TidyFS: A Simple and Small Distributed File System

Dennis Fetterly
Microsoft Research, Silicon Valley
fetterly@microsoft.com

Maya Haridasan
Microsoft Research, Silicon Valley
mayah@microsoft.com

Michael Isard
Microsoft Research, Silicon Valley
misard@microsoft.com

Swaminathan Sundararaman
University of Wisconsin, Madison
swami@cs.wisc.edu

Abstract

This paper describes TidyFS, a simple and small distributed file system that provides the abstractions necessary for data parallel computations on clusters. In recent years there has been an explosion of interest in computing using clusters of commodity, shared nothing computers. Frequently the primary I/O workload for such clusters is generated by a distributed execution engine such as MapReduce, Hadoop or Dryad, and is high-throughput, sequential, and read-mostly. Other large-scale distributed file systems have emerged to meet these workloads, notably the Google File System (GFS) and the Hadoop Distributed File System (HDFS). TidyFS differs from these earlier systems mostly by being simpler. The system avoids complex replication protocols and read/write code paths by exploiting properties of the workload such as the absence of concurrent writes to a file by multiple clients, and the existence of end-to-end fault tolerance in the execution engine. We describe the design of TidyFS and report some of our experiences operating the system over the past year for a community of a few dozen users. We note some advantages that stem from the system's simplicity and also enumerate lessons learned from our design choices that point out areas for future development.

1 Introduction

Shared-nothing compute clusters are a popular platform for scalable data-intensive computing. It is possible to achieve very high aggregate I/O throughput and total data storage volume at moderate cost on such a cluster by attaching commodity disk drives to each cluster computer.

In recent years data-intensive programs for shared-nothing clusters have typically used a data-parallel framework such as MapReduce [9], Hadoop [13], Dryad [16], or one of the higher level abstractions layered on top of them such as PIG [22], HIVE [14], or DryadLINQ [28]. In order to achieve scalability and fault-tolerance while remaining relatively simple, these computing frameworks adopt similar strategies for I/O workloads, including the following properties.

- Data are stored in streams which are striped across the cluster computers, so that a single stream is the logical concatenation of parts stored on the local file-systems of the compute machines.
- Computations are typically parallelized so that each part of a distributed stream is streamed as sequential reads by a single process, and where possible that process is executed on a computer that stores that part.
- In order to get high write bandwidth on commodity hard drives I/O is made sequential as far as possible. In order to simplify fault-tolerance and reduce communication the computing frameworks do not implement fine-grain transactions across processes. Consequently, modifications to datasets are made by replacing an entire dataset with a new copy rather than making in-place updates to an existing stored copy.
- In order to achieve high write bandwidth the streams are written in parallel by many processes at once. In order to reduce communication and simplify lock management however, each part is typically written sequentially by a single writer. After all the parts have been output in their entirety they are assembled to form a stream.
- In order to provide fault-tolerance when computers or disks become unavailable, the frameworks automatically re-execute sub-computations to regenerate missing subsets of output datasets.

Distributed file systems have been developed to support this style of write-once, high-throughput, parallel streaming data access. These include the I/O subsystem in River [4], the Google File System (GFS) [11], and the Hadoop Distributed File System (HDFS) [6, 24]. Unsurprisingly there are similarities in the designs of these systems: metadata for the entire file system is centralized and stored separately from stream data, which is striped across the regular compute nodes. They differ in their level of complexity and their support for general filesystem operations: for example GFS allows updates in the middle of existing streams, and concurrent appends by multiple writers, while the HDFS community has struggled with the tradeoff between the utility and complexity of even single-writer append operations [10] to a stream that can be concurrently read. In order to provide fault-tolerance all the systems replicate parts of a distributed stream, and provide reliable write semantics so that a stream append is replicated before the write is acknowledged to the client.

This paper presents TidyFS, a distributed file system which is specifically targeted *only* to workloads that satisfy the properties itemized above. The goal is to simplify the system as far as possible by exploiting this restricted workload. Parts of a distributed stream are invisible to readers until fully written and commited, and subsequently immutable, which eliminates substantial semantic and implementation complexity of GFS and HDFS with appends. Replication is lazy, relying on the end-to-end fault tolerance of the computing platform to recover from data lost before replication is complete, which allows us to eliminate custom I/O APIs so parts are read and written directly using the underlying compute node file system interface.

Sections 2 and 3 outline the data model and architecture of TidyFS in more detail. Section 4 describes some of our experiences operating TidyFS for over a year. We describe related work in Section 5 and then conclude with a discussion of some of the tradeoffs of our design choices.

2 TidyFS usage

This section describes the TidyFS data model and the typical usage patterns adopted by clients. As noted in the introduction, the design aims to make TidyFS as simple as possible by exploiting properties of its target workload. Wherever features are introduced in the following discussion that might seem to go beyond the simplest necessary functionality, we attempt to justify them with examples of their use.

2.1 Data Model

TidyFS makes a hard distinction between data and metadata. Data are stored as blobs on the compute nodes of the cluster and these blobs are immutable once written. Metadata describe how data blobs are combined to form larger datasets, and may also contain semantic information about the data being stored, such as their type. Metadata are stored in a centralized reliable component, and are in general mutable.

TidyFS exposes data to clients using a stream abstraction. A stream is a sequence of parts, and a part is the atomic unit of data understood by the system. Each part is in general replicated on multiple cluster computers to provide fault-tolerance. A part may be a single file accessed using a traditional file system interface or it may be a collection of files with a more complex type—for example, TidyFS supports SQL database parts which are pairs of files corresponding to a database and its log. The operations required by TidyFS are common across multiple native file systems and databases so this design is not limited to Windows-based systems.

The sequence of parts in a stream can be modified, and parts can be removed from or added to a stream; these operations allow the incremental construction of streams such as long-lived log files. A part can be a member of multiple streams, which allows the creation of a snapshot or clone of a particular stream, or a subset of a stream's parts. Clients can explicitly discover the number, sizes and locations of the parts in a stream, and use this information for example to optimize placement of computations close to their input data.

Each stream is endowed with a (possibly infinite) lease. Leases can be extended indefinitely, however, if a lease expires the corresponding stream is deleted. Typically a client will maintain a short lease on output streams until they are completely written so that partial outputs are garbage-collected in the case of client failure. When a stream is deleted any parts that it contained which are not contained in any other stream are also scheduled for deletion.

Each part and stream is decorated with a set of metadata represented as a key-value store. Metadata include for example the length and fingerprint of a part, and the name, total length and fingerprint of a stream. Rabin fingerprints [7] are used so that the stream fingerprint can be computed using the part fingerprints and lengths without needing to consult the actual data. Applications may also store arbitrary named blobs in the metadata, and these are used for example to describe the compression or partitioning scheme used when generating a stream, or the types of records contained in the stream.

2.2 Client access patterns

A client may read data contained in a TidyFS stream by fetching the sequence of part ids that comprise the stream, and then requesting a path to directly access the data associated with a particular part id. This path describes a read-only file or files in the local file system of a cluster computer, and native interfaces (e.g., NTFS or SQL Server) are used to open and read the file. In the case of a remote file, a CIFS path is returned by the metadata server. The metadata server uses its knowledge of the cluster network topology to provide the path of the part replica that is closest to the requesting process. The metadata server prioritizes local replicas, then replicas stored on a computer within the same rack, and finally replicas stored on a computer in another rack.

To write data, a client first decides which stream the data will be contained in, creating a new empty stream if necessary. The client then asks TidyFS to "pre-allocate" a set of part ids associated with that stream. When a client process wishes to write data, it selects one of these pre-allocated part ids and asks TidyFS for a write path for that part. Typically the write path is located on the computer that the client process is running on, assuming that computer has space available. The client then uses native interfaces to write data to that path. When it has finished writing the client closes the file and adds the new part to the stream, supplying its size and fingerprint. At this point the data in this part is visible to other clients, and immutable. If a stream is deleted, for example due to lease expiration, its pre-allocated part ids are retired and will not be allocated to subsequent writers.

Optionally the client may request multiple write paths and write the data on multiple computers so that the part is eagerly replicated before being committed, otherwise the system is responsible for replicating it lazily. The byte-oriented interface of the TidyFS client library, which is used for data ingress and egress, provides the option for each write to be simultaneously written to multiple replicas.

The decision to allow clients to read and write data using native interfaces is a major difference between TidyFS and systems such as GFS and HDFS. Native data access has several advantages:

- It allows applications to perform I/O using whatever access patterns and compression schemes are most suitable, e.g., sequential or random reads of a flat file, or database queries on a SQL database part.

- It simplifies legacy applications that benefit from operating on files in a traditional file system.

- It avoids an extra layer of indirection through TidyFS interfaces, guaranteeing that clients can achieve the maximum available I/O performance of the native system.

- It allows TidyFS to exploit native access-control mechanisms by simply setting the appropriate ACLs on parts, since client processes operate on behalf of authenticated users.

- It gives clients precise control over the size and contents of a part so clients can, for example, write streams with arbitrary partitioning schemes. Pre-partitioning of streams can lead to substantial efficiencies in subsequent computations such as database Joins that combine two streams partitioned using the same key.

The major disadvantage would appear to be a "loss of control" on the part of the file system designer over how a client may access the data, however our experience is that this, while terrifying to many file system designers, is not in practice a substantial issue for the workloads that we target. The major simplification that we exploit is that data are immutable once written and invisible to readers until committed. The file system therefore does not need to mediate between concurrent writers or order read/write conflicts. The detection of corruption is also simplified because data fingerprints are stored by TidyFS. A malicious client is unable to do more than commit corrupted data, or (access-controls permitting) delete or corrupt existing data. In both these cases the corruption will be discovered eventually when the fingerprint mismatch is detected and the data will be recovered from another replica or discarded if no good replicas are available. This is no worse than any other file system: if a client has write access to a file it can be expected to be able to destroy the data in that file.

Systems such as HDFS and GFS perform eager replication in their client libraries. Although the TidyFS client library provides optional eager replication for data ingress, TidyFS gains simplicity and performance in the common case by making lazy replication the default. The potential loss of data from lazy replication is justifiable because of the underlying fault tolerance of the client computational model: a failure of lazy replication can be treated just like a failure of the computation that produced the original part, and re-run accordingly. This is even true for workloads such as log processing, which are often implemented as a batch process with the input being staged before loading into a distributed file system. We believe this reliance on end-to-end fault tolerance is a better choice than implementing fault tolerance at multiple layers as long as failures are uncommon: we optimize the common case and in exchange require more work in error cases.

A drawback to giving clients control over part sizes is that some may end up much larger than others, which can complicate replication and rebalancing algorithms. Both GFS and HDFS try to split streams into parts of moderate sizes, e.g., around 100 MBytes. In principle a system that controls part boundaries could split or merge parts to improve efficiency, and this is not supported by TidyFS since each part is opaque to the system. In our experience, however, such rebalancing decisions are best made with semantic knowledge of the contents of the stream, and we can (and do) write programs using our distributed computational infrastructure to defragment streams with many small parts as necessary.

The biggest potential benefit, given our current workloads, that we can see from interposing TidyFS interfaces for I/O would come if *all* disk accesses on the compute nodes were mediated by TidyFS. This would potentially allow better performance using load scheduling, and would simplify the allocation of disk-space quotas to prevent clients from denying service to other cluster users by writing arbitrary sized files. We discuss some pros and cons of this direction in the final section.

Of course, the major tradeoff we make from the simplicity of TidyFS is a lack of generality. Clients of GFS use multi-writer appends and other features missing in TidyFS to implement services that would be very inefficient on our system. Again, we address this tradeoff in the Discussion.

2.3 SQL database parts

As mentioned in section 2.1, we have implemented support for two types of TidyFS parts: NTFS files and SQL databases. In the case of SQL parts, each part is a Microsoft SQL server database, consisting of both the database file and the database log file. The TidyFS metadata server stores the type of each part, and this information is used by the node service so that it will replicate both files associated with the part. The HadoopDB evaluation [1] shows that for some data-warehouse applications it is possible to achieve substantial performance gains by storing records in a database rather than relying on flat files. We believe that the ease of supporting SQL parts in TidyFS, compared with the additional mechanisms required to retrofit HadoopDB's storage to HDFS, provides support for our design choice to adopt native interfaces for reading and writing parts. As a bonus we achieve automatic replication of read-only database parts. There remains the problem of targeting these partitioned databases from client code, however, DryadLINQ [28] can leverage .NET's LINQ-to-SQL provider to operate in a hybrid mode shipping computation to the database where possible, as described in the referenced paper.

3 System architecture

The TidyFS storage system is composed of three components: a metadata server; a node service that performs housekeeping tasks running on each cluster computer that stores data; and the TidyFS Explorer, a graphical user interface which allows users to view the state of the system. The current implementation of the metadata server is 9,700 lines of C++ code, the client library is 5,000 lines of mixed C# and C++ code, the node service is 948 lines of C# code, and the TidyFS Explorer is 1,800 lines of C#. Figure 1 presents a diagram of the system architecture, along with a sample cluster configuration and stream. Cluster computers that are used for TidyFS storage are referred to in the following text as "storage computers."

3.1 Metadata server

The metadata server is the most complex component in the system and is responsible for storing the mapping of stream names to sequences of parts, the per-stream replication factor, the location of each part replica, and the state of each storage computer in the system, among other information. Due to its central role, the reliability of the overall system is closely coupled to the reliability of the metadata server. As a result, we implemented the metadata server as a replicated component. We leverage the Autopilot Replicated State Library [15] to replicate the metadata and operations on that metadata using the Paxos [18] algorithm. Following the design of systems such as GFS, there is no explicit directory tree maintained as part of the file system. The names of the streams in the system, which are URIs, create an implied directory tree based on the arcs in their paths. When a stream is created in the system, any missing directory entries are implicitly created. Once the last stream in a directory is removed, that directory is automatically removed. If the parent directory of that directory is now empty, it is also removed, and the process continues recursively up the directory hierarchy until a non-empty directory is encountered.

The metadata server tracks the state of all of the storage computers currently in the system. For each computer the server maintains the computer's state, the amount of free storage space available on that computer, the list of parts stored on that computer, and the list of parts pending replication to that computer. Each computer can be in one of four states: `ReadWrite`, the common state, `ReadOnly`, `Distress`, or `Unavailable`. When a computer transitions between these states, action is taken on either the list of pending replicas, the list of parts stored on that computer, or both. If a computer transitions from `ReadWrite` to `ReadOnly`, its pending replicas are reassigned to other computers that are in the `ReadWrite` state. If a computer transitions to the `Distress` state, then all parts, including any which are pending, are reassigned to other computers that are in the `ReadWrite` state. The `Unavailable` state is similar to the `Distress` state, however in the `Distress` state, parts may be read from the distressed computer while creating additional replicas, while in the `Unavailable` state they cannot. The `Distress` state is used for a computer that is going to be removed from the system, e.g., for planned re-imaging, or for a computer whose disk is

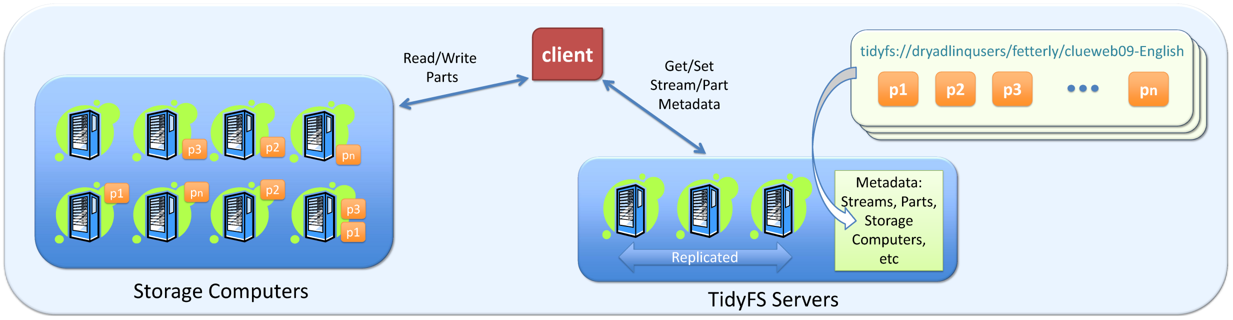
Read/Write Parts
client
Get/Set Stream/Part Metadata
tidyfs://dryadlinqusers/fetterly/clueweb09-English
p1
p2
p3
pn
Metadata: Streams, Parts, Storage Computers, etc
Replicated
Storage Computers
TidyFS Servers

Figure 1: TidyFS System Architecture

showing signs of imminent failure. The `Unavailable` state signifies that TidyFS should not use the computer at all.

In the current TidyFS implementation computers transition between states as a result of an administrator's command. We have found this manual intervention to be an acceptable burden for the clusters of a few hundred computers that we have been operating. A more widely deployed system should be able to automatically detect failure conditions and perform transitions out of the `ReadWrite` state without operator action. This could be implemented using well-known techniques such as the watchdogs employed in the Autopilot system [15].

The metadata server also maintains per-stream and per-part attributes. There is a set of "distinguished" attributes for each stream or part which are maintained by the system automatically or as a side-effect of system API calls, and users may add arbitrary additional attributes as key-value pairs to store client-specific information. For streams, the distinguished values are creation time, last use time, content fingerprint, replication factor, lease time, and length. For parts, the distinguished values are size and fingerprint.

Clients of the metadata server, including the other TidyFS components, communicate with the server via a client library. This client library includes RSL code that determines which metadata server replica to contact and will fail over in case of a server fault.

3.2 Node service

In any distributed file system there is a set of maintenance tasks that must be carried out on a routine basis. We implemented the routine maintenance tasks as a Windows service that runs continuously on each storage computer in the cluster. Each of the maintenance tasks is implemented as a function that is invoked at configurable time intervals. The simplest of these tasks is the periodic reporting, to the metadata server, of the amount of free space on the storage computer disk drives. The other tasks are garbage collection, part replication, and part validation, which are described in the following paragraphs.

Due to the separation of metadata and data in TidyFS and similar systems, there are many operations that are initially carried out on the metadata server that trigger actions on the storage computers in the system. The deletion of streams, via either user action or lease expiration, is one such operation. Once all of the streams that reference a particular part are deleted, every replica of that part should be removed from the storage computer that holds it. The metadata server is responsible for ensuring that there are sufficient replicas of each part, as calculated from the maximum replication factor of all streams the part belongs to. Once the replicas have been assigned to particular computers, the node services are responsible for actually replicating the parts.

In order to determine what parts should be stored on a storage computer, each node service periodically contacts the metadata server to get two lists of parts: the first is the list of parts that the server believes should be stored on the computer; and the second is the list of parts that should be replicated onto the computer but have not yet been copied.

When the node service processes the list of parts that the metadata server believes should be stored on the computer, the list may differ from the actual list of parts on the disk in two cases. The first is that the metadata server believes a part should be there but it is not. This case is always an error, and will cause the node service to inform the metadata server of the discrepancy, which in turn will cause the metadata server to set in motion the creation of new replicas of the part if necessary. The second is that the metadata service believes a part that is present on the disk should not be there. In this (more common) case the part id is appended to a list of candidates for deletion. Once the entire list is processed, the list of deletion candidates is sent to the metadata server, which filters the list and returns a list of part ids approved for deletion. The node service then deletes the files corresponding to the filtered list of part ids. The complete function pseudocode is listed in Algorithm 1, where `ListPartsAtNode`, `RemovePartReplica`, and `FilterPendingDeletionList` are all calls that contact the metadata server.

The reason for this two phase deletion protocol is to prevent parts that are in the process of being written from being deleted. The metadata server is aware of the part ids that have been allocated to a stream but not yet committed, as outlined in Section 2.2, however these pending part ids are not included in the list of part ids stored on any storage computer.

Algorithm 1 Garbage collection function

```
partIds = ListPartsAtNode();
filenames = ListFilesInDataDir();
List pdList;
for all file in filenames do
   id = GetPartIdFromFileName(file);
   if !partids.Remove(id) then
      pdList.Add(id);
   end if
end for
for all partId in partids do
   RemovePartReplica(partId);
end for
partIdsToDelete = FilterPendingDeletionList(pdList);
for all partId in partIdsToDelete do
   DeletePart(partId);
end for
```

If the list of parts that should be replicated to the node but are not present is non-empty, the node service contacts the metadata server for each listed part id to obtain the paths to read from and write to for replicating the part. Once the part has been replicated the fingerprint of the part is validated to ensure it was correctly copied, and the node service informs the metadata server that it has successfully replicated the part, after which the part will be present in the list of ids that the metadata believes are stored at that computer.

As we will show in Section 4, there is a substantial fraction of parts that are not frequently read. Latent sector errors are a concern for the designers of any reliable data storage system [5]. These errors are undetected errors where the data in a disk sector gets corrupted and will be unable to be read. If this undetected error were to happen in conjunction with computer failures, the system could experience data loss of some parts. As a result, the node service periodically reads each part replica and validates that its fingerprint matches the stored fingerprint at the metadata server; if not, the node service informs the metadata server that the part is no longer available on that computer, potentially triggering a re-replication.

3.3 TidyFS Explorer

The two TidyFS components already described deal with the correct operation of the system. The final component is the graphical user interface for the distributed file system, named the TidyFS Explorer. It is the primary mechanism for users and administrators to interact with the system. Like all other TidyFS clients, the TidyFS Explorer communicates with the metadata server via the client library. For users, TidyFS Explorer provides a visualization of the directory hierarchy implied by the streams in the system. In addition to the directory hierarchy, the TidyFS Explorer exposes the sequence of parts that comprise a stream, along with relevant information about those parts. Users can use the GUI to delete streams, rename streams, manipulate the sequence of parts in a stream, as well as copy parts between streams. Cluster administrators can use the TidyFS Explorer to monitor the state of computers in the system, including determining what computers are healthy, what replications are pending, and how much storage space is available. Administrators can also manually change the state of computers in the system and interact with the node service.

3.4 Replica Placement

When choosing where to place replicas for each part, we would like the system to optimize two separate criteria. First, it is desirable for the replicas of the parts in a particular stream to be spread across the available computers as widely as possible, which allows many computers to perform local disk reads in parallel when processing that stream. Second, storage space used should be roughly balanced across computers. Figure 2 shows a histogram of part sizes in a cluster running TidyFS. Due to this non-uniform distribution of part sizes, assigning part to replicas is not as simple as assigning roughly equal numbers of parts to each computer.

The location of the data for the first copy of a part is determined by the identity and state of the computer that is writing the part. We would like to bias writes to be local as often as possible, so we simply use the local computer if it is “legal” to write there, meaning that the computer is a storage computer in the `ReadWrite` state that has enough space available. We do not know the length of the part before the data is written, so we make a conservative estimate based on typical usage. If the disk fills up, the writer will fail and the computational framework’s fault-tolerance will reschedule the computation elsewhere.

Subsequent replication of parts should optimize the two criteria above. We have avoided complex balancing algorithms that directly optimize our desired criterion in favor of simpler greedy algorithms.

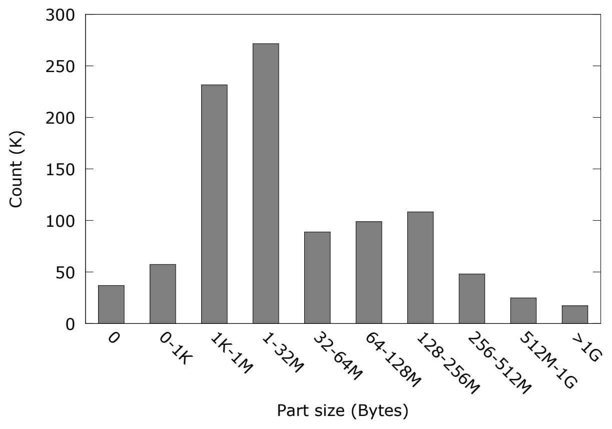

Figure 2: Histogram of part sizes (in MB)

We initially implemented a policy that assigns a replica to the legal computer in the system with most free space. Unfortunately many streams contain very small parts, and after adding one such small part to a given computer it often continues to be the one with the most free space. Since many parts from a stream are replicated in a short time period when the stream is created, this problem resulted in very poor balance for streams with small parts: one computer would hold the second copy of many or all of that stream's parts.

This lack of balance led us to implement a second policy, similar to the one used in the Kinesis project [19] and techniques described in [21]. This approach uses the part identifier to seed a pseudo-random number generator, then chooses three legal computers using numbers drawn from this pseudo-random number generator and selects the computer with the most free space from this set as the destination for the part. As we report in Section 4.4, this second policy results in acceptable balance for streams.

Over time the system may become unbalanced with some computers storing substantially more data than others. We therefore implemented a rebalancing command. This simply picks parts at random from overloaded computers and uses the replication policy to select alternate locations for those parts. The metadata server schedules the replication to the new location, and when the node service reports that the copy has been made, the metadata server schedules deletion from the overloaded computer.

3.5 Watchdog

We recently started to prototype a set of watchdog services to automatically detect issues and report them to an administrator for manual correction. The issues fall into two categories: error conditions, such as the failure to replicate parts after an acceptable period; and alert conditions, such as the cluster becoming close to its storage limit or computers becoming unresponsive. In the month or so that the watchdogs have so far been deployed they have reported two alerts and no errors. When we have more confidence in the watchdogs we may integrate them into an automatic failure mitigation system to reduce the cluster management overhead.

```
void AddStorageComputer(string
    storageComputerName, ulong freeSpace,
    string managedDirectory, string
    uncPath, string httpPath, int
    networkId);

bool DeleteStorageComputer(string
    storageComputerName);

StorageComputerInfo
    GetStorageComputerInformation(string
    storageComputerName);

void SetFreeSpace(string
    storageComputerName, ulong freeSpace);

void SetStorageComputerState(string
    storageComputerName,
    StorageComputerState
    storageComputerState);

string[] ListStorageComputers();

ulong[] ListPartsAtStorageComputer(string
     storageComputerName);

ulong[] GetPartReplicaList(string
    storageComputerName);
```

Figure 3: TidyFS API - Operations involving storage computers

3.6 API

For completeness we list the TidyFS API here. Most operations involve storage computers, streams and parts. As previously described, these operations are used by other TidyFS components (node service and TidyFS Explorer) and by external applications that wish to read and write TidyFS data.

Figure 3 includes a representative set of operations involving storage computers. These include commands, typically used by cluster administrators, for adding, modifying and removing computers. `SetFreeSpace` and `SetStorageComputerState` are useful for updating the state of the cluster, and are used both by admin-

```
string[] ListDirectories(string path);

string[] ListStreams(string path);

void CreateStream(string streamName);

ulong CreateStream(string streamName,
    DateTime leaseTime, int numParts);

void CopyStream(string srcStreamName,
    string destStreamName);

void RenameStream(string srcStreamName,
    string destStreamName);

bool DeleteStream(string streamName);

void ConcatenateStreams(string
    destStreamName, string srcStreamName);

void AddPartToStream(ulong[] partIds,
    string streamName, int position);

void RemovePartFromStream(ulong partId,
    string streamName);

ulong[] ListPartsInStream(string
    streamName);

PartInfo[] GetPartInfoList(string
    streamName);

DateTime GetLease(string streamName);

void SetLease(string streamName, DateTime
     lease);

ulong RequestPartIds(string streamName,
    uint numIds);

byte[] GetStreamBlobAttribute(string
    streamName, string attrName);

void SetStreamAttribute(string streamName,
     string attrName, byte[] attrValue);

void RemoveStreamAttribute(string
    streamName, string attrName);

string[] ListStreamAttributes(string
    streamName);
```

Figure 4: TidyFS API - Operations involving streams

```
void AddPartInfo(PartInfo[] pis);

void AddPartReplica(ulong partId, string
    nodeName);

void RemovePartReplica(ulong partId,
    string nodeName);

void GetReadPaths(ulong partId, string
    nodeName, out StringCollection paths);

void GetWritePaths(ulong partId, string
    nodeName, out StringCollection paths);
```

Figure 5: TidyFS API- Operations involving parts

istrators and by the node service. Operations that allow listing all computers and all parts at a computer are also provided for diagnostic purposes and are used by the TidyFS Explorer. `GetPartReplicaList` is a command for listing all new replicas that have been assigned by TidyFS to a specific storage computer, but that have not yet been created. This call is invoked periodically by the node service running on a storage computer to fetch the list of parts that it needs to fetch and replicate.

Figure 4 lists operations involving streams. Due to space restrictions we have only included the most often used ones and have omitted some operations that are similar to others already shown. For example, while we only show operations for getting and setting stream attributes of blob data type, similar commands exist for attributes of different types. The figure includes operations for listing the contents (both subdirectories and streams) of directories, and stream manipulation commands including operations for adding and removing parts from a stream. Parts may be added at any position in the stream, and streams may be concatenated, which causes all parts from one stream to be appended to another.

Finally, in Figure 5 the remaining operations involving parts are listed. The `AddPartInfo` command is used by clients to inform TidyFS that a part has been fully written, and to pass information such as the part size and fingerprint. Operations for adding and removing replicas from a storage computer (`AddPartReplica` and `RemovePartReplica`) are used by the node service to inform the metadata server when replicas have been created at a particular computer. Other important operations include `GetReadPaths` and `GetWritePaths`, which return a list of paths where a part may be read from, or written to. These are used by clients prior to reading or writing any data to TidyFS. There are also operations for getting and setting part attributes of various data types that are similar to those provided for streams and are omitted here for brevity.

The RSL state replication library allows some API calls to execute as "fast reads" which can run on any replica using its current state snapshot and do not require a round of the consensus algorithm. Fast reads can in principle reflect quite stale information, and are not serializable with other state machine commands. Given our read-mostly workload, by far the most common API calls are those fetching information about existing streams, such as listing parts in a stream or read paths for a part. We therefore allow these to be performed as fast reads and use all replicas to service these reads, reducing the bottleneck load on the metadata server. As we report in Section 4 most reads are from streams that have existed for a substantial period, so staleness is not a problem. If a fast read reports stale data such as a part location that is out of date following a replication, the client simply retries using the slow-path version that guarantees an up to date response.

4 Evaluation and experience

TidyFS has been deployed and actively used for the past year on a research cluster with 256 servers, where dozens of users run large-scale data-intensive computations. The cluster is used exclusively for programs run using the DryadLINQ [28] system. DryadLINQ is a parallelizing compiler for .NET programs that executes programs using Dryad [16]. Dryad is a coarse-grain dataflow execution engine that represents computations as directed-acyclic graphs of processes communicating via data channels. It uses TidyFS for storage of input and output data, following the access patterns set out in Section 2.2. Fault-tolerance is provided by re-execution of failed or slow processes. Dryad processes are scheduled by a cluster-wide scheduler called Quincy [17] that takes into account the size and location of inputs for each process when choosing which computer to run the process on. The cluster hardware is as described in our Quincy paper [17].

Dryad queries TidyFS for the locations of its input parts and passes this information to Quincy so that most processes end up scheduled close in network topology to at least one replica of their input parts. DryadLINQ makes use of TidyFS attributes to store type information about the records in output streams and to record partitioning information—both of these types of attribute are used by subsequent DryadLINQ programs that consume the streams as inputs. Our cluster infrastructure also include Nectar [12] which is a cache manager for computations. Nectar and DryadLINQ communicate to determine sub-computations whose results have already been stored to TidyFS, to save unnecessary recomputation of complex quantities. Nectar makes use of TidyFS stream and part fingerprints to determine when the inputs of two computations are identical.

TidyFS was designed in conjunction with the other cluster components listed above, so it naturally has APIs and performance that is well suited to their requirements. As noted in Section 2.2 we believe these requirements are shared by other systems including MapReduce and Hadoop. At the end of the paper we include more discussion of the wider applicability of TidyFS.

4.1 Data volume

On a typical day, several terabytes of data are read and written to TidyFS through the execution of DryadLINQ programs. We collected overall statistics on the usage of the system through logs maintained by TidyFS.

Figures 6 and 7 present daily read and write loads, and volumes of data deleted, on TidyFS during a sample two-week period. The purpose of these figures is to give the reader a sense of the scale of system usage. The amount of data read and written on a specific day varies over the days mostly because of the diverse nature of jobs being run on the cluster, and on other factors such as day of the week.

As earlier described, Quincy attempts to place processes close to the data they will read, and when satisfying clients' requests for read paths TidyFS prioritizes copies of the data that are stored locally at the computer where the client is running, followed by copies stored within the same rack, and finally cross racks. The default replication factor in the cluster is two, so there are generally two replicas per part from which data can be read. Figure 6 classifies reads as local, within rack, cross rack or remote. Remote reads refer to reads where the client is outside the compute cluster. As expected, the majority of reads are local, followed by reads that occur within the same rack, indicating that the goal of moving computation close to the data is very often achieved.

DryadLINQ does not perform any eager replication, so each part is written once by DryadLINQ and subsequently replicated lazily by TidyFS. Figure 7 shows the amount of data committed by DryadLINQ per day during the sample period. The vast majority of these writes are local, and there is an equivalent amount of data written during lazy replication since the default replication count is two. The volume of data deleted shown in the figure corresponds again to the volume of primary part data: due to the replication factor of two, the space freed across the cluster disks is actually twice as large.

4.2 Access patterns

More insight can be gained into cluster usage by studying how soon data is read after it has been initially written.

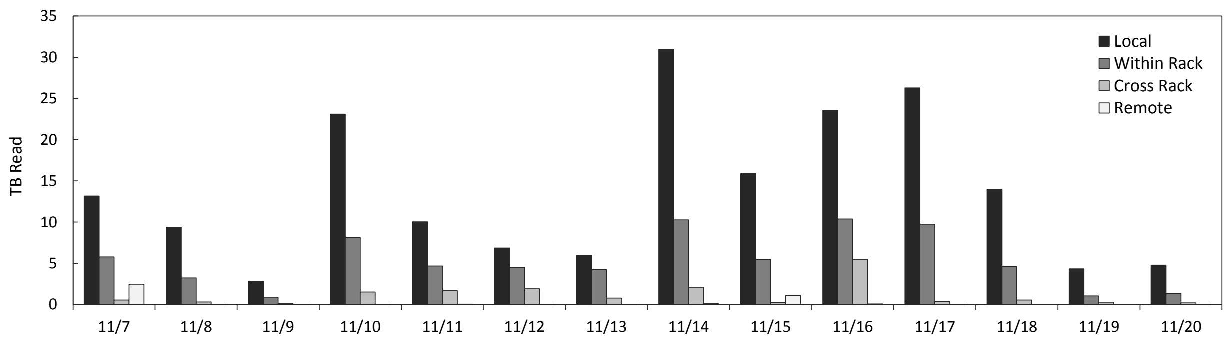

Figure 6: Terabytes of data read per day, grouped by local, within rack, cross-rack, and remote reads.

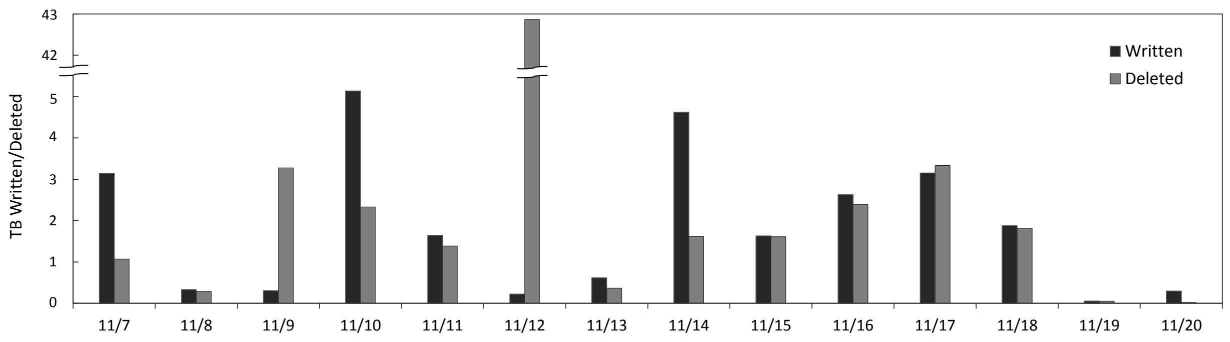

Figure 7: Terabytes of primary part data written and deleted per day.

We computed, for every read that happened over a period of three months, the age of the read as the time interval between the time the read occurred and the time the part being read was originally written. Figure 8 presents the cumulative distribution of data read as the read age increases. As shown in the figure only a small percentage of data is read within the first five minutes after it has been created. Almost 40% of the data is read within the first day after the data being read has been created, and around 80% of the data is read within one month of creation.

Given the small percentage of reads that occur within the first minute of writing a part, the node service's periodic task of checking for pending replicas is configured to run every minute. This implies a delay of up to a minute before lazy replication of a part begins, and reads that occur in smaller windows of time will have fewer choices of where to read from.

This effect can be observed in Figure 9, which shows the proportion of local, within rack and cross rack data out of all reads that happen for different read ages. As observed, most reads that happen within the first minute after a part is written are remote reads, since many of those parts would only have one copy at that time. However, as observed from Figure 8 the total number of reads that happen at that time interval is very small relative to all reads. For part reads that occur after longer periods of time since the part's creation, local and within rack reads predominate.

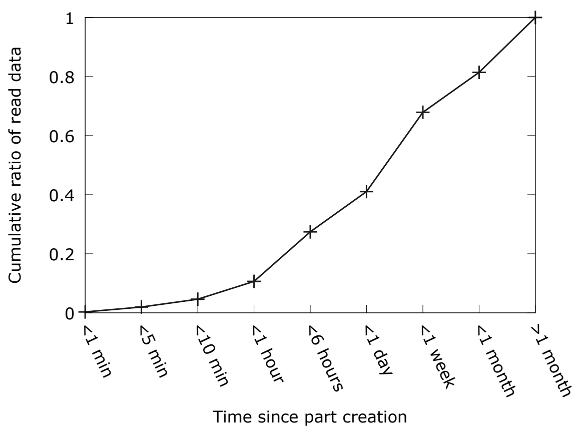

Figure 8: Cumulative distribution of read ages (time when read occurs - time when data was originally written) for reads occurring over a period of three months.

To further characterize our cluster's usage pattern we analyzed the relationships in timing and frequency between reads and writes of the same part. In Figure 10 we show how often parts are read once they have been written. Once again, we analyzed data over a period of three

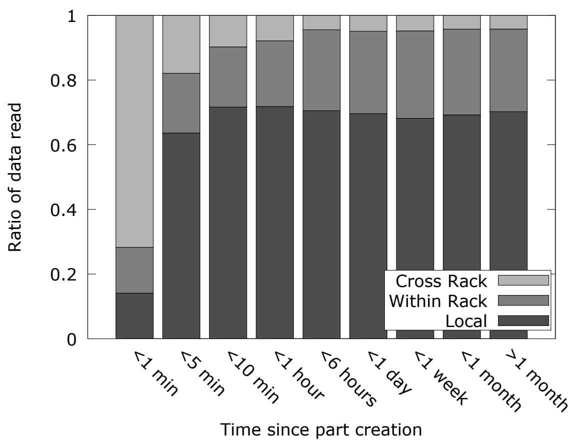

Figure 9: Proportion of local, within rack and cross rack data read grouped by age of reads.

months, considering all writes and subsequent reads that happened in the period. For each part written we counted the number of times the part was subsequently read. As observed from the graph, many parts are read only once or a small number of times. There is also a large number of parts which are never read again by future jobs.

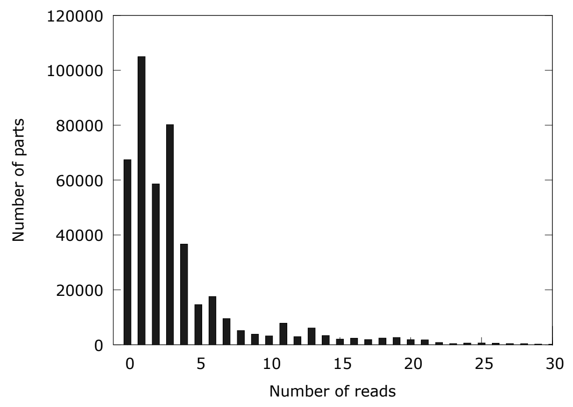

Figure 10: Number of times a part is read after it has been originally written.

Finally, in Figure 11 we present results on the last access time of parts. For every part in the system, we identify the last time it was read, up to a maximum of sixty days, and plot the cumulative ratio of parts as a function of their last read time. Approximately thirty percent of the parts had not been read in the period of sixty days, and read ages are evenly distributed over the sixty day period.

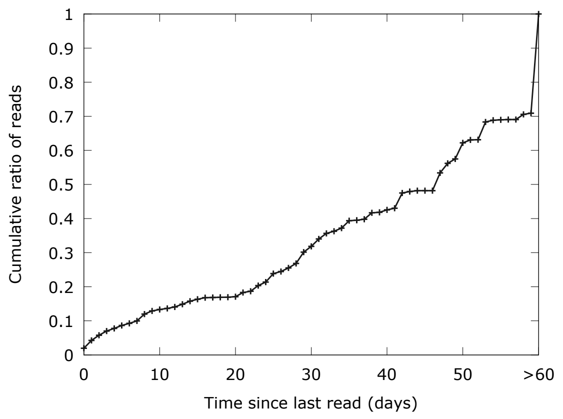

Figure 11: Cumulative distribution of parts over time since parts' last access.

4.3 Lazy versus Eager Replication

In order to evaluate the effectiveness of lazy replication, we gathered statistics about the average time before parts are replicated. Table 1 shows the mean time between a part being added to the system and the creation of a replica for that part over a three month time window. Nearly 70% of parts have a replica created within 60 seconds and 84% within 2 minutes. 96% of parts have a replica within one hour. Parts that are not replicated within one hour are due to the node service on the storage computer where the replica has been scheduled being disabled for system maintenance. The data in these parts is still available from the original storage computer. Therefore, we find that lazy replication provides acceptable performance for clusters of a few hundred computers. We have been experiencing around one unrecoverable computer failure per month, and have not so far lost any unreplicated data as a consequence.

Mean time to replication (s)	Percent
0 - 30	6.7%
30 - 60	62.9%
60 - 120	14.6%
120 - 300	1.1%
300 - 600	2.2%
600 - 1200	4.5%
1200 - 3600	3.4%
3600 -	4.5%

Table 1: Mean time to replication over a three month time interval.

4.4 Replica Placement and Load Balancing

As described in Section 3.4, we would like TidyFS to assign replicas to storage computers using a policy that balances the spread of parts per stream across computers as well as the total free space available at storage computers. We compare the two policies we implemented: the initial space-based policy that led to poorly-balanced streams, especially for those streams with many small parts; and the subsequent best-of-three random choice policy.

We define a load-balancing coefficient for each stream by calculating the L^2 distance between a vector representing the number of parts from a particular stream assigned to a specific storage computer and the perfectly-balanced mean vector. The coefficient is computed as follows: $\sqrt{\sum_{i=1}^{n}(p_i - \frac{rp}{n})^2}$ where r is the stream replication factor, p is the number of parts in the stream, p_i is the number of part replicas stored at node i, and n is the number of computers in the `ReadWrite` state in the cluster. We normalize so that a coefficient of 1 corresponds to the worst-case situation where just r computers are used to store all the parts, which leads to the following complete equation:

$$\frac{1}{\sqrt{r(p - \frac{rp}{n})^2 + (n-r)(\frac{rp}{n})^2}} \sqrt{\sum_{i=1}^{n}(p_i - \frac{rp}{n})^2} \quad (1)$$

The load-balancing behavior was analyzed over two periods of time: in the first one, the space-based policy was used; in the second one, the randomized policy. We computed, at the end of each day, the load balancing coefficient of each stream, as given by Equation 1, and the overall average over all streams. Figure 12 presents the average coefficient for each day over these two periods. As shown in the figure, streams were significantly better balanced during the second period when the randomized policy was being used.

5 Related Work

The TidyFS design shares many characteristics with other distributed storage systems targeted for data-intensive parallel computations, but wherever possible simplifies or removes features to improve the overall performance of the system without limiting its functionality for its target workload. While several systems only focus on delivering aggregate performance to a large number of clients, one of the main goals with TidyFS is to also encourage moving computation close to the data by providing interfaces that allow applications to locate all replicas of parts.

Like several distributed storage systems such as Frangipani [26], GPFS [23], GFS [11], HDFS [6, 24],

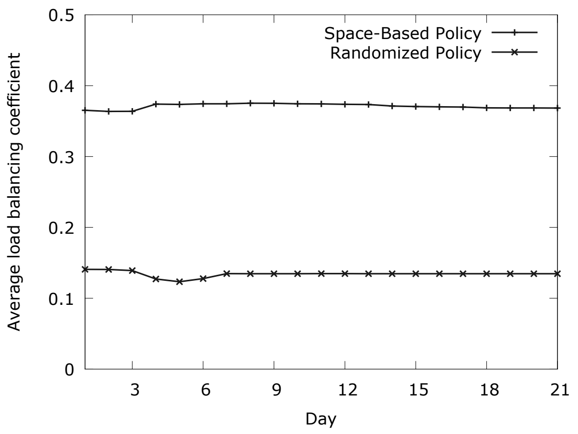

Figure 12: Load balancing coefficient when using space-based policy against randomized policy for replica placement.

PanasasFS [27] and Ursa Minor [25], TidyFS separates metadata management from data storage, allowing data to be transparently moved or replicated without client knowledge. Some of these systems do not maintain the metadata in centralized servers and support complex read and write semantics, either by relying on distributed locking algorithms (Frangipani, GPFS), or by migrating metadata prior to executing metadata operations (Ursa Minor).

TidyFS is most similar to GFS [11] and HDFS [6, 24]. It follows their centralized metadata server design and focuses on workloads where data is written once and read multiple times, which simplifies the needed coherency model. Despite the similarities with GFS and HDFS, TidyFS's design differs from these systems in important ways. The most significant difference is that TidyFS uses native interfaces to read and write data while GFS and HDFS both supply their own data access APIs. This design choice leads to related differences in, for example, replication strategies and part-size distribution.

TidyFS also differs from GFS in how it achieves resilience against metadata server crashes. GFS maintains checkpoints and an operation log to which metadata operations should be persisted before changes become visible to clients. TidyFS instead maintains multiple metadata servers, all of which keep track of all the metadata in the system, and uses the Paxos [18] algorithm to maintain consistency across the servers. A similar approach is used in BoomFS [2], a system similar to HDFS built from declarative language specifications.

TidyFS's ability to handle database parts enables a hybrid large-scale data analysis approach that exploits the performance benefits of database systems, similarly to the approach taken for HadoopDB [1]. HadoopDB combines MapReduce style computations with database systems to achieve the benefits of both approaches, although the databases accessed by HadoopDB are not stored in HDFS or a replicated file system. The MapReduce framework is used to parallelize queries, which are then executed on multiple single-node database systems. Database parts stored in TidyFS can be queried using Dryad and DryadLINQ in similar ways.

6 Discussion

TidyFS is designed to support workloads very much like those generated by MapReduce and Hadoop. It is thus natural to compare TidyFS to GFS and HDFS, the file systems most commonly used by MapReduce and Hadoop respectively. The most consequential difference is the decision for TidyFS to given clients direct access to part data using native interfaces. Our experience of the resulting simplicity and performance, as well as the ease of supporting multiple part types such as SQL database, has validated our assumption that this was a sensible design choice for the target Dryad workload. The main drawback is a loss of generality. Other systems built on GFS, such as BigTable [8], use the ability to persist small appends and make them visible to other clients in order to achieve performance and reliability, and the desire to support appends in HDFS is related to the desire to implement similar services such as HBASE on top of that file system [10]. A key point in [20] is that the GFS semantics were not a good fit for all of the applications rapidly built on GFS. Some issues that are described in [20], such as the small file problem, can be addressed in client libraries. Other issues, including inconsistent data returned to the client depending on which replica was read and latency issues because GFS was designed for high-throughput, not low-latency, cannot be addressed in client libraries. We believe however that rather than complicating the common use case of a data-intensive parallel file system it makes more sense to add a separate service for reliable logging or distributed queues. This was done for example in River [4] and the Amazon Web Service [3] and would be our choice if we needed to add such functionality to our cluster.

Another feature that TidyFS lacks as a result of our choice of native interfaces is automatic eager replication, with the exception of optional eager replication in the data ingress case. Again we are happy with this tradeoff. In the year we have been operating TidyFS we have not had a single part lost before replication has completed. Clearly this is primarily because of the relatively small size of our deployment, however it suggests that leveraging the client's existing fault-tolerance to replace lost data is a reasonable alternative to eager replication, despite the additional work spent in the rare failure cases.

The final major difference is our lack of control over part sizes. DryadLINQ programs frequently make use of the ability to output streams with exact, known partitioning, which leads to sometimes significant performance improvements. However we do also have to deal with problems caused by occasional parts which are very much larger than the average. This caused problems with our original simple replication policy that fortunately were easy to fix with the slightly more sophisticated best-of-three random policy. We believe that the existence of very large parts also adds to disk fragmentation across our cluster. If ignored, we have found that this fragmentation results in devastating performance penalties as parts are split into thousands of fragments or more, preventing the sequential I/O that is necessary for high read throughput. We have recently started to aggressively defragment all disks in the cluster to mitigate this problem.

While we motivate the TidyFS design using general properties of data intensive shared-nothing workloads, in practice it is currently used almost exclusively by applications executing using Dryad. Rather than making TidyFS more general, one direction we are considering is integrating it more tightly with our other cluster services. If all I/O on the cluster were reads or writes from TidyFS, Dryad intermediate data shuffling, and TidyFS replication traffic, then substantial performance benefits might accrue from integrating I/O into the Quincy scheduling framework, and possibly even adopting circuit-switched networking hardware to take advantage of these known flows. As mentioned in Section 2.2 this tighter integration might conflict with the choice to allow clients unfettered access to native I/O. On the other hand if the only client were Dryad, which is trusted to obey scheduling decisions, the benefits of I/O scheduling might still be achieved. Tighter integration with Dryad would also let us revisit a design alternative we had originally considered, which is to eliminate the node service altogether and perform all housekeeping tasks using Dryad programs. We abandoned this potentially simplifying approach primarily because of the difficulty of ensuring that Dryad would run housekeeping tasks on specific computers in a timely fashion with its current fairness and locality policies.

We have recently reimplemented the metadata server on top of a replicated SQL database instead of the C++ and RSL implementation described in this paper. This radically reduces the number of lines of novel code in the system by relying on the extensive but mature SQL Server codebase. Our main concern is whether comparable performance can be easily attained using SQL, which is unable to perform fast reads as described in Section 3.6 without the addition of a custom caching layer, and we are currently evaluating this tradeoff.

Overall we are pleased with the performance, simplicity and maintainability of TidyFS. By concentrating on a single workload that generates a very large amount of I/O traffic we were able to revisit design decisions made by a succession of previous file systems. The resulting simplifications have made the code easier to write, debug, and maintain.

Acknowledgments

We would like to thank Mihai Budiu, Jon Currey, Rebecca Isaacs, Frank McSherry, Marc Najork, Chandu Thekkath, and Yuan Yu for their invaluable feedback, both on this paper and for many helpful discussions about the TidyFS system design. We would also like to thank the anonymous reviewers and our shepherd, Wilson Hsieh, for their helpful comments.

References

[1] ABOUZIED, A., BAJDA-PAWLIKOWSKI, K., ABADI, D. J., SILBERSCHATZ, A., AND RASIN, A. HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads. In *Proceedings of the 35th Conference on Very Large Data Bases (VLDB)* (Lyon, France, 2009).

[2] ALVARO, P., CONDIE, T., CONWAY, N., ELMELEEGY, K., HELLERSTEIN, J. M., AND SEARS, R. BOOM analytics: Exploring data-centric, declarative programming for the cloud. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys)* (Paris, France, 2010).

[3] Amazon Web Services. http://aws.amazon.com/.

[4] ARPACI-DUSSEAU, R. H., ANDERSON, E., TREUHAFT, N., CULLER, D. E., HELLERSTEIN, J. M., PATTERSON, D., AND YELICK, K. Cluster I/O with River: making the fast case common. In *Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems (IOPADS)* (Atlanta, GA, USA, 1999).

[5] BAIRAVASUNDARAM, L. N., GOODSON, G. R., PASUPATHY, S., AND SCHINDLER, J. An analysis of latent sector errors in disk drives. In *Proceedings of the Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)* (San Diego, CA, USA, 2007).

[6] BORTHAKUR, D. HDFS architecture. Tech. rep., Apache Software Foundation, 2008.

[7] BRODER, A. Some applications of rabin's fingerprinting method. In *Sequences II: Methods in Communications, Security, and Computer Science* (1993), Springer Verlag, pp. 143–152.

[8] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI))* (Seattle, WA, USA, 2006).

[9] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)* (San Francisco, CA, USA, 2004).

[10] File appends in HDFS. http://www.cloudera.com/blog/2009/07/file-appends-in-hdfs/.

[11] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. In *Proceedings of the 19th Symposium on Operating Systems Principles (SOSP)* (Bolton Landing, NY, USA, 2003).

[12] GUNDA, P. K., RAVINDRANATH, L., THEKKATH, C. A., YU, Y., AND ZHUANG, L. Nectar: Automatic management of data and computation in data centers. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)* (Vancouver, Canada, 2010).

[13] Hadoop wiki. http://wiki.apache.org/hadoop/, April 2008.

[14] The HIVE project. http://hadoop.apache.org/hive/.

[15] ISARD, M. Autopilot: automatic data center management. *SIGOPS Oper. Syst. Rev. 41*, 2 (2007), 60–67.

[16] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of 2nd European Conference on Computer Systems (EuroSys)* (Lisbon, Portugal, 2007).

[17] ISARD, M., PRABHAKARAN, V., CURREY, J., WIEDER, U., TALWAR, K., AND GOLDBERG, A. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the 22nd Symposium on Operating Systems Principles (SOSP)* (Big Sky, MT, USA, 2009).

[18] LAMPORT, L. The part-time parliament. *ACM Trans. Comput. Syst. 16*, 2 (1998), 133–169.

[19] MACCORMICK, J., MURPHY, N., RAMASUBRAMANIAN, V., WIEDER, U., YANG, J., AND ZHOU, L. Kinesis: A new approach to replica placement in distributed storage systems. *ACM Trans. Storage 4*, 4 (2009), 1–28.

[20] MCKUSICK, M. K., AND QUINLAN, S. GFS: Evolution on fast-forward. *Queue 7* (August 2009), 10:10–10:20.

[21] MITZENMACHER, M., RICHA, A. W., AND SITARAMAN, R. The power of two random choices: A survey of techniques and results. In *Handbook of Randomized Computing* (2001), Kluwer, pp. 255–312.

[22] OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R., AND TOMKINS, A. Pig Latin: A not-so-foreign language for data processing. In *Proceedings of the 28th International Conference on Management of Data (SIGMOD)* (Vancouver, Canada, 2008).

[23] SCHMUCK, F., AND HASKIN, R. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the 1st Conference on File and Storage Technologies (FAST)* (Monterey, CA, 2002).

[24] SHVACHKO, K. V. HDFS scalability: The limits to growth. *;login 35*, 2 (April 2010).

[25] SINNAMOHIDEEN, S., SAMBASIVAN, R. R., HENDRICKS, J., LIU, L., AND GANGER, G. R. A transparently-scalable metadata service for the Ursa Minor storage system. In *Proceedings of the USENIX Annual Technical Conference (USENIX)* (Berkeley, CA, USA, 2010).

[26] THEKKATH, C. A., MANN, T., AND LEE, E. K. Frangipani: a scalable distributed file system. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP)* (Saint Malo, France, 1997).

[27] WELCH, B., UNANGST, M., ABBASI, Z., GIBSON, G., MUELLER, B., SMALL, J., ZELENKA, J., AND ZHOU, B. Scalable performance of the Panasas parallel file system. In *Proceedings of the 6th Conference on File and Storage Technologies (FAST)* (San Jose, CA, USA, 2008).

[28] YU, Y., ISARD, M., FETTERLY, D., BUDIU, M., ERLINGSSON, Ú., GUNDA, P. K., AND CURREY, J. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)* (San Diego, CA, USA, 2008).