GatorShare: A File System Framework for High-Throughput Data Management

Jiangyan Xu and Renato Figueiredo
Advanced Computing and Information Systems Laboratory
University of Florida
jiangyan@ufl.edu, renato@acis.ufl.edu

ABSTRACT

Voluntary Computing systems or Desktop Grids (DGs) enable sharing of commodity computing resources across the globe and have gained tremendous popularity among scientific research communities. Data management is one of the major challenges of adopting the Voluntary Computing paradigm for large data-intensive applications. To date, middleware for supporting such applications either lacks an efficient cooperative data distribution scheme or cannot easily accommodate existing data-intensive applications due to the requirement for using middleware-specific APIs.

To address this challenge, in this paper we introduce GatorShare, a data management framework that offers a file system interface and an extensible architecture designed to support multiple data transfer protocols, including BitTorrent, based on which we implement a cooperative data distribution service for DGs. It eases the integration with Desktop Grids and enables high-throughput data management for unmodified data-intensive applications. To improve the performance of BitTorrent in Desktop Grids, we have enhanced BitTorrent by making it fully decentralized and capable of supporting partial file downloading in an on-demand fashion.

To justify this approach we present a quantitative evaluation of the framework in terms of data distribution efficiency. Experimental results show that the framework significantly improves the data dissemination performance for unmodified data-intensive applications compared to a traditional client/server architecture.

Keywords

Desktop Grids, Voluntary Computing, BitTorrent, Data Management, High-Throughput Computing

1. INTRODUCTION

In recent years a computing paradigm called *Voluntary Computing* has emerged and gained considerable popularity. Using Voluntary Computing systems, scientists around the world can collaborate and share computing resources and data. This collaboration allows small and middle sized organizations to conduct experiments without hefty investment on IT resources. In addition, by utilizing idle resources which are otherwise wasted, this approach builds a more economic distributed computing system.

The main contribution of this work is two-fold: 1) we present GatorShare, a data management framework with both a POSIX file system interface and a REST Web Service interface; 2) we describe the implementation of a BitTorrent service using this framework that provides efficient cooperative data distribution. In this way we combine the benefits of cooperative data distribution and file system interface to provide high-throughput, efficient data distribution and easy integration with existing unmodified applications. Unlike many grid middleware frameworks, it has a simple and identical setup process on each machine which requires no special components. This enables auto-configuration capabilities for clients to join and leave the network freely.

Furthermore, the GatorShare framework has the following features: a) its file system interface allows intuitive interaction with files and folders as if they were local; b) it is an extensible framework that allows easy addition of protocols and file system services by registering desired file system structures and implementing service handlers. The GatorShare BitTorrent service has the following features: a) it supports tracker-less fully decentralized BitTorrent; b) it provides an on-demand piece-level BitTorrent sharing scheme. Issues with security and NATs are not directly addressed in GatorShare but it seamlessly integrates with IPOP [25] virtual network to provide end-to-end security and NAT traversal across the WAN.

2. BACKGROUND AND MOTIVATIONS

Voluntary Computing systems are also frequently referred to as *Desktop Grids* or *Public-Resource Computing systems* since they are often comprised of desktop systems and the resources usually come as donations from the general public. Throughout this paper, we use these terms interchangeably. In Voluntary Computing systems, organizations and individuals contribute their computing resources to a pool that other participants can access. A very popular Voluntary Computing platform is BOINC [15, 3]. In recent years, projects based on BOINC system such as SETI@home [12, 16] have been very successful and assembled considerable computing power. As of January 2010, SETI@home processes data averaging about 730 TeraFLOPS [13].

2.1 Barrier to Entry in Voluntary Computing

One problem with many existing (e.g. BOINC-like) Voluntary Computing systems is that the collaboration is under an asymmetric relationship, as pointed out by Anderson [15]. In such systems, not everyone who contributes can easily use the resources because to use such systems the users need to develop their applications with the system APIs. Even though the goals of the BOINC system include lowering the barrier of entry and minimizing development effort, this might not be easy for ordinary users and/or for large projects with non-trivial sized existing applications.

In another example, Fedak et al. [22] present a comprehensive framework for data management for Desktop Grids. It uses metadata to tag data objects and provides multiple transfer protocols, including FTP and BitTorrent to disseminate data. To use their framework, developers have to utilize their specific APIs which may require extensive application modification in some cases.

Grid Appliance. One approach to address the barrier of entry and support unmodified applications in Desktop Grids is the Grid Appliance [8, 31]. Grid Appliances are virtual appliances connected by an IPOP virtual network [25] to create a self-configuring Desktop Grid that makes traditional, unmodified applications available to all participants via Condor [4]. The Grid Appliance forms the core infrastructure of Archer [24, 1], a voluntary computing infrastructure for computer architecture research. In Archer, computer architecture simulation software is installed on each Grid Appliance instance. Scientists can submit jobs and contribute computing resources from any node in this DG. The introduction of Grid Appliances and Archer not only simplifies the deployment and usage of DGs but also opens up the possibility for users to carry the experiment code and data around and do research anywhere without being restricted to a fixed location. As a result, users can enjoy the high portability of DG nodes thanks to the auto-configuration, virtual machines and virtual networking features.

However, being able to autonomically construct a pool of virtualized resources is not enough for data-intensive applications using such systems. In Archer, data sharing between peer nodes is achieved by a wide-area NFS file system over the encrypted tunnels provided by the virtual network. The Archer NFS setup self-configures read-only user file systems by adding features including auto-mounting file system exports of a certain node by their host name. This approach is sufficient for pools within a single site or sites connected by high-speed Internet. However, with limited bandwidth, especially upload bandwidth which could as low as 512 Kbps, in a typical residential network, it is prohibitively slow for users to submit jobs with large input data to multiple workers. A cooperative data distribution method is needed to address this problem.

In typical Archer simulation applications, the data needed to be distributed over the wide-area network are of the order of GBs each. In some cases only a fraction of the data is accessed while in others whole files are used. One example is in full-system architecture simulators, such as VirtuTech Simics, which boot simulated systems from disk images with an entire O/S installation, but only touch a small fraction of the data blocks during a given simulation. Under such circumstances, even regular BitTorrent is inefficient because it transfers the entire data. An on-demand partial data distribution method would be much helpful in this case.

2.2 Data Distribution Efficiency

How to provide fast, cost-efficient and reliable data distribution for DGs, especially when bandwidth is limited, is still an unsolved problem. Moreover, the integration of transfer techniques including BitTorrent with existing applications not targeted at such Voluntary Computing systems remains a challenging task. There has been much research on cooperative data distribution schemes. Shark [17] and WheelFS [29] are distributed file systems with a POSIX interface and use cooperative data distribution protocol. Both of the systems provide prototypes and not robust implementations. BitTorrent [21], in contrast, is proven to be robust and scalable.

Researchers have conducted studies using BitTorrent-like P2P methods to distribute data in DGs. Study results [30] show that BitTorrent is a promising technique to facilitate data distribution in scientific Grids for data-intensive applications, but modification may be necessary to suit certain usage scenarios [20]. There are existing studies on the performance of BitTorrent-like P2P transports as a means to distribute data in scientific computing. As Al-Kiswany et al. [14] conclude, such P2P data dissemination techniques could bring unjustified overhead to over-provisioned networks but as user communities grow, P2P based mechanisms will outperform techniques.

Chen et al. [20] take advantage of BitTorrent and FUSE [6] to provision services as VMs in high-speed networks. They leverage a Piece-On-Demand mechanism, which is similar to ours, to give high priority to data chuncks requested by applications via FUSE. Therefore, services in VMs can be started prior to full distribution of VMs images. The scope of [20] is in high-speed networks and enterprise environment where dedicated deployment servers are used in the distribution process. Moreover, they do not address or evaluate the ease of deployment and auto-configuration issues.

Research projects on data management in conventional Data Grids such as Stork [28] and Storage Resource Broker (SRB) [18] take relevant approaches. Stork and SRB both aim to bridge the heterogeneity of data resources by providing a uniform interface and handle different storage types or transport protocols dynamically. Being in production for years, these system are fully functional and reliable. They have a full range of features and have been proven successful. However, being targeted at conventional Data Grids where centralized administration and manual setup is the norm, they are not suitable for Desktop Grids where churn of participants often happens. Our solution, on the other hand, focuses on self-configuration that deals with this characteristic of DGs.

The rest of this paper is organized as follows. In Section 3, we introduce the concept and usage of GatorShare interfaces. In Section 4, we describe the architecture of GatorShare. Section 5 gives detailed description of how the components in GatorShare work together to achieve the tasks. In Section 6 we demonstrate the usage scenario for GatorShare integrated with Grid Appliances. In Section we give the evaluation on the performance of GatorShare. Section 8 and 9 are future work and conclusions.

3. USING GATORSHARE

GatorShare provides data distribution functionalities via a web service interface and a file system interface. We design the file system interface as a lightweight client in terms of computing and storage resource consumption that uses the web service to provide a virtual file system in which users can deal with remote data objects as local files. The web service can also be used independently so that other clients using HTTP protocol can be developed and one service can serve multiple clients. This allows possible deployment of one such data distribution web service per LAN. All machines in a LAN can access this one service either through the virtual file system interface or a web browser, which requires no installation of extra software, on each machine. In this section we introduce the usage of both file system and web service interfaces which we call *GatorShare Client* (GSClient) and *GatorShare Server* (GSServer), respectively.

3.1 Data Distribution APIs

We first define the essential set of operations in an abstract form. The abstract operations can be mapped to concrete services such as BitTorrent, dictionary services that we have developed in GatorShare so far.

***service*.STORE(*uid*, *data*)** Store the data to the service by the uid. This could be the *publish* operation in BitTorrent or the *put* operation in a dictionary.

***data service*.RETRIEVE(*uid*)** Retrieve the data from the service by the uid. This could be a *download* in BitTorrent or a *get* in a dictionary.

GatorShare stores and retrieves data by *unique identifiers* (UIDs). To make naming more user-friendly, UIDs are composed of two elements: a *namespace* and a *name* under that namespace. The two elements are concatenated together to construct a UID. The purpose of the namespace is that it specifies a globally unique ID to a user (`e.g. jiangyan.ufl.edu`) or a group (`e.g. GA_Releases`) to avoid naming collisions. GatorShare itself does not manage the assignment or collision control of the namespaces. It can be configured as whatever makes sense in each deployment. For example, in Grid Appliances, we configure the namespace to be the hostname of each node, which is derived from its virtual IP. Currently we do not authenticate the owner of a namespace either. Under a namespace, users can name the data with local knowledge. Naming collision within a namespace can be prevented through local mechanisms.

Using the APIs, clients can access services such as BitTorrent and dictionary service in GatorShare. A dictionary service is a map of keys and values. It can be implemented with a plethora of underlying structures, such as Distributed Hash Tables, cloud services, or even centralized databases. Distributed Hash Table (DHT) is a type of P2P system that provides reliable key-value pair storage and efficient lookup. Previous work [26] describes the implementation of DHT in Brunet [19], a P2P overlay we use as the backend of our virtual network solution, IPOP. As the API is an abstraction of all services of the same usage and we generally use DHT in a production deployment for its reliability and decentralization features, in this paper we use DHT to refer to this type of dictionary service regardless of the concrete implementation when there is no confusion. The abstract APIs apply to both web service and file system interface.

3.2 Data Distribution Web Service

GSServer exposes core data distribution operations as a set of RESTful [23] web services. REST, or REpresentational State Transfer, is an architectural style of web service design. RESTful architecture enables simple and scalable implementations of web services which are built around the concept of "resources".

The APIs for the primarily used BitTorrent service in this work are summarized in Table 1. The APIs in the table follow the design principles of a RESTful web service architecture. We choose to use RESTful web service because the concept of "resources" intuitively depicts the services provided by GSServer. By using the APIs, clients accomplish data management tasks as if they were dealing with local resources. For example, to manipulate the BitTorrent service for a particular dataset, the client sees the service as a resource identified by the namespace and name of the item under the namespace. Then it either GETs it or POSTs it, with optional parameters to tweak the behavior of such GET or POST. In compliance with the RESTful web service conventions, we use URL segments to represent the mandatory arguments passed to the service, HTTP verbs to represent the nature of the operation, and URL parameters to represent service options.

3.3 Virtual File System

GatorShare's virtual file system, like other FUSE based systems, provides a means to access data-related (e.g. storage, sharing, transfer) services through a file system based interface. GatorShare defines an intuitive file system structure that users can comfortably use. The users perceive the file system as a regular local file system and can access "files" directly (e.g. `cat` command on a file reads it and prints the content). The changes to the file system are propagated in a way defined by the concrete file system service implementation.

Different file system services may have different semantics. For instance, modifications to files are not published in the BitTorrent file system but are put into the DHT in the DHT file system. Also in a BitTorrent file system, the data is published after the publisher creates a new file but the data distribution happens on-demand only when a data consumer reads the file on his/her machine. Figure 1 shows the structure of the BitTorrent file system.

For example, the GatorShare file system can be mounted as `/mnt/gatorshare`, which resembles a mounted network file system. The file system is identical to both data publishers and consumers across all the nodes in the Desktop Grid. This symmetry gives the feeling to users that both publishers and consumers use the same, shared network file system. Inside the mounted file system, path segments are parsed and interpreted as parameters to services. The way virtual paths are interpreted is similar to the handling of patterns of the REST APIs of GSServer. In stead of using HTTP verbs, we identify whether the operation is a read or a write and take actions based on the path and the file system "verb".

In Figure 1 the example path, with segments concatenated together, is `/bittorrent/jiangyan.ufl.edu/Data.dat`. Additional parameters can be attached to file names. For example, reading the path `/bittorrent/jiangyan.ufl.edu/Data.dat?od=true` enables on-demand partial file download of `Data.dat`. In this example we show a file system struc-

#	Operation	URL Template	Verb	Req. P/L	Resp. P/L
1	Publish through BT	/bt/{namespace}/{name}	POST	None	None
2	Download through BT	/bt/{namespace}/{name}	GET	None	Metainfo of the data
3	"Peek" data availability	/bt/{namespace}/{name}?peek=true	GET	None	Metainfo of the data
4	Download part of a file through BT	/bt/{namespace}/{name}?offset={offset}& btr={bytesToRead}	GET	None	Content of the bytes read.

Table 1: RESTful APIs of BitTorrent Service on GSServer. Columns: Operation on the resource, URL of the resource, Verb to use in HTTP requests, Payload of the request, Payload of the response. Operation #3 checks data availability without downloading it. Operation #3,4 together achieves On-Demand BitTorrent downloading.

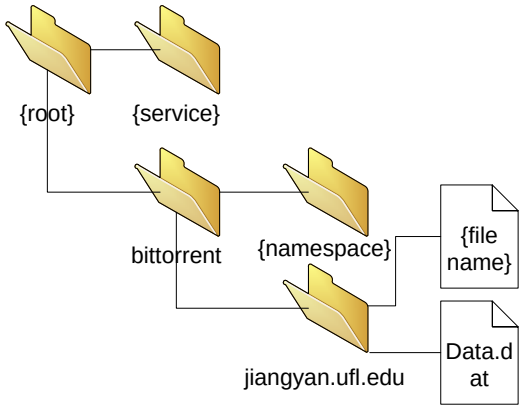

Figure 1: Virtual File System. The names enclosed in braces indicate the path segments' corresponding parameters in the APIs defined in Section 3.1.

ture used by BitTorrent file system that resembles the URL structure of REST APIs in Table 1. However, thanks to the fact that in GatorShare framework services and file systems are registered and dispatched using URL templates, the file system structure is configurable.

Assuming the structure in Figure 1, from the user's perspective, under the root folder mounted for GatorShare service, he first enters the directory that corresponds to the *service* he intends to use (bittorrent, in this case). Then he enters next level, the *namespace* directory. The directory contains all the data belonging to the namespace. The data could be files or a directory of files. The *name* of the item at the third level, whether for a file or a directory, is used along with the namespace to globally identify the datum. He can then read, create, modify files and directories in the namespace directory.

Despite the difference in constructing the path and identifying the action, in our design the user manipulates the data the same way using file system and web service: the combination of path (URL) and action (verb) decides the behavior. *Reading* a path downloads the data while *writing/creating* a path distributes it.

4. GATORSHARE ARCHITECTURE

In this section we describe the architecture and components of GatorShare. We design GatorShare to be a generic framework that can be used in various scenarios. In the scope of Desktop Grids, though, the GatorShare BitTorrent implementation has a Peer-to-Peer (P2P) architecture. On each node we have an identical setup (*GSClient* and *GSServer*). GSClient interfaces with FUSE; FUSE performs I/O operations on behalf of user applications, or simply, users.

In a Desktop Grid, participants join a pool of peers. Each node running GatorShare can both publish and consume data. If allowed by the DG's system functionality and policy, each of them can be a job submitter as well as a worker through a batch system. Figure 2 shows the overall architecture of GatorShare BitTorrent.

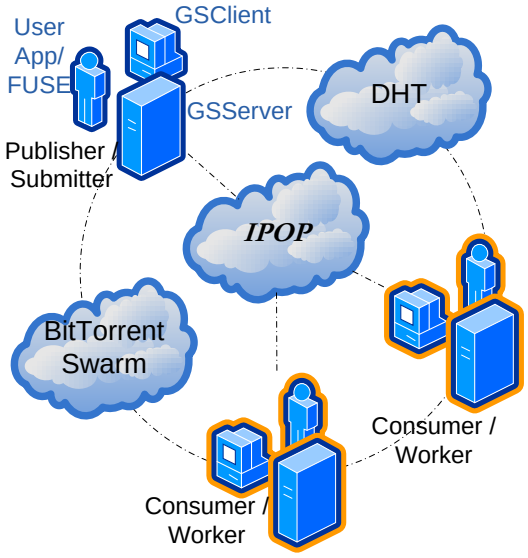

Figure 2: GatorShare BitTorrent High Level Architecture. This figure shows a decentralized architecture with the exception of a possible batch system master node (not shown).

GatorShare leverages three types of P2P overlays: BitTorrent swarm, DHT and IPOP overlay. GatorShare relies on DHT to lookup related information (BitTorrent torrent files and swarm information which traditionally are provided by a tracker). GatorShare peers form a BitTorrent swarm and collaboratively disseminate data across the Grid. Moreover, GatorShare can optionally run on IPOP virtual network to enable features such as NAT traversal and end-to-end security across the WAN.

4.1 Components within a Node

Figure 3 shows the overall software component architecture of GatorShare. There are two major GatorShare components on each node: a file system client (GSClient) and a server running data distribution services (GSServer).

GSServer implements the core logic for data distribution, interfaces with BitTorrent, DHT and other essential services. It manages a data store and torrent files associated with the data. GSClient integrates with underlying file system and fulfills file system operations requested by FUSE. GSClient manages a shadow file system, which stores metadata associated with files. We design the two components with the goal that GSClient should be as lightweight as possible.

By separating functionalities into Client/Server, GatorShare can be easily extended. Also, multiple GSClients can share one GSServer, which is preferable in some cases inside a single LAN, as described in Section 3.3. Thus, while GatorShare is globally P2P, it has a Client/Server architecture locally.

As Figure 3 shows, GSClient and GSServer have similar architecture and they communicate with HTTP protocol. Within each of them, there is an interface layer (Virtual File System and REST API), which defines the functionalities and presents an abstract view of the underlying system (resource accessible by file system and web interface). The second layer are dispatchers that examine requests and match them with registered services and then pass them along. The bottom layer is where the services reside. Both GSClient and GSServer have services that can be registered to some criteria. Services are represented as tabs and can be added and removed by configuration. In this paper we primarily look at BitTorrent services. Note that *Redirect File System*, *Dictionary Service* and *BitTorrent Library* are services and library external to but dependent by BitTorrent Service.

Subcomponents inside BitTorrent Service and BitTorrent File System are described in Section 4.2 and 4.3. The remainder of the section explains the components in details.

4.2 Distribution Service: BitTorrent

Distribution service provider (GSServer) provides the essential services for data management in GatorShare. While GSServer has a pluggable architecture for implementing new services, in the effort to build a high-throughput DG system, we focus on BitTorrent as the transport protocol.

GSServer allows publishing a file, downloading a whole file and downloading part of a file through BitTorrent. In a traditional BitTorrent downloading process, a user first downloads a torrent file for the data from a web server. BitTorrent client loads the torrent file and contacts a tracker that manages information of the BitTorrent swarm associated with the torrent and gets a list of peers. BitTorrent client then starts the downloading process by communicating with the peers and exchanging data pieces. The publishing process involves creating a torrent file for the data, uploading it to a web server and informing downloaders the URL for the torrent file.

In GatorShare we automate the process and modify the way that torrent files, swarm information are managed. Torrent files are stored in a dictionary service and thus are uploaded and retrieved by unique IDs. We implement GSTracker, a proxy on each peer handles standard tracker requests by putting and getting torrent metadata information and torrent swarm information to and from the dictionary service and eliminate the need for a centralized tracker.

Traditional BitTorrent applications do not support on-demand partial file downloading scheme so we need to modify BitTorrent protocol to achieve that. In BitTorrent, data chuncks are shared by the unit called pieces. The size of a piece is decided by the creator of torrents and the size information is stored in torrent files. In our on-demand downloading scheme, we round the requested file part to pieces and download whole pieces of a torrent. We implement a *Piece Information Service* on each peer so that each peer that has the whole data can serve requests for BitTorrent pieces. The process is explained in detail in Section 5.3.

As shown in Figure 3, *BitTorrent Manager* provides functionalities such as DownloadFile, PublishFile. *BitTorrent Piece Manager* is responsible for managing requested pieces for a torrent download. *BitTorrent Library* is the 3rd-party BitTorrent library that we use to handle BitTorrent tasks.

GSServer maintains its own file system structure as a data store to provide services such as BitTorrent publishing and downloading. Size of the data store dominates the disk space used by GatorShare on a computer.

4.3 File System Client

File system client (GSClient) uses FUSE to intercept user-space file system calls. It is programmed using an event-driven model. Whenever a file system operation is called, the *Event Dispatcher* looks at the call and matches the path and the operation with the registered file system services. (e.g. The BitTorrent File System in Figure 3). If a match is found, it fires an event and file system service handles the event, communicates with GSServer, does necessary file system operations and return control to FUSE.

GSClient manages a *shadow* file system and a *metadata* file system that provide support for the virtual file system. They both have the same structure as the virtual file system. Redirect File System, shown in Figure 3, redirects all the FUSE file system calls (e.g. `mkdir`) to the the two file system except those intercepted by Event Dispatcher.

The difference between the two file systems is that when FUSE writes a file, it is first written to the shadow file system and then gets uploaded while when FUSE reads a file, it first reads from metadata file system. All files in metadata file system are virtual files that contains nothing but meta information about the real files that reside in GSServer data store. Based on information in the virtual file, GSClient can fetch data from the real file.

The rationale behind this design is that 1) By separating the physical file systems that reads and writes are redirected to, there is no confusion about whether a file is a virtual file; 2) we store a bare minimum amount of data with GSClient and since GSClient and GSServer are on the same host or connected by a high-speed network, data bytes can be requested and served on the fly once the file is downloaded.

5. DATA DISTRIBUTION PROCESS

In this section we describe the process of data distribution among GatorShare peers, specifically through BitTorrent, in detail. Distribution process starts with a file system operation. In FUSE, a high-level read or write that users observe takes multiple file system operations. Event Dispatcher processes some of them and simply redirects others to the shadow file system. Specifically, it matches `getattr`

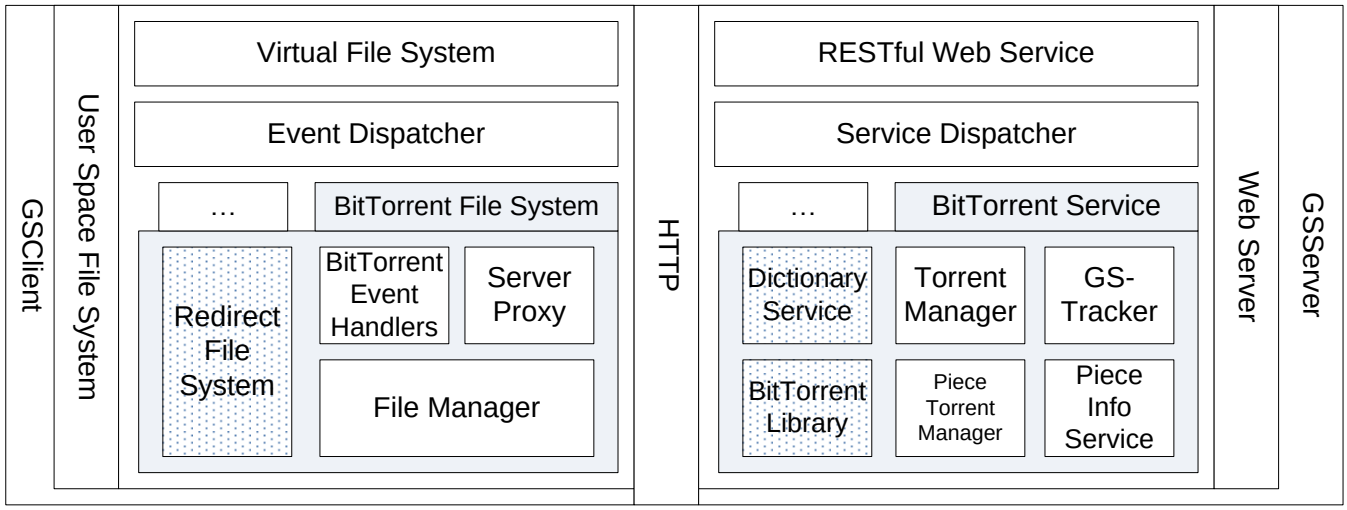

Figure 3: GatorShare Software Component Architecture. Inside the "container" is a tabbed environment where each service is represented as a tab. Components that this tab depends on but are not integral parts of this tab have stipple patterns.

(`getxattr`), `read` and `release` operations with the patterns that the BitTorrent file system registers. If the path is matched, an event is fired and the BitTorrent Event Handler takes the control.

We define the following events (in braces are the corresponding FUSE operations) in GSClient event handlers and the following description lists the actions to handle them.

ReleasedHandle (`release`): The file handle has been released and data has been written to the file. We upload it if the file is newly created.

GettingPathStatus (`getattr`): This is when FUSE is requesting the status of the path. We check the dictionary service to determine whether a file exists and potentially download it if it is associated with a read operation.

ReadingFile (`read`): The content of a file is requested and we need to either return it from locally downloaded file or download the requested bytes in the case of on-demand downloading.

Because of the multiple stages in reading and writing files, in GatorShare, a publish or download also takes multiple steps. Before going into detailed explanation of the data distribution process, we summarize what file system operations each BitTorrent operation involves.

Publish Data: `write` is redirected to the shadow directory; `release` publishes the file.

Download Entire File: `getattr` downloads the file, `read` retrieves the content from the file.

Download File Part: `getattr` downloads metadata about the file part, `read` downloads file part on-demand.

5.1 Publishing Data

A user creates(writes) a file under a namespace directory (as shown in Figure 1) in order to publish it. When FUSE performs a write operation, GSClient first redirects the operation to the shadow directory. After all the bytes have been written, FUSE releases the file. The process of handling `release()` is illustrated in Figure 4: 1) FUSE calls `release()`; 2) the call triggers the *ReleasedHandle* event in GSClient; 3) then this event is then handled by *BitTorrentFilesysEventHandler*, which stages in the data to server data store. Depending on the relative locations of the client and server, the "stage-in" could be using methods such as local file move, symbolic link, or network copying; 4) Afterward, it sends an HTTP request "POST `/bittorrent/jiangyan.ufl.edu/Data.dat`" to GSServer through the *ServerProxy*; 5) Upon receiving this request, GSServer generates the torrent file for the data, registers it with the GSTracker which puts the data into the DHT; 6) it starts to seed the data using the BitTorrent client.

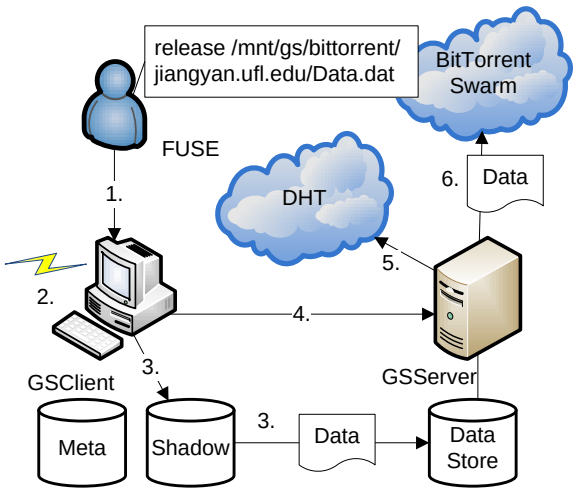

Figure 4: Handling `release()` call in Publishing Data [1].

[1]In the figures describing the data distribution process, we use arrows to indicate both message and data flow directions. We also annotate the data being transferred when it is not clear from the context.

5.2 Downloading Entire Data

To download data, the user application simply reads the path under the GatorShare virtual file system. The publisher can send the path it uses to publish the data to other nodes and they can use the same path to download it.

In GatorShare it is possible to have a file fully downloaded before the read or alternatively, a user can read part of file on-demand and GatorShare only downloads the part needed to accomplish the read.

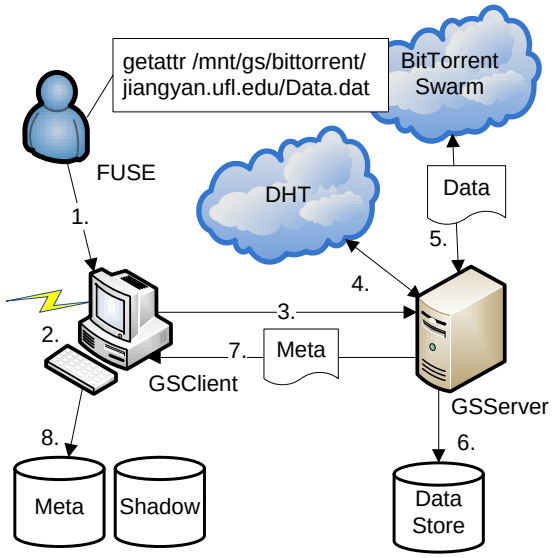

Figure 5: Handling `getattr()` call in Full Data Retrieval.

When FUSE reads a file, it first tries to get the attributes of the path. This is when the downloading process is initiated. Figure 5 demonstrates the handling of `getattr()` call: 1) FUSE calls `getattr` operation; 2) it triggers *GetPathStatus* event in GSClient; 3) ServerProxy sends "GET `/bittorrent/jiangyan.ufl.edu/Data.dat`" request for this path to GSServer; 4) GSServer in turn queries the DHT about using a key derived from the namespace and item name. If DHT indicates there is no such file available, GSServer returns an error and GSClient forwards it to FUSE. If there is such a file, DHT returns the torrent file; 5) GSServer parses it; 6) BitTorrent starts to download data to the server data store. No matter what tracker URL is provided in the torrent file, BitTorrent client always queries local GS-Tracker for swarm information. GSClient blocks while the download is in process; 7) After successful download, GSServer returns the meta file which contains the path to the data to GSClient; 8) GSClient then places the meta file in the meta directory and reports to FUSE that there is a file under such path.

After GSClient returns path status to FUSE, FUSE then `read`s the file, with *offset* and *read length* as parameters. This triggers the *ReadingFile* event. GSClient reads the meta file, uses the path in the meta data to locate the file and read the bytes from it and then returns them to FUSE.

The GS-Tracker is installed on each peer. It acts as an adapter between BitTorrent tracker clients and the DHT which keeps track of the torrent swarm information. The torrent creator in GatorShare specifies the local GS-Tracker as the sole tracker in torrent files. Each request to the tracker is translated into dictionary service operations.

5.3 Support for Partial Downloads

Users can embed the parameter `od=true` in file names to request on-demand partial downloads. It takes a similar process as full data downloading. The difference between them is that the BitTorrent download process does not start when FUSE requests the status of the path in the case of partial downloads. When GSClient sees the `od=true` parameter in the virtual path, it first peeks the availability of the file by attaching the `peek=true` parameter in the HTTP request to GSServer. To fulfill the request, GSServer downloads the torrent file and generates the meta info from the torrent file without downloading the data. It adds an entry in the meta info to indicate that this file is going to be downloaded on-demand.

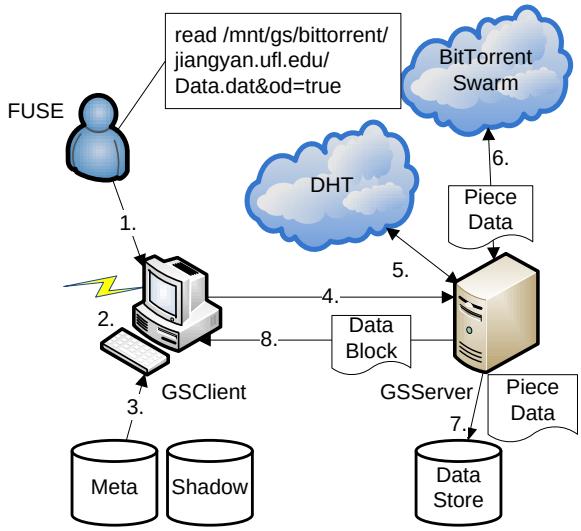

Figure 6: Handling `read()` in On-Demand Data Retrieval.

The handling of the subsequent `read()` operation is shown in Figure 6: 1) FUSE makes `read()` call; 2) GSClient triggers "ReadingFile" event, which has offset and bytesToRead as properties; 3) same as handling full data downloading, it reads the meta info and but in this case it sees the *on-demand* indicator; 4) it then sends a request GET `/bittorrent/jiangyan.ufl.edu/Data.dat?offset={offset}&btr={bytesToRead}` to GSServer with the *offset* and *bytesToRead* parameters; 5) GSServer then calculates the piece(s) need to download for the offset and bytesToRead based on the piece size stored in the torrent file and downloads the torrent file for the piece; 6) It loads the torrent file and gets information of the piece(s) and start the downloading process; 7) downloads are saved to server data store; 8) after successfully downloading the pieces, it seeks the data chunk requested and returns it to GSClient. GSClient now gives the bytes back to FUSE.

The collaborative downloading process for data pieces are like regular BitTorrent but we need some special steps to bootstrap the sharing of pieces. We now describe the protocol for piece-level BitTorrent. In the current system we treat torrent pieces as single-piece torrents. To enable data shar-

ing at a piece level, we implement a *Piece Information Service* (PINS) on each peer. The publisher, along with other peers that have the entire data, can bootstrap the sharing process. When a downloader needs to download a piece on-demand, it first checks the dictionary service whether there is already a torrent for this piece: if yes, it starts the process to download the piece with regular BitTorrent protocol; if no, it asks a peer in the peer list for the full data to serve such a piece by sending a request to the PINS on that peer. That peer then reads the piece from the whole data and generates a single-piece torrent for it and publishes it as a regular torrent. The downloader then is notified with the availability of the piece torrent so it can download the single-piece data via regular BitTorrent.

5.4 Implementation

Our goal is to create a multiple-platform framework so we strive to abstract the framework and make extending the framework easy. In GSClient, we separate system-independent logic from system dependent interfaces. We abstract the common properties and actions of a virtual file system and implement a FUSE-binding that works on Unix-like systems with FUSE [6] or MacFUSE [9]. A Windows binding can be implemented using Dokan [5].

With regard to GSServer, we use ASP.NET MVC [2] to create a pluggable architecture for the web service. ASP.NET MVC allows developers to simply specify a controller for each service and register it with a certain URL template on the web server. As a result, services other than BitTorrent or DHT can be easily added in the future. GSServer can be run on Mono XSP or with mod-mono on Apache HTTP Server [10]. For the BitTorrent service, we use and extend MonoTorrent [11], a C# BitTorrent library.

6. INTEGRATION WITH GRID APPLIANCE

In Section 3.3 we have showed the general usage of GatorShare's file system interface. In this section we describe the integration of GatorShare with Grid Appliance as a realistic example to apply the collaborative high-throughput data distribution to real Voluntary Computing environments. In a Grid Appliance pool, scheduling is handled by Condor. Jobs can be submitted from any node in the pool. Before the implementation of GatorShare, jobs are submitted with input data transported to workers via either a shared WAN NFS or direct transfer. Neither of such cases is efficient for data-intensive applications. Job results are transferred back by the same mechanisms.

6.1 Usage Scenario

We envision such a usage scenario that best demonstrates the power of GatorShare with the Grid Appliance.

A researcher, Alice, works with a university lab. This lab, along with other universities, has established a Grid Appliance pool that comprises machines from several LAN clusters across the US. She does experiments that run data-intensive applications and usually has jobs with input data. Machines within clusters and even across the clusters have high-speed network connections. Her Internet service at home has 10 Mbps downlink and 1 Mbps uplink. She often takes work home and wants to use the pool from the residential network at home.

6.2 Working with Grid Appliance

To integrate GatorShare with Grid Appliances, it is installed on each node in the pool. It provides an alternative data distribution method to users. GatorShare mounts a virtual file system at `/mnt/gs`. For small input data and non data-intensive applications, users can specify data dissemination method to be Condor file transfers or the Grid Appliance NFS setup (`/mnt/ganfs`). For large input data or data-intensive applications, users can specify `/mnt/gs` as the shared file system.

To submit a Condor job using GatorShare system, she first prepares the input data by creating them under `/mnt/gs`. GatorShare publishes the data and starts to serve them using BitTorrent. Here the publishing process only involves putting the related data information into DHT and starting to seed the data on the publisher machine. No data dissemination occurs at this point. Then she submits the Condor job and references the input data in the job description file as if the job were using a shared file system to transfer the data.

Then Condor schedules to run the job on a number of nodes. The downloading process starts when the daemon process reads the GatorShare file system on worker nodes. GatorShare processes on worker nodes independently query the DHT to obtain information and download the input data for the job. The publisher starts to transfer data after BitTorrent clients on the workers discover it and request data pieces. The collaboration happens gradually and workers start trading data pieces with each other. Then the bandwidth usage and load on the publisher gradually drop. Since the output data are usually small, Alice specifies results to transferred back via traditional methods after the job finishes.

7. EVALUATIONS

In this section we present the evaluation on functionality and performance of GatorShare. In the experiments we analyze the following performance metrics: a) latency of data distribution with GatorShare from a data publisher (job submitter) to a single downloader (worker) or multiple downloaders; b) bandwidth usage on data publisher; c) network overhead to accomplish the collaboratively downloading scheme.

The scalability of the platform as a whole has not been evaluated but the distributed systems (i.e. DHT and BitTorrent) we select to implement our service are well known to scale well. Scalability evaluation of IPOP and Brunet, which we use to provide additional functionalities, can be found in [25, 19].

7.1 Experiment Setup

Our experiments are conducted on Amazon Elastic Cloud (EC2) service. We deploy virtual machines of EC2 "small instance" type (1.7GB RAM, 1 virtual core, 32-bit platform). Each of the virtual machines has Ubuntu Linux operation system 9.10 and Mono C# 2.6 installed.

In the experiment we deploy one VM as the *master* node, which simulates the job submitter machine in a residential network. To simulate the low-bandwidth residential network environment, we use Linux's traffic control (tc) tool to throttle the bandwidth of the data publisher machine to enforce the limit of 10 Mbps down-link and 1 Mbps up-link. We deploy other VMs as *worker* nodes without throttling

to simulate the LAN environment or clusters connected by high-speed Internet.

For the experiment we have implemented a dictionary web service on Google App Engine [7], a hosting platform for cloud services, to simulate the DHT. Google App Engine provides reliable web server, data storage and monitoring tools. The dictionary service we have developed exposes dictionary operations as web services and GSServer communicates with it using HTTP protocol. Using such a platform we can monitor dictionary operation requests and responses, storage space for the metadata that GatorShare relies on.

7.2 Distribution Performance

In this experiment the master publishes data via either BitTorrent or NFS. Workers run a simple job that reads the input data from the FUSE file system of GatorShare or the mounted NFS file system and verifies the data using md5sum. We evaluate the distribution performance by measuring the distribution time and the overhead of the network transport.

We first show the distribution time between a single publisher/worker pair in Figure 7. To distinguish our modified version of BitTorrent from the original BitTorrent, we refer to our version as GS-BT. The result shows that under such one-to-one scenario where no collaboration happens, the distribution delay is similar among GS-BT, NFS and the ideal case.

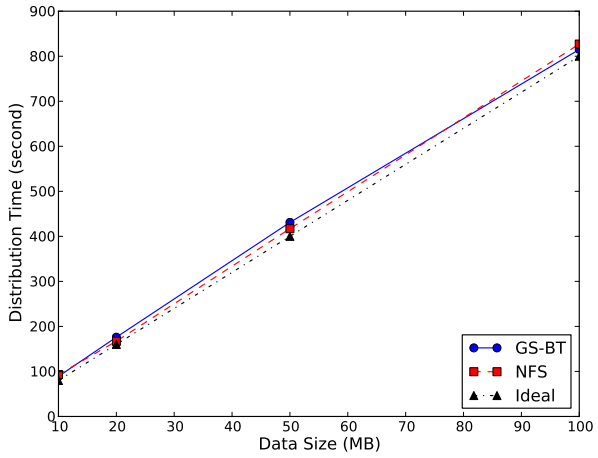

Figure 7: Distribution time between a single publisher/worker pair. The bandwidth on the publisher node is limited to 1 Mbps. The ideal distribution time (Ideal) is *Data Size / Upload Speed of Publisher*.

In Figure 8 we illustrate the scenario where one publisher is distributing data to ten workers. Under such scenario, each downloader gets an average of 0.1 Mbps, or 1/10 of the 1 Mbps, of the uploader bandwidth.

The result proves that (a) when an uploader is serving multiple downloaders and the bandwidth of the uploader is not abundant, GS-BT outperforms the ideal performance for a non-collaborative distribution mechanism; (b) NFS does not perform well when the shared bandwidth is scarce; and (c) the performance of GS-BT approaches the ideal case for collaborative distribution mechanisms. In the experiment,

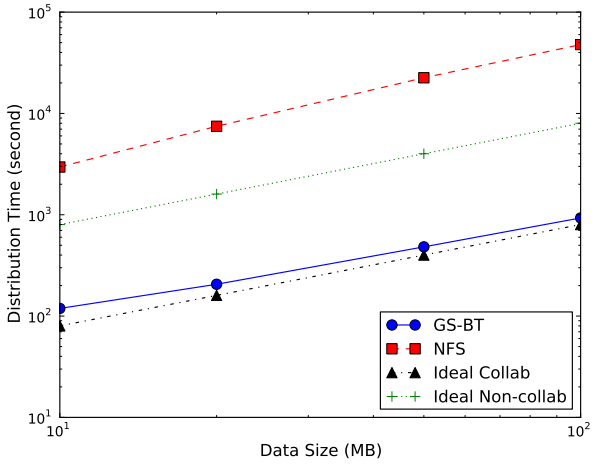

Figure 8: Distribution time between one publisher and ten workers. Both axes use logarithmic scales to show the linearity of growth. The bandwidth on the publisher node is limited to 1 Mbps. The ideal delay for collaborative distribution (Ideal Collab) is *Data Size / Upload Speed of Publisher*. The ideal delay for non-collaborative distribution (Ideal Non-collab) is *Data Size / (Upload Speed of Publisher / Number of downloaders)*.

we also find that the network traffic overhead of NFS is several times more than the actual payload size while the overhead of GS-BT is trivial.

7.3 On-Demand Partial Downloading Performance

In this experiment we use the ten-worker setup same as the whole data downloading experiment but the workers read only 200 KB from the start of the file. We measure the metrics shown in Table 2. The results show that our On-demand piece-level BitTorrent (OD-BT) implementation slightly outperforms NFS in terms of delay. However, the distribution time is highly unstable among workers. Due to optimization of file system I/O, the operating system downloads more than 200 KB. This could reduce the delay for future reads of the applications. In fact, in this experiment, as a 100 MB data has the piece size of 256 KB in our system, OD-BT already starts to download the second piece when the measurements are taken. As we can see in Table 2, the OD-BT downloads almost twice that of NFS and its throughput is twice the number of NFS.

Even if the delay for such a small read does not differentiate the performance of two protocols, the bandwidth consumption on the publisher node could serve as a metric for cost. We further look at the bandwidth consumption on the publisher node. Figure 9 shows that the bandwidth consumption decreases as peers start to collaborate in the case of OD-BT. The result suggests that data publisher consumes more bandwidth in the case of NFS. The time used for querying the dictionary service hosted on Google App Engine contributes to the startup delay of OD-BT.

7.4 Integration with Grid Appliance

Transport	Delay Avg. (sec)	Delay Stdev. (sec)	RX Avg. (MB)	Throughput Avg. (Mbps)
OD-BT	9.8	1.62	0.56	0.46
NFS	10.3	7.23	0.29	0.23

Table 2: Partial file distribution performance comparison between OD-BT and NFS for a read of 200 KB from a 100MB file. The columns are Transport protocol, Average of distribution delay, Standard deviation of distribution delay, Average of data received, Average of reception bandwidth.

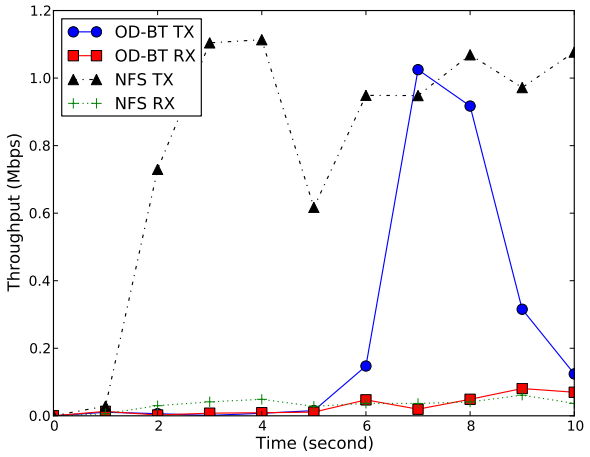

Figure 9: Comparison of Bandwidth Consumption on the submitter between OD-BT and NFS for a read of 200 KB from a 100MB file. RX and TX stand for reception and transmission, respectively.

To evaluate GatorShare integrated with Grid Appliance, we set up a Grid Appliance pool on EC2 with one job submitter, one server node running Condor Central Manager, and ten worker nodes. Same as with previous experiments, the job submitting node has only 1 Mbps up-link bandwidth. The publisher submits a job that computes the MD5 hash of the input data. The input data we use has the size of 100 MB. The submitter requests the pool to run the job ten times. The job gets published to ten workers as there are ten workers in total. The result matches the one in Figure 8. The ten copies of the job are started on workers after 85 seconds on average and terminated 952 seconds on average after it is started.

8. FUTURE WORK

The GatorShare architecture presented in this paper provides a basis for the creation of distributed file systems for Desktop Grids. Building upon this architecture, future work will focus on providing support for important features that include:

Caching: For datasets that are reused by applications, data locality enabled by caching can greatly improve the performance. Currently GSServer checks if requested data with the same unique id has already been downloaded, it is returned immediately. We plan to implement caching with a configurable size on GSServer so that users can set a limit on the disk space GatorShare can use and when the limit is reached, cache eviction policies will be consulted to evict old data to make space for new cache entries. In addition, with caching on GSClient we can save communication between GSClient and GSServer if the same result is "fresh" enough. The fact that we use ASP.NET to develop HTTP web services helps us leverage existing tools and protocols for HTTP caching. With caching enabled, GSClient can save the communication with GSServer so that file system operations on local cached content can be made faster.

Versioning: Some data, such as application binaries, are evolving throughout the lifetime of a DG pool. When a new version of application binaries come out, it needs to be distributed to machines in the pool. Currently we do not provide a native versioning mechanism and data associated with a unique name is considered immutable. Users need to explicitly put version number in file name and therefore two versions of a file are two files with different names.

On-Demand BitTorrent: On-demand BitTorrent saves the time and bandwidth for file to be fully downloaded when only part of it is used. Currently we only support on-demand piece-level BitTorrent download by users explicitly adding parameters to virtual paths. We will investigate ways to learn the usage patterns of certain data types and applications in order to autonomically control the means used to retrieve data.

Full File Replication: With the cache size limit, old data can be evicted so it is possible that a piece of data is removed automatically from all nodes except the publisher. If the data publisher then leaves the pool, the piece of data is lost. Full file replication can prevent data loss caused by publisher going offline. We plan to add this feature so that the data publisher can choose to fully migrate some data to other nodes.

Access Control: Currently access control is not addressed in GatorShare. Even though by using IPOP, it is possible to limit access to a machine by restricting connections to authenticated peers, there is no finer-grained access control. We are developing an intuitive and easy-to-use access control scheme for distributed file systems and will integrate it with GatorShare in the future.

Workflow: By applying workflow techniques, new customized services can be easily added as a set of clearly defined and reusable smaller tasks associated with the GatorShare file system events. A similar implementation is introduced in a Rule-Oriented Data System based on SRB, iRODS [27]. We are investigating methods to add the functionality to GatorShare.

9. CONCLUSIONS

In this paper we have introduced GatorShare, a file system based framework for high-throughput data management in the Desktop Grid environment. The major contributions of this work include that it tackles integration obstacles, performance bottleneck of distribution over the Internet by combining the file system interface and collaborative sharing mechanism, i.e. BitTorrent. Furthermore, GatorShare does not need centralized server or steps to set up dedicated components, nor does it need applications to be modified to use particular APIs. The same application is installed on each node and users publish and download data just by launch-

ing the application. We have shown that our implementation largely improves the data distribution performance for Desktop Grid environment when bandwidth is limited. With the collaborative distribution solution provided to unmodified applications, one can use Voluntary Computing pools from work, home, or even on the go.

References

[1] Archer project. http://www.archer-project.org/.

[2] Asp.net mvc. http://www.asp.net/mvc/.

[3] Boinc. http://boinc.berkeley.edu/.

[4] Condor project. http://www.cs.wisc.edu/condor/.

[5] Dokan. http://dokan-dev.net/en/.

[6] Filesystem in userspace. http://fuse.sourceforge.net/.

[7] Google app engine. http://code.google.com/appengine/.

[8] Grid appliance. http://www.grid-appliance.org/.

[9] Macfuse. http://code.google.com/p/macfuse/.

[10] Mono asp.net. http://www.mono-project.com/ASP.NET.

[11] Monotorrent. http://projects.qnetp.net/projects/show/monotorrent.

[12] Seti@home. http://setiathome.berkeley.edu/.

[13] Seti@home status at boincstats.com. http://boincstats.com/stats/project_graph.php?pr=sah.

[14] S. Al-Kiswany, M. Ripeanu, A. Iamnitchi, and S. Vazhkudai. Beyond music sharing: An evaluation of peer-to-peer data dissemination techniques in large scientific collaborations. *Journal of Grid Computing*, 7(1):91–114, March 2009.

[15] D. Anderson. Boinc: a system for public-resource computing and storage. In *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*, pages 4 – 10, nov. 2004.

[16] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. Seti@home: an experiment in public-resource computing. *Commun. ACM*, 45(11):56–61, November 2002.

[17] S. Annapureddy, M. J. Freedman, and D. Mazières. Shark: scaling file servers via cooperative caching. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 129–142, Berkeley, CA, USA, 2005. USENIX Association.

[18] C. Baru, R. Moore, A. Rajasekar, and M. Wan. The sdsc storage resource broker. In *In Proceedings of CASCON*, 1998.

[19] P. O. Boykin, J. S. A. Bridgewater, J. S. Kong, K. M. Lozev, B. A. Rezaei, and V. P. Roychowdhury. A symphony conducted by brunet. Sep 2007.

[20] Z. Chen, Y. Zhao, C. Lin, and Q. Wang. Accelerating large-scale data distribution in booming internet: effectiveness, bottlenecks and practices. *Consumer Electronics, IEEE Transactions on*, 55(2):518–526, August 2009.

[21] B. Cohen. Incentives build robustness in bittorrent, 2003.

[22] G. Fedak, H. He, and F. Cappello. Bitdew: A data management and distribution service with multi-protocol file transfer and metadata abstraction. *Journal of Network and Computer Applications*, 32(5):961–975, September 2009.

[23] R. T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, 2000.

[24] R. Figueiredo and et al. Archer: A community distributed computing infrastructure for computer architecture research and education. In *CollaborateCom*, November 2008.

[25] A. Ganguly, A. Agrawal, P. O. Boykin, and R. J. O. Figueiredo. Ip over p2p: enabling self-configuring virtual ip networks for grid computing. In *IPDPS*. IEEE, 2006.

[26] A. Ganguly, D. Wolinsky, P. Boykin, and R. Figueiredo. Decentralized dynamic host configuration in wide-area overlays of virtual workstations. In *International Parallel and Distributed Processing Symposium*, March 2007.

[27] M. Hedges, A. Hasan, and T. Blanke. Management and preservation of research data with irods. In *CIMS '07: Proceedings of the ACM first workshop on CyberInfrastructure: information management in eScience*, pages 17–22, New York, NY, USA, 2007. ACM.

[28] T. Kosar and M. Livny. Stork: Making data placement a first class citizen in the grid. In *ICDCS '04: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pages 342–349, Washington, DC, USA, 2004. IEEE Computer Society.

[29] J. Stribling and et al. Flexible, wide-area storage for distributed systems with wheelfs. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI '09)*, Boston, MA, April 2009.

[30] B. Wei, G. Fedak, and F. Cappello. Scheduling independent tasks sharing large data distributed with bittorrent. In *GRID '05: Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, pages 219–226, Washington, DC, USA, 2005. IEEE Computer Society.

[31] D. I. Wolinsky and R. J. O. Figueiredo. Simplifying resource sharing in voluntary grid computing with the grid appliance. In *IPDPS*, pages 1–8. IEEE, 2008.