevent others from viewing. The level of privacy requested by a user could determine how it can be enforced. It could be enforced through authenticated access, encrypted data transfers, or both. In most cases, authenticated data access is sufficient.

Private versions of shared and unique data cannot use the same optimization as public data. For example, the data could not be transferred using a caching daemon if authenticated access is required. Transferring data unauthenticated, even encrypted, is dangerous due to susceptibility to brute force decryption.

An example policy for a cache that is composed of public shared data could be:

```
Requirements = MY.DiskUsage < TARGET.TotalDisk
ReplicationMethods = "BITTORRENT, DIRECT"
```

This policy will allow the CacheD to proactively replicate the caches to available CacheDs. Further, it will allow the CacheD to replicate using the BitTorrent method, or the Direct method if the remote CacheD does not support or prefer BitTorrent.

For private shared data, an example policy would look like:

```
Requirements = (MY.DiskUsage < TARGET.TotalDisk) &&
   (TARGET.CacheRequested =?= true)
ReplicationMethods = "DIRECT"
```

In this method, the cache would only be replicated to nodes where it is requested. This will minimize the number of nodes that have this private data stored. Second, it will use the DIRECT method of distribution, which uses an authenticated and optionally encrypted method of transfer.

Unique data cannot benefit from caching. But it can benefit from faster transfers. As we have seen in Chapter 4, even when caching is not turned on, the unique transfer methods that the CacheD uses can benefit the stage-in time for this unique data. It is possible to package many jobs worth of unique data into a cache and transfer the cache for each job. Then, on subsequent execution that require unique data from the package, it will be immediately available from the CacheD's cache.

Packaged unique private data can be transferred with the same policies as shared private data. Unique public data can be transferred with shared public policies.

5.4.1 Policies Utilizing Extended Attributes

Since ClassAds are schema-less and extendable, attributes can be added that can help in matching.

One example policy is to ban BitTorrent at certain OSG sites. This policy is useful if sites have policies against certain transfer methods. While running our experiments from the previous chapter, we were contacted by the administrators of the clusters at Brookhaven National Lab (BNL). They discovered that we were running BitTorrent on their clusters, contrary to their policy. The following policy would only use the Direct transfer method at BNL.

```
Requirements = (MY.DiskUsage < TARGET.TotalDisk) &&
   (TARGET.CacheRequested =?= true)
ReplicationMethods = ifThenElse(regexp(".*.bnl.gov$",
   TARGET.Name), "DIRECT", "BITTORRENT,DIRECT")
```

The `Requirements` are similar to previous policies. The `ReplicationMethods` uses the `ifThenElse` ClassAd function in order determine which replication method to use. It uses a regular expression to determine if the target CacheD is from BNL. It tests if the `Name` ends with the domain name of the `bnl.edu`, the domain for BNL. If it does match, then the Direct method is chosen. If the `Name` does not match `bnl.edu`, then both BitTorrent and Direct methods are allowed.

Administrators can add custom attributes using the CacheD Cron. For example, administrators could add attributes that list the organizations that own the storage resources. When this attribute is available, both the CacheD and the caches can make policies that will use it.

An example CacheD policy:

```
CacheDOwners = "CMS,HCC"
Requirements = (MY.TotalDisk > TARGET.DiskUsage) &&
   stringListIMember(TARGET.CacheOwner, MY.CacheDOwners)
ReplicationMethods = "BITTORRENT,DIRECT"
```

In this policy, the CacheD would only accept replications of caches which have the attribute `CacheDOwners` and it is set to either `CMS` or `HCC`. There is no method of enforcing the requirement that only members of the CMS or HCC organizations have this attribute; therefore, you must create a trust relationship with CacheDs that are allowed to communicate.

A cache policy that would match this policy is:

```
CacheOwner = "HCC"
Requirements = (MY.DiskUsage < TARGET.TotalDisk)
```

```
ReplicationMethods = "BITTORRENT,DIRECT"
```

Further, a cache could set a policy to only replicate to resources that are owned by the same organization to which they belong:

```
CacheOwner = "HCC"
Requirements = (MY.DiskUsage < TARGET.TotalDisk) &&
   stringListIMember(MY.CacheOwner, TARGET.CacheDOwners)
ReplicationMethods = "BITTORRENT,DIRECT"
```

5.5 User Scenario

In order to illustrate a user's experience with the policy language, we will use the BLAST example that is used in the previous chapter's experiments. In this example, we have three types of data:

BLAST Database: A shared public database which will be used by every execution of the job. The database is widely distributed; therefore, there is no private data to hide.

BLAST Executables: Public executables that are originally from the NIH website. They will be used by every job in the workflow.

Query Files: Multiple small files that may not be public. Each query file is used only by one job.

The BLAST database was shown to be optimally distributed in Chapter 4 with the CacheD and BitTorrent. Therefore, the user would first create the cache, then give it the policy to be proactively replicated to caches with BitTorrent.

```
$ importCache nrdb <path/to/db>/nrdb*
$ setReplicationPolicy "MY.DiskUsage < TARGET.TotalDisk"
   "BITTORRENT,DIRECT"
```

Next, the BLAST executables are much smaller than the BLAST database, but they are used by every execution. Therefore, they would benefit from caching and we will use the CacheD. Since executables are small, the replication will not take long, and there is no need to proactively replicate the executables. But, there is nothing private in the BLAST executables, therefore they will be distributed over BitTorrent or the Direct method.

```
$ importCache blast_executables <path/to/exes>/blast*
$ setReplicationPolicy "(MY.DiskUsage < TARGET.TotalDisk)
   && (TARGET.CacheRequested =?= true)"
   "BITTORRENT,DIRECT"
```

Finally, the BLAST queries may be private. The queries are stored in many small files. The queries may or may not be optimally transfered and managed by the CacheD. If the BLAST jobs are short, then the submission node could constantly be transferring the query files to resources that need them. If the number of running jobs is significant, then this could become a bottleneck. Using the CacheD, the researcher could avoid significant transfers from the submit host. By grouping all the queries into a single cache, then transferring that cache for each job, any other jobs on that same node would request the file from the cache rather than the submission node. If the jobs are short, then this cache would be used often, at the cost of transferring all of the queries the first time.

Since the queries are private, the CacheD must transfer them with the Direct method so that the transfers are authenticated and encrypted. Further, the CacheD must only replicate them to the nodes that actually request them, so as not to expose the queries on more nodes than necessary.

```
$ importCache blast_executables <path/to/exes>/blast*
$ setReplicationPolicy "(MY.DiskUsage < TARGET.TotalDisk)
   && (TARGET.CacheRequested =?= true)" "DIRECT"
```

5.6 Conclusions

In a distributed computing system, independent agents are designed to act on behalf of entities such as users, hosts, or entire clusters. A policy language must exist so that the entities can express their goals to the agents. In this chapter, we have designed a policy language based on the HTCondor ClassAds that can be used for expressing policy for data distribution. We have identified three interaction points for caching agents and designed the semantics for their interaction.

Additionally, we described different input data scenarios and possible replication policies for them. We gave a recommended policy configuration for the popular BLAST application, including explanations about the policy configuration. We also included the commands to create and set the policy configuration.

The policy language has been implemented in the CacheD, where it has been tested in Chapter 4. Further, we illustrated a user scenario of setting this policy language.

Chapter 6

Conclusion

In this dissertation we optimized distributed computing workflows on a campus grid. We were interested in optimizing a researcher's use of the computational and storage resources on the campus to increase the reliability and decrease the time to solution for scientific results. Prior work was enhanced to allow greater access to opportunistic computational resources for researchers on a campus. We then expanded our work to the data needs of modern workflows on the campus.

Bosco is used to effortlessly create a remote submission endpoint on a cluster without requiring the administrator to install any software. Bosco is a remote submission framework based upon HTCondor. It uses the SSH protocol to submit and monitor remotely submitted jobs. Additionally, it performs file transfers using the same SSH connection.

Improving the user experience was a primary goal of Bosco. We addressed the user experience by improving the interaction with the user during the installation and configuration. Another problem area we found is when a user must debug issues with distributed software. In order to address this, we created a traceroute-like utility. The traceroute utility tests every step of the job submission process, from network access to a properly configured remote scheduler. If an error is found at any step of the traceroute, a useful message is given to the user, including possible steps to fix the problem.

Bosco and the Campus Factory combine to make an easy-to-use framework that can distribute jobs to many computational clusters on a campus. Users are able to effectively distribute their processing to multiple clusters using this framework. We showed that Bosco transparently and effectively distributes computational jobs across multiple clusters on a campus while maintaining a simple user experience.

Bosco's usage has increased since we originally published the Bosco paper. For example, it is heavily used by the University of Chicago in order to submit OSG processing to opportunistic resources around the country. They find Bosco useful since it does not require the installation of any software on the remote cluster. Additionally, it has been used in several publications by the CMS experiment when they have used opportunistic resources for data processing.

For data distribution on the campus, we have presented the HTCondor CacheD, a framework to decrease the stage-in time for large shared input datasets. The experiments proved that the CacheD decreases stage-in time for these datasets. Additionally, the transfer method that the CacheD used can significantly affect the stage-in time of the jobs.

The BitTorrent transfer method proved to be an efficient method to transfer caches from the originator to the execution hosts. In fact, the transfer time for jobs did not increase as the number of distinct nodes requesting the data increased. Any bottlenecks that surround the cluster are therefore irrelevant using the BitTorrent transfer method. In addition, we found that the CacheD using BitTorrent transfer method out-performed the popular HTTP transfer method on the Open Science Grid.

In a distributed computing system, independent agents are designed to act on behalf of entities such as users, hosts, or entire clusters. A policy language must exist so that the entities can express their goals to the agents.

We have designed a policy language based on the HTCondor ClassAds that can be used for expressing policy in data distribution. We have identified three interaction points for caching agents and designed the semantics for their interaction. The policy language has been implemented in the CacheD.

Through Bosco, we have created a framework for easy-to-use job submission on the campus. Through the CacheD, we have created an efficient data distribution agent for the campus. And through a policy language we created, storage agents can negotiate and represent the users' and resource owners' goals. This has created a comprehensive framework for campus computing.

6.1 Future Work

6.1.1 Debugging

Debugging in distributed computing has always been a challenge. Many different systems working together can create barriers for message and error propagation.

Debugging in Bosco has always been difficult. To alleviate some of the debugging burden, we created the `traceroute` utility described in Section 3.2.3. But that is not enough to solve every issue. See Figure 6.1 for the flow of job submission in Bosco.

In Figure 6.1, the Bosco job submission starts at the user, and it goes through six daemons before the submission reaches the Slurm Scheduler on the remote cluster. An error can occur at each step in this process. The error must propagate back to the user if there is one. But, propagating the error is very difficult, since each daemon has different methods of internal error propagation. For example, the GridManager

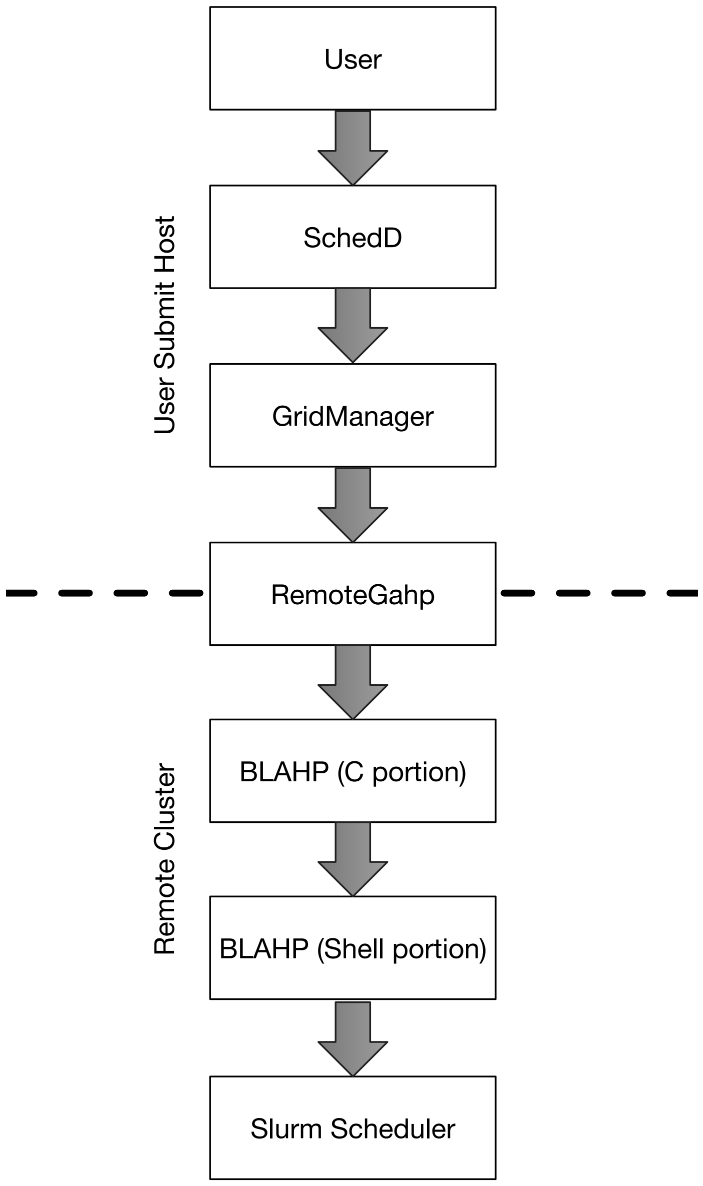


Figure 6.1: Bosco Job Submission Flow From Submission Host to Remote Cluster

can propagate an error back to the SchedD through HTCondor ClassAds. But the BLAHP's shell portion cannot propagate an error back except through the Linux standard error. Further work needs to be done, and parts of the Bosco submission chain need to be modified or refined in order to enable improved error propagation.

6.1.2 Flexible Transfer Types

The CacheD currently has support for two transfer methods, the Direct and the BitTorrent methods. The BitTorrent method uses the libtorrent library directly for transfers.

The Direct method uses HTCondor's file transfer service. The file transfer service is expandable through file transfer plugins. The CacheD would benefit from enabling similar file transfer plugins in order to allow proper negotiation of file transfer methods.

A possible solution is to build off of the file transfer plugins method. An executable advertises its transfer capabilities to the CacheD, which in turn uses those capabilities to negotiate with other CacheDs for transferring caches.

6.1.3 Co-Scheduling of Data and Computation

Scheduling of jobs with the caching data could optimize job placement. Currently, there is no knowledge of the cache placement when scheduling a job. If the scheduler knew where replicas of the cache were located, it could schedule the jobs to be run in those locations. Running the job near the cache will eliminate stage-in time.

Currently, the CacheD reports each cache stored locally to the cache's origin. The origin CacheD keeps a data structure of the locations of the cache replicas. However, the job needs to specify which caches are required for the job to run. The scheduler would then call out to the origin CacheD to see if the cache is located at the target node. This could be done using the HTCondor ClassAd library plugins.

6.1.4 Quantify Increase in Productivity

Bosco and the HTCondor CacheD were designed to make distributed computing on the campus easy to use. Additionally, they should increase the productivity of researchers performing distributed computing.

It is difficult to measure the increase in productivity of researchers. A survey could be performed to aggregate the opinions of researchers in regards to Bosco and CacheD. Additionally, we could measure how long it takes researchers to begin with a local application and convert it to run on the campus's distributed resources. Measuring this would provide a method to quantify that Bosco and CacheD increase the productivity of their users.

Bibliography

[1] Comprehensive r archive network. `http://cran.r-project.org/`. Accessed: 6/28/15.

[2] Extreme science and engineering discovery environment (xsede). `https://www.xsede.org/`. Accessed: 6/28/15.

[3] Osg operations. `https://twiki.opensciencegrid.org/bin/view/Operations/WebHome`. Accessed: 07/24/2015.

[4] Package snow. `http://cran.r-project.org/web/packages/snow/index.html`. Accessed: 5/22/15.

[5] Package parallel. `https://stat.ethz.ch/R-manual/R-devel/library/parallel/doc/parallel.pdf`. Accessed: 5/22/15.

[6] rsync. `https://rsync.samba.org/`. Accessed: 5/22/15.

[7] William Allcock, John Bresnahan, Rajkumar Kettimuthu, Michael Link, Catalin Dumitrescu, Ioan Raicu, and Ian Foster. The globus striped gridftp framework and server. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 54. IEEE Computer Society, 2005.

[8] Kent Alstad. The average web page has almost doubled in size since 2010. `http://www.webperformancetoday.com/2013/06/05/web-page-growth-2010-2013/`. Accessed: 7/2/15.

[9] Stephen F Altschul, Warren Gish, Webb Miller, Eugene W Myers, and David J Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, 1990.

[10] E.C. Amazon. Amazon elastic compute cloud. `http://aws.amazon.com/ec2/`, Jaunary 2011. Accessed: 07/24/2015.

[11] Garhan Attebury, Andrew Baranovski, Ken Bloom, Brian Bockelman, Dorian Kcira, James Letts, Tanya Levshina, Carl Lundestedt, Terrence Martin, Will Maier, et al. Hadoop distributed file system for the grid. In *Nuclear Science Symposium Conference Record (NSS/MIC), 2009 IEEE*, pages 1056–1061. IEEE, 2009.

[12] L Bauerdick, D Benjamin, K Bloom, B Bockelman, D Bradley, S Dasu, M Ernst, R Gardner, A Hanushevsky, H Ito, et al. Using xrootd to federate regional storage. In *Journal of Physics: Conference Series*, volume 396, page 042009. IOP Publishing, 2012.

[13] LAT Bauerdick, K Bloom, B Bockelman, DC Bradley, S Dasu, JM Dost, I Sfiligoi, A Tadel, M Tadel, F Wuerthwein, et al. Xrootd, disk-based, caching proxy for optimization of data access, data placement and data replication. In *Journal of Physics: Conference Series*, volume 513, page 042044. IOP Publishing, 2014.

[14] John Bent, Venkateshwaran Venkataramani, Nick LeRoy, Alain Roy, Joseph Stanley, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Miron Livny. Flexibility, manageability, and performance in a grid storage appliance. In *High*

Performance Distributed Computing, 2002. HPDC-11 2002. Proceedings. 11th IEEE International Symposium on, pages 3–12. IEEE, 2002.

[15] Kurt Binder and Dieter W Heermann. *Monte Carlo simulation in statistical physics: an introduction*. Springer, 2010.

[16] Jakob Blomer, Predrag Buncic, and Thomas Fuhrmann. Cernvm-fs: delivering scientific software to globally distributed computing resources. In *Proceedings of the first international workshop on Network-aware data management*, pages 49–56. ACM, 2011.

[17] Brian Bockelman. Stashcache: Data services for the osg. `https://indico.fnal.gov/getFile.py/access?contribId=47&sessionId=5&resId=0&materialId=slides&confId=8580`, March 2015.

[18] D. Bradley, O. Gutsche, K. Hahn, B. Holzman, S. Padhi, H. Pi, D. Spiga, I. Sfiligoi, E. Vaandering, et al. Use of glide-ins in CMS for production and analysis. In *Journal of Physics: Conference Series*, volume 219, page 072013. IOP Publishing, 2010.

[19] Miguel Branco, David Cameron, Benjamin Gaidioz, Vincent Garonne, Birger Koblitz, Mario Lassnig, Ricardo Rocha, Pedro Salgado, and Torre Wenaus. Managing atlas data on a petabyte-scale with dq2. In *Journal of Physics: Conference Series*, volume 119, page 062017. IOP Publishing, 2008.

[20] Cyril Briquet, Xavier Dalem, Sébastien Jodogne, and Pierre-Arnoul de Marneffe. Scheduling data-intensive bags of tasks in p2p grids with bittorrent-enabled data distribution. In *Proceedings of the second workshop on Use of P2P, GRID and agents for the development of content networks*, pages 39–48. ACM, 2007.

[21] Bram Cohen. The bittorrent protocol specification, 2008.

[22] Adaptive Computing and Green Computing. Torque resource manager. `http://www.adaptivecomputing.com/products/open-source/torque/`. Accessed: 07/24/2015.

[23] Platform Computing. Lsf scheduler. `http://www-03.ibm.com/systems/platformcomputing/products/lsf/`. Accessed: 6/28/15.

[24] George Coulouris. Blast benchmark. `http://fiehnlab.ucdavis.edu/staff/kind/Collector/Benchmark/Blast_Benchmark`. Accessed: 07/24/2015.

[25] Ewa Deelman and Ann Chervenak. Data management challenges of data-intensive scientific workflows. In *Cluster Computing and the Grid, 2008. CC-GRID'08. 8th IEEE International Symposium on*, pages 687–692. IEEE, 2008.

[26] Jochen Dinger and Oliver P Waldhorst. Decentralized bootstrapping of p2p systems: a practical view. In *NETWORKING 2009*, pages 703–715. Springer, 2009.

[27] Alvise Dorigo, Peter Elmer, Fabrizio Furano, and Andrew Hanushevsky. Xrootd-a highly scalable architecture for data access. *WSEAS Transactions on Computers*, 1(4.3), 2005.

[28] I. Foster and C. Kesselman. *The grid: blueprint for a new computing infrastructure.* Morgan Kaufmann, 2004.

[29] Ian Foster. The globus toolkit for grid computing. In *Cluster Computing and the Grid, IEEE International Symposium on*, pages 2–2. IEEE Computer Society, 2001.

[30] Ian Foster. Globus online: Accelerating and democratizing science through cloud-based services. *IEEE Internet Computing*, 15(3), 2011.

[31] Ian Foster and Carl Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of High Performance Computing Applications*, 11(2):115–128, 1997.

[32] Ian Foster and Carl Kesselman. The globus toolkit. *The grid: blueprint for a new computing infrastructure*, pages 259–278, 1999.

[33] D. Fraser. OSG campus grids meeting, October 2010. `http://indico.fnal.gov/conferenceDisplay.py?confId=3674`.

[34] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke. Condor-G: A computation management agent for multi-institutional grids. *Cluster Computing*, 5(3):237–246, 2002.

[35] Gabriele Garzoglio, Tanya Levshina, Mats Rynge, Chander Sehgal, and Marko Slyz. Supporting shared resource usage for a diverse user community: the osg experience and lessons learned. In *Journal of Physics: Conference Series*, volume 396, page 032046. IOP Publishing, 2012.

[36] Wolfgang Gentzsch. Sun grid engine: Towards creating a compute power grid. In *Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on*, pages 35–36. IEEE, 2001.

[37] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999.

[38] Chen He, Derek Weitzel, David Swanson, and Ying Lu. Hog: Distributed hadoop mapreduce on the grid. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 1276–1283. IEEE, 2012.

[39] Robert L Henderson. Job scheduling under the portable batch system. In *Job scheduling strategies for parallel processing*, pages 279–294. Springer, 1995.

[40] Jerry L Hintze and Ray D Nelson. Violin plots: a box plot-density trace synergism. *The American Statistician*, 52(2):181–184, 1998.

[41] Dirk HUFNAGEL, Oliver GUTSCHE, David MASON, Marco MAMBELLI, Anthony TIRADANI, Parag MHASHILKAR, Krista LARSON, and Burt HOLZMAN. Enabling opportunistic resources for cms computing operations. In *Journal of Physics: Conference Series*, 2015. To be published.

[42] Cluster Resources Inc. Cluster resources :: Products - TORQUE Resource Manager:. `http://www.clusterresources.com/pages/products/torque-resource-manager.php`, Jaunary 2011. Accessed: 07/24/2015.

[43] KDnuggets. Top languages for analytics, data mining, data science, 2013.

[44] Tevfik Kosar and Miron Livny. Stork: Making data placement a first class citizen in the grid. In *Distributed Computing Systems, 2004. Proceedings. 24th International Conference on*, pages 342–349. IEEE, 2004.

[45] Peter Kreuzer, Dirk Hufnagel, D Dykstra, O Gutsche, M Tadel, I Sfiligoi, J Letts, F Wuerthwein, A McCrea, B Bockelman, et al. Opportunistic resource usage in cms. In *Journal of Physics: Conference Series*, volume 513, page 062028. IOP Publishing, 2014.

[46] Arnaud Legout, Nikitas Liogkas, Eddie Kohler, and Lixia Zhang. Clustering and sharing incentives in bittorrent systems. In *ACM SIGMETRICS Performance Evaluation Review*, volume 35, pages 301–312. ACM, 2007.

[47] Michael J Litzkow, Miron Livny, and Matt W Mutka. Condor-a hunter of idle workstations. In *Distributed Computing Systems, 1988., 8th International Conference on*, pages 104–111. IEEE, 1988.

[48] Zhuoqing Morley Mao, Jennifer Rexford, Jia Wang, and Randy H Katz. Towards an accurate as-level traceroute tool. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 365–378. ACM, 2003.

[49] Zach Miller, Dan Bradley, Todd Tannenbaum, and Igor Sfiligoi. Flexible session management in a distributed environment. In *Journal of Physics: Conference Series*, volume 219, page 042017. IOP Publishing, 2010.

[50] J.H. Morris, M. Satyanarayanan, M.H. Conner, J.H. Howard, D.S. Rosenthal, and F.D. Smith. Andrew: A distributed personal computing environment. *Communications of the ACM*, 29(3):201, 1986.

[51] A Nordberg. Rasterbar libtorrent. *Available on http://www. rasterbar. com/products/libttorrent, Last access Dec. 1st*, 2011.

[52] University of Wisconsin. Glow. `http://www.cs.wisc.edu/condor/glow/`, January 2003. Accessed: 07/24/2015.

[53] OpenBSD. Openssh. `http://www.openssh.com/`. Accessed: 5/22/15.

[54] Stefan PIPEROV, Dirk HUFNAGEL, David MASON, Marco MAMBELLI, Meenakshi NARAIN, Anthony TIRADANI, Alan MALTA RODRIGUES, and

Lothar BAUERDICK. Operational experience with opportunistic use of allocation based hpc facilities for cms data processing. In *Journal of Physics: Conference Series*, 2015. To be published.

[55] Ruth Pordes, Don Petravick, Bill Kramer, Doug Olson, Miron Livny, Alain Roy, Paul Avery, Kent Blackburn, Torre Wenaus, Frank Würthwein, et al. The open science grid. In *Journal of Physics: Conference Series*, volume 78, page 012057. IOP Publishing, 2007.

[56] Jon Postel and Joyce Reynolds. File transfer protocol. 1985.

[57] Jeffrey S Racine. Rstudio: A platform-independent ide for r and sweave. *Journal of Applied Econometrics*, 27(1):167–172, 2012.

[58] Arcot Rajasekar, Reagan Moore, Chien-yi Hou, Christopher A Lee, Richard Marciano, Antoine de Torcy, Michael Wan, Wayne Schroeder, Sheau-Yen Chen, Lucas Gilbert, et al. irods primer: integrated rule-oriented data system. *Synthesis Lectures on Information Concepts, Retrieval, and Services*, 2(1):1–143, 2010.

[59] Rajesh Raman, Miron Livny, and Marvin Solomon. Matchmaking: Distributed resource management for high throughput computing. In *High Performance Distributed Computing, 1998. Proceedings. The Seventh International Symposium on*, pages 140–146. IEEE, 1998.

[60] D. Rebatto, F. Prelz, G. Fiorentino, M. Mezzadri, E. Martelli, and E. Molinari. Blahp: A local batch system abstraction layer for global use. Poster, 2006.

[61] J Rehn, T Barrass, D Bonacorsi, J Hernandez, I Semeniouk, L Tuura, and Y Wu. Phedex high-throughput data transfer management system. *Computing in High Energy and Nuclear Physics (CHEP) 2006*, 2006.

[62] K Rexer. Data miner survey-2013 survey summary report. *Rexer Analytics, Winchester*, 2013.

[63] Mathilde Romberg. The unicore grid infrastructure. *Scientific Programming*, 10(2):149–157, 2002.

[64] Robert R Schaller. Moore's law: past, present and future. *Spectrum, IEEE*, 34(6):52–59, 1997.

[65] Frank B Schmuck and Roger L Haskin. Gpfs: A shared-disk file system for large computing clusters. In *FAST*, volume 2, page 19, 2002.

[66] Igor Sfiligoi. glideinwmsa generic pilot-based workload management system. In *Journal of Physics: Conference Series*, volume 119, page 062044. IOP Publishing, 2008.

[67] Arie Shoshani, Alex Sim, and Junmin Gu. Storage resource managers: Middleware components for grid storage. In *NASA Conference Publication*, pages 209–224. Citeseer, 2002.

[68] Squid. Squid: optimizing web delivery. http://www.squid-cache.org/, 2015. Accessed: 2015-02-28.

[69] Miklos Szeredi et al. Fuse: Filesystem in userspace, 2010.

[70] R Core Team et al. R: A language and environment for statistical computing. 2012. `http://www.r-project.org/`.

[71] RDevelopment Core Team et al. R: A language and environment for statistical computing. *R foundation for Statistical Computing*, 2005.

[72] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: The Condor experience. *Concurrency and Computation: Practice and Experience*, 17(2-4):323–356, 2005.

[73] Douglas Thain, Jim Basney, Se-Chang Son, and Miron Livny. The kangaroo approach to data movement on the grid. In *High Performance Distributed Computing, 2001. Proceedings. 10th IEEE International Symposium on*, pages 325–333. IEEE, 2001.

[74] Douglas Thain and Miron Livny. Multiple bypass: Interposition agents for distributed computing. *Cluster Computing*, 4(1):39–47, 2001.

[75] Douglas Thain and Miron Livny. Parrot: An application environment for data-intensive computing. *Scalable Computing: Practice and Experience*, 6(3):9–18, 2005.

[76] Rick Wagner, Mahidhar Tatineni, Eva Hocks, Kenneth Yoshimoto, Scott Sakai, Michael L Norman, Brian Bockelman, Igor Sfiligoi, Matevz Tadel, James Letts, et al. Using gordon to accelerate lhc science. In *Proceedings of the Conference on Extreme Science and Engineering Discovery Environment: Gateway to Discovery*, page 59. ACM, 2013.

[77] Dan Walsh, Bob Lyon, Gary Sager, JM Chang, D Goldberg, S Kleiman, T Lyon, R Sandberg, and P Weiss. Overview of the sun network file system. In *Proceedings of the Winter Usenix Conference*. Dallas, TX, 1985.

[78] Dennis Wegener, Thierry Sengstag, Stelios Sfakianakis, Stefan Ruping, and Anthony Assi. Gridr: An r-based grid-enabled tool for data analysis in acgt clinico-genomics trials. In *e-Science and Grid Computing, IEEE International Conference on*, pages 228–235. IEEE, 2007.

[79] Baohua Wei, Gilles Fedak, and Franck Cappello. Collaborative data distribution with bittorrent for computational desktop grids. In *Parallel and Distributed Computing, 2005. ISPDC 2005. The 4th International Symposium on*, pages 250–257. IEEE, 2005.

[80] Baohua Wei, Gilles Fedak, and Franck Cappello. Scheduling independent tasks sharing large data distributed with bittorrent. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, pages 219–226. IEEE Computer Society, 2005.

[81] Baohua Wei, Gilles Fedak, and Franck Cappello. Towards efficient data distribution on computational desktop grids with bittorrent. *Future Generation Computer Systems*, 23(8):983–989, 2007.

[82] D Weitzel, I Sfiligoi, B Bockelman, J Frey, F Wuerthwein, D Fraser, and D Swanson. Accessing opportunistic resources with bosco. *Journal of Physics: Conference Series*, 513(3):032105, 2014.

[83] Derek Weitzel. Campus factory. `http://djw8605.github.io/campus-factory`. Accessed: 07/24/2015.

[84] Derek Weitzel. Campus grids: A framework to facilitate resource sharing. Master's thesis, University of Nebraska, 2011.

[85] Derek Weitzel, Brian Bockelman, and David Swanson. Distributed caching using the htcondor cached. In *Proceedings for Conference on Parallel and Distributed Processing Techniques and Applications*, 2015. Accepted.

[86] Tom White. *Hadoop: the definitive guide.* O'Reilly, 2012.

[87] Tatu Ylonen and Chris Lonvick. The secure shell (ssh) protocol architecture. 2006.

[88] Andy B Yoo, Morris A Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In *Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer, 2003.

[89] M. Zvada, D. Benjamin, and I. Sfiligoi. CDF GlideinWMS usage in Grid computing of high energy physics. In *Journal of Physics: Conference Series*, volume 219, page 062031. IOP Publishing, 2010.