Boxwood: Abstractions as the Foundation for Storage Infrastructure

John MacCormick, Nick Murphy, Marc Najork,
Chandramohan A. Thekkath, and Lidong Zhou

Microsoft Research Silicon Valley

Abstract

Writers of complex storage applications such as distributed file systems and databases are faced with the challenges of building complex abstractions over simple storage devices like disks. These challenges are exacerbated due to the additional requirements for fault-tolerance and scaling. This paper explores the premise that high-level, fault-tolerant abstractions supported directly by the storage infrastructure can ameliorate these problems. We have built a system called Boxwood to explore the feasibility and utility of providing high-level abstractions or data structures as the fundamental storage infrastructure. Boxwood currently runs on a small cluster of eight machines. The Boxwood abstractions perform very close to the limits imposed by the processor, disk, and the native networking subsystem. Using these abstractions directly, we have implemented an NFSv2 file service that demonstrates the promise of our approach.

1 Introduction

Implementing distributed, reliable, storage-intensive software such as file systems or database systems is hard. These systems have to deal with several challenges including: matching user abstractions (e.g., files, directories, tables, and indices) with those provided by the underlying storage, designing suitable data placement, prefetching, and caching policies, as well as providing adequate fault-tolerance, incremental scalability, and ease of management. Indeed, it is generally believed that building a distributed file system or a distributed database with all these properties is an unrealistic ideal. Our hypothesis in this paper is that this perceived difficulty can be considerably lessened through the use of suitable abstractions such as trees, linked lists, and hash-tables, provided directly by the storage subsystem, without compromising performance, scalability, or the manageability of the storage system or the higher-level subsystems built on top of it. We have built a system called Boxwood to explore the feasibility and utility of providing such high-level abstractions or data structures as the fundamental storage infrastructure. Using these abstractions directly, we have implemented a highly available and scalable NFSv2 server that runs on multiple machines, coherently exporting the same underlying file system.

Although Boxwood's approach to storage is a significant departure from traditional block-oriented interfaces provided by disks—whether physical, logical [17], or virtual [21]—we think it provides some key advantages. One advantage, as evidenced by our experience with the multi-machine NFS server, is that by directly integrating data structures into the persistent storage architecture, higher-level applications are simpler to build, while getting the benefits of fault-tolerance, distribution, and scalability at little cost. Furthermore, abstractions that can inherently deal with sparse and non-contiguous storage free higher level software from dealing with address-space or free-space management. In contrast, even sophisticated virtual disk systems that provide scalability and ease of management require higher layers like the file system to deal with free space management, data placement, and maintaining user-visible abstractions [32]. A third advantage is that using the structural information inherent in the data abstraction can allow the system to perform better load-balancing, data prefetching, and informed caching. These mechanisms can be implemented once in the infrastructure instead of having to be duplicated in each subsystem or application.

Our earliest experience with Boxwood convinced us that there is no single universal abstraction to storage that will serve the needs of all clients. Our current prototype provides two: a *B-tree* abstraction, which allows typical operations like lookups, insertions, deletions, and enumerations, and a simpler *chunk store*, where variable sized data items can be allocated, freed, written, and read, in much the same way that memory is today.

Our specific choices were motivated by several obser-

vations. First, B-trees are a very useful abstraction for many storage needs found in file systems, databases and the like. Second, building a high-performance, scalable and *distributed* B-tree is a considerable challenge (even building a centralized version with good concurrent performance is difficult). We believe that our experience with this data structure will complement the existing literature in building distributed data structures like hash tables. Third, we believe that our simple chunk store abstraction provides a better match for applications that do not need the strict atomicity guarantees or the rigid structure of a B-tree. This simpler abstraction offers good performance and much flexibility to client applications while offloading the details of free space (or in a virtual disk environment, address space) management.

Our prototype is implemented by a collection of "server nodes", each containing a CPU, RAM, one or more disks, and a network interface packaged as a rack-mounted server. One can imagine alternative implementations of server nodes ranging from individual disk units to disk controllers that control sets of disks.

In addition to its focus on distributed storage abstractions, this paper also offers some insights into the structure of fault-tolerant distributed systems. The classic approach to building such systems is to use Lamport's replicated state machines with Paxos [20]. Our approach, although highly reliant on Paxos for consensus, uses a fault-tolerant distributed lock service and the simple notion of shared memory (or more precisely shared store) programming to deal with the inherent complexity of a distributed system with independent failures. We believe the lessons learned may be valuable in the design of other fault-tolerant distributed systems.

2 Boxwood System Structure

The overall goal of the Boxwood project is to experiment with data abstractions as the underlying basis for storage infrastructure. Generally, the term storage infrastructure connotes several requirements, a few of which are: *redundancy and backup schemes* to tolerate failures, *expansion mechanisms* for load and capacity balancing, and *consistency maintenance* in the presence of failures. Thus, ideally Boxwood needs to go well beyond providing distributed data structures. Our current status does not satisfy this ideal, but we have made much progress. For example, to deal with transient failures, we provide services for logging and transaction recovery. To deal with the correctness of replication in the presence of failures, automatic reconfiguration and expansion, we provide mechanisms (e.g., a Paxos consensus module, a lock service, and a failure detector) to insure a correct inventory of the components in the system and to provide a consistent view of the overall system.

In this section, we describe parts of our design as it relates to data abstractions and storage infrastructure mechanisms. We envisage our system being deployed in a machine room or in an enterprise cluster as the principal storage infrastructure used by file systems, database systems, and other services. This environment justifies several assumptions that impact our design choices. We first enumerate these assumptions and design principles before describing our system in greater detail.

2.1 Preliminaries

The Boxwood system is targeted at an environment that has multiple processing nodes each with locally attached storage, interconnected by a high-speed network. These processors run the Boxwood software components to implement abstractions and other services. Software running on a processor communicates with locally attached disks using a low-level interface similar to the UNIX raw device interface. We use remote procedure call (RPC) to access resources and services executing on remote processors.

The Boxwood system is organized as several interdependent services. We use layering as a way of managing the complexity in a Boxwood system. For example, the B-tree and the chunk store services mentioned earlier in Section 1 are constructed by layering the former on top of the latter. The chunk store service, in turn, is layered on top of a simple *replicated logical device* abstraction (to be described in Section 3.4). Although layering has the potential for reducing performance because of context switching overheads, our design avoids these problems by running all layers within a single address space.

Our interconnection network is Gigabit Ethernet; we therefore feel justified in providing fault-tolerance by *synchronous replication* of data on two disks attached to separate machines. With this scheme, under fault-free operations, the primary replica must wait for the secondary to finish writing its copy before it can return to the client. This wait can be large on a slow network, but is tolerable in a high-speed LAN. We also feel justified in assuming that the cost of making RPCs is small and that the network can be scaled by adding more switches. Our implementation results bear out these assumptions.

We use a security model that is appropriate to the target environment. Specifically, we assume that the machines are within a single administrative domain and are physically secure. We therefore send messages between machines in the clear and make no special provisions for encryption, authentication, or security.

We assume that CPUs, disks, and networks can fail. Such failures can be transient or permanent. Examples of transient failures that we can tolerate are: a faulty

power supply takes down a machine (and its attached disks), which will come back up without the contents of its RAM after power is restored; or the operator mistakenly unplugs a network cable. Examples of permanent failures are: a disk suffers catastrophic media failure, or a server's log is destroyed beyond repair. We assume, realistically, that the failure of a disk affects only that disk, but the failure of a machine affects it and all the disks attached to it. Although unlikely, we assume that networks can partition. Failures are assumed to be fail-stop.

Our fundamental mechanism for protecting data against catastrophic media failure is chained-declustered replication [14]. Thus, the permanent failure of a single disk will not cause data loss or data unavailability. In fact, chained-declustering prevents data loss even in the presence of many combinations of multiple disk failures as well, but not against all combinations of multiple disk failures. We also deal with the failure mode when all machines suffer a transient power outage and come back having lost the contents of RAM.

In our design, each service consists of software modules executing on multiple machines. Each service independently arranges for failover and high availability in the presence of multiple failures. For instance, our Paxos consensus service (described in Section 3.1) works as long as a majority of Paxos servers are running. Thus, double failures can be tolerated by running five Paxos servers and triple failures with seven. Similarly, our lock service (described in Section 3.3) uses a single master and one or more slaves as standby. Depending on the number of slaves we choose to run, we can tolerate multiple permanent failures. Our B-tree and NFS services described in Sections 3.7 and 5 impose no additional availability constraints as long as at least one instance of each module is running and the underlying services (e.g., locking, consensus, and replicated data) are available.

As the scale of the deployment increases, the probability of multiple failures increases. Our design is most vulnerable to increased disk failures in this regard. If the probability of double disk failures becomes a serious problem, we can use a different data protection scheme (e.g., triplexing or erasure coding) at the lowest layers without changing the design of any of the other services.

The principal client-visible abstractions that Boxwood provides are a B-tree abstraction and a simple chunk store abstraction provided by the *Chunk Manager*. Figure 1 shows the organization of these abstractions relative to each other. We introduce them briefly here, but defer a fuller description to later sections.

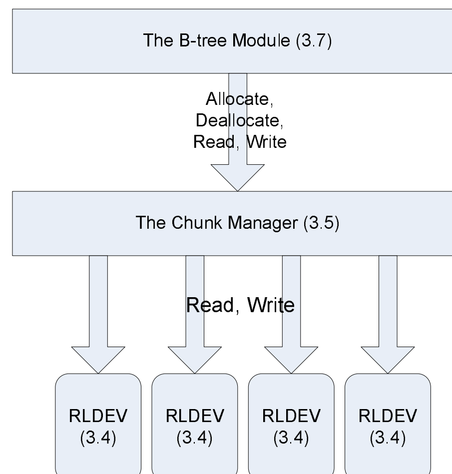

Figure 1: **Design of the Boxwood abstractions.** The B-tree is layered atop the chunk manager, which is layered on top of the replicated logical device. The numbers in parentheses refer to the section describing the design of the module.

2.2 B-tree Abstraction

B-trees and their variants are widely viewed as the best general-purpose data structure for implementing a dictionary (supporting insertion, lookup, deletion, and enumeration of key-value pairs) on secondary storage. B-trees are also complex enough to exercise fully the features and foibles of a distributed storage architecture. Therefore, they seemed an excellent candidate for the first data abstraction to be implemented within Boxwood.

The Boxwood B-tree module is a distributed implementation of Sagiv's [28] B-link tree algorithm, which is a variant of the Lehman-Yao B-link tree algorithm [22]. Sagiv's algorithm has the desirable property that locking is considerably simplified from traditional B-tree algorithms without compromising concurrency or atomicity. Sagiv's original B-link tree algorithm (like its classic B-tree, B^+-tree, or B^*-tree counterparts) runs in a single machine environment, uses locks for synchronizing accesses amongst multiple threads, and stores data either in memory or persistently on disk. Sagiv's algorithm is well suited for a distributed implementation, an observation independently made by Johnson and Colbrook [16].

Since the algorithm to implement the B-link tree can already deal with thread concurrency within a single machine, our design extends this design to multiple machines by ensuring two simple constraints are met:

- Threads executing on multiple machines use global locks for synchronizing access to shared data.
- Data stored by one thread running on a machine can be accessed by another thread on any other.

The first constraint is readily provided by our distributed lock service described in Section 3.3. To meet the second

requirement, we could use an existing virtual (or logical) disk or logical volume manager, but we decided against this for the following reasons. Existing logical/virtual disk systems would still require us to do our own management of physical/virtual space. Most systems we know of did not support our needs for fault-tolerance, incremental expansion, and scalability. The few systems that do (e.g., Petal [21] or FAB [9]), implement their own logging and recovery schemes, which duplicate much of the logging and recovery required at the B-link tree level, increasing our bookkeeping overheads, and making it difficult to implement certain optimizations that were possible in our design.

2.3 Chunk Data Store

The data store used by the B-tree abstraction is provided by a *chunk manager*. The principal function of the chunk manager is to hide the details of the physical storage media and to provide a level of address mapping so that the B-tree algorithm can deal with opaque pointers to stored data. The chunk manager acts much like a memory allocator, in that it hands out variable length chunks of the data store that can subsequently be written to, read from, or deallocated.

The chunk manager carves out chunks of storage by using the services of a lower layer called the *replicated logical device (RLDev)* layer. Each RLDev provides access to a fixed amount of chain-declustered storage. An RLDev is implemented on two machines using two separate physical disk drives for replication.

2.4 Infrastructure Services

In addition to the software modules that implement the various abstractions, Boxwood contains three important modules that provide essential distributed system services. These are heavily used within the Boxwood system to implement the abstractions, and can also be used directly by the external clients of the Boxwood system. These services are:

- **Paxos service.** This is an implementation of the Paxos part-time parliament algorithm [20]. It is used to store global system state such as the number of machines and the number of RLDevs in the system. It is also used by the distributed lock service to keep track of client information and the identity of the lock master for recovery when the lock service has a transient failure.

- **Lock Service.** This provides a distributed lock service that handles multiple-reader, single-writer locks. This service is used by the RLDev module, the chunk manager, and our multi-node NFS server.

- **Transaction Service.** This service provides a redo-undo logging facility and transaction support for the recovery of the B-tree module and in the NFS server.

3 Boxwood Design

This section describes the design of the various components of the Boxwood system in more detail. Since there are several interdependencies amongst these components, we describe them in an order that minimizes forward references.

3.1 Paxos Service

The Paxos service is a "general-purpose" implementation of the state machine approach using the Paxos part-time parliament algorithm. We refer to our service as general-purpose because clients of the service can define arbitrary client-specific state and pass client-specific "decrees" to modify and query this state by making RPC calls to the Paxos service. The state maintained by the Paxos service is replicated on a collection of independent machines. The Paxos algorithm provably guarantees that the state changes occur in the same order on each replica and that the state is available and is consistent as long as a majority of these replicas are non-faulty.

Boxwood depends on three different types of client states maintained within Paxos. Each client state typically refers to the essential state required by an internal Boxwood service such as the lock service, or the RLDev layer, or the chunk manager. This state is consulted by each service as appropriate to perform recovery or reconfiguration of that layer. Typically, the state includes the machines, disks, and other resources like network port identifiers used by the client.

The Paxos service is implemented on a small collection of machines, typically three, with two machines constituting a quorum or majority. We do not dedicate an entire machine to implementing the service; the Paxos service instance on a machine is restricted to a single server process. Paxos state is maintained on disks that are locally attached to the machines hosting the server. We do not rely on these disks to be fault tolerant, but merely that they are persistent. The failure of the disk storage or the machine is considered as a failure of the Paxos server. Our choice of three machines is arbitrary; it allows us to tolerate the failure of one machine in our small cluster. We can tolerate the failure of k machines by running the service on $2k + 1$ machines.

We draw a clear distinction between the characteristics of the Paxos service and the characteristics of the rest of the storage- and abstraction-related services. We have isolated the scaling of the overall system from the

scaling of Paxos. Client state stored on Paxos can be dynamically changed with the addition of new RLDevs, new chunk managers, new machines, or new disks. This does not require us to dynamically change the number of Paxos servers that store the state. It is conceivable that at extremely large scales, we might wish to increase on the fly the number of Paxos servers in the system to guard against increased machine failures. If so, we will need to implement the protocols necessary to increase or decrease dynamically the number of Paxos servers *per se*.

We ensure that Paxos is only involved when there are failures in the system or there are reconfigurations of the system. This allows us to avoid overloading the servers for the common case operations such as reads and writes, which need to complete quickly.

We implement a slightly restricted form of Paxos by decreasing the degree of concurrency allowed. In standard Paxos, multiple decrees can be concurrently executed. In our system, we restrict decrees to be passed sequentially. This makes the implementation slightly easier without sacrificing the effectiveness of the protocol for our purposes.

To ensure liveness properties in a consensus algorithm like Paxos, it can be shown that it is only necessary to use a failure detector with fairly weak properties [5]. In principle, we too only need such a weak failure detector. However, we need failure detectors with stronger guarantees for our RPC and lock server modules. Rather than implement multiple failure detection modules, we use a single one with more restrictions than those required by Paxos.

3.2 Failure Detector

Our failure detection module is implemented by having machines exchange periodic *keepalive* beacons. Each machine is monitored by a collection of *observer* machines with which it exchanges keepalive messages. A machine can check on the status of any machine by querying the observers. A machine is considered failed only when a majority of the observer machines have not heard keepalive messages from it for some threshold period. The invariants we maintain are that, (a) if a machine dies, the failure detector will eventually detect it, and that, (b) if the failure detector *service* (not to be confused with an individual observer) tells a client that a machine is dead, then that machine is indeed dead.

Figure 2 sketches the message protocol assuming a single observer, rather than a majority. Our messages are sent using UDP and may fail to arrive or arrive out of order, albeit with very low probability on a LAN. We do not assume a synchronized clock on each machine, but we do assume that the clocks go forwards, the clock drift on the machines is bounded, and UDP message delays are non-zero.

A client (Client A in the figure) periodically (at intervals of ΔT) sends out beacon messages to the observer. These messages may or may not be delivered reliably. The observer echoes each beacon message it receives back to the client. At any point in time, the observer considers a client dead if it has not received a beacon from the client in the preceding `GracePeriod`. The client considers the receipt of the echo as an acknowledgement from the observer. The client keeps track of the last time it sent a beacon that was acknowledged by the observer. If more than `GracePeriod` time elapses without an acknowledgement, it considers itself dead and kills itself.

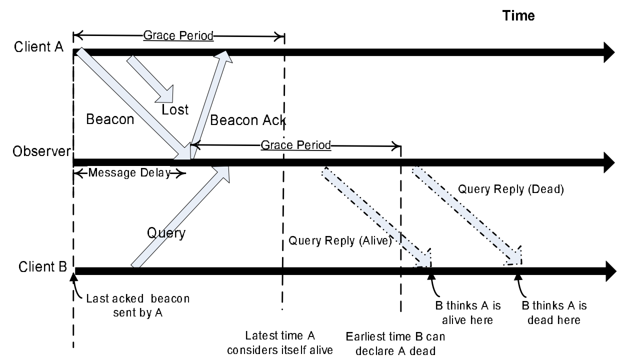

Figure 2: **Message protocol for the failure detector assuming a single observer and two clients.** Time advances to the right. If B thinks A is dead, then A must be dead.

Another client (Client B in the figure) that wishes to monitor the first client (Client A) sends a query message to the observer. The observer then sends B its view about A. If B receives a reply from the observer claiming A is dead, it considers A dead; otherwise it considers A alive. Given our assumptions about clock drift and non-zero message delay, this protocol is conservative and maintains our invariants.

In reality, we don't use a single observer, but use a collection of observers for reasons of fault-tolerance. In this case, client B in the figure pronounces A dead only if it gets replies from a majority of observers (instead of *the* observer) that all pronounce A dead. If B does not receive a majority of responses that pronounce A dead, it assumes that A is alive. A, on the other hand, considers itself dead as soon as `GracePeriod` time elapses without an acknowledgement from a majority. This protocol can lead to a state where A considers itself dead, while B thinks A is alive. But more to the point, if B thinks A is dead (because a majority pronounces A dead), then A cannot have received an acknowledgement from a majority, and will consider itself dead. This maintains our invariant that if B considers A dead (because the failure detector said so), then A must be dead.

Our protocol, as sketched, only works if clients that die do not get resurrected later and start sending beacons. We ensure this in practice by having each client use a monotonically increasing incarnation number each time it has a transient failure.

The values of `GracePeriod`, ΔT, and the number of observers are tunable parameters. We have found one second, 200 ms, and three to be suitable in our environment. We use the same observer machines for all the clients in the system for convenience, although it is feasible to use different subsets of machines as observers for each specific client.

3.3 Distributed Lock Service

The lock service provides a simple general purpose locking mechanism that allows its clients to acquire multiple-reader single-writer locks. Our design borrows techniques used in earlier work [4, 6, 12, 32]; we describe the details of our scheme to underscore our rationale for particular choices.

Locks are uniquely identified by byte arrays on which the lock service does not impose any semantics. Although locks have no explicit timeouts associated with them, the failure detector is used to time out unresponsive clients. So in essence, our locks act as degenerate leases [12].

Clients of the lock service have a clerk module linked into their address spaces. Leases are cached by the clerk and are only revoked by the service if there is a conflicting request by another clerk. A clerk blocks an incoming revocation request until all currently outstanding local uses of the lease have completed. An optimization, which we don't currently support, is for the clerk to release a lease when it has not used it for some time.

Clients can optionally arrange with the lock service to call a recovery function on another instance of the same client if the first instance were to fail. The lock service guarantees that the leases acquired by a client that has subsequently failed will not be released until this recovery is successfully completed. This is a modest extension to Gray and Cheriton's standard lease mechanism (which primarily focused on write-through client caching) to deal with residual state that exists in a client after the lease has timed out. An example of such usage can be found in the B-tree module described in Section 3.7.

The failure of a lock client is determined by the failure detector. Notice that for our scheme to work correctly, both the client and the lock service must use the failure detector in a consistent fashion. Otherwise, the lock service could revoke the lease, while the client believed it had the lease. We ensure correct behavior by requiring two conditions of our failure detector. First, if a client dies, then the failure detector will eventually notice that it is dead. Second, if the failure detector claims that a client is dead, then the client must have died some time prior (but perhaps is alive now if it was a transient failure). We also ensure that a client that comes alive and finishes recovery always assumes that it holds no leases and registers with the lock service.

For fault-tolerance, the lock service consists of a single master server and one or more slave servers running on separate machines. In our cluster, we typically use only a single slave server, but if multiple machine failures are common, additional slaves can be used. Only the master server, whose identity is part of the global state in Paxos, hands out leases. The lock service also keeps the list of clerks as part of its Paxos state.

If the failure detector pronounces the current master dead, the slave takes over after passing a Paxos decree that changes the identity of the current master. The new master recovers the lease state by first reading Paxos state to get a list of clerks. Then it queries the clerks for their lease state. It is possible that some of the clerks are dead at this point. In this case, the lock service calls recovery on behalf of these clients on the clients that are alive, and considers all leases held by the dead clients as free. If no recovery procedure has been established for a dead client, the lock service considers all leases held by that client as free.

Lock service clerks query Paxos to determine the identity of the master server. This information is cached until an RPC to the currently master returns an exception, at which point it is refreshed. RPCs to a machine return an exception if the failure detector claims the target is dead.

The lock service fails if all (both) servers fail. Clients cannot make forward progress until it is re-established.

We use a simple master-slave design for the lock service because we believe other more elaborate, scalable schemes are unnecessary in most storage-centric environments. Our rationale is that even in elaborate schemes with several active lock servers, a specific lock will be implemented by a single lock server at any given time. If this is a highly contended lock, then lock contention due to data sharing becomes a performance problem on the clients long before the implementation of the lock server itself becomes a bottleneck. Our experience with deploying Petal/Frangipani, which had a more scalable lock service, seems to bear this out.

3.4 RLDevs: Replicated, Logical Devices

Boxwood implements storage replication through a simple abstraction we call a *replicated, logical device* (RLDev). An RLDev is logically a block device in that it expects block-aligned accesses in multiples of block units.

We chose to implement replication at a fairly low level in the abstraction hierarchy for two principal reasons. First, by providing replication at a low level, all higher layers, which are typically more complex in nature, can depend on fault-tolerant storage, which makes the logic of the higher layers simpler to reason about. For instance, our implementation of the chunk manager (to be described in Section 3.5) was considerably easier because of the RLDev layer. Second, by replicating at a low level of abstraction, the relevant replication, mapping, and failover logic, as well as internal data structures, can be made simple.

RLDevs implement chained declustering. A single RLDev is of fixed size and consists of two *segments* of equal size located on disks on two different machines. A single disk will contain segments from multiple RLDevs. The list of RLDevs, the segments belonging to them, the identity of machines that host the primary and the secondary segments, and the disks are all part of the global state maintained in Paxos. If an RLDev is added or if the locations of the segments belonging to an RLDev are changed, a Paxos decree must be passed.

The replication protocol is fairly standard. One replica is designated the primary, and the other the secondary. On initialization, a replica reads its state from the Paxos service and monitors its peer using the failure detector. When both replicas are up, writes are performed on both and reads on either. A client sends write requests to the primary, which forwards the request to the secondary and waits for completion.

Clients of the RLDev use hints to determine where the replicas are located. Hints can sometimes be wrong and can be updated by reading the Paxos state. An RLDev clerk linked in with the clients handles the details of dealing with hints and refreshing them as appropriate.

When one of the replicas dies, the surviving replica continues to accept writes (and reads). We call these *degraded mode* writes because the system is accepting new data but not replicating it on the dead replica. A subsequent failure of the surviving replica (the one that has accepted the degraded mode data) before the first replica has finished recovering will lead to data loss. Before it accepts these "degraded mode" writes, the survivor passes a decree to that effect so that if the dead replica were to come back after a transient failure, it knows to reconcile its stale data, and more importantly, not to accept new reads or writes if it cannot reconcile its stale data. This can happen because the replica that was working in "degraded mode" now happens to be dead. All blocks that have degraded mode writes on them are put in a log (called the *degraded mode log*) so that reconciliation is fast and only involves the affected blocks.

Notice that in the worst case, the entire segment could have been written in degraded mode. Thus, when a previously dead replica comes up, we have to be prepared to copy the entire segment. We can leverage this mechanism to implement the automatic *reconfiguration* of an RLDev. By a reconfiguration operation, we mean the redistribution of the data in an RLDev to a different disk and/or machine to enable load balancing. Making an RLDev relatively small makes it easy to copy its data quickly on a high-bandwidth LAN link, thereby cutting down reconfiguration time.

As mentioned previously, when both replicas are up, the primary waits for the secondary to commit the write before returning to the client. In order to cope with a transient failure when the writes are in flight, an RLDev implements a *dirty region log*, which serves a different purpose than the degraded mode log mentioned previously. The dirty region log on the primary keeps track of writes that are in flight to the secondary. When the secondary replies, the log entry can be removed from the primary in principle. To recover from a transient failure, the primary consults its dirty region log to determine the writes that were in flight and sends the secondary the current contents of its disks for these writes.

In cases where the client can deal with the two replicas differing after a crash, RLDevs allow the client to turn off the dirty region log. Such clients must explicitly read and write each replica to reconcile the differences after a crash. In Boxwood, all such clients already maintain a log for other reasons, and there is no added cost for maintaining the equivalent of the dirty region information, except a modest violation in layering. We felt this tradeoff was justified for the performance gain of saving an additional disk write.

Recovering an RLDev is fairly straightforward. There are two failure cases: a permanent failure of a disk or the transient failure of processor and its attached disks. When there is a permanent disk failure, the RLDevs that are hosted on that disk must be reconstituted on a new (or perhaps more than one) disk, but the recovery of each RLDev proceeds independently. An RLDev recovering from a permanent disk failure contacts the RLDev on the machine that hosts its surviving segment and copies the contents of the entire segment. In contrast, if the failure was transient, then the data retrieved from the peer is limited to any degraded mode writes the peer has for the recovering segment. After degraded mode writes have been applied on the recovering segment, the dirty region log of the surviving segment is read and sent to the recovering server to apply any in-flight writes. When the recovering segment is up to date, the recovering server passes another Paxos decree indicating that the state of the RLDev is normal and stops recording degraded mode writes.

3.5 The Chunk Manager

The fundamental storage unit in Boxwood is the *chunk*. A chunk is a sector-aligned sequence of consecutive bytes on an RLDev, allocated in response to a single request. Every chunk is identified by an *opaque handle* that is globally unique in the system.

Chunks are managed by the chunk manager module, which supports four operations: *allocate*, *deallocate*, *read*, and *write*. Deallocated handles are guaranteed never to be reused. Reading or writing an unallocated handle raises an exception.

Since an RLDev can be accessed by any machine with a suitable clerk linked in, it is possible, in principle, to have any chunk manager allocate chunks from any RLDev. In fact, as long as a single chunk manager is alive, we can manage all the RLDevs that are non-faulty. However, for simplicity, ease of load balancing, and performance, we designate a pair of chunk managers running on two different machines to manage space from a given set of RLDevs. Typically, these RLDevs will have their primary and secondary segments located on disks that are local to the two chunk managers. This reduces the number of network hops required to perform chunk operations. One of the pair of chunk managers acts as the primary initiating all allocations and deallocations, while either can perform reads and writes.

The mapping between the opaque handle and the RLDev offset is replicated persistently on RLDevs. This mapping information is accessed often: an allocate call requires a new mapping to be created; a deallocate call deletes the mapping; reads and writes require this mapping to be consulted. We therefore cache the mapping on both the primary and the secondary.

A *map lock* from the lock service protects mappings. Only the primary makes changes to the mappings; this lock is therefore always cached on the primary in exclusive mode. When the primary changes the mappings, it writes the new mapping to the RLDev and sends an RPC to the secondary, which directly updates its cached copy of the mappings without acquiring the lock.

On startup, the secondary has to read the latest mapping state from the RLDev so that subsequent RPCs from the primary can update it correctly. In order to get the latest state from the RLDev, the secondary acquires the *map lock* in shared mode, reads the mapping from the RLDev, releases the lock, and never acquires it again. The primary, on the other hand, always acquires the *map lock* in exclusive mode before making mapping changes. The ordinary locking mechanism will then ensure the consistency of the data.

If the primary dies, the secondary will notice it via the failure detector, whereupon it acquires the *map lock* in exclusive mode and acts as a primary. If the secondary dies, the primary will also detect it via the failure detector. It continues to update the mappings on the RLDev, but does not make RPCs until it gets a revocation for the *map lock* indicating that the secondary has come alive and wants to read the state.

Our design of the chunk manager is very simple, largely because of our decisions to (a) implement replication below the level of the chunk manager, and (b) use the locking service to do the failover.

Figure 3 shows the relationship of the chunk manager and the RLDev layer.

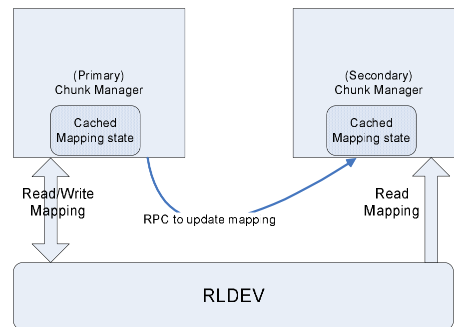

Figure 3: **The chunk manager pair relies on a shared RLDev and RPCs to keep the mapping information consistent.**

Mapping From Opaque Handles to Disk Offsets

An opaque handle consists of a 32-bit chunk manager identifier and a 64-bit handle identifier. A chunk manager identifier corresponds to a pair of chunk managers. The locations of the primary and secondary chunk manager, which may change over time, are maintained in Paxos and cached by the chunk manager clerk. The clerk uses this mapping to direct chunk requests to an appropriate manager. If the cache is out of date, there is no correctness issue with misdirecting the RPC because the incorrect chunk manager will return an error, which causes the clerk to refresh its mappings.

The translation of the handle identifier to the RLDev is performed by the chunk manager responsible for the handle. It consults its cached mapping table to translate to the RLDev offset. It then invokes the RLDev clerk, which accesses the physical disks as necessary. In most cases, at least one segment of the RLDev will be on a local disk, so accesses incur little network overhead.

The chunk manager module guarantees that an opaque handle is never reused. It enforces this condition by never reusing a handle identifier once it is deallocated. The 64-bit identifier in our current implementation seems adequate for this purpose; but we may reconsider that decision after we gain more experience with the system.

The mapping between handles and disk offsets is stored as stable state on an RLDev. The state on the RLDev consists of a checkpoint and a separate list of incremental changes that have not been applied to the checkpoint. The list is updated synchronously whenever the mapping changes, but the checkpoint is only accessed infrequently. We periodically apply the changes in the list to the checkpoint and truncate the list.

Recovering the chunk manager is simple as long as the RLDev holding the stable state is available. The primary chunk manager applies the list of incremental changes to the current checkpoint to get the updated state, which it caches in memory and stores stably on disk. Then it truncates the list and is ready to service new requests. Since the chunk manager depends on the RLDev for its recovery, the RLDevs have to be recovered first.

3.6 Transaction and Logging Service

Boxwood provides simple transaction and logging support to clients. We provide logging to perform both *redo* and *undo* operations of transactions. Logs are duplexed by storing them on an RLDev so that they are resilient to single failures and universally accessible from any machine. Thus, recovery for a service can be done on any machine. The logging system supports group commits and also allows clients the option of selectively flushing the volatile in-core log to disk on every transaction commit.

Our transaction system does not provide isolation guarantees; instead clients explicitly take out locks using the lock server. Clients do deadlock avoidance by using lock ordering. With the current set of clients in our system, deadlock avoidance is the more attractive alternative to providing deadlock detection as in a traditional transaction system.

3.7 The B-tree Module

We assume the reader is familiar with the B-tree [3], and its variant the *B*-tree* [33], which has the same structure as a B-tree, except that all genuine keys reside in the leaf nodes. Non-leaf nodes contain shadow keys acting purely as an index for finding the desired key in a leaf. A *B-link tree* [22] is a B*-tree with one extra pointer per node: this extra link points to the next node of the tree at the same level as the current node. The extra links make it trivial to enumerate keys in a B-link tree (one just follows the extra links from one leaf node to the next). But even more importantly, the extra links make efficient concurrent operations possible. We do not give details here, but the main intuition is that certain operations can recover from unexpected situations by following the extra links, so some concurrent operations that would otherwise require several locks can proceed with fewer locks.

The B-link tree, together with algorithms for efficient concurrent operations, was first introduced by Lehman and Yao [22], and significantly improved by Sagiv [28]. Sagiv's algorithms require no locks for lookups. Insertions require only a single lock held at a time even if the insertions cause node splits at many levels of the tree. Each lock protects a single tree node that is being updated. Deletions are handled like insertions, and hold a single lock on the updated node. Thus, these operations are provably deadlock free. Deletions can leave a node empty or partially empty; and such nodes are fixed up as a separate activity by one or more background compression threads. Each compression thread must lock up to three nodes simultaneously. Each B-link tree operation provides all ACID properties when only a single tree is involved. ACID properties across multiple trees must be maintained by clients using the transaction and locking services described earlier.

We implement a distributed version of Sagiv's algorithm in a conceptually simple fashion by using the global lock server instead of simple locks, and by using the chunk manager for storage instead of ordinary disks. Each instance of the B-tree module maintains a write-ahead log on an RLDev.

Recovering the B-tree service is done by replaying the write-ahead log. The records in the log refer to handles implemented by the chunk manager. Thus, before the B-tree service can be recovered, both the RLDev service and the chunk manager must be available. If a B-tree server were to crash, the active log records (i.e., records for those operations that haven't made it to disk) will be protected by leases. When the leases expire, the lock service will initiate recovery using some other B-tree server and the write-ahead log. (Recall from Section 3.3 that clients of the lock service can designate a recovery function to be invoked on lease expiration, when the lock service will consider the client as having failed.) When recovery is complete, the expired leases are made available for other servers to acquire. It is possible that all B-tree servers are dead, in which case, the lock server will not be able to call recovery on any machine when the leases expire. It therefore defers the recovery until the first B-tree server registers with it, at which point all server logs will be replayed before any new leases are handed out.

4 Performance of the Boxwood Prototype

4.1 Experimental Setup

Boxwood is deployed in our lab on a cluster of eight machines connected by a Gigabit Ethernet switch. Each machine is a Dell PowerEdge 2650 server with a single

2.4 GHz Xeon processor, 1GB of RAM, with an Adaptec AIC-7899 dual SCSI adapter, and 5 SCSI drives. One of these, a 36GB 15K RPM (Maxtor Atlas15K) drive, is used as the system disk. The remaining four 18GB 15K RPM drives (Seagate Cheetah 15K.3 ST318453LC) store data.

Each machine executes the Boxwood software, which runs as a user-level process on a Windows Server 2003 kernel. The failure detector observers, the Paxos service, and the lock service run as separate processes; the rest of the layers are available as libraries that can be linked in by applications. Boxwood software is written in C#, a garbage-collected, strongly typed language, with a few low-level routines written in C.

The networking subsystem provided by the kernel is capable of transmitting data at 115 MB/sec using TCP. Using a request response protocol layered on TCP, our RPC system can deliver about 110 MB/sec.

4.2 RLDev Performance

On each machine we run an RLDev server, which manages several RLDevs on its locally attached disks. For some of these RLDevs, the server is a primary and for others it is a secondary. We measure the performance using a simple benchmark program that performs read and write accesses.

Size bytes	Write			Read		
	Xput MB/s	Util. % CPU	Util. % Disk	Xput MB/s	Util. % CPU	Util. % Disk
512	0.3	2	99	0.3	20	100
8192	5	9	99	5.3	15	99
64K	40	73	98	48	70	85
256K	110	52	76	110	70	60

Table 1: **Aggregate random write and read throughput on a two machine configuration**. Disk queue length is constrained to be one. Each write request involves one local disk write and one remote disk write. Each read request involves one RPC call and a disk read on a remote disk.

Table 1 shows the performance of replicated random writes and reads in the smallest (two machine) configuration. All the write accesses made by the benchmark are to RLDevs where the primary server is local. Thus, each write will result in a local disk write and an RPC to a remote server to update the mirror. All read accesses are made to a remote RLDev server. CPU utilization represents the average CPU usage on each server. We constrain the kernel disk access routines to allow only a single outstanding request per disk. This represents the most pessimistic performance.

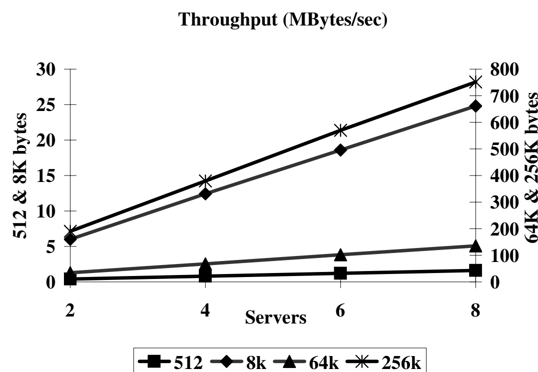

Figure 4: **Scaling of writes in the RLDev.**

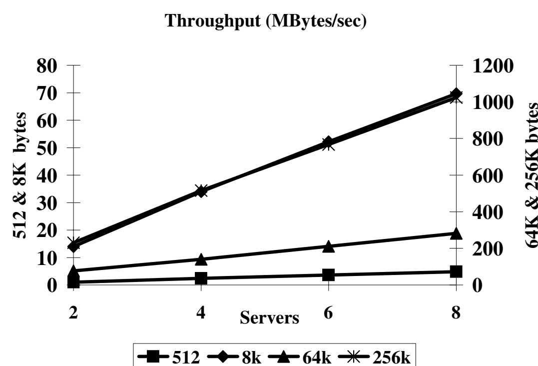

Figure 5: **Scaling of reads in the RLDev.**

Figures 4 and 5 show the scaling as the number of servers is increased from 2 to 8. Read are slightly faster than writes in general. This is because the disk inherently has slightly better (about 5%) read performance than write according to the manufacturer's specification sheet. At small packet sizes, the throughput is limited by disk latency. For large packet sizes, we get performance close to the RPC system imposed limit. In all cases, we observe good scaling.

4.3 Chunk Manager Performance

We next discuss the performance of the chunk manager. The read and write performance of the chunk manager closely matches the performance of the RLDev described above and is not repeated here. We instead describe the performance impact of allocations and deallocations. Our chunk manager implements a performance optimization that defers some of the work of an allocation to a later time. When possible, we log the allocation locally on the client, but avoid an RPC to the server. A whole set of allocation requests are subsequently sent in a batch to the server.

Table 2 shows the performance effect as the amount of batching is varied. As part of allocating a handle, the

Batch Size	Amortized Latency (ms)
1	24
10	3.3
100	1.0
1000	1.0

Table 2: **Effect of batching on allocations.** Each allocation is for a single 8KB region, which is zeroed out on disk. Latency is amortized over the batch size.

storage allocated for it is zeroed out on disk. Thus allocation costs will vary somewhat with chunk size; we show a typical size of 8KB. For single allocations (i.e., with no batching), the latency cost is high. This is because we require three RLDev writes to stably record the new mapping. However, batching is very effective even at small numbers; we typically more than a dozen allocations that can be batched in our tests.

Deallocations are somewhat easier than allocations. The cost of a single deallocation as seen by a client is about 5.3 milliseconds and is independent of the number of servers. In addition, if a handle whose allocation request has not yet been sent to the server is deallocated, we can avoid making both operations at the server. This brings the time down even further.

4.4 B-tree Performance

We report on the performance of two sets of experiments for the B-tree. In the first experiment we have a number of machines each inserting keys into a separate B-tree. In a subsequent phase, each machine looks up these keys and in a final phase deletes them. Before each of the three phases, all entries in the B-tree caches are evicted. Since the B-trees are private, there is no lock contention, but there is contention for storage on the underlying RLDevs, which are evenly distributed across all machines.

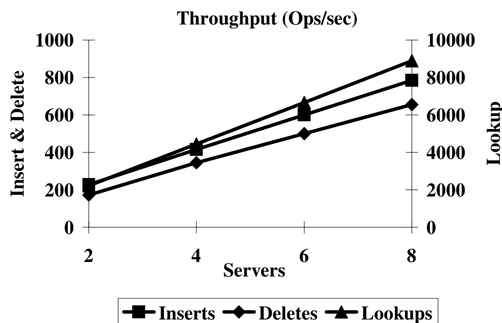

Figure 6: **Performance of B-link tree operations.** Each machine performs operations on a separate tree, starting with a cold cache.

Figure 6 reports on the aggregate steady state throughput across all machines. For all phases, we observe good scaling within experimental error. This is not surprising since there is no contention for the locks, and the RLDev performance is known to scale from Figures 4 and 5.

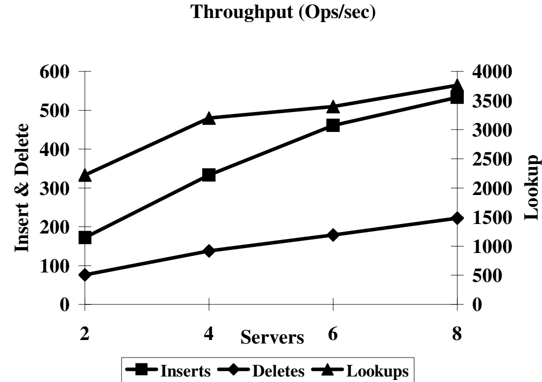

Figure 7: **Performance of B-link tree operations.** All machines perform operations on a single shared tree, starting with a cold cache.

We next study the performance of B-tree operations under contention. In this experiment, we redo the previous experiment but use a single B-tree that all machines contend for. To avoid the extreme case where every insertion contends for a global lock, we partition the keys so that each client acts on a disjoint region of the key space. This arrangement still leads to lock contention in the upper levels of the tree. Once again we perform three phases with cold caches. Figure 7 shows the aggregate throughput across all machines. Under contention, the performance of the B-link operations tends to flatten relative to Figure 6. Also, relative to the first experiment, the size the shared tree is proportionately larger in the second experiment. This leads to lowered hit rates in the cache, which slows the performance of lookups in the second case.

In general, the performance of B-trees is dependent on several parameters, which include the branching factor, the distribution of keys, and the size of the keys and data. We have not yet done an exhaustive characterization of our B-tree module at various parameter values. The particular trees we measure have 10-byte keys and data and each tree node has a branching factor of 6.

5 BoxFS: A Multi-Node NFS Server

The previous sections showed the functional characteristics of Boxwood's abstractions. To test these abstractions, we built BoxFS: a file system using Boxwood B-trees, exported using the NFS v2 protocol. It runs at user-level and directly or indirectly depends on all the services

described earlier.

BoxFS implements a conventional file system containing directories, files, and symbolic links. Each of these is stored as a Boxwood B-tree. Each tree contains a single well-known *distinguished key* to hold attributes such as access times, protection and owner information similar to a UNIX *inode*. In addition, directories, files, and symbolic links also have *regular keys* as described below.

A B-tree storing a directory is keyed by the name of the file, directory, or symbolic link. The data corresponding to the key refers to a byte array that is the on-wire representation of the NFS file handle. The NFS file handle is not interpreted by the client, but BoxFS stores in it bookkeeping information including the B-Tree handle that represents the file, directory, or symbolic link.

A B-tree storing a file is keyed by block number. The data corresponding to the key is an opaque chunk handle. The actual data in the file is stored in the chunk as uninterpreted bytes. Our usage of B-trees for directories and chunks for file data is consistent with the idea that strict atomicity guarantees are needed in the file system metadata, but not in the file user data. Alternatively, we could have stored user data as part of the file B-tree, but every file write would translate into a relatively expensive B-tree insert operation, with attendant overheads of redo and undo logging.

A B-tree storing a symbolic link is a degenerate tree containing a single special key and the byte array referring to the target of the link.

Obviously, all the B-trees are maintained via Boxwood B-tree operations — and any single such operation has all the ACID properties even when the same B-tree is simultaneously being accessed by different machines. However, many file system operations require multiple B-tree operations to be performed atomically. For example, a *rename* operation requires the combination of a delete from one B-tree and an insertion into another B-tree to be atomic. This atomicity is achieved using the transaction service described in Section 3.6. But as we explained earlier, clients have to provide isolation guarantees themselves. BoxFS ensures isolation by acquiring locks on the appropriate file system objects from the Boxwood lock service in a predefined lock order. After the locks are acquired, it performs the required operation as part of a transaction, commits, and releases the locks.

All B-tree operations benefit from the B-tree cache built into Boxwood's B-tree module. In contrast, file data is not stored in B-trees; it is accessed via the chunk manager interface and requires a separate cache within BoxFS. This cache is kept coherent by acquiring the appropriate shared or exclusive lock from the Boxwood lock service for every data read or write.

Typical file operations that do not involve user file data access result in one or more B-tree operations encapsulated within a single transaction. These B-tree operations will generate log entries that are committed to the log when the transaction commits. Where possible the transaction system will do a group commit. In any event, a single transaction commit will cause only a single disk write in the usual case unless the log is full. The actual B-tree nodes that are modified by the transaction will remain in the B-tree's in-core cache and do not make it to disk. Thus, BoxFS can perform metadata intensive file operations efficiently.

BoxFS makes three simplifying assumptions to achieve acceptable performance. We do not believe these assumptions materially affect the usability of our system or the validity of our hypothesis that it is easy to build file systems given higher-level abstractions. By default, the data cache is flushed once every 30 seconds, which is not strictly in accordance with NFS v2 semantics. Also, the B-tree log that contains the metadata operations is flushed to stable storage with the same periodicity. Thus, file system metadata is consistent, but we could lose 30 seconds of work if a machine crashes. Finally, we do not always keep the access times on files and directories up to date.

The BoxFS system runs on multiple machines simultaneously and exports the same file system. The file system is kept consistent by virtue of the distributed lock service. However, since we are exporting the file system using the NFS protocol, clients cannot fully exploit the benefits of coherent caching.

Locking in BoxFS is quite fine-grained. Since all file system metadata is stored as key-data pairs in a B-tree, metadata modifications made by different machines are automatically locked for consistency by the B-tree module through the global lock server. In addition, BoxFS protects individual file blocks by taking out locks for each block. Thus, two machines writing different blocks in the same file will contend for the metadata lock to update the file attributes, but not for the individual block. This reduces lock contention as well as cache coherence traffic due to false sharing.

Using the Boxwood abstractions enabled us to keep our file system code base small. The actual file system code is about 1700 lines of C# code, which is a more verbose language than C. The BoxFS code implementing a simple LRU buffer cache is an additional 800 lines, which is largely a reuse of the code in the B-tree cache module. The size of the code compares favorably with Mark Shand's classic user-level NFS daemon, which is about 2500 lines of C for read-only access to a UNIX file hierarchy [30]. In addition, we wrote code to support NFS RPC (both UDP and TCP) and XDR in C#, which accounted for an additional 2000 lines, and about 1000 lines of C to interface BoxFS to the native Windows networking libraries.

5.1 BoxFS Performance

Table 3 shows Connectathon performance benchmarks (available at http://www.connectathon.org/nfstests.html) for BoxFS. For comparison we show the performance of a stock NFS server running on the same hardware. We run BoxFS on a single machine in our cluster. The RLDevs are on four locally attached disks and have no replication enabled. The stock NFS server is the Windows 2003 Services for Unix (SFU) NFS server that runs in-kernel on the NTFS local file system on the same four local disks. Our client is a Linux 2.4.20-8 NFS client in both cases. We mount the file system so that NFS RPCs are made with UDP send and receive sizes of 8KB, the maximum supported by the kernel.

Description	**Time (secs)**	
	BoxFS	**NFS**
Create 155 files and 62 dirs	0.7(1.9)	1.0
Remove 155 files and 62 dirs.	0.5(1.7)	0.4
500 Getwd and stat calls	0.1(0.1)	0.8
1000 Chmods and stats	1.5(4.8)	8.4
Write a 1MB file 10 times	3.7(12.4)	10.8
Read a 1MB file 10 times	0.2(0.2)	0.2
Reads 20500 dir. entries and 200 files and unlinks them	1.5(2.7)	0.6
200 Renames and links on 10 files	0.9(2.3)	1.9
400 Symlinks and readlinks on 100 files	1.4(3.7)	6.7
1500 Statfs calls	1.2(1.3)	1.1

Table 3: **Connectathon benchmarks**. BoxFS writes file data asynchronously. The test that reads a 1MB file repeatedly is an anomaly because the reads hit in the cache on the client. The numbers in parenthesis indicate performance when the in-core metadata log is flushed to disk on each transaction commit instead of periodically.

We show the performance of BoxFS when the metadata log is lazily flushed to disk and also when it is not. In general, the performance of BoxFS is comparable to the native NFS server on top of a local NTFS file system. We have a slight edge on some meta-data intensive operations, which we attribute to our usage of B-trees. Depending on whether we synchronously update the log or not, the fifth test shows a marked difference (12.4 versus 3.7 seconds) even though file data is being updated asynchronously in both cases. This is because file write requires a metadata update, which involves a log write.

To show the benefit of lazily updating the metadata log where possible, we report our results from running PostMark, a metadata intensive benchmark. This benchmark models the expected file system work load imposed by electronic mail, web based commerce, and netnews [18]. It creates an initial set of files of varying sizes. This measures the performance of the file system in creating small files. Next it measures the performance of a predetermined number of "transactions", where a transaction is either the creation or deletion of a file followed by either a read or append to a file. We ran the benchmark with the settings shown in Table 4. Our results are shown in Table 5. Since the benchmark emphasizes metadata intensive operations, BoxFS performs well.

Parameter	**Value**
Num. Initial Files	500
Num. of Transactions	500
File Sizes	0.5–9.8KB
Create/Delete Ratio	5
Read/Append Ratio	5
Read/Write Block Size	512

Table 4: **Postmark parameters**. We use Unix buffered I/O and use a default random number seed of 42.

Metric	**BoxFS**	
	Async.	**Sync.**
File Creations/sec	63	27
File Read/sec	81	24
File Append/sec	85	25
File Deletions/sec	63	27
Data Read KB/sec	116	50
Data Write KB/sec	379	162

Table 5: **Postmark Results**. We gain substantial performance by flushing the meta data log to disk periodically, without significant change in the semantics of the file system.

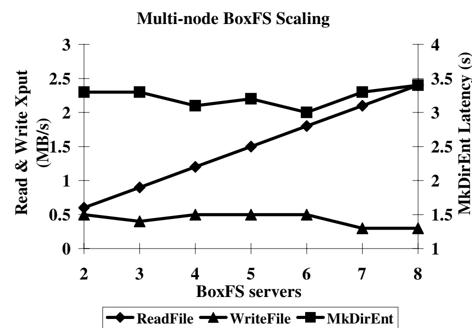

Figure 8: **Performance of BoxFS sharing experiments.** MKDirEnt Latency is the elapsed time to insert all 100 files.

Since benchmarks for multi-node file systems like BoxFS are difficult to obtain, we show scaling characteristics of BoxFS on three simple experiments in Figure 8. In all three experiments, we run the Boxwood abstractions on all 8 machines in our cluster. We export a *single* shared NFS volume from one or more of these machines.

By the nature of NFS mounting, the single volume appears as separate file systems to the client.

In the first experiment (*ReadFile* in Figure 8), a single 1 MB NFS file is read from multiple mount points, with a cold client cache and a cold server cache. As we would expect, the aggregate throughput increases linearly as the number of servers is increased since there is no contention.

In the second experiment (*MkDirEnt* in the figure), we create 100 files with unique names in the root directory of each NFS volume. This results in NFS Create RPC requests to each BoxFS machine, which in turn modifies the B-tree corresponding to the single shared directory. There are two potential sources of conflict traffic. First, we are inserting into a shared B-tree, which will result in locking and data transfer in the B-tree module; second, BoxFS has to acquire a lock to ensure that the metadata update and the B-tree insertion is consistently done. Since our experiments in Figure 7 show that we can perform in excess of 300 insertions a second into the B-tree, we believe the performance bottleneck is the traffic to keep the metadata up to date. For comparison, the performance when there is no sharing (single machine case) is 0.6 ms.

In the final experiment (*WriteFile* in the figure), we write data to a shared file, but at non-overlapping offsets. Each write results in an NFS Write RPC to a different BoxFS machine. Since BoxFS implements fine-grained locking at the file block level, the coherence traffic and lock contention, which determines scaling, is limited to what is required to keep the file metadata up to date. For comparison, the performance in the single machine (no contention) case is 4 MB/sec.

6 Related Work

Some early seeds of our work are present in the "scalable distributed data structures" (SDDSs) of Litwin *et al.* [24, 25, 26]. Litwin's SDDSs offer algorithms for interacting in a scalable fashion with a particular family of data structures. Our focus in Boxwood is significantly different, in that we are concerned with systems issues such as reliability and fault-tolerance in general abstractions—issues that were largely ignored in Litwin's work.

The "scalable distributed data structure" approach of Gribble *et al.* [13] is the previous work most similar to Boxwood. We view our work as complementary to theirs. The implementation of different data structures (hash tables versus B-trees) offers very different tradeoffs. Furthermore, our failure models are different, which also lead to very different system design techniques. Gribble *et al.* designed their system assuming the availability of uninterruptible power supplies, so transient failures that take down the entire cluster are extremely rare. They rely on data always being available in more than one place (in RAM or on disk on multiple machines) to design their recovery protocols.

Production systems like BerkeleyDB [15] and DataBlade are related to our approach and were sources of early inspiration for our work. These systems offer much more functionality than we do at the moment, but neither is distributed.

Petal [21] is another related approach to scalable storage. Petal provides similar durability and consistency guarantees to Boxwood, but it presents applications with a single sparse address space of blocks. Applications must coherently manage this space themselves, and implement their own logging and recovery mechanisms for any data structures they employ. Many of the basic services that Petal uses are similar in spirit to ours. Similarly, the overall structure of the B-link tree service is reminiscent of Frangipani [32].

Distributed databases are another approach to scalably storing information (e.g., [23]). However, we view our system as lower-level infrastructure that we hope database designers will use (and we plan to use it in this way ourselves). We have deliberately avoided mechanisms that we felt are unnecessary in the lower levels of storage architecture: query parsing, deadlock detection, and full-fledged transactions.

Distributed file systems also enable storage to scale in some respects (e.g., [2, 10] to name but two). But, once again, our view is that these will be layered over the facilities of Boxwood.

Boxwood's layering of a file system on top of B-trees is related to Olson's Inversion File System, which is layered on top of a full-fledged Postgres database [27]. The Inversion File System is capable of answering queries about the file system that our implementation cannot. Since our file system is layered on a simpler abstraction, all things being equal, we expect a performance increase at the expense of reduced querying capability.

Recent work on Semantically Smart Disks [31] is also related to Boxwood. Like Boxwood, semantically smart disks strive for better file system performance by exploiting usage patterns and other semantics within the lower layers of the storage system. Unlike Boxwood, semantically smart disks present a conventional disk interface (e.g., SCSI) to higher levels, but try to encapsulate (or infer) semantic knowledge about the file system to enhance performance.

Many projects, including FARSITE [1], OceanStore [19], and the rapidly-growing literature on distributed hash tables (e.g., [34]), have attempted to provide scalable storage over a wide area network. These all address a set of trade-offs quite different to those of Boxwood. The wide-area solutions must deal

with untrusted participants, frequent reconfiguration, high network latency, and variable bandwidth. Thus, these solutions do not take advantage of the more benign conditions for which Boxwood was designed.

There are also many projects using local area clusters to provide scalable services or persistent distributed objects. Typically, these rely on a file system or database to manage their storage (e.g., [7, 8]) or build an application-specific storage service (e.g., [29]) and do not offer easily-utilized, atomically-updating data structures.

Network attached secure disks (NASD) [11] are also related to Boxwood in that they provide a storage abstraction at a higher level than raw blocks.

Our choice of the B-link tree was motivated by the observation that it is well suited for distributed implementation. A similar observation was made independently by Johnson and Colbrook [16], who proposed B-link tree variants called DE- and DB-trees for shared memory multiprocessors. Their scheme explicitly replicates internal nodes on specific processors and stores contiguous sets of keys on a processor. Our implementation is more dynamic and our algorithm is simpler at the cost of having a distributed lock service.

7 Conclusions

Our initial experience indicates that using scalable data abstractions as fundamental, low-level storage primitives is attractive. To be sure, it appears difficult to settle on a single, universal abstraction that will fit all needs. However, our particular combination of abstractions and services seem to offer a sound substrate, on top of which multiple abstractions may be readily built.

Our use of the chunk manager as a generalized storage allocator obviating the need for address space management elsewhere in the system seems to be widely applicable. In our case, it enabled us to distribute a complicated data structure with modest programming effort.

Our strategy of isolating the use of the Paxos module to a relatively small part of the system has worked well in practice. It allows us to continue to scale the rest of the system dynamically without much hindrance.

Does the design of BoxFS support our claim that building scalable applications using the Boxwood infrastructure is easy? This is a subjective question, but our feeling on this is positive. BoxFS is a distributed file system with good resilience and scalability. Yet it does not require complicated locking, logging, or recovery schemes, making it very easy to implement. We need further performance analysis to make more objective comparisons.

Acknowledgements

We wish to thank current and former colleagues Andrew Birrell, Mike Burrows, Úlfar Erlingsson, Jim Gray, Ed Lee, Roy Levin, and Mike Schroeder for many interesting discussions related to the ideas in this paper. Ahmed Talat, Roopesh Battepati, and Ahmed Mohamed helped us understand Windows networking and SFU/NFS. Qin Lv and Feng Zhou, who were interns during two successive summers, helped us understand several subtle issues with our design. We also thank our anonymous reviewers and our shepherd, Jason Flinn, for their comments on the paper.

References

[1] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. 5th Symp. Operating Systems Design and Implementation (OSDI)*, pages 1–14, December 2002.

[2] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless network file systems. In *Proc. 15th Symp. Operating Systems Principles (SOSP)*, pages 109–126, December 1995.

[3] R. Bayer and C. McCreight. Organization and maintenance of large ordered indexes. *Acta Inf.*, 1(3):173–189, 1972.

[4] W. M. Cardoza, F. S. Glover, and W. E. Snaman, Jr. Design of the TruCluster multicomputer system for the Digital UNIX environment. *Digital Technical Journal*, 8(1):5–17, 1996.

[5] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM (JACM)*, 43(4):685–722, July 1996.

[6] M. Devarakonda, B. Kish, and A. Mohindra. Recovery in the Calypso file system. *ACM Transactions on Computer Systems*, 14(3):287–310, August 1996.

[7] P. Ferreira, M. Shapiro, X. Blondel, O. Fambon, J. Garcia, S. Kloosterman, N. Richer, M. Robert, F. Sandalky, G. Coulouris, J. Dollimore, P. Guedes, D. Hagimont, and S. Krakowiak. PerDiS: Design, implementation, and use of a persistent distributed store. In *Recent Advances in Distributed Systems*, volume 1752 of *Lecture Notes in Computer Science*, chapter 18, pages 427–452. Springer-Verlag, February 2000.

[8] A. Fox, S. Gribble, Y. Chawathe, E. Brewer, and P. Gauthier. Cluster-based scalable network services. In *Proc. 16th Symp. Operating Systems Principles (SOSP)*, pages 78–91, October 1997.

[9] S. Frølund, A. Merchant, Y. Saito, S. Spence, and A. Veitch. FAB: Building reliable enterprise storage systems on the cheap. In *Proc. 11th Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2004.

[10] S. Ghemawat, H. Gobioff, and S. Leung. The Google file system. In *Proc. 19th Symp. Operating Systems Principles (SOSP)*, pages 29–43, December 2003.

[11] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proc. 8th Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 92–103, October 1998.

[12] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proc. 12th Symp. Operating Systems Principles (SOSP)*, pages 202–210, December 1989.

[13] S. Gribble, E. Brewer, J. Hellerstein, and D. Culler. Scalable, distributed data structures for internet service construction. In *Proc. 4th Symp. on Operating Systems Design and Implementation (OSDI)*, pages 319–332, October 2000.

[14] H. Hsaio and D. J. DeWitt. Chained declustering: A new availability strategy for multiprocessor database machines. In *Proc. of the 6th International Conference on Data Engineering*, pages 456–465, February 1990.

[15] SleepyCat Software Inc. *Berkeley DB*. New Riders Publishing, 2002.

[16] T. Johnson and A. Colbrook. A distributed data-balanced dictionary based on the B-link tree. Technical Report MIT/LCS/TR-530, MIT, 1992.

[17] W. de Jonge, M. F. Kaashoek, and W. C. Hsieh. The logical disk: A new approach to improving file systems. In *Proc. 14th Symp. Operating Systems Principles (SOSP)*, pages 15–28, December 1993.

[18] J. Katcher. PostMark: A new file system benchmark. Technical Report TR3022, NetApp, 2004.

[19] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proc. 9th Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 190–201, November 2000.

[20] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[21] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proc. 7th Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 84–92, October 1996.

[22] P. L. Lehman and S. B. Yao. Efficient locking for concurrent operations on B-trees. *ACM Trans. Database Systems*, 6(4):650–670, 1981.

[23] B. Lindsay. A retrospective of R*. *Proc. IEEE*, 75(8):668–673, 1987.

[24] W. Litwin. Linear hashing: A new tool for file and table addressing. In *Proc. 6th Int. Conf. Very Large Data Bases (VLDB)*, pages 212–223, October 1980.

[25] W. Litwin, M.-A. Neimat, and D. Schneider. RP*: A family of order preserving scalable distributed data structures. In *Proc. 20th Int. Conf. Very Large Data Bases (VLDB)*, pages 342–353, September 1994.

[26] W. Litwin, M.-A. Neimat, and D. A. Schneider. LH* — a scalable, distributed data structure. *ACM Trans. Database Systems*, 21(4):480–525, 1996.

[27] M. A. Olson. The design and implementation of the Inversion file system. In *Proc. of the USENIX Winter 1993 Technical Conference*, pages 205–217, January 1993.

[28] Y. Sagiv. Concurrent operations on B*-trees with overtaking. *Journal Computer and System Sciences*, 33:275–296, 1986.

[29] Y. Saito, B. Bershad, and H. Levy. Manageability, availability and performance in Porcupine: A highly scalable, cluster-based mail service. In *Proc. 17th. Symp. Operating Systems Principles (SOSP)*, pages 1–15, December 1999.

[30] M. A. Shand. UNFSD–User-level NFS server. *comp.-sources.unix*, 15(No. 1 and 2), May 1988.

[31] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-smart disk systems. In *2nd USENIX Conference on File and Storage Technologies (FAST)*, pages 73–88, March 2003.

[32] C. A. Thekkath, T. P. Mann, and E. K. Lee. Frangipani: A scalable distributed file system. In *Proc. 16th Symp. Operating Systems Principles (SOSP)*, pages 224–237, October 1997.

[33] H. Wedekind. On the selection of access paths in a data base system. In J. W. Klimbie and K. L. Koffeman, editors, *Data Base Management*, pages 385–397. North-Holland, Amsterdam, 1974.

[34] B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, and J. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal Selected Areas in Communications (JSAC)*, 22(1), 2004.