TidyFS: A Simple and Small Distributed File System

Dennis Fetterly
Microsoft Research, Silicon Valley

Maya Haridasan
Microsoft Research, Silicon Valley

Michael Isard
Microsoft Research, Silicon Valley

Swaminathan Sundararaman
University of Wisconsin, Madison

Microsoft Research Technical Report
MSR-TR-2010-124

Abstract

In recent years, there has been an explosion of interest in computing using clusters of commodity, shared nothing computers. In this paper, we describe the design of TidyFS, a simple and small distributed file system that provides the abstractions necessary for data parallel computations on clusters. Similar to other large-scale distributed file systems such as the Google File System (GFS) and the Hadoop Distributed File System (HDFS), the prototypical workload for this file system is high-throughput, write-once, sequential I/O. The primary user visible unit of storage in this system is the stream, which is a sequence of partitions distributed across the local storage of machines in the cluster. The mapping of streams to sequences of partitions is performed by the TidyFS metadata server, which also tracks the locations of each of the partition replicas in the system, the state of each storage machine in the cluster, and per-stream and per-partition attributes. The metadata server is implemented as a state machine and replicated for scalability and fault tolerance. In addition to the metadata server, the system is comprised of a graphical user interface which enables users and administrators to view the state of the system and a small service installed on each cluster machine responsible for replication, validation, and garbage collection. Clients read and write partitions directly to get the best possible I/O performance.

1 Introduction

For more than the last decade, there has been substantial interest in building compute clusters out of commodity components. Programs to utilize these clusters are often written using a data-parallel framework such as map-reduce [8], Dryad [12], or one of the higher level abstractions layered on top of them such as PIG [15], HIVE [1], or DryadLINQ [17]. Programs written using these data-parallel frameworks typically have I/O access patterns that differ from traditional High Performance Computing (HPC) frameworks such as MPI [2]. As an example of the workload for these file systems, consider the load generated when running TeraSort [3] on a cluster of 240 compute nodes, each containing 4 SATA hard drives. When reading the input stream, each of the compute nodes will read from its local disks at 240 MB/s for a aggregate read rate of 56 GB/s. And once the input has been distributed and sorted, those 240 machines each writing at 160 MB/s will have an aggregate write rate of 37 GB/s.

The I/O subsystem in River [4], the Google File System (GFS) [9], and the Hadoop Distributed File System (HDFS) [6, 16] are distributed file systems that are designed for use in these commodity computing clusters with a prototypical workload consisting of write-once, high-throughput, sequential I/O. These systems typically store the metadata separate from the actual data. The storage component of the River system grouped single disk collections into par-

allel collections which could be mirrored onto multiple machines. The metadata describing these groupings is stored in NFS, and access to this metadata is serialized though a single instance of the application. Both GFS and HDFS store the file system metadata in a single node and the data in machines that are part of the cluster.

In this paper, we present the design of TidyFS, which is a simple distributed file system designed specifically for these prototypical workloads. Clients of this file system primarily operate on streams of data, typically by having multiple computations each independently process partitions of the stream data in parallel.

2 System Design

TidyFS stores data in streams, which are defined as a sequence of partitions. Partitions are stored as files on storage machines in the cluster. The sequence of partitions can be modified, and partitions can be removed from or added to a stream as necessary. These operations are just operations on metadata, and do not require modification of the underlying data. Partitions can be members of multiple streams. This, combined with the ability to modify the partition sequence for a stream, lends itself nicely to performing computations on sliding windows of data, e.g. the last seven days of log files.

A client may access data contained in a TidyFS stream by fetching the sequence of partition ids that comprise a particular stream, and then requesting a path to directly access the data associated with a particular partition id. In the case of writing the partition data, when writing is completed and the client is prepared to declare the partition immutable, the client will close the file and provide the size and fingerprint to the system. This design choice requires a trusted writer, since the writer provides checksum. However, since corrupt data will be discarded by the system, it is in the writers interest to provide a valid checksum.

Allowing applications to directly access the partition data has several advantages. First, it allows applications to perform I/O using whatever access pattern (i.e. sequential or random) and data compression technique that best fit their needs, removing TidyFS components from involvement in the actual reading or writing of data from disk. Furthermore, it provides applications the flexibility to operate on files that exist in a traditional file system. Of course, there are disadvantages to providing direct access to partitions in the system as well, such as a lack of portability gained via a level of indirection, and the ability of the system to split and merge partitions as necessary to obtain roughly uniform partition sizes. For our specific design point of a distributed file system optimized for data-parallel distributed computing, streams that are the result of a DryadLINQ computation also store metadata describing the schema of the data in the stream and providing the partitioning and compression information. Given this information, the system is able to split and merge partitions belonging to these streams as well as change the type of compression used by the stream. Partitions can be of multiple types; we have currently implemented partition types for NTFS files as well as SQL database files. Among other partition types, it would also be possible to have a partition be an immutable directory of files, where the directory is replicated as a unit, yet the files exist as individual entities in the file system so that they can be processed using legacy libraries without the entire contents of the directory being read by the application.

Our general approach is to design the set of systems (Dryad [12], DryadLINQ [17], and TidyFS) so that the end to end system is fault tolerant. As a result, each layered component does not need each service to be fault tolerant if fault tolerance can be achieved via a system at another level. To that end, each individual write to a TidyFS stream does not need to be immediately replicated to several machines for fault tolerance, as long as it is possible to ensure that each partition is replicated once it has been completely written. In the case of a TidyFS stream that is being written by a Dryad job, the fault tolerance is provided by the Dryad job manager. Dryad jobs either fail or complete successfully. In the case of job failure, no output stream is generated. In the case of successful completion, each vertex successfully writes its outputs generating a complete stream. This sim-

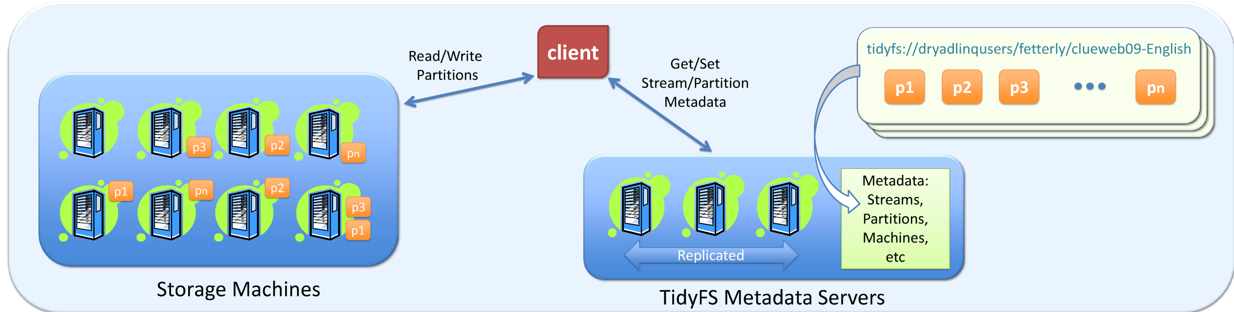

Figure 1: TidyFS System Architecture

plifies the TidyFS design: once the metadata server is informed that the partition is complete it schedules replication of this partition for fault tolerance. In order to ensure that data is safe against machine failure once the stream is written, it is possible for the Dryad job manager to wait until there are at least k replicas of each partition in an output stream before exiting.

The TidyFS storage system is composed of three components: a metadata server, a node service that runs on each storage machine in the system to perform required housekeeping tasks, and the TidyFS Explorer, a graphical user interface which allows users to view the state of the system. Figure 1 presents a diagram of the system architecture, along with a sample cluster configuration and stream.

2.1 Metadata server

The metadata server is the primary component in the system and is responsible for storing the mapping of stream names to sequences of partitions, the per-stream replication factor, the location of each partition replica, and the state of each storage machine 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 have implemented the metadata server as a replicated component. We leverage the Autopilot Replicated State Library [11] to replicate the metadata and operations on that metadata using the Paxos [13] algorithm. Similar to GFS, there is no actual 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 names. When a stream is created in the system, any missing directory entries are created. Once the last stream in a directory is removed, that directory is automatically removed.

The metadata server tracks the state of all of the storage machines currently in the system. For each machine, the metadata server maintains the machine's state, the amount of free storage space available on that machine, the list of partitions stored on that machine, as well as the list of partitions pending replication to that machine. Each machine can be in one of four states: `ReadWrite`, the common state, `ReadOnly`, `Distress`, or `Unavailable`. Machines transition between states as the result of an administrator's command. When a machine transitions between these states, action is taken on either the list of pending replicas, the list of partitions stored on that machine, or both. If a machine transitions from `ReadWrite` to `ReadOnly`, the pending replicas are reassigned to other machines that are in the `ReadWrite` state. If a machine transitions to the `Distress` state, then all partitions, including any which are pending, are reassigned to other machines that are in the `ReadWrite` state. The `Unavailable` state is similar to the `Distress` state, however in the `Distress` state, partitions may be read from the distressed machine while creating additional replicas, while in the `Unavailable` state they cannot.

The metadata server also maintains per-stream

and per-partition attributes, some of which are *distinguished* attributes which are present for each stream or partition. For streams, the distinguished values are creation time, last use time, content fingerprint, replication factor, lease time, and length. For partitions, the distinguished values are size and fingerprint. In addition to these lists, clients of the file system are able to specify per-stream and per-partition attributes as key-value pairs, where the key is a string and the value is a string, integer, or blob.

The fingerprints used for partitions and streams are 64-bit Rabin fingerprints [7], which have the nice property that the stream fingerprint can be computed using the partition fingerprints and lengths, without needing to consult the actual data. While the partition fingerprints are used by the system to ensure data integrity, the stream fingerprints are used by clients of the file system, such as the Nectar query caching system [10], which uses both the stream and partition fingerprints.

Stream leases are used to set a future expiration date for a stream. If the stream's expiration date is reached, then the stream is deleted and any partitions that are no longer referenced by another stream are removed from the system. The Dryad job manager utilizes leases to cleanup any partial output that remains after job failure. When users delete streams via the TidyFS Explorer, the stream is renamed to add "Recycle Bin" as the first arc in the pathname, and a lease is set for 6 hours, in order to allow users to recover streams they have accidentally deleted.

Clients of the file system, including the other TidyFS components, communicate with the metadata server via a client library. This client library is responsible for determining which metadata server replica to contact and will failover in case of a server fault.

2.2 Node service

In any distributed file system, there are 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 machine 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 of the amount of free space on the storage machine's disk drives. The other tasks are garbage collection, partition replication, and partition 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 need to be eventually carried out on the storage machines 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 partition are deleted, that partition can be deleted on each storage machine that stores a replica.

In order to determine what partitions should be stored on the local storage machine, each machine service periodically contacts the metadata service to get the list of partitions that should be stored on the local storage machine. There are two purposes to this task. The first purpose is to delete any partitions that should no longer be stored on this machine, either because all streams referencing those partitions have been deleted or the partitions have been moved to other machines as part of a load balancing operation, as described later in this paper. The second purpose is to ensure that the storage machine actually has all of the partitions that the metadata server believes are stored on that machine.

Once this list is obtained, the node service compares it against the partition data files stored in its data directory. For any partitions that are in the list obtained from the metadata server where the relevant files are not in the local data directory, which is an error condition, the system attempts to recover by calling `RemovePartitionReplica` to remove this storage machine from the list of replicas for this partition. This will trigger creation of additional replicas of this partition, if necessary. In the more common case where a partition is not in the list of partitions that should be stored on this machine, but a file is contained in the local data directory, that partition 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 vets

the list and returns a list of partition ids approved for deletion. The node service then deletes the files corresponding to the vetted list of partition ids.

The reason for this two phase deletion protocol is to prevent partitions that are in the process of being written from being deleted. The metadata server is aware of the partition ids that have been allocated, but have not been completely written. As a result, these pending partition ids will not be included in the list of partition ids stored on any storage machine. The complete function pseudocode is listed in Algorithm 1.

Algorithm 1 Garbage collection function

```
partitionIds = ListPartitionsAtNode();
filenames = ListFilesInDataDir();
List pdList;
for all file in filenames do
  id = GetPartitionIdFromFileName(file);
  if !partitionids.Remove(id) then
    pdList.Add(id);
  end if
end for
for all partitionId in partitionids do
  RemovePartitionReplica(partitionId);
end for
partIdsToDelete = VetPendingDeletionList(pdList);
for all partitionId in partIdsToDelete do
  DeletePartition(partitionId);
end for
```

There are a substantial fraction of partitions that are not frequently read. As demonstrated in [5], latent sector errors are a concern for the designers of any reliable data storage system. 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 a machine or multiple machine failure, the system would experience data loss for that partition. As a result, the node service periodically reads each partition replica and validates that its fingerprint matches the stored fingerprint at the metadata server.

The metadata server is responsible for ensuring that there are sufficient replicas of each partition, as calculated from the maximum replication factor of all streams the partition belongs to. Once the replicas have been assigned to particular machines, the node service is responsible for actually replicating the partitions. To do so the node service contacts the metadata server requesting the list of partition identifiers that should be replicated to this machine. For each partition identifier in this list, the node service will contact the metadata server to obtain the paths to read from and write to for replicating the partition. Once the partition has been replicated, the fingerprint of the partition will be validated to ensure it was correctly replicated, and the node service will inform the metadata server that it has successfully replicated the partition.

2.3 TidyFS Explorer

The two TidyFS components that have been previously described primarily 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 partitions that comprise a stream, along with relevant information about those partitions. Users can use the GUI to delete streams, rename streams, manipulate the sequence of partitions in a stream, as well as copy partitions between streams. Cluster administrators can use the TidyFS Explorer to monitor the state of machines in the system, including determining what machines are healthy, what replications are pending, and how much storage space is available. Administrators can also manually change the state of machines in the system and interact with the node service.

2.4 Interactions with Dryad

When a Dryad job's input is a TidyFS stream, the Dryad job manager first contacts the TidyFS metadata server to determine the partition id, size, and

replica location for each partition in the input stream. The job manager uses this information to set the location affinities for each vertex and then sets the input file URI for the vertex to identify the TidyFS partition that should be read. When each vertex starts, it contacts the TidyFS metadata server and requests a read path for the given partition, providing the location of the running vertex as a hint. The metadata server uses its knowledge of the cluster network topology to provide the path to the closest partition replica. The metadata server prioritizes local replicas, then replicas stored on a machine within the same rack, and finally replicas stored on a machine in another rack. Once the vertex receives the read path for the partition, which identifies the location of the data file for that partition replica, the vertex proceeds using the appropriate reader for that partition type, as indicated by the protocol in the read path URI.

When a Dryad job's output is a TidyFS stream, the job manager contacts the TidyFS metadata server to create a temporary output stream, requesting the number of partition ids to be the number of output vertices in the final stage of the Dryad computation. This temporary stream has a short lease set which will lead to the deletion of the stream if the job does not complete. These partition ids are assigned to vertices as they are scheduled, and the job manager maintains a map of vertex id to partition id. When the initial pool of partition ids is exhausted, either due to vertex failures or duplicate scheduling, the job manager requests additional partition ids which are associated with the temporary output stream. When each vertex starts, it contacts the TidyFS metadata server and requests a write path for the given partition id. In most cases, the returned path is on the local machine. However, if the local machine is not in the `ReadWrite` state, then the initial partition will be written to another machine chosen randomly from the set of machines in the `ReadWrite` state. Once the vertex has finished writing that partition, the vertex supplies the metadata server with the partition's id, size, fingerprint, and location of the initial replica, at which point the metadata server will schedule additional replicas. Once all vertices in the output stage have completed successfully, the job manager creates the final output stream and adds the sequence of partition ids from the successful vertices, at which point the temporary stream is deleted.

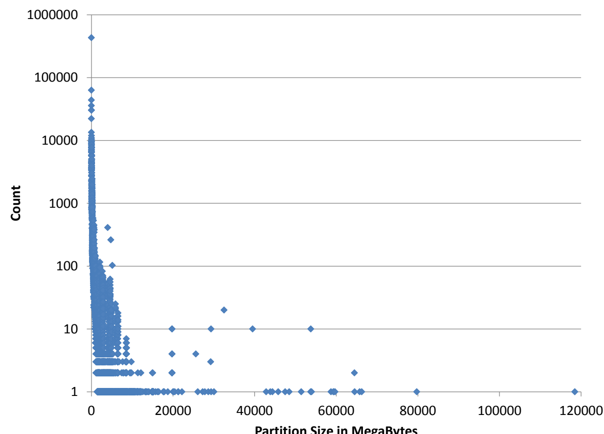

Figure 2: Histogram of partition sizes (in MB)

2.5 Replica Placement

When choosing where to place replicas for each partition, the system attempts to optimize two separate criteria. First, it is desirable for the replicas of the partitions in a particular stream to be spread across the available machines as widely as possible, which allows many different machines to perform local disk reads when processing that stream. Second, storage space used should be roughly balanced across machines. Figure 2 shows a histogram of partition sizes in a cluster running TidyFS. There are 215,575 partitions that are more than 128 MB in size, which is 16% of the partitions and 86% of the storage used in this cluster. Due to this non-uniform distribution of partition sizes, assigning partition to replicas is not as simple as assigning roughly equal numbers of partitions to each machine.

The location of the data for a partition is determined as the result of a call to `GetWritePath`, where the client provides both the partition identifier and the name of the machine that the client is running on. If that machine is configured as a TidyFS storage machine, and is in the `ReadWrite` state, a local path-

name will be returned to the client. Once the client has completed writing the partition, replicas of the partitions will be scheduled by the metadata server as described previously. We have implemented the policy for replica assignment in two different ways. Initially, we implemented a policy that would assign a replica to the machine that had the most free space, was in the `ReadWrite` state, and did not already contain a replica of that partition. After implementing this policy, we observed performance problems due to many partitions from the same stream residing on the same machine. This led us to implement a second policy, similar to the one used in the Kinesis project [14], which uses the partition identifier to seed a pseudo-random number generator, which allows deterministic execution across all state machine replicas, then chooses three machines in the `ReadWrite` state using numbers drawn from this pseudo-random number generator, and then finally chooses the machine with the most free space from this set. This provides an efficient mechanism to choose replicas based on both stream distribution and available space.

3 Conclusions

This paper has presented the design of the TidyFS distributed file system, which is designed explicitly for sequential, read-mostly data parallel workloads. Of course, while the file system is primarily designed for use in conjunction with Dryad and DryadLINQ, it can be used by other clients desiring a read-mostly distributed storage system. The file system is comprised of three components: a replicated metadata server, a service run on each storage machine in the cluster, and a graphical user interface.

References

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

[2] Open MPI. http://www.open-mpi.org/.

[3] Sort benchmark. `http://research.microsoft.com/barc/SortBenchmark/`.

[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 *IOPADS '99: Proceedings of the sixth workshop on I/O in parallel and distributed systems* (New York, NY, USA, 1999), ACM, pp. 10–22.

[5] Bairavasundaram, L. N., Goodson, G. R., Pasupathy, S., and Schindler, J. An analysis of latent sector errors in disk drives. In *SIGMETRICS '07: Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems* (New York, NY, USA, 2007), ACM, pp. 289–300.

[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] 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)* (Dec. 2004), pp. 137–150.

[9] Ghemawat, S., Gobioff, H., and Leung, S.-T. The Google file system. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles* (New York, NY, USA, 2003), ACM Press, pp. 29–43.

[10] 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)* (October 4-6 2010).

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

[12] Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of European Conference on Computer Systems (EuroSys)* (March 2007), pp. 59–72.

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

[14] MacCormick, J., Murphy, N., Ramasubramanian, V., Wieder, U., Yang, J., and Zhou, L. Kinesis: A new approach to replica placement in distributed storage systems. *Trans. Storage 4*, 4 (2009), 1–28.

[15] Olston, C., Reed, B., Srivastava, U., Kumar, R., and Tomkins, A. Pig Latin: A not-so-foreign language for data processing. In *International Conference on Management of Data (Industrial Track) (SIGMOD)* (Vancouver, Canada, June 2008).

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

[17] 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)* (December 8-10 2008).