BoomFS: A Declarative Approach To Building Distributed File Systems

Peter Alvaro, Neil Conway

December 19, 2008

Abstract

While architectures for distributed computing are changing rapidly, techniques for building distributed systems have remained stagnant. As distributed computation becomes the common case, traditional techniques for building such systems will become increasingly burdensome, because they force programmers to deal with the mundane details of constructing reliable distributed systems rather than concentrating on the desired computation. This yields programs that are difficult to construct, understand, modify, and adapt to new environments. We propose BOOM, an ongoing project to develop concise declarative specifications for a broad class of scalable distributed systems. In this paper, we describe the first application in the BOOM stack: BoomFS, a distributed file system that is implemented using a combination of Java and declarative logic. We show that BoomFS is easy to understand and modify, and achieves competitive performance in a preliminary performance study.

1 Introduction

With the widespread adoption of cloud computing, mobile clients, and manycore processors, computing architectures are undergoing a period of disruptive change. In the near future, nearly every non-trivial software system will be physically dispersed: distributed programming will be the common case.

Traditional techniques for building distributed systems are ill-suited to this new environment. Despite considerable research, developing fault-tolerant distributed systems remains enormously difficult and expensive, and is typically only attempted by experienced programmers with extensive training in the field. As more journeyman developers encounter the challenges of distributed computing as a matter of course, this situation will become increasingly untenable. Better techniques for constructing distributed systems are urgently needed.

Inspired by prior work on declarative networking [13, 10], we propose a new architectural style for distributed programs in which policy and protocol are specified in a declarative logic language, while the operational mechanisms of the program are written in a traditional imperative language. We are engaged in the Berkeley Orders of Magnitude (BOOM) project, which aims to use this style to build cloud computing infrastructure components that operate at orders of magnitude greater scale but are specified in orders of magnitude less code than traditional approaches. In Section 2, we discuss the problems with traditional approaches to building distributed systems in more detail, and outline the BOOM vision in contrast.

The first system we have built using this architectural style is BoomFS, a distributed file system similar to the Google File System [7]. Our goal in building this system was not novelty of design: although BoomFS supports multiple master nodes, its architecture is otherwise very similar to GFS. Instead, our aim was to show that a GFS-like file system can be easily and concisely implemented in the BOOM style. The resulting system should achieve competitive performance, and be easy to understand and adapt to new environments and policies. In Section 3, we describe the basic architecture of BoomFS. In Section 4, we detail how this architecture was realized using a mixture of declarative logic and Java. In Section 5, we report on a preliminary performance study comparing BoomFS and the Hadoop File System (HDFS), an open source implementation of GFS [2]. We discuss plans for future work in Section 6, describe related research in Section 7, and conclude in Section 8.

2 The BOOM Vision

The fundamental problem with traditional techniques for building distributed systems is that they provide the wrong abstractions to the programmer. Distributed systems are typically implemented with tools developed for single-machine programs and only superficially adapted to the challenges of a distributed setting. Programmers are forced to deal with the tedious details of communication, synchronization, and distribution. As a result, the essence of the distributed computation is obscured by a thicket of boilerplate details. Evidence for this can be seen in the fact that distributed algorithms such as Paxos can be stated in a page of pseudocode, but require many thousands of lines of code to implement using standard tools [5].

Traditional tools operate at a low level of abstraction because they force programmers to intermingle the specification of *what* a distributed program should do with *how* it should be achieved. That is, mechanism and policy are specified together, and implemented in the same language. This approach has two primary problems: it yields fragile programs that cannot easily adapt to change, and it results in programs that are difficult to understand.

For example, consider a client that wants to compute a function over data stored in a compute cloud. Should the function's code be sent to the server, should the data be sent to the client, or should both code and data be sent to an intermediary? These alternatives can be viewed as "query plans" that accomplish the same objective, but have different performance characteristics. The optimal plan depends on factors including the relative costs of network bandwidth, server-side computation and client-side computation, how much data is required, how expensive the function is, and the frequency with which the function is invoked or the input data is modified. All of these parameters are likely to fluctuate, both within a single environment over time,[1] and when the distributed program is deployed to a new environment. Traditional approaches to constructing distributed systems require hardcoding assumptions about these parameters. This yields fragile programs that are expensive to modify and difficult to adapt to new environments.

In addition to being inflexible, the failure to separate mechanism from policy it harder to understand and reason about these programs. The policies and protocols of a distributed program are typically its most interesting parts, and often the hardest to get right. In contrast, the mechanisms that achieve those policies are often straightforward but verbose. By implementing policy together with mechanism, the mundane details of the latter obscure the essential nature of the former, harming comprehensibility. Furthermore, a language that is appropriate for implementing mechanism is unlikely to be ideal for specifying policy, and vice versa. There are many examples of algorithms that can be concisely expressed in an appropriate declarative language, but are much harder to write in an imperative language (e.g. [14]).

Inspired by the data independence provided by the relational model, we aim to provide *network scale independence* for distributed systems by separating the programmer's intent from its concrete realization [10]. In BOOM, a distributed system is composed of two types of components:

1. *imperative* components implement the basic units of functionality of a distributed system. Imperative components are typically used to perform tasks like I/O and numerical computation. These tasks are usually best stated in an imperative language like Java or C++, particularly because these components often involve interaction with the operating system or native libraries.

2. *declarative* components specify the bulk of the logic of the distributed system. These components are written as a collection of logical rules that describe the coordination and composition of the imperative components. Essentially, the declarative components are responsible for deciding "what" a member of the distributed system should do; the imperative components are responsible for realizing those actions. A declarative component is essentially a join between a stream of events and a database. Evaluating this query over an event stream results in producing more events (either at the local node or a remote node), inserting new database tuples, or invoking imperative components.

   Declarative components are implemented in a network-aware declarative logic language, such as Overlog [13]. This requires the state of the distributed program to be represented as relations that are partitioned over the nodes of the system.

In BoomFS, imperative components are used to

[1]Internal heterogeneity and performance variability in virtualized environments such as Amazon's Elastic Compute Cloud has been well-documented in recent work [18].

efficiently transfer data between hosts. The vast majority of the file system's complexity resides in the declarative components, which decide when and where data should be transferred. We describe the realization of BoomFS as a set of imperative and declarative components in Section 4.

2.1 Declarative Specification of Distributed File Systems

In many systems, there is a structural separation between control and data paths. This can be motivated either by principle (e.g. the separation of mechanism and policy [12]), or by practical concerns: data and control paths typically differ in average message size, and in requirements for consistency and latency. It is undesirable for data path congestion to delay control messages.

We argue that in many distributed systems, the bulk of the intellectual complexity resides in the control path; this is clearly true of the GFS design. This is not to say that operational components (zero-copy I/O, fast checksumming, careful use of caches at various levels) are not critical to system performance, but merely that the bulk of the design and implementation deal with enforcing and maintaining the various invariants that make the system behave correctly, including replica placement decisions, failure handling, load balancing and consensus among agents in shared computations.

Combining these observations, we see that GFS-like distributed file systems are attractive candidates for the BOOM approach to implementing distributed systems.

3 System Architecture

We now turn to the architecture, implementation, and performance characteristics of BoomFS. The architecture of the system is directly inspired by the Google File System. The design goals and system architecture of GFS are well-documented elsewhere [7, 2], so we include only a brief discussion of them here.

Like GFS, BoomFS does not attempt to be a general-purpose distributed file system. Instead, it is designed to perform well for a particular class of workloads and to operate on a particular cluster architecture. We focus on achieving good performance for large sequential reads and writes. Files are assumed to be very large, and are therefore divided into large *chunks* that far exceed typical file system block sizes (64 MB). We focus on delivering high sequential read and write throughput and efficient network utilization, rather than achieving low latency or minimizing total space requirements. Members of a BoomFS system are assumed to be unreliable commodity machines, provisioned with relatively modest resources. Reliability is achieved by storing multiple copies of each chunk and scalability is achieved by spreading file system content over a large cluster of machines (typically hundreds or thousands).

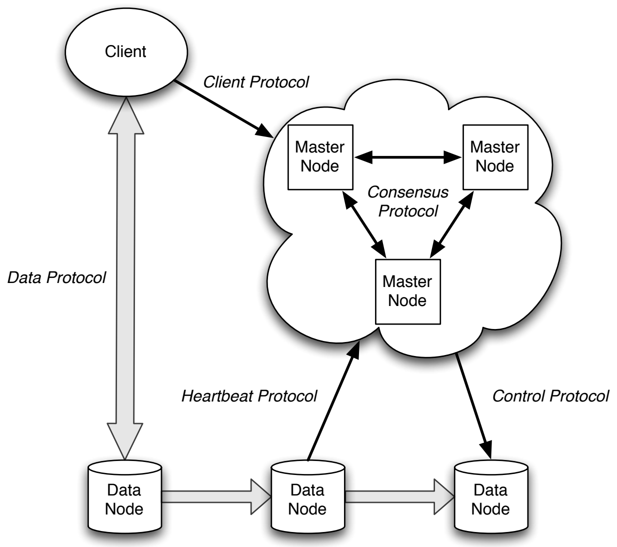

Figure 1: BoomFS architecture

3.1 System Overview

The major components of BoomFS are depicted in Figure 1. There are three types of nodes in BoomFS: *master nodes*, *data nodes*, and *clients*. Master nodes contain the canonical description of the structure of the file system. The file system is described by a mapping from file names to file identifiers, and from file identifiers to the sequence of chunks that contain the file's content. To avoid a single point of failure, our design allows for multiple master nodes, which are kept in synchrony using a consensus protocol (currently Paxos [11]).

Data nodes are responsible for storing chunks. They have no knowledge of the structure of the file system or the identities of any other data nodes. Each data node periodically sends a *heartbeat* to the master nodes. This notifies the masters that the data node is still alive, and contains a list of the chunks currently stored at the data node. The masters use this information to update the mapping from chunk identifiers to the set of data nodes that might be holding that chunk. Note that the canonical description of the content of a data node resides at the data node itself, not at the master nodes; the master's copy of this information is updated lazily.

Finally, client nodes represent application programs that wish to read and write files. We expect that applications will interact with the system through a client library that provides a convenient stream-like abstraction for files stored in BoomFS. In the future, we plan to implement the HDFS API, to allow BoomFS to easily replace HDFS in existing Hadoop installations.

3.2 File System Operations

To append to a file, a client node begins by contacting one of the master nodes and requesting that a new chunk be added to a particular file.[2] The master first generates a new chunk identifier. To ensure that the state of the file system is consistent across all the masters, the master uses a consensus protocol to ensure that all masters will agree to add the new chunk identifier at the same position in the chunk list for the appropriate file. Once consensus has been reached, the master replies to the client with the new chunk identifier and a list of data nodes that might be appropriate locations for the new chunk.

The client is then responsible for transferring the chunk's content to a sufficient number of data nodes. There are various policies a client could use to do this. For example, the client could directly connect to all the data nodes and send the chunk content itself, or it might only send the data to a single data node and instruct that node to propagate the data onward. Once the content of a chunk has been completely received by a data node and written to disk, this fact will be reflected in the next heartbeat that the data node sends to the masters. In turn, this will make the data node available for subsequent operations on the chunk.

To read a file, a client once again begins by contacting one of the master nodes to fetch the list of chunks that make up the target file. For each chunk identifier in the list, the client consults the master to determine the set of data nodes that have copies of that chunk. The client then reads the chunk by connecting to one of the data nodes directly.

To delete a file, a client contacts a master node. The master passes the request through the consensus protocol. When consensus has been reached, all the masters remove the file from their internal metadata. There is no need to eagerly contact data nodes to remove the chunks that make up the deleted file: the next time the data node sends a heartbeat to a master, the master replies with a list of the chunks on the data node that is has no knowledge of. These orphaned chunks can be garbage collected at the data node's leisure.

Other file system operations are straightforward. For example, clients interact with master nodes to obtain directory listings, and create and remove directories.[3]

4 System Realization

The file system design described in Section 3 is abstract, and might reasonably be implemented using a variety of techniques. In this section, we describe how we realized the BoomFS design using a combination of Java and Overlog. In the following discussion, we assume that the reader has some familiarity with Overlog [13].

We validate our implementation strategy in two ways: by demonstrating how a declarative specification enables easy modification of file system policies in Section 4.6, and by conducting a performance study in Section 5.

4.1 Software Architecture

In BoomFS, masters, data nodes, and clients are all implemented using a combination of Java and Overlog. We used JOL [3], a Java-based Overlog implementation that allows an Overlog evaluator to be embedded inside a Java program, and allows Overlog programs to contain Java objects and invoke Java methods. Java programs can insert and delete tuples from Overlog relations, and register callbacks that are invoked when a particular state is reached by the Overlog instance.

Each node type in BoomFS is structured in a similar way. Upon startup, the node runs Java code that

[2] In the current prototype, random writes are not supported and each append operation creates a new chunk — we expect that each append will be large enough to justify occupying an entire chunk. We plan to relax this constraint in the future.

[3] Our prototype implementation of BoomFS does not currently support directories, but we expect that this implementation shortcut will be easy to fix.

bootstraps a JOL instance, and installs the Overlog files for BoomFS.

This architecture is quite similar to the familiar structure of an application that uses a DBMS: the database is responsible for managing data, and the application uses queries to retrieve and modify the state of the database. However, there are some crucial differences between this architecture and the design of BoomFS. For example, Overlog relations are partitioned over multiple nodes. While one-time queries are expressible in Overlog, continuous queries are the more common case, represented syntactically as a recursive join and semantically as a cyclic dataflow graph.

4.2 State Representation

Any file system contains two kinds of state: data and metadata. In BoomFS, data is stored as a collection of chunks distributed over the data nodes. Each data node stores chunks as normal files on its local file system.

All the metadata in BoomFS is represented as relations; some of these relations are partitioned over multiple nodes.

4.3 Control Path

In GFS, the control path comprises three separate protocols: the client protocol spoken between clients and masters to modify file system metadata, the control protocol between master and data node used for replica migration or deletion instructions, and the heartbeats regularly sent from data nodes to the masters. The replication mechanism, which in GFS involves a log flush to multiple slave masters, can be considered a fourth protocol.

In our design, instead of using separate state machines to implement the various communication protocols, all control path messaging is specified declaratively. Logical inference rules, like database views, perform select-project-join over locally materialized relations, and the newly projected tuples are sent over the network if their location specifier indicates that they belong in a different partition, and hence a different node in the system. At the destination, an incoming tuple's schema describes its content and determines which rules, if any, should be reevaluated with reference to the new data.

For read-only operations like many client protocol interactions, a single declarative rule on the client (to transfer the injected request tuple to the master) and a pair of rules on the master (one to return the results to the client, and an error handler) suffice to express the messaging logic. Requests that modify file system metadata follow the same structure, but trigger a more complicated chain of inferences on the master. In particular, the master must achieve consensus among master replicas, attempt the modification, and indicate success to the client only if these steps succeed. Heartbeats from data nodes to the master are structured similarly to client requests, but are triggered by a periodic timer rather than by human interaction. Finally, control protocol messages from master to data node are fired when conditions specified by certain aggregate queries indicate that system invariants are unmet, as when the number of replicas drops below the specified replication factor.

Let us take a concrete example beginning with a client request to append a stream of data to a file. The Java shell code running at the client injects a tuple into a local table, triggering a send rule that causes the tuple, which contains the client's address, the file identifier, and an opcode indicating that this is a request for a new chunk identifier, to the master. This initiates a chain of inferences that generate a new id, achieve consensus on the assignment of this id to the given file's list of chunks, and select a set of possible candidate data nodes that may store the new chunk. The resulting chunk identifier and candidate data node list are then embedded in a tuple and returned to the client. The client applies its local policy, which may be a simple distance function, to sort the set of candidate data nodes and select a next hop. From here, the data path is invoked for the remainder of the operation.

4.4 Data Path

The data protocol is a straightforward mechanism for transferring the content of a chunk between two hosts. The server side of the protocol is implemented as a simple imperative component that runs on each data node. It listens on a dedicated TCP port and handles each client connection using a separate thread. Data transfer is done using the `FileChannel.transferTo()` mechanism provided by Java NIO, which allows an underlying zero copy API to perform the bulk data transfer (such as `sendfile(2)` on Linux). The data protocol consists of two simple operations: writing a chunk from the client to the server, and reading a chunk from the server to the client. The protocol client is either a client node or another data node.

While the data protocol is unicast, a client node typically wants to write a new chunk to multiple data nodes. As noted in Section 3.2, there are several ways in which this could be accomplished. The current BoomFS prototype does "source routing": after obtaining a new chunk identifier and list of candidate data nodes from a master node, the client decides on the sequence of data nodes to which it wants to write a new chunk, and then transfers the chunk to the first node in the sequence. That node then transfers the chunk to the second node in the sequence, and so on. While this approach is effective at utilizing network bandwidth, it is subject to partial failures, and the current implementation is not pipelined. More importantly, routing decisions of this nature should be specified in Overlog, not hardcoded in Java. We plan to rectify this shortcoming in the near future.

4.5 Fault Tolerance

Both BoomFS and GFS are intended to be deployed on large clusters of commodity machines, in which "component failures are the norm rather than the exception" [7]. In this section we explore several of the failure modes of the system, and discuss how our integration of a declaratively-specified Paxos implementation addresses shortcomings in the original GFS and HDFS design.

4.5.1 Data Node Failure

As long as the replication factor of a file is greater than one, the loss of a single data node will be transparent to applications using the system. The last heartbeats of the lost node will quickly expire from the soft-state tables on the masters, and clients will no longer be directed to this server for read or write requests. The updated soft state will cause aggregates in the BoomFS logic to be recomputed, which in turn cause new migration events to be fired, selecting another replica as a target for the chunks whose replication factor is now too low. GFS and HDFS both use this kind of replication strategy, though Ghemawat et al. discuss the possible use of other forms of redundancy, such as parity [7].

4.5.2 Data Node Disk Failure

The effect of this failure is more or less the same as for data node failure, assuming that all the file system data is stored on a single disk. The data node will continue to send heartbeats to the master, but one of these will include a delta record describing the loss of the files on the disk. The same chain of inferences described in the lost data node scenario described above will fire for these chunks. It should be noted that our current prototype will throw an exception after trying to read from the lost directory, and the data node service will stop, resulting in *identical* behavior to the failed data node scenario.

4.5.3 Data Corruption

GFS and HDFS implement data integrity checking at different granularities: in HDFS, chunk (called "blocks" in HDFS) checksums are calculated over the entire 64 MB chunk, while in GFS, a chunk is subdivided into 64 KB blocks, each of which has an associated 32 bit checksum. Our prototype did not implement any integrity checks, but we plan to add this feature in the near future.

4.5.4 Master Failure

GFS keeps all file system metadata on a single master server to which all client requests are directed; the mapping is maintained through the DNS. Several secondary masters are run in a log replay mode similar to the log-shipping approach employed by database systems: a metadata change is not considered committed until the log tail has been flushed not only to stable local storage, but to all secondary masters. This implies the use of a two-phase commit protocol, but the implementation is not discussed in detail. If the primary master fails, the failure must be detected by an external application, which then remaps the DNS entry to one of the secondaries.

HDFS supports the concept of a "Secondary NameNode" or backup master that asynchronously applies log entries to a checkpoint image. The primary master supports writing log entries to multiple directories, one of which can be a remote file system such as NFS. A secondary master can then read the log from this file system. Presumably, this adds considerable overhead to the performance of the master.

In our prototype, a configurable number of masters operate in lock-step using the Paxos consensus protocol, also implemented in the Overlog language. This means that like GFS, a mutation of file system metadata in not complete until it is reflected on all participating masters. Unlike GFS, BoomFS backed by Paxos is a multimaster system that supports "update anywhere anytime" semantics [8], although for performance reasons it is preferable to have all clients communicate with a single primary master. Assuming non-byzantine faults, a BoomFS installation with m masters can survive $m/2 - 1$ failures, as consensus requires merely simple minority. If the primary fails, we also incur the cost of a timeout on each client on the first request to the failed master; after that, the client's master list is updated to reflect the loss. No external monitoring or indirection are necessary to support this level of fault tolerance.

4.6 Implementation Process

Our task in developing the BoomFS prototype was to represent the distributed state of the file system as a single relational database, with relations partitioned across nodes. Both policy (which expresses constraints over these relations, and conditions under which the contents of the relations may change) and protocol (which expresses how data may move from one node to another) can then be expressed in a network-aware recursive query language that operates over the relations.

As we built the prototype, we concentrated first on the correctness of the specification and on exploiting the economy of mechanism exposed by flattening system state. For the imperative components, including data transfer and source-routing for the data path, we implemented simple, best-effort modules as place holders while we built the collection of logical rules comprising the declarative specification of the system. Once this was correct, it was simple to return to the imperative components and tune them to improve performance, without worrying about violating the correctness guarantees made by the policy layer.

It was equally straightforward to extend the declarative components of the system. With the global state flattened, it was easy to return to many initially deferred optimizations. For example, our initial implementation of the system sent the entire state of each data node in every heartbeat message. While this was correct, it is clearly suboptimal as file system size increases. Changing the heartbeats to reflect only chunk status deltas was a simple matter of sending the set difference of the local chunk relation and the relation of chunk heartbeat messages that have been acknowledged by the master. Deletion deltas are simply the same set minus with the terms reversed. Our initial policy for replica placement was purely distance-based, and had poor distribution of chunks when a small number of topologically close clients were appending data. Extending this to a multi-level ordering that considers data node chunk load was a simple matter of adding an aggregate query computing chunk count per node, and performing a second bottom-k computation to select the lowest loaded nodes that are not more than some specified distance from the client.

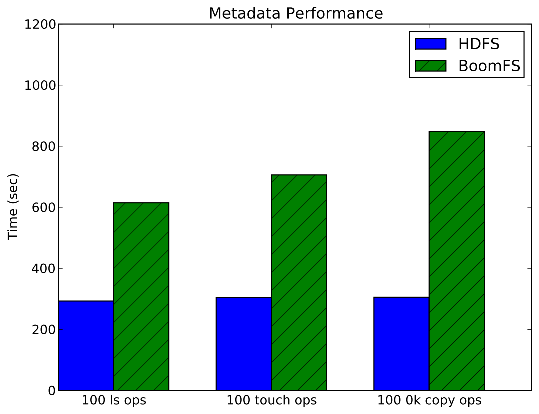

Figure 2: Performance for metadata operations

5 Performance Evaluation

To show that our approach to building BoomFS is competitive with traditional techniques for constructing distributed systems, we compare the performance of BoomFS and HDFS on two different workloads. When designing BoomFS, our goal was to achieve high performance on the large append and read workload that GFS-like file systems are designed for, despite the increased control overhead of our Overlog runtime. In this section, we evaluate how successful we were at achieving this goal.

We performed our experiments on the Amazon Elastic Compute Cloud (EC2), a virtualized "utility computing" environment [1]. In Section 5.1, we compare the cost of metadata operations between our prototype and HDFS. In the following section, we simulate a MapReduce sort workload, comparing the read and write performance of BoomFS and HDFS as we scale the number of concurrent clients and data nodes together.

5.1 Metadata Operations

In our first experiment, we compare the latency of metadata operations affecting the control path. A directory listing uses the client protocol and requires a round trip between the client and master and a lookup on the master, as does touching a file on the file system. Copying a zero-byte file requires two round trips to the master: one to get a new chunk identifier for the append operation, and another to request a set of data nodes who can accept the new chunk.[4] The results of running 100 of each of these metadata operations are shown in Figure 2. As we anticipated, the overhead of these operations is significantly higher in our implementation than in HDFS. This is partly due to the current performance of the JOL implementation. Another factor is the presence of client-side caching, enabled in HDFS and not yet implemented in BoomFS. However, it would be simple to locally materialize lookup results into relations that can be queried before visiting the master.

5.2 Sort Benchmark

Our next experiment evaluates read and write performance under a typical MapReduce workload. The sort benchmark of Dean and Ghemawat [6] is a simple and practical test that puts equal stress on the read and write components of a distributed file system.

The MapReduce specification of sorting is trivial, largely because sorting (over *some* key) is a side-effect of the framework. Thus, the map function simply returns the sort key and the entire line as the key's value, and the reduce function is simply the identity function. In an execution of the sort, disjoint sections of a single input file are read in parallel by all of the mappers, which apply the hash function and write the bucketed results to their local disks. The reducers then read these files via RPC calls, apply the built-in merge sort, and write as many files back to the GFS as there are reducers.

For this experiment, we are interested only in the costs associated with the parallel reads at the start of the workflow described above, and the parallel writes described at the end. Hence, we dispense with the actual sorting, hashing and crossbarring, and focus merely on file system performance as the number of concurrent clients and data nodes are scaled up together. For a given concurrency level D, we provision an EC2 cluster with a master node and D data nodes, and prime the file system by creating D 100 MB files with a replication factor of 2. We then spawn D client processes (one on each data node), which read one of the files from the file system, write it to local disk, then create a new distributed file system file and append the contents of the local file to it. Thus, each client reads (writes) 100 MB to (from) the file system concurrently in each sort benchmark.

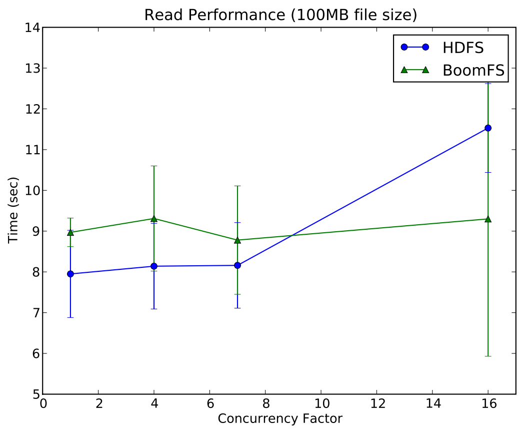

Figure 3: Read performance for the sort benchmark

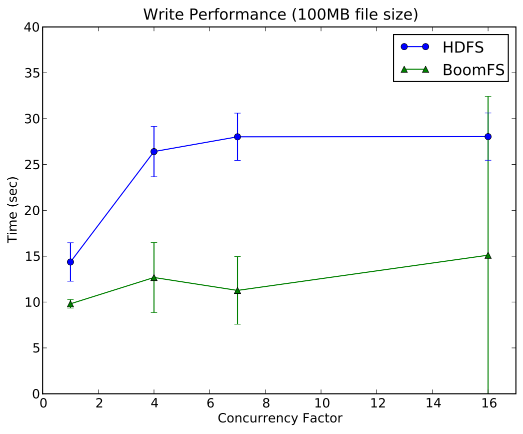

Figure 4: Write performance for the sort benchmark

Our results are summarized in Figures 3 and 4. At lower concurrency levels, BoomFS is comparable in read performance and handily outperforms HDFS in write performance. These results support our intuition that the relatively high cost of metadata operations is quickly amortized by the highly efficient data path under the large transfers characteristic of MapReduce workloads.

In HDFS, read latency increases gradually as load increases, but write latency reacts very quickly to concurrency, increasing by almost two times as we increase the number of clients from 1 to 4. Beyond that

[4]Combining these responses into a single message is a simple optimization not yet implemented in our prototype.

point, HDFS gracefully handles increasing load with low variance. BoomFS shows similar read results, but significantly lower write latency. We believe that BoomFS's superior write performance is due partly to implementation shortcuts (e.g. BoomFS performs no checksumming), and partly to the efficiency of the data path implementation.

At 16 concurrent clients, BoomFS began highly variable performance. The variance for $x = 16$ in Figure 4 reflects the fact that 4 of the 48 operations took longer than 60 seconds, triggering a timeout. If we omit those 4 operations, the mean write response time was 10.12 seconds, which is consistent with BoomFS performance at lower concurrency levels. Although we have not yet isolated the cause of this performance variability, its relatively rare occurrence suggests it is more likely a bug than a systemic or fundamental issue.

6 Future Work

The current version of the BoomFS prototype confirms that the BOOM architectural style can be successfully applied to the construction of a distributed file system. However, we feel that the most exciting research on this topic remains to be done.

We believe that interesting issues in computer system design often only arise when technology is deployed in production environments. Therefore, we plan to complete the implementation of BoomFS and deliver a production-quality file system. This requires implementing support for directories, fine-grained concurrent appends, and persistent metadata. We also want to ensure that BoomFS is able to scale to petabyte-range data sets and thousands of nodes. We plan to validate BoomFS's usability and performance by implementing the HDFS API, and then using BoomFS to replace HDFS in real-world Hadoop installations.

We also plan to leverage the flexibility allowed by BoomFS's declarative specification to explore new file system features and policies. We closely followed the GFS design primarily to demonstrate that a faithful reimplementation is possible using declarative techniques, but we plan to consider a much broader space of design alternatives in the future. For example, it should be possible to partition the file system metadata over the master nodes, removing a major scalability barrier.

Finally, we plan to explore the cross-layer optimizations that are enabled when multiple elements of the software stack are implemented using the same declarative techniques. For example, colleagues are completing an implementation of the Hadoop job scheduler in Overlog. By running this on top of BoomFS, it would be possible for an optimizer to make intelligent costing decisions for file system operations that automatically reflect the characteristics of the current Hadoop workload.

7 Related Work

To our knowledge, this work represents the first attempt to specify a distributed file system using declarative techniques. However, it can be argued that GFS and related designs (including BoomFS) are "file systems" only in a loose sense, because they do not implement POSIX semantics and provide looser consistency guarantees than traditional file systems. In this sense, BoomFS is clearly related to prior work on declarative specifications of storage infrastructure, such as DHTs [14] and data replication systems [4].

Recent work on a declarative file system integrity checker [9] provides another example of how sophisticated policies and invariants can be concisely stated in an appropriate high-level declarative language. Similar tools for distributed consistency checking, garbage collection, monitoring, and periodic optimization should be easy to write on top of BoomFS, because all file system metadata is already available for declarative queries. We plan to explore this topic in the future.

Both BoomFS and GFS emphasize the distinction between control and data paths. This division has a long history in computer science, and can be found in protocols ranging from FTP [16] and DOT [17], to file systems like Berkeley FFS [15] and ext3.[5] Similarly, the separation of control and data is a well-established principle in computer system design [12].

8 Conclusion

We have implemented BoomFS, a distributed file system that is similar to GFS and HDFS. BoomFS achieves competitive performance with HDFS. Unlike HDFS, BoomFS avoids a single point of failure

[5]In these file systems, metadata and data are subject to different consistency requirements and durability guarantees, and are implemented using different on-disk data structures.

through its support for multiple master nodes and automatic failover.

More importantly, BoomFS is implemented using the BOOM architectural style: the complex protocols and policy of the control path are specified in declarative logic, while the simple, high-performance data path is written in Java. BoomFS was straightforward to implement. Because the implementation is concise and comprehensible, the file system is easy to modify and adapt to new environments. BoomFS is the first of several pieces of data center infrastructure we plan to implement using declarative techniques as part of the BOOM project.

References

[1] Amazon Elastic Compute Cloud. `http://aws.amazon.com/ec2`.

[2] HDFS architecture. `http://hadoop.apache.org/core/docs/current/hdfs_design.html`.

[3] Java Overlog Library (JOL). `https://svn.declarativity.net/lincoln/java`.

[4] N. Belaramani, M. Dahlin, A. Nayate, and J. Zheng. PADRE: A Policy Architecture for building Data REplication Systems. `http://www.cs.utexas.edu/users/dahlin/papers/padre-may-2008-extended.pdf`, May 2008.

[5] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing*, pages 398–407, 2007.

[6] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the Sixth Symposium on Operating System Design and Implementation*, pages 137–150, 2004.

[7] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating System Principles*, pages 29–43, 2003.

[8] J. Gray, P. Helland, and D. Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173–182, 1996.

[9] H. S. Gunawi, A. Rajimwale, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. SQCK: A declarative file system checker. In *Proceedings of the Tenth Symposium on Operating System Design and Implementation*, 2008.

[10] J. M. Hellerstein. Toward network data independence. *SIGMOD Record*, 32(3):34–40, 2003.

[11] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, December 2001.

[12] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in hydra. In *Proceedings of the Fifth ACM Symposium on Operating Systems Principles*, pages 132–140, 1975.

[13] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking: language, execution and optimization. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 97–108, 2006.

[14] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, volume 39, pages 75–90, 2005.

[15] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, 1984.

[16] J. Postel and J. Reynolds. RFC 959: File transfer protocol. `http://www.ietf.org/rfc/rfc959.txt`, 1985.

[17] N. Tolia, M. Kaminsky, D. G. Andersen, and S. Patil. An architecture for Internet data transfer. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI)*, May 2006.

[18] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce performance in heterogeneous environments. In *Proceedings of the Tenth Symposium on Operating System Design and Implementation*, 2008.