Communicating Process Architectures – 2003
Jan F. Broenink and Gerald H. Hilderink (Eds.)
IOS Press, 2003

The Grid Block Device

Bardur ARANTSSON Brian VINTER
Department of Mathematics and Computer Science, University of Southern Denmark

Abstract. In this paper we propose a distributed, replicated block device for The Grid based on the the replication algorithm of Y. Amir[1]. The end goal is to support application-specified semantics, both wrt. replication strategy and the consistency model used for reads and writes.

We list the proposed features of the Grid Block Device and discuss various methods of implementation.

1 Introduction

With the ever-increasing need for computing capacity it has become clear that using clusters of relatively cheap personal computers for some classes of computational problems is much more cost-effective than using mainframe computers.

Similarly, the storage demands of cluster applications, or, more specifically, scientific cluster applications, are increasing at a similar rate. In this paper we describe a proposal for a distributed block device, which, in addition to supporting the very large datasets required for such applications, is also fully replicated, and tolerant of both node failures and network partitions.

Before getting into the specifics, let us first examine traditional cluster computer configurations. The local storage on each individual PC is usually divided between the operating system and a "scratch" area where applications can write intermediate results and temporary data. Currently, the operating system occupies a very small portion of the total available storage – typically on the order of 1% – and the relative amount of operating system data vs. scratch data is only going to grow as storage density on hard drives grows.

In order to access shared data such as user home directories and large sets of experimental data[1], there is typically also a network-accessible drive. This networked drive usually resides on a central server using a large array of disks for redundancy and performance, simply because the administration is much easier that way.

There are a few inherent problems with this approach to cluster computing which we will consider below.

Firstly, the scratch area not explicitly utilized by the particular application that the cluster is running goes to waste. This can amount to quite considerable amounts of wasted storage capacity.

Secondly, and more imporantly, server performance may become a limiting factor for access to the shared data. In addition, if multiple clusters are connected over a Wide Area Network (WAN) then the bandwidth of this link may become a limiting factor for the performance that can be achieved by an application. The incurred extra cost may be unacceptable in many instances, even when the WAN link is a high-speed dedicated link.

Finally, each server is a single point of failure for its part of the file system namespace. As cluster systems grow to several thousands of nodes, such a single point of failure becomes unacceptable.

To solve the problems of using the above approach, the proposed Grid Block Device (GBD) is focused on providing a fault tolerant block device which is used as the sole data storage facility for each node participating in a computation. The proposed block device has many desirable features, among them replication, automatic fail-over, and disconnected operation.

Why a block device instead of a file system? There are a several reasons for this decision, a few of which will be enumerated below.

Firstly, implementing a distributed block device is simpler than implementing a fully-fledged distributed file system, because the semantics of file systems are largely irrelevant to the implementation. We can thus focus on the essentials that are relevant for efficient replication and efficient access to the replicated data. Also, some features, such as global snapshots, are actually much easier to deal with on the block level than the file level.

A distributed block device could also prove useful for distributed database-like applications. Since database applications have much more predictable access patterns than most applications, they usually bypass the file system layer entirely, and often also the caching mechanism of the operating system, avoiding the overhead associated with these. File systems are almost always optimized for much more unpredictable usage patterns which makes them unsuitable for this type of application. This would seem to indicate that it might be more efficient to implement distributed database applications using distributed block-oriented storage, rather than basing the implementation on a more general distributed file system.

Finally, implementing a distributed file system on top of the distributed block device should be relatively simple since the infrastructure for replication and disconnected operation would already exists. To elaborate further on this last point, a very simple implementation of a distributed file system based on the GBD could simply use the old UNIX idea of using trees of i-nodes to represent files and directories. It might seem like a bad idea to have a single root i-node and letting everything else "hang off" that single i-node because each file lookup requires access to the root i-node. However, it has been observed that i-nodes closer to the root have a tendency to be quite static over time[2], whereas i-nodes deeper in the tree have a tendency to change much more frequently. In the context of the GBD this would mean that the root i-node could be replicated to all servers at very little cost because any (expensive) updates to it would occur very rarely. Since all searches for files start at the root i-node, replicating it, and possibly its children, to each node could be significant win. Of course, supporting more high-level features such as truly atomic writes of arbitrary length would require a much more complicated file system structure, but the infrastructure provided by the GBD should make implementing such "extras" much simpler than it would otherwise have been.

By avoiding the complications caused by tying the file system implementation to the replication engine implementation we believe that a much cleaner and more efficient solution to the data distribution problem can be developed.

2 Previous Work

2.1 Spread

The low-level communication layer utilized by the replication algorithm is the group communication system Spread [2], which we give a short overview of here.

Spread provides efficient group communication with reliability and ordering guarantees over WANs. It is based on UDP/IP and implements its own packet ordering and reliability. An overview of how the spread network is structured is given in figure 1.

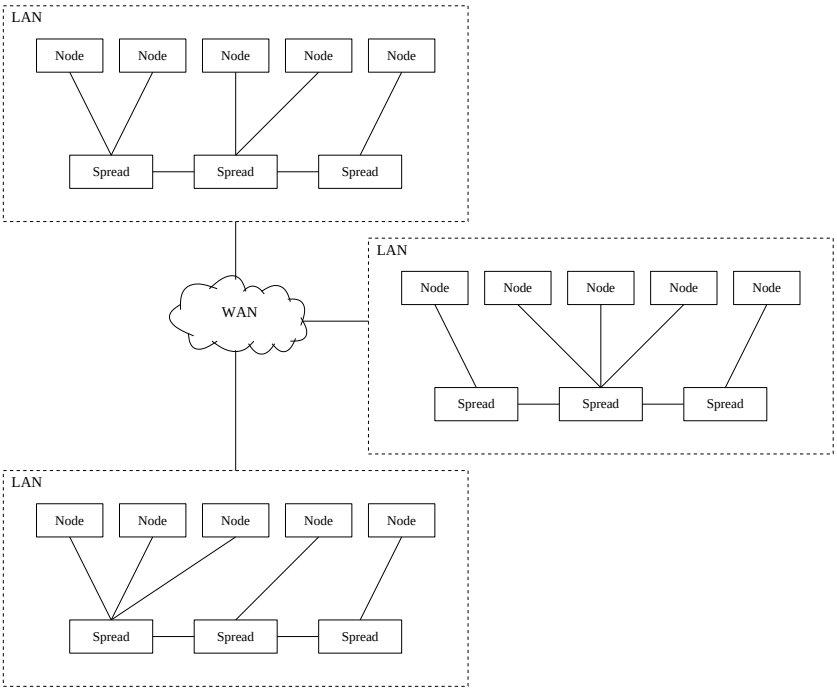


Figure 1: Overview of the Spread network structure. Some (not all!) nodes in a LAN run a Spread daemon (`spreadd`) which communicates with the other Spread daemons in the LAN. Each node initiates a connection to a designated Spread daemon which actually handles all the communication. Spread daemons from multiple LANs communicate via a WAN.

The network is overlaid with a multicast tree network which is based on the network topology and typical latencies in the network, which are, in turn, provided by the user through a static configuration file. The multicast tree is used to implement efficient multicast by simply only sending packets that are absolutely necessary for a multicast message to reach all its intended destinations; see figure 2 for a simple example.

The reason that Spread is based on UDP/IP instead of an ordered, reliable protocol such as TCP/IP is simple: If a reliable point-to-point protocol with built-in packet ordering were used, the latency would suffer in scenarios such as the one shown in figure 3. Looking at the figure, we see that node A sends a sequence of three packets to node C through B. In the example shown, the first packet is lost and must be retransmitted. Observe that all packets transmitted after p1 could easily have been sent on after arriving at B, but when using the ordered reliable protocol they all have to wait for the first packet to arrive at B before being

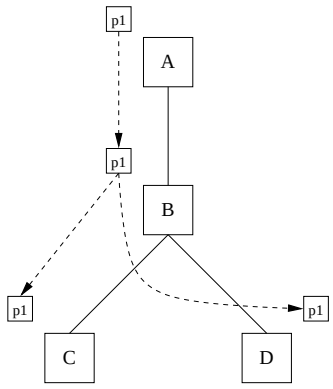
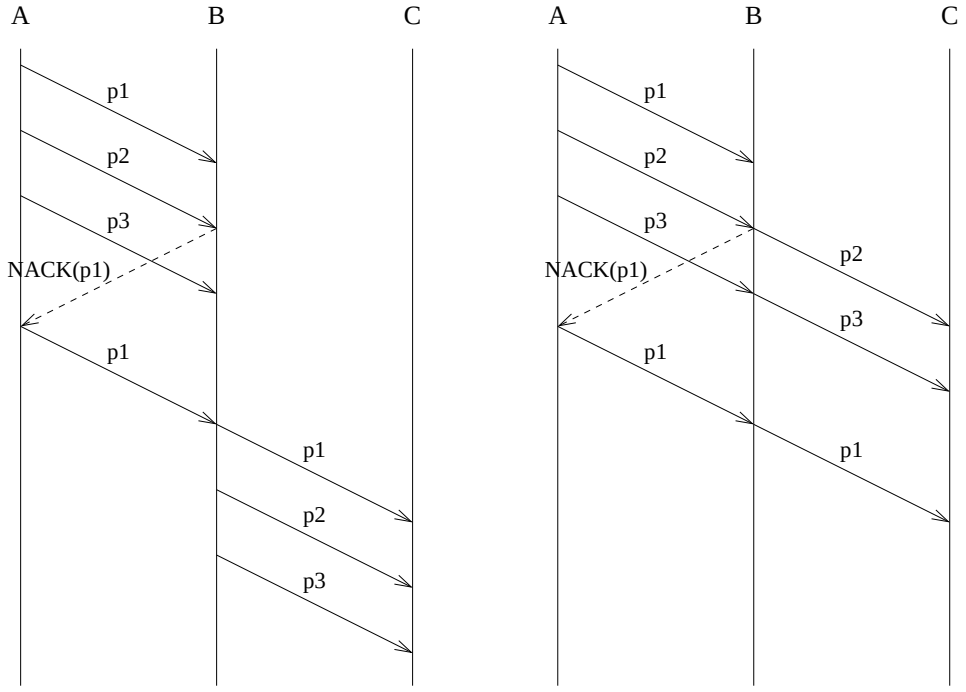


Figure 2: Efficient multicast in Spread: If A wants to multicast a message to C and D, only *one* message is sent from A to B, and B assumes responsibility for transmitting the message to C and D.



(a) Using a reliable PTP protocol.

(b) Using an unreliable PTP protocol.

Figure 3: Difference in total latency between using a reliable/unreliable point-to-point (PTP) transport protocol for routing packets through intermediate hops along a route.

sent on. As shown, this can lead to completely unnecessary communication delays, even in relatively reliable networks. To avoid such unnecessary delays, Spread assigns responsibility for packet ordering to the final destination of the packet instead of forcing each intermediate hop to perform its own packet sequencing. In the above scenario, this means that all packets following the first packet would be sent to the intermediate node B, and, while p1 was being retransmitted, node B would forward the packets p2 and p3 to C. The packet ordering would happen at the final destination C, just before delivery to the application layer.

Another aspect of the Spread communication layer which is leveraged by the replication algorithm is the fact that the communication layer provides Virtual Synchrony[5] semantics, which effectively means that notification is provided whenever the configuration – i.e. the set of connected servers – changes. There are two types of configuration changes: Transient and regular configuration changes. Transient configuration changes are a signal to the application that the group membership has changed due to a network event, but that the new configuration is not known yet. Regular configuration changes occur when the new network configuration following such an event becomes known. These messages are delivered *interlaced* with the regular data messages and in an order consistent with the "stream" of data messages. This allows for the introduction of a state where the network is "in between" fully connected and partitioned.

2.2 Replication Algorithm

One of the most interesting replication algorithms to emerge in the last few years is the algorithm of Y. Amir [1]. It is based on the Spread group communication system and uses the Extended Virtual Synchrony [7] concept to provide fault tolerant replication whilst con-

siderably reducing the need for – and in the common case avoiding – the most expensive part of most other replication algorithms, namely end-to-end acknowledgements. Avoidance of end-to-end acknowledgements on a per-action basis can yield much higher performance than algorithms such as 2PC and COReL as demonstrated in [3]. Network partitions are also handled correctly.

The replication algorithm is designed mainly for database-like scenarios where the goal is to replicate the complete contents of the database to all participating servers. Instead of taking the traditional approach of replicating *data* to participating servers, it instead replicates all *actions* taken upon the data, and ensures that all these actions are applied across all replicas in a global total order. While ensuring that all participating servers have identical[3] replicas, this also affords a lot of flexibility wrt. allowing different semantics for different actions. For example, an application could choose to use the dirty query semantics over the typical sequential consistency if its needs are more in the realm of interactivity than accuracy. Apart from these two models, the algorithm also supports weak consistency.

The aforementioned actions are small transactional units consisting of a *query* section and an *update* section. Since the replication algorithm cannot give guarantees about the precise timing of the application of updates to the database[4] we must somehow prevent updates from being performed if they are based on stale data. This is achieved using the *query* section of an action as a precondition for performing the updates in the *update* section. As an example of how this works, consider the following: An application which increases values in a database by 10% reads that a particular piece of data that it intends to modify, x, has value 5. To perform the update, the application creates a new action, containing $x = 5$ in the query section, and $x \leftarrow 5.5$ in the update section. The action is submitted to the replication engine which orders it according to the global total order. If, when the action is applied to the database, the value of x has changed, then the action is rejected and the client program is informed of this. Using this technique it can be ensured that the update will only be performed if the value of the data x has not been changed by another action in the time between the read and the update. This can be generalized by associating version numbers with individual blocks and preconditioning operations on these version numbers instead of relying on particular data values.

Originally, the replication algorithm in [1] was restricted to a preconfigured set of replication servers, but it was extended in [3] to allow for dynamic addition/removal of replicas. Our intention is to incorporate this extension into the Grid Block Device implementation.

3 Overview of The Grid Block Device

The overall architecture of the Grid Block Device is rather simple; it simply sits as a layer between the replication engine and the application (as shown in figure 4). Note that the "application" which is shown is not necessarily a user application, but could just as easily be a distributed file system implementation, or even a Linux device driver that would effectively allow using the GBD as a simple means of integrating with Linux filesystems.

In the following sections, the motivation and implementation ideas for each of the proposed features of the GBD will be described in some detail.

4 Features

The main proposed design features of the Grid Block Device (GBD) are as follows:

- Widely Distributed Servers

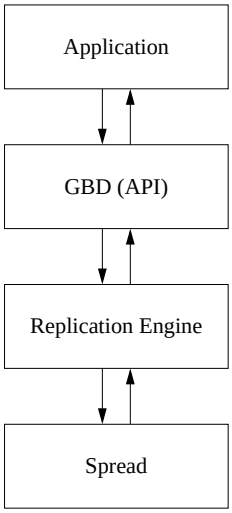


Figure 4: Overview of the Grid Block Device.

- Automatic Replication and Migration
- Disconnected Operation
- Global Snapshots
- Application-specific Semantics
- Legacy Application Support

In the following sections each of these proposed features will be discussed in more detail.

4.1 Widely Distributed Servers

Having widely distributed servers is critical to high availability and low latency for widely distributed clients accessing the system. Using widely distributed servers has traditionally been problematic because the inter-server communication has usually been done using protocols and algorithms (e.g. 2PC) with a lot of overhead, or, which simply could not operate correctly in partitioned networks. Clearly, network partitioning is to be expected in any reasonably-sized WAN, simply because of the distances involved.

Since the algorithm of [1] can operate in partitioned networks and avoids per-action end-to-end acknowledgements, it is relatively cheap to employ widely distributed servers in this context.

While increasing the number of servers usually also increases the performance and availability of the system there is also an increase in the amount of overhead involved in maintaining the replication engine state and in the group communication layer itself. Thus, a balance between the number of servers and the incurred overhead needs to be found through experimentation.

4.2 Automatic Replication and Migration

Automatic replication and migration of data to where it is needed is among the most important features of the proposed Grid Block Device. Much research has been conducted in this area; see [6] for an overview of early work. However, the assumptions underlying many of the proposed algorithms sometimes vary wildly, and may or may not apply practical applications. Obviously, the optimal choice of algorithm may also vary from application to application, and so the best strategy for implementing any replicated/distributed system

is probably to have a generally applicable algorithm as the default, but giving the user the option of choosing/implementing a different algorithm if necessary.

Any generally useful algorithm for replication and migration needs to satisfy a few properties: Decision-making must be truly distributed, i.e. with no centralized decision-making, and decisions must be taken quickly. It is also very desirable to have as low overhead as possible, particularly wrt. the space requirements for any bookkeeping data. Finally, the algorithm should be quick to adapt to changes in situation such as network congestion, imbalance in server load, and changes in user-imposed resource utilization constraints.

There are, of course, a number of heuristics that apply to this problem. If, for example, a particular block of data is accessed frequenty by a particular client, then it makes sense to try to move the data closer to that client. Also, if a particular block of data is accessed very infrequently on a particular server, then it may be possible and desirable to simply remove the replicated data. Of course, we must be careful to obey any redundancy constraints imposed by the user, and possibly network partitioning, before removing replicated data.

There are also some randomized algorithms for the replica placement problem that are quite simple to implement which also do well in practice; see [4], for example.

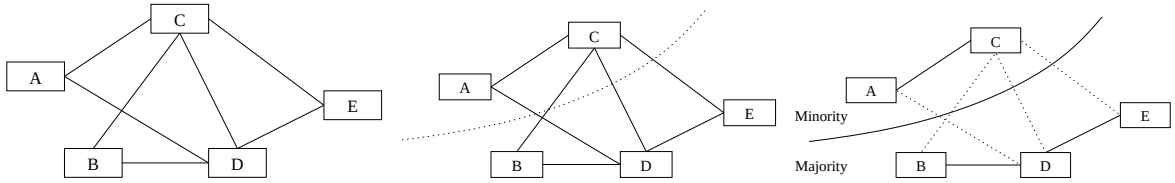
It should be noted that the basic unit of replication in the GBD is the *block*. Each block is therefore – in principle – distributed completely independently of other blocks, which also means, that no single node needs to have sufficient storage for the complete contents of the block device. This makes it possible to support vast amounts of storage on relatively cheap (PC-class) hardware. There may of course be significant benefits to keeping neighbouring blocks together, especially wrt. the amount of metadata required, but there are no absolute constraints on the replication/movement of individual blocks of data.

4.3 Disconnected Operation

We distinguish between two forms of disconnected operation, namely *involuntary* disconnection and *voluntary* disconnection. As the names suggest, the former occurs when the network becomes partitioned or individual network links go down, and the latter occurs when e.g. temporarily moving a few participating servers.

Such events should not have any significant consequences for the performance of the rest of the network. Obviously, there are instances where a performance impact cannot be avoided. As an example, consider a network which becomes partitioned, causing all servers with a given piece of data to become isolated from the rest of the network (see figure 5). In this case it is simply impossible to avoid waiting for the network to become fully connected again before any reads or updates can be performed upon the affected data. Furthermore, aside from replicating all data to all network nodes, there is no possible way of avoiding this scenario, since it is impossible to predict where network partitions may occur. However, it should be possible to give the replication layer some *hints* as to which links are more likely to go down than others. This could allow replication/migration decisions to be based on such external information instead of being based solely upon information which is internal to the system, and, which is thus also limited to what can be gathered through internally observable properties of the network.

Another important consideration wrt. disconnected operation is how to handle updates and reads that occur while the network is partitioned. The replication engine automatically selects a new primary component which is allowed to continue unhindered, but if sequential consistency is to be maintained, actions in other components must be delayed until the network becomes connected again. Clearly, choosing a particular consistency model for all applications to adhere to is either going to limit their efficiency, or cause inconsistencies to arise in the case where the model is too weak for the particular application. Therefore, the



(a) Fully connected network. (b) Partition occurs. (c) Partitioned network.

Figure 5: Impact of network partitions: If node A and node C have some data which is required by node B, D or E, then a network partition may prevent all access to the data until the network becomes whole again. Note, however, that the majority of the nodes may continue issuing and performing updates on data which is *shared* between nodes on either side of the network partition without causing any consistency problems.

choice should be left up to the applications themselves.

There is also the issue of "defensive" replication to consider here: Since network partitions lead to the network separating into two or more connected components there is a likelihood that they each contain a number of replicas of any particular block device. If we are lucky they will contain an equal number of replicas, but we might be unlucky, and one of them might contain only one replica. This could clearly be a dangerous situation for the system to be in since the side containing all the remaining replicas could be completely wiped out from e.g. natural distaster without the other side of the partition knowing anything about such an event before it was too late, i.e. when one of its own servers containing a part of the replica crashed. In order to defend against such failures, we propose a RAID-like redundancy scheme where *transient* replicas are introduced when a network partition occurs. These transient replicas serve as a sort of backup if all (regular) replicas should fail on either side of a network partition.

4.4 Global Snapshots

Supporting global snapshots is not trivial by any means, because snapshots have very precise and strict semantics: A snapshot represents a *consistent* and *unchanging* image of the data, in our case the contents of the block device, at the time the snapshot was taken. Global snapshots are extremely useful because they allow backups in systems which cannot be taken off-line for extended periods of time. Additionally, snapshots can serve as a hint to the system that no further changes to the given data are expected. This could allow the system to optimize for the read-often, write-rarely scenario.

There are two basic techniques for supporting snapshots:

1. Create a read-only duplicate of the data.
2. Avoid overwriting data that is part of a snapshot; instead, write new copies of changed blocks. When the snapshot is released, any subsequent changes can be incorporated back into the "live" block device.

The former of these techniques is *extremely* expensive, since it requires creating a complete copy of a potentially huge data set. This requires a lot of storage, and also requires a lot of time to perform the actual copying operation. Additionally, in a replicated system care must be taken to replicate the snapshot itself to avoid bandwitdth bottlenecks and vulnerability to crashes on the server(s) containing the snapshot. Without such replication a crash of any server containing any portion of such a snapshot could tie up any processes reading from the snapshot. One positive property of using this technique is the fact that actually *accessing* a

snapshot is completely trivial and carries no overhead whatsoever, since the snapshot is just a read-only block device containing an exact copy of the original.

If we examine the latter approach, we see that it requires a certain amount of extra bookkeeping, but that it has some significant advantages. Firstly, a global snapshot only requires extra storage on the order of $O(n)$ where n is the number of *modified* blocks since the snapshot was taken. In the worst case this is exactly the same amount of space required using the former technique, excluding the negligable bookkeeping overhead. Under normal circumstances this should result in significant space savings.

A relatively simple way to implement this approach is to apply versioning to individual blocks and use this version information to avoid overwriting/releasing data which is part of a snapshot. Creating a snapshot is then merely a matter of noting the version number at which the snapshot was made, and avoiding making direct changes to the affected blocks while the snapshot is active. Since this requires no copying of data this is also quite cheap time-wise, but it does impose a slight overhead for each request while snapshots are active.

4.5 Application-specific Semantics

Application-specific semantics are really a two-part issue. First we consider application-specific replication/migration policies, and then we consider application-specific read and update semantics.

A simple example of an application-specific replication policy would be allowing applications to specifiy a minimal number of replicas which the system must maintain at all times – if at all possible. Note that it may not actually be possible to always guarantee, say, precisely 5 replicas of a given file if the network becomes partitioned. However, upon discovering the partition, the system can start introducing transient replicas of the affected blocks until the replication count reaches 5 again. While this "rebuilding" is taking place, the system runs in a sort of "degraded" mode akin to RAID-like systems. When the network becomes connected again, the transient replicas can be removed.

Supporting application-specific read and update semantics can be an important aid in optimizing for specific access patterns. For example, there may be applications that write sequential blocks a vast majority of the time, while other applications only perform random-access reads of data. Additionally, it may be worthwhile to let applications signal to the system that they only require *dirty read* semantics. This could be useful for an application that continually reads data from a GBD, but which has interactivity constraints. In this case, the read data should converge in some sense, but the interactivity is valued more than showing data which is necessarily guaranteed to be the most up to date. Note also, that network partitions can force normal reads to block until the network becomes connected again. This is obviously a bad idea for such "real-time" data.

4.6 Legacy Application Support

An important parameter for adoption of any new technology is the level of support offered for legacy applications. This support can come in two forms: Either the new techonology supports the legacy applications outright, or it provides an easy-to-implement migration layer/mechanism. Implementing transparent support for legacy applications might be a worthwhile goal for the simple reason that it would promote adoption, but there are several problems associated with it which we will summarize below.

Firstly, the POSIX file system interface[5], which consists mainly of the `open`, `read`, `write`, and `close` operations, does not lend itself well toward a transaction-based view of the file system. These operations all assume in some way that the application is in complete

control of the file, i.e. has an exclusive lock on the file. In some distributed file systems this is dealt with by the introduction of shared/exclusive locks which are held by applications while performing read/write operations on any given block of data. But exclusive locking can cause horrendous performance degradation, particularly in the case where applications are accessing completely different regions of the same file. This has been addressed by region-based locking of files through the `fnctl` system call which is part of the POSIX standard, but these locks are quite error-prone and consequently quite hard for programmers to use correctly. Compared to these, the transaction-like semantics of actions seem much more intuitive.

It is unclear exactly how these operations should be mapped to the transaction-like semantics of the GBD. The trivial solution is to simply lock the entire device on `open` and unlock the device on `close`, but this completely eliminates the usefulness of the GBD. The problem is, of course, that since the POSIX file operations were not defined in the context of concurrent reads/updates, their semantics are too weak to have any meaning in a distributed context. Even the *append* file mode in POSIX does not define any sort of mechanism for dealing with concurrent writes to a given file, even though there is the obvoius approach of just appending anything written to the file. This weakness in the POSIX semantics for file operations means that we either have to simply discard the old interface, use locking to achieve POSIX semantics, or make our own assumptions about the intended semantics of the POSIX file operations.

One option is to impose something resembling causal ordering in the following manner: Firstly, version numbers are added to each block. When an application performs a `read`, it causes a certain version of a block of data to become "known" to the application. Any subsequent `write` of that particular block is preconditioned on the block version being the same as that which was read from the device. Subsequent reads of any *other* block are preconditioned on previously read blocks having the same version numbers as when they were read. This stems from the observation that reads are almost always dependent on each other in some way.

Subsequent `writes` of any other block is preconditioned on all "known" blocks having the same version numbers as when they were first read.

Opening or closing a file flushes the "known" blocks for that file. This is based on the assumption that closing and reopening a file means that an application cannot assume *anything* about the file's contents after reopening it. This seems a reasonable assumption since closing a file implicitly removes any locks held on it.

This way of handling the POSIX file operations implements a kind of causal ordering which may give resonable results for applications that do not have strict locking requirements, i.e. most user programs should work properly with these semantics since they usually do not access data which is shared between multiple processes.

Unfortunately, storing version numbers for all these "known" blocks is not something which requires a trivial amount of storage. Actually, if a legacy application – supported using this scheme – were to read a large file sequentially, it would require storage proportional to the file size. It would be prohibitively expensive, probably impossible, to have this much data in memory to handle legacy applications which were streaming in nature. It could be workable for applications which only require a few random accesses to the block device, but in general it would not be feasible nor advisable to use such a scheme.

The best approach is, of course, to rewrite applications to use the native API in which such problems do not occur because dependencies are handled explicitly by using the *query* section as a precondition for the *update* section of actions. This could potentially also result in much higher performance because access patterns can be tailored specifically to the given application.

Another issue is the matter of handling failed updates, i.e. updates which fail because they are performed on stale data, i.e. data which was changed by another process without the application having any way of becoming aware of the change. The return codes for the POSIX file operations simply have no equivalent for this situtation. One would have to make a choice between using one of the existing error codes, and risk problems because of the different semantics of the situations in which the error occurs, or, introducing a new error code and risk problems arising from error handlers not written to cope with this new error code. Either of these solutions is unsatisfactory.

5 Conclusion

Work is currently focused on developing a prototype Grid Block Device based on the various ideas presented in this paper. When finished, it could provide valuable insight as to the merits of the individual ideas/techniques that have been discussed.

When such a prototype is available, the API can be solidified, and a framework for testing performance objectively can be developed to aid in further evaluation. Objective evaluation in real-world scenarios is needed for at least the following aspects of the Grid Block Device: Performance as a function of number of servers and server connectivity, replica placement, and disconnected operation.

In conclusion, a considerable amount of work remains to be done, but we believe the algorithms and ideas presented here to be a sound foundation upon which to base the implementation of the Grid Block Device.

Notes

[1]As an example of what we consider a large data set in this context, the LHC experiments at CERN are expected to result in an estimated 6PB of data annually.

[2]It helps to think of the UNIX file system hierarchy when considering this. For example, the contents of the file system root, `/`, almost never change whereas the contents of a user's home directory, `/home/user/`, change much more frequently.

[3]In the sense that they contain *equivalent*, not necessarily physically identical, data.

[4]Such guarantees are, in fact, impossible to give in partitionable networks since they require the equivalent of a global clock which is provably impossible to maintain correctly in such networks.

[5]Which is also the block device interface, because of the UNIX philosophy of treating everything like a file.

References

[1] Yair Amir. *Replication Using Group Communication Over a Partitioned Network*. PhD thesis, 1995.

[2] Yair Amir, Claudiu Danilov, and Jonathan Stanton. A low latency, loss tolerant architecture and protocol for wide area group communication. In *FTCS 2000*, 2000.

[3] Yair Amir and Ciprian Tutu. From total order to database replication. Technical report, Johns Hopkins University, 2002.

[4] Yair Bartal. *Lecture Notes in Computer Science 1442*, chapter 5. Springer, 1998.

[5] Kenneth P. Birman and Thomas Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 123–138, November 1987.

[6] D. Dowdy and D. Foster. Comparative models of the file assignment problem.

[7] Louise E. Moser, Yair Amir, P. Michael Melliar-Smith, and Deborah A. Agarwal. Extended virtual synchrony. In *The 14th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 56–65, 1994.