FreEBS - A Virtual Block Storage System

Igor Canadi, Rebecca Lam, James Paton
University of Wisconsin-Madison
{canadi,rjlam,jpaton}@cs.wisc.edu

1. INTRODUCTION

2. RELATED WORK

3. DESIGN

FreEBS is a virtual block device backed by replication and log-structured virtual disks for high reliability and availability. For all intents and purposes, the FreEBS storage system appears to the kernel as a normal block device and thus reads and writes are issued to FreEBS without knowledge of its internals. The architecture of FreEBS is depicted in Figure 1. As we observe, the most important component in the system is the *driver*, which is a mountable kernel module that interfaces between the OS and the rest of the system. It accepts normal block device commands and forwards the requests appropriately to the *replicas*. Our current configuration uses three replicas, but our design is flexible enough that we can arbitrarily add more if necessary. We discuss replication in more detail in Section 4.2.

Replicas are essentially server machines that communicate with each other and the driver across the network using a custom TCP protocol. Each replica runs a userspace process called `sdaemon` that is in charge of propagating write requests and synchronizing replicas. It is also responsible for interfacing with its private copy of the underlying storage that actually holds the device data, *Log Structured Virtual Disk* (LSVD). LSVD is backed by a file on the server file system that uses a dynamically growing file format. It provides checkpointing and versioning for high data integrity. A more in-depth discussion about LSVD is in Section 4.3.

4. IMPLEMENTATION

4.1 Reads and Writes

There are two operations that are issued by the driver — read and write. The driver keeps track of the most recent version for each replica in a list. This structure is updated on a write and used to decide which replica to request a read from.

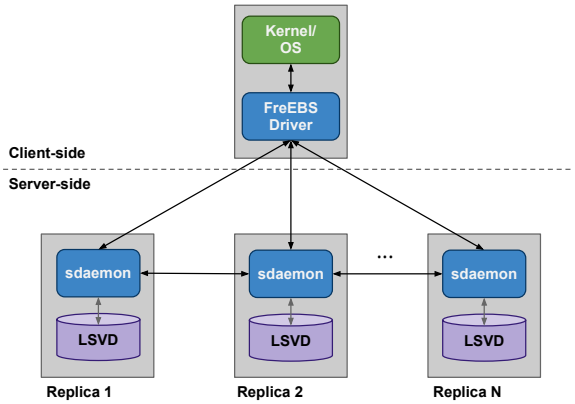
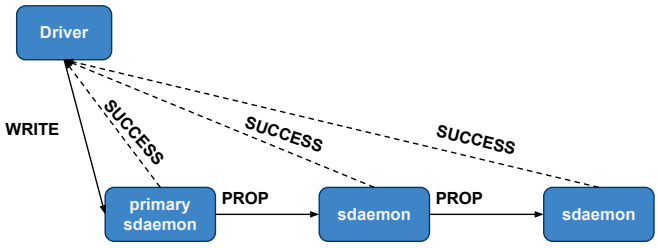


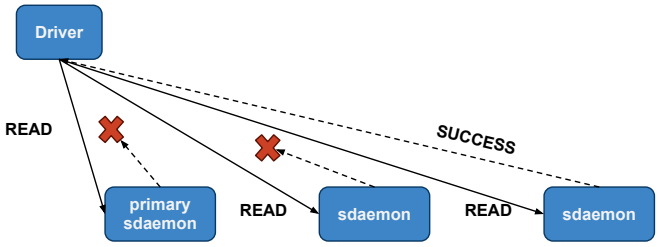
Figure 1: The system architecture for FreEBS. A driver on the client machine communicates with processes on the replicas on server machines to read and write data.

Figure 2a shows the message flow on a write request. Notice that we utilize chained replication, where the closer the replica is to the head of the chain the more more up-to-date it is. The choice for this chained scheme is to reduce the network load to the driver. Thus, when the driver sends a write request, `sdaemon` issues the write to its local LSVD copy and then propagates the request to the next replica in the chain. After writing to its own LSVD, each replica sends back a response to the driver with a status and its current version number. The driver then updates its replica list with the version in the sent response. If a majority of replicas responds to the driver with a SUCCESS message, then the driver reports a success back to the kernel. Otherwise, it will report a failure.

In contrast, a read only needs one replica to respond for a success. In Figure 2b we observe that when the driver issues a read request it issues the request to the replica with the most recent version. Typically this is the primary replica. If the replica responds with a SUCCESS, then the driver serves the request back to the kernel. Otherwise, it will issue the request to the next replica in the chain with the most recent version. I no



(a) The message flow for a write operation. A quorum of SUCCESS responses must be received for a successful write.



(b) The message flow for a read operation. A read request is issued to the replica with the most recent version until a SUCCESS is received or if no replicas with the most recent version responds with SUCCESS.

Figure 2: Message flows for read and write operations

replicas respond, then the driver reports a failure.

4.2 Replication

Replication is managed by `sdaemon` using write propagation and synchronization. Propagation was discussed in Section 4.1. Synchronization is performed whenever a replica joins the chain and also periodically to keep replicas near the tail from becoming increasingly out of date. Figure 3 shows how replication is performed. As we see, replica C sends a SYNC request with the version number of its local LSVD copy to the previous replica in the chain, replica B. Replica B then responds with all the writes from C.version to B.version. Replica C then applies all the writes to its LSVD. After the synchronization operation, Replica C's LSVD version is equal to B's version.

To handle failure, a replica controller is used to inform the driver and neighboring replicas that a failure has occured. This controller keeps track of the liveliness of each replica by recording the elapsed time since the last hearbeat message, which is sent every T seconds by each replica. After $2T$ seconds the controller flags the replica as *inactive* and after $4T$ seconds the replica is marked as failed. Upon failure, the controller informs the kernel and the neighboring replicas using update messages. Upon receipt of the update messages, the two neighboring replicas connect to each other. The latter replica sends a SYNC request, and then normal operation resumes. Meanwhile, the controller spins off

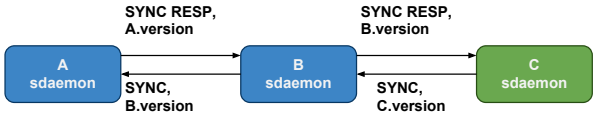


Figure 3: The message flow for a synchronization operation. Replica C sends a SYNC request to its predecessor, which responds with all the writes since C's version.

a new `sdaemon` process that is appended to the end of the chain. Note that currently the replica controller is not been implemented.

4.3 Log Structured Virtual Disk

Figure 4: The LSVD file structure.

5. EVALUATION

5.1 File System Benchmarks

5.2 Checkpointing evaluation

6. CONCLUSION

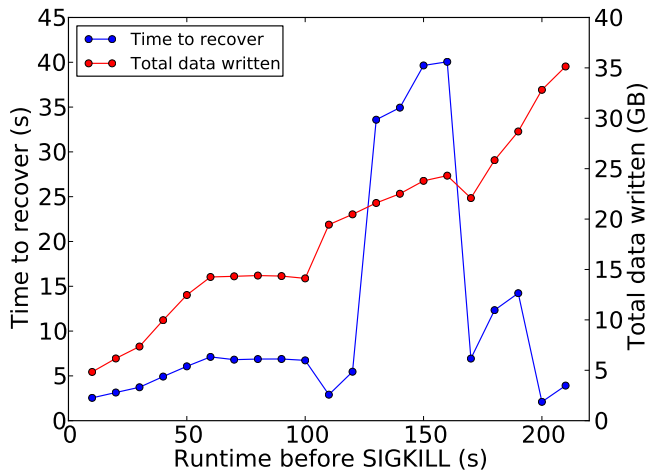


Figure 5: TODO