RJ 10316 (A0403-006) March 5, 2004
Computer Science

IBM Research Report

The Design and Evaluation of Network RAID Protocols

Deepak R. Kenchammana-Hosekote, Richard A. Golding,
Claudio Fleiner, Omer A. Zaki
IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

The design and evaluation of network RAID protocols

ABSTRACT

Scalable distributed storage systems are being built out of collections of small storage bricks, which are network-connected systems with a small amount of storage and little internal redundancy. In these systems, data must be replicated across multiple bricks for good reliability. In this paper we investigate "network RAID protocols": the protocols used to read and write redundant data over a network. several distributed storage projects have implemented such protocols, but without a systematic basis for designing the protocols, choosing features, or comparing one protocol to another. The choice of protocol depends on factors such as the relative performance of client systems to storage bricks, the semantic guarantees desired, and the offered workload. We offer a taxonomy of the possibilities and a performance evaluation to help choose among them. We classify protocols by three main features: the structure of the data path between clients and storage; the method used for serializing operations; and the method used to ensure atomicity and failure tolerance. We find that system load and performance depend primarily on the data path structure, with direct client-to-storage communication fastest but placing the greatest load on client systems. Under normal conditions pessimistic and optimistic serialization methods give nearly equal performance, and there is little or no penalty for providing multi-block write atomicity.

1. INTRODUCTION

Several new storage systems are built out of collections of small, cooperating servers, or *bricks*, connected by a standard network. Each brick is small in computing power and capacity when compared to even a mid-range disk array, but aggregating them yields a distributed storage system with more computation and storage capacity than even the largest disk arrays.

Systems built this way have several attractive properties:

- Data can be moved between clients and bricks in parallel, yielding excellent bandwidth, up to the limits of the network connecting them.
- These systems can scale to be very large when they are built without central points of control.
- Each brick adds a reasonable increment of storage, processing, cache, and networking as one package, relieving the system administrator of having to purchase and configure many different types of hardware.

In addition, recent work in object storage [4] shows that these systems can also be secure.

However, bricks are relatively small, and there is nearly always a tradeoff between a brick's cost and its reliability and performance. Today, the sweet spot is often a brick with only modest performance and reliability, lacking redundant internal structures. Instead, system performance and reliability must come from aggregating multiple bricks. Performance comes from striping data, while reliability comes from storing data redundantly among bricks.

In traditional, centralized RAID systems, the RAID controller performs a similar aggregation, of simple, non-redundant disk drives. The controller sits in the data path between the client and the disks, and handles the jobs of writing redundant copies of the data, reading data from the appropriate disks, and recovering data after a failure.

Decentralized systems that use network-attached bricks must provide equivalent features by distributing the RAID controller function amongst the bricks. The clients and bricks use a "network RAID" protocol to read, write, and recover data reliably. A few different network RAID protocols have been proposed or implemented [3, 10, 9, 11]. There are many options: should a client talk directly to bricks, or through a controller? Where should parity calculations be done? Who should detect and recover from failure?

Given all these possible protocols, how is one to choose amongst them? Indeed, what exactly are the options to choose from?

Note that we are *not* concerned here with the RAID layout, as typified by the various RAID levels [18]. Network RAID protocols are used to execute the IO operations that read and write mirrored storage, parity-protected storage using RAID 4 or similar layouts, as well as storage protected using more complex coding schemes.

First, in §2, we present a taxonomy that describes the different ways that network RAID protocols can be designed. The taxonomy outlines the options for how data flows from client to storage, and how serialization, atomicity, and failure are handled. We found that this approach led us to examine combinations that we did not initially envision. It

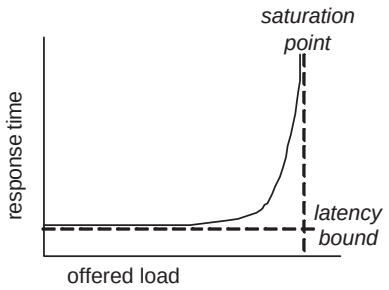

Figure 1: Concerns in performance evaluation. The *latency bound* governs the minimum response latency of the system at light load, and is dominated by the length of the sequence of synchronous messages and processing. The *saturation point* determines the maximum reasonable offered load, and is determined by the load that different operations impose on system components.

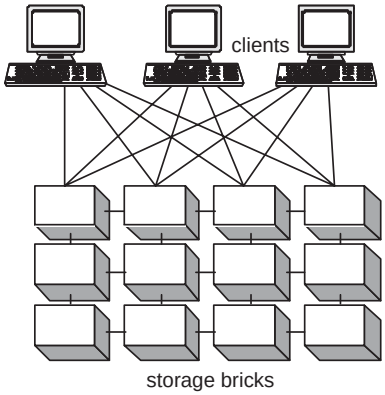

Figure 2: Components in the system: *clients* generate IO requests, while *bricks* store data. The bricks are connected with a high-performance, dense network, while clients may be less well connected.

also helps to explain the differences between existing systems, so we survey a few distributed storage systems.

We then turn our attention to how to choose among the protocols based on their performance. For each protocol, we are concerned with two aspects of performance, as shown in Figure 1: the minimum latency bound that it places on processing one request, and the offered load at which the protocol will saturate some part of the system. In §3 we give an evaluation of the load that different protocols place on clients and bricks, which determines the IO rate at which something saturates. Then in §4 we report on the latency we observe for different IO requests.

1.1 System model

The kind of distributed storage system that concerns us has nodes filling two roles (Figure 2): what we will call *bricks* and *clients*. (The SCSI model calls these roles *targets* and *initiators* respectively.)

A brick's main purpose is to store data. Each brick is typically relatively small, and not internally redundant: a single processor, some cache memory, between one and twelve disk

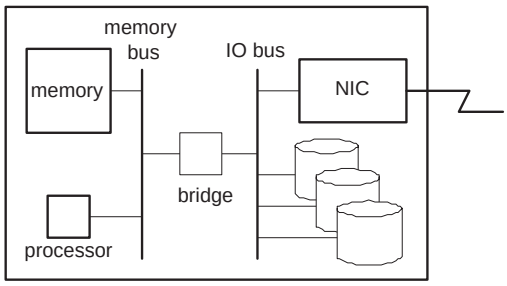

Figure 3: Structure of a brick.

drives or so. The hardware and software in a brick is designed to be at least as reliable as any non-redundant server, and to provide good performance for the data stored on that brick. Figure 3 shows the structure of a typical brick.

A client, on the other hand, is running some application that wants to read and write data. The client expects that once it sends write data to the storage system, the data will be stored reliably until it or some other client reads the data. In some environments clients will often share data, while in other environments data sharing will be rare.

Some protocols introduce other roles as well, such as lock servers. We assume that these servers run on non-client bricks.

All the nodes in the system—both bricks and clients—are connected by a high-performance network. Informally, we assume that the network is generally but not perfectly reliable. Different installations will use different network topologies, ranging from dense, highly-reliable networks that provide uniform high bandwidth between any pair of nodes, to sparse networks that exhibit widely varying performance. This model covers many different kinds of storage systems, including SANs built from networks like FibreChannel as well as those connected by IP networks.

Different network RAID protocols can make different assumptions about the formal reliability and synchrony of the system. We generally assume a system under the timed asynchrony model [8]. Some of the protocols use timeouts to detect failure; in practical terms, this means that there must be reasonable bounds on the relative rate of progress of clocks at different nodes. Some other protocols will assume the presence of loosely-synchronized clocks, which are maintained by network time synchronization protocols that also typically assume bounded clock drift rates [17].

2. PROTOCOL DESIGN OPTIONS

There are many ways to build a network RAID protocol, but they all provide consistent read and write on redundant or striped data in a distributed storage system. In addition, any network RAID protocol that aims to provide failure tolerance must implement a mechanism for rebuilding data after failure.

Different design options yield different levels of consistency and fault tolerance. Some design options can take advantage of underlying system features such as particular network topologies or synchronized clocks. And finally, the load imposed on different nodes depends on protocol design choices.

In this section we discuss the features and guarantees that network RAID protocols can provide, and present a tax-

onomy of the approaches for implementing those features. We then present several protocols based on combinations of these features, and discuss how some existing systems fit the taxonomy.

2.1 Levels of protocol guarantees

One-copy serializability [2] is the traditional definition of correct reading and writing of redundant data. Roughly speaking, the values observed by the outside entities performing the reads and writes are correct if they are equivalent to some serialization of those operations on a single copy of the data. This definition eliminates problematic cases such as writing a value to one replica but not the other, then getting a different response depending on which replica one reads from.

The one-copy serializability definition leaves many details open, and there are several variants possible. We refine that definition, and classify protocols according to three kinds of guarantees [3]:

- Atomicity. Disk drives have traditionally provided a weak level of atomicity: on a write, each individual sector will either be written in its entirety, completely unwritten, or its contents are scrambled in a way that will be reliably detected; however, there is no guarantee that more than one sector will be written atomically together. If a multisector read and write to the same sectors are processed concurrently, the read may observe part but not all ofthe write. Stronger guarantees are possible, and can be important for building file and database systems: atomically-written and read blocks (fixed ranges of sectors), arbitrary-length sequential sectors or blocks, and discontiguous ranges of sectors or blocks.

- Serialization. Some systems—including disk drives—assume that the clients are coordinating their activities and are not sending concurrent, conflicting operations. Other systems ensure that concurrent operations are serialized in some consistent order.

- Failure tolerance. The weakest tolerance is that the protocol will maintain its atomicity, serialization, and durability guarantees as long as only storage devices fail, and that they crash fail in a reliably detectable way. A stronger protocol can tolerate any system component crash failing. Other protocols can tolerate storage devices occasionally returning stale data or a different data item on a read, which reflects actual failure behaviors of existing disk drives. Some protocols go so far as to tolerate Byzantine failures of some fraction of the clients and storage devices.

2.2 Taxonomy

Even for a given level of atomicity, serialization, or failure tolerance, there are multiple ways to construct network RAID protocols. Some of the options include:

- Through what path does data flow between clients and storage bricks?

- Where is serialization enforced, and in what order?

- How is atomicity achieved?

- Who detects and handles failing participants?

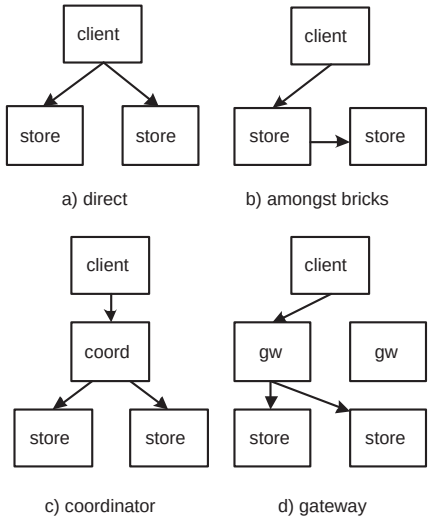

Figure 4: Data path options. Arrows show the direction data flows on a write.

For convenience, we will name protocols using a tuple of the form **(data path, serialization, atomicity)**, where each of the fields in the tuple refers to one way to implement that feature of the protocol. The following sections present options for each of these fields.

2.2.1 Data path structure.

Network RAID protocols can be compared by the route that data takes from the client to the storage bricks. Some entity in the IO path must convert the single IO request into requests on multiple devices, and ensure that the devices are consistent. In addition, some RAID layouts like RAID 4 require computing partially redundant data, which has to be done somewhere.

We have considered four data path structures (Figure 4).

Direct. The client transfer data directly to and from storage bricks, and the client compute parity. This approach is simple and can result in scalable bandwidth, but it has the disadvantage of putting greater load on the client's CPU and potentially of transferring parity data over long-distance network links. Each client will have to coordinate its actions with other clients to ensure that serialization (§2.2.2) and atomicity (§2.2.3) hold—for example, by using locking.

Single coordinator. Alternately, one brick can be designated coordinator for the data object. That brick will be responsible for sending the data onward to any other brick and for computing redundancy codes. This single brick can also perform serialization decisions centrally, without coordinating its actions. The coordinator brick is a single point of failure unless there is a failover mechanism.

Gateways. The single coordinator approach can be modified so that there are multiple gateways for a particular data item, and a client is free to contact any of them to process an IO. The gateways must coordinate amongst themselves, just as clients do when talking directly to bricks. This approach can increase storage system scalability at the cost of greater overhead for communication and more complex implementation than a single centralized sequencer. Using a set of gateways can mean that the failure of a single gateway node does not cause disruption beyond those operations it

was processing when it failed.

Amongst bricks. Finally, the client can send one copy of the data to the affected bricks, and those bricks will update the bricks that store a redundant copy or parity. For example, when updating data stored in a RAID 4 layout, the client would write data to each of the affected data disks, and each of those disks would forward an XOR of old and new data to the parity disk, which would XOR the received values into its data.

2.2.2 Serialization

Serializable execution of operations ensures that the effects of two concurrent, redundant updates are consistent. As an example, consider two concurrent updates, A and B, to the same mirrored storage location (sometimes called a write-write conflict). One-copy serializability requires that later reads will consistently read either A or B but not both. One way to achieve this is to have both copies either process A first and then B, or B first and then A. Enforcing serialization involves introducing a delay into one or more operations. Different serialization methods can be compared by where and how they introduce this delay.

Some of the ways to implement serialization follow.

Pessimistic approaches. One way to ensure serialization is to require that the clients coordinate their activities so that they do not send conflicting concurrent operations. This technique is the most commonly used today, especially for systems that use the **direct** data path model. Clients can use a lock service, requiring that the client acquire locks on all the data locations it needs to update before sending updates. Locking-based serialization delays operations by blocking lock requests at the lock server until conflicting locks are released. Locks can be implemented using a centralized lock server, or by having each storage brick provide a lock server for the data it holds [1]. We call the centralized lock server approach **slock** and the brick-based lock server approach **dlock**.

Sequencers. Another simple approach is to use a centralized operation sequencer. This fits naturally with a single **coordinator** data path, similar to existing RAID controllers and file servers. We call this mechanism **seq**. In brick-based systems, the sequencer can be one of the bricks that is storing part of the data, so saving some network communication. The sequencer defines a serialization order based on the order in which it received read and write requests, and then maintains an internal scoreboard that ensures that each operation is delayed until there are no conflicts with preceding operations. On the positive side, the sequencer approach offloads nearly all the work from client systems. However, the centralization can cause the coordinator to become a bottleneck. One solution is to partition the data in the system and have a separate sequencer for each portion of the data. Sequencers also need some form of fail-over so that the system can continue operation after a sequencer fails.

The **seq-opt** variant uses the sequencer for writes and degraded reads, but allows clients to read directly from storage bricks. This approach removes the centralized bottleneck on reads. The cost is a limitation on the span of atomicity (§2.2.3): when reading and writing data that spans multiple bricks, the client can see partially-completed writes.

Timestamp-based approaches. Optimistic concurrency control techniques aim to maximize performance when concurrency is low [3, 11, 13] and to distribute the work of serialization between clients and storage bricks. For example, when a client wants to perform an IO, it communicates directly with each storage brick that holds part of the data it is accessing—the **direct** data path model. The client labels its IOs with a timestamp that the storage bricks use to globally order conflicting operations. The storage brick persistently tracks the timestamps of the last operations to read and write each data item.

There are several ways to get the timestamps. The key requirement is that any two timestamps can be compared to establish a global order. One common method uses samples from (weakly) synchronized clocks among clients (and possibly storage bricks); the clients use a protocol such as NTP [17] to synchronize their clocks. Other approaches include using a node or set of nodes hand out sequence numbers, and using Lamport clocks [14].

There are several variants of timestamp-based serialization. The simplest maintains read and write timestamps for each data item, and rejects an IO operation if its timestamp is "in the past". We call this approach **ts**; it delays operations by rejecting and retrying them. Rejections can only occur when there are concurrent operations, or when clocks or network latencies are widely skewed, which can cause a storage brick to falsely conclude there is concurrency.

The timestamp-based approach can be taken further by maintaining a timestamped version of a data item for every write [11]. On a read, clients obtain the latest version from each affected storage brick. If the versions the client gets are not consistent, then the client can read older versions until it has obtained a consistent set of versions from which it can construct an image of the data being read. We call this approach **version**. Protocols in this family never reject operations, though read operations may on rare occasion have to communicate more than once with some storage bricks.

2.2.3 Atomicity and failure

Atomicity is the property that read operations only see all the effects of a write operation, or none of its effects. Storage systems usually extend this by allowing a third state: the data is corrupted at all subsequent reads (until overwritten). Atomicity is important when reads can proceed concurrently with writes. It also matters when failures occur, since a failure could block the completion of a write operation and cause some copies to reflect the update while other copies remain unaffected.

The span of the atomicity guarantee determines the complexity of the protocol that provides it. Possibilities include a single sector or fixed block, a variable range of adjacent sectors, or arbitrary sets of blocks or sectors. Fixed spans are simplest, because they limit the number of bricks that must be coordinated, while arbitrary sets can require coordinating arbitrary sets of bricks.

The span of atomicity that a network RAID protocol provides determines the units that need to be serialized (§2.2.2). If the atomicity guarantee only applies to individual blocks, then updates that span multiple blocks can result in a different serialization in each block. On the other hand, a larger span of atomicity will require the serialization protocol to ensure the consistency of more data. At the extreme, if arbitrary blocks of data are guaranteed to be written atomically, then the serialization mechanism may have to coordi-

nate operations among an arbitrary set of bricks.

A protocol that provides non-trivial atomicity must consider the effects of failure. The brick should be able to discard effects of a write by a client that failed during its execution by rolling back all changes.

Failure-tolerant network RAID protocols can be characterized by two properties: what kinds of failures they can withstand, and how many of different kinds of nodes can fail. The simplest protocols only handle crash failures of the storage bricks; some protocols can cope with some number of storage bricks failing arbitrarily and clients failing non-maliciously.

We use three criteria to classify implementations of atomicity:

- Multi-brick atomicity mechanism. The options include enforcing atomicity at write time using a two-phase commit (**2pcw**), which will block reads while a write remains uncommitted, and which allows for rolling back a write after failure. A second approach is to check atomicity at the client at read time (**read**). A third is to use non-overwriting, log-structured storage (**log**). The last option is simply not to provide any special atomicity mechanism (**none**).

- Recording span. If the protocol provides for a span of atomicity greater than a single block, then the protocol will need to transmit and record what that span is. The two main approaches are to record a range of sequential blocks, or a record of discontiguous data (also called the "linkage record" [10]).

- Failures tolerated. The protocol can tolerate different kinds of failure (crash, timing failures, up to Byzantine failures). The tolerance can depend on what component is failing—client, storage brick, or other server, and how many of them fail. Most storage systems have only considered crash failure, though tolerance of Byzantine failure is beginning to be addressed [11, 7].

2.3 Protocols

In this paper, we will be evaluating several protocols, including basic centralized RAID controllers as well as several protocols that distribute control. This set of protocols listed in Table 1 is the combination of each of the data path structures with a pessimistic or an optimistic serialisation method. Many such protocols have already been used in existing systems; Table 2 shows a representative sample. Other protocols are new.

We found some of these protocols because the taxonomy exposed combinations we had not seen or considered before. For example, the gateway-based schemes became apparent when we formalized the coordinator-based data path structure. Similarly, the variants of the protocols that communicate amongst bricks came from mixing different serialization methods with the **amongst** data path structure.

In addition, the taxonomy makes some odder combinations apparent. For example, the **(direct,dlock,read)** protocol uses locking for serialization, but handles failures by checking for consistency on read.

2.4 Summary

In this section we have presented a taxonomy of network RAID protocols for building distributed storage systems.

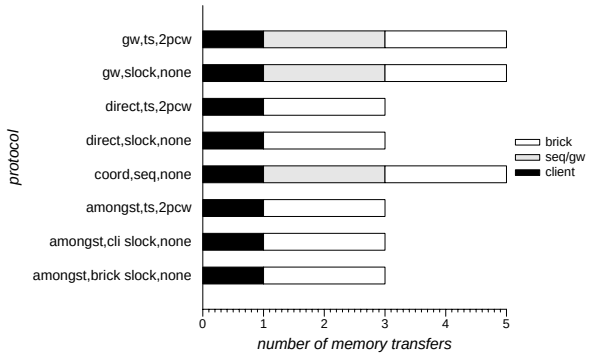

Figure 5: Memory load, RAID 1 reads. The dark portion of the bar shows client load; the grey portion shows coordinator or gateway load; the white portion shows storage brick load.

This taxonomy classifies protocols according to the pattern of communication as data are read and written, and by the serialization, atomicity, and failure tolerance the protocols provide. Existing protocols fit well into this classification scheme, and the scheme suggests some new combinations of features that are worth considering.

In the rest of the paper, we evaluate several of the protocols in order to suggest ways to select amongst them according to performance expectations and the environment they run in.

3. LOAD ON COMPONENTS

The load that a protocol places on the system determines when components will saturate, giving an upper bound on the throughput that can be achieved.

Load is therefore a way to choose protocols when brick or client resources are scarce. When bricks are relatively less powerful than clients, then more load should be placed on clients than bricks, and vice versa. The resources available on a client are often much less than the basic hardware would suggest, since most of the client system is expected to be used to run an application.

We consider here two kinds of load: memory traffic and network messages. Memory traffic determines the load on a brick's memory bus, which is often relatively slow compared to the processor; moreover, data must often be copied many times over the memory bus. Network traffic determines the number of interrupts that the processor must field, and interrupt processing is often a significant intrusion on processor performance. (We do not consider the traffic going to disks because disk load is determined by the RAID layout, not by the network protocol.)

To evaluate the load, we built analytic models of many different protocol feature combinations for each of the following operations: RAID 1 and 4 reads, RAID 1 and 4 small writes (with a read-modify-write cycle), and RAID 4 full-stripe writes. We then counted the number of times that data must be moved across a memory bus and the number of message send and receive operations that were performed at each participant. For optimistic serialization methods, we ignored the effect of operation rejection and retry because we expect it to be a rare event.

Table 1: Set of protocols evaluated.

Protocol	Description
(coord,seq,none)	Data transferred through a single coordinator; sequences IO; no atomicity.
(direct,slock,none)	Direct client-to-brick data movement; serialized by a lock server; block-granularity atomicity.
(gw,slock,none)	Data transferred to/from client through one of several gateways; server locking; no atomicity.
(amongst,cli slock,none)	Data transferred to one brick, which forwards it on to others; client uses lock server.
(amongst,brick slock,none)	Data transferred from one brick to another; first brick uses lock server.
(direct,ts,2pcw)	Direct client-to-brick data movement; optimistic timestamps; atomicity via 2PC writes.
(gw,ts,2pcw)	Data transferred through one of several gateways; gateways use timestamps; atomicity via 2PC.
(amongst,ts,2pcw)	Data transferred from one brick to another; first brick uses timestamps; atomicity via 2PC.

Table 2: A classification of existing storage systems.

System	Classification	Data path	Serialization	Atomicity
Basic RAID	**(coord,seq,none)**	coordinator	Where: controller Ordering: arrival/scoreboard Delay: controller	Span: block Mechanism: log + commit Tolerance: disk crash failure
Swift/RAID [16]	**(direct,lock,none)**, **(amongst,lock,none)**	direct, amongst	Where: lock server, brick Ordering: lock server arrival Delay: lock server	Span: block Mechanism: crash recovery Tolerance: single crash failure
Zebra [12]	**(direct,slock,log)**, **(direct,version,log)**	direct	Where: file manager Ordering: manager arrival (locks), version (internal) Delay: at manager (locks)	Span: block Mechanism: non-overwrite logs Tolerance: single crash failure
TickerTAIP [6]	**(amongst,seq,2pcw)**	amongst	Where: brick Ordering: dependency graph Delay: at brick	Span: block range Mechanism: two-phase commit Tolerance: crash failure
Petal [15]	**(amongst,seq-opt, none)**	amongst with primary	Where: sequencer (primary) Ordering: arrival Delay: at primary	Span: block Mechanism: none Tolerance: crash failure
Palladio [3, 10]	**(direct,ts,2pcw)**	direct	Where: client and brick Ordering: client timestamp Delay: brick rejection	Span: arbitrary blocks Mechanism: 2PC write Tolerance: minority crash failure, any client failure
FAB [9]	**(gw,ts,2pcw+read)**	gateway	Where: brick Ordering: brick timestamp Delay: brick delay and operation rejection	Span: block range Mechanism: 2PC on writes, consistency check and reconciliation on reads Tolerance: minority crash failure
PASIS [11]	**(direct,version, log+read)**	direct	Where: at brick Ordering: client timestamp Delay: operation retry on read	Span: block Mechanism: client consistency check and reconciliation, nonoverwriting log Tolerance: up to byzantine

Figures 5 through 7 show the memory load results for RAID 1 and RAID 4 reads, RAID 4 small writes, and RAID 4 full-stripe writes respectively. We considered the eight protocol combinations listed in §2.3.

We assume that reads do not require synchronization, since they only go to one brick in the small read case illustrated. Figure 5 shows the number of times data being read is copied in memory systems: always once onto the client, always twice on the brick containing the data (once getting it off disk into memory and once from memory to the network), and possibly twice more if there is a coordinator or gateway in the path.

RAID 4 small writes are much more expensive operations because they involve a read-modify-write cycle and XOR calculation in addition to just transmitting data to the data and parity bricks. As Figure 6 shows, the read-modify-write causes between three and four times as much total memory traffic (and the number of messages increases similarly). Direct protocols use client resources to read old data and compute XORs, so that the client sees 7 or 8 times the memory traffic as when the client offloads the work to another brick. However, offloading to a gateway or coordinator just shifts the work and adds the cost of moving the data from the client to the coordinator. Protocols that move data amongst bricks have the lowest memory load (as was observed in the Swift system [16]).

The RAID 4 full-stripe write case (Figure 7) appears to be transferring the greatest amount of data—but it is writing four blocks, one full stripe's worth of data, not just one block, and the load per byte written is lower than for small

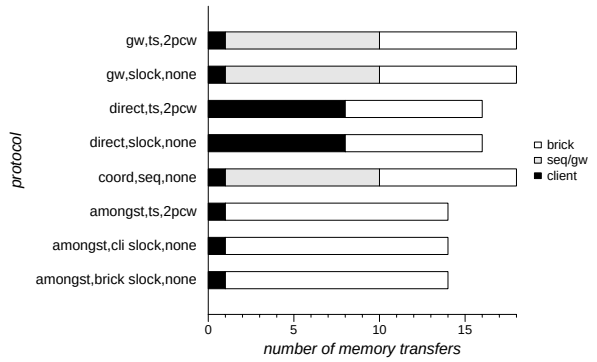

Figure 6: Memory load, RAID 4 small writes, for a 4 data + 1 parity layout.

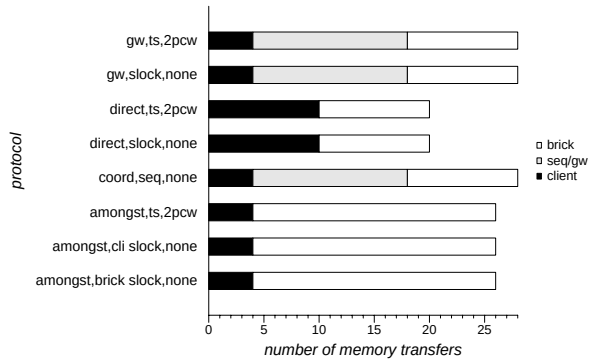

Figure 7: Memory load, RAID 4 full-stripe writes, for a 4 data + 1 parity layout.

write. Direct client-to-brick transfer creates the least overall work in the system, but puts the greatest load on the client because the client computes the parity XOR. Going through a gateway reduces the load on the client, but all the data must still be copied out of the client onto the gateway, so the overall work is greater.

The first conclusion is that the choice of topology is independent of the choice of serialization: for each operation, the ranking of the best serialization mechanism for a given topology is the same, and vice versa.

In all the cases we examined, there is no difference in memory traffic between using server-based **slock,none** and timestamp-based **ts,2pcw** serialization. The two methods are also almost identical in number of messages sent and received—in most cases, **slock,none** exchanges two messages with the lock server, while **ts,2pcw** does two commit exchanges with storage bricks. The primary difference in a RAID 4 full-stripe write is that **slock,none** sends one lock message for the whole stripe, while **ts,2pcw** must send one commit for each of five bricks.

Finally, the greatest sensitivity appears to be to the read-write mix and where the system bottleneck is likely to be. For reads, all protocols examined produce minimal client load, while the direct and amongst topologies load bricks the least. For writes, the **gw** and **coord** data path structures successfully offload the client, while the **direct** data path loads bricks the least.

4. PERFORMANCE

The choice of protocol features is based on the latency of the protocols as well as the load they place on the system. In this section we consider how different protocols are affected by the kind of operation in the workload, the amount of contention, and the network topology.

4.1 Simulation environment

We used a simulator to obtain latency measurements. The simulator is a derivative of *ns-2* [19], a packet-based, event-driven simulator to which we have added many storage extensions, including host models with memory and I/O buses; RAID levels 0, 1, and 4; disk models (RAM disks, Disksim [5] models); SCSI and ATA disk interfaces; synthetic and trace-driven traffic generators; switches; and various network topologies.

For most experiments reported in this paper, the system was modeled as some number of clients connected to storage bricks through a very fast network switch. For some of the experiments we varied the performance of links between some nodes and the switch. (We also evaluated more complex network topologies, such as switch fabrics and meshes. We found that the experiments reported here fully explain the results of using more complex topologies.)

All of the storage was used by all the clients, and was distributed among the bricks. It was organized into extents; extent sizes were multiples of stripe sizes. Extents were laid out over random subset of bricks. RAID stripe units were 8KB, and the cache page size was 8KB. Rank size for RAID 4 was 5: four data disks plus one parity. The extent size for the experiments was 256KB.

Each client or brick had a memory bus and an I/O bus (Figure 3). Both network packets and data to/from disk traversed both the memory and I/O bus. Clients and bricks included an XOR engine when the protocol needed it. Data to and from the XOR engine traversed the memory bus only. The memory bus ran at 8 Gb/s, the I/O bus at 4 Gb/s. We did not model the effect of processor cycles, because we expect memory or I/O bus traffic to be the bottleneck. Each brick had a 128 MB block buffer cache.

Each brick was modeled to have 8 disks. The disk parameters were chosen to be consistent with current 100 GB, 7200 RPM disks: 40 MB/s sequential transfer rate, and single track seek time of 4 ms. The disk interconnect consists of 100 MB/s ATA buses. In some experiments we used a RAM disk model where the device had zero head seek, switch and rotational delays. However, data transfer on the ATA bus added transfer delay.

The network switch was modeled as a full crossbar interconnect. The simulation model assumed no blocking within the crossbar switch. Each link between a brick and the switch ran at 1 Gb/s, with 0.01 ms latency.

The results presented here are from 16 simulations runs for each configuration (each with a different seed). The plotted points are averages of the values from each run. The error bars show 95% confidence intervals. Transience during start up was filtered by discarding data during an initial warm up period.

4.2 Effect of protocol topology

We found that the choice of topology – **direct**, **coord**,

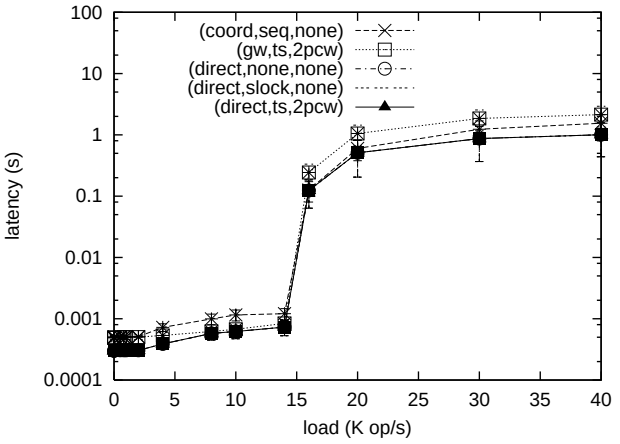

Figure 8: Response time for small reads for both RAID 1 and 4 from one client, usng RAMDisk. All protocols saturate the client network link at about the same IO rate.

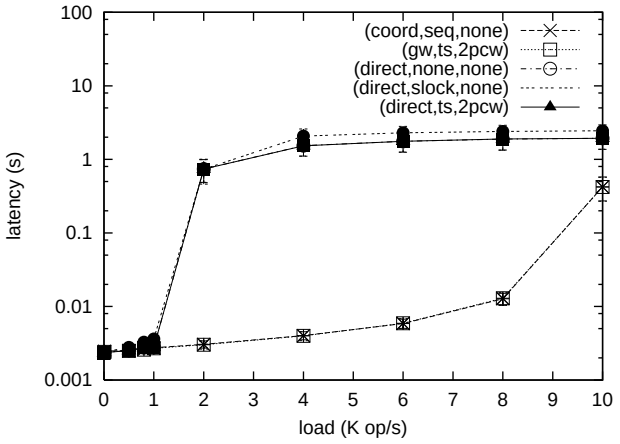

Figure 9: Response time for RAID 4 small writes from one client using RAMDisk. The curves for (coord,seq,none) and (gw,ts,2pcw) overlap as the lower curve. Client-based protocols saturate the client link at low load, while protocols that place the RAID computation load on many bricks saturate at a higher load.

and so on – is a major determinant of performance. The fundamental difference is whether the protocol is doing most of its work on the client or on the brick. If one client is generating a lot of traffic, then the **direct** topology will cause the client to saturate first, consistent with the load analysis in §3. However, if a client is below saturation, the **direct** topology generally gives lower latency than the others.

For this evaluation, we modeled bricks as having RAM disks rather than rotating magnetic disks. This models the system's behavior when the workload can be handled by caches in the bricks, which in turn places the greatest load on the network and memory subsystems. This approach also factors out the effect of the disks in a brick, since the amount of disk traffic generated depends on the RAID layout and not the network protocol.

Figures 8 and 9 show the latency that results when one client performs reads or RAID 4 small writes (respectively) at increasing rates. We modeled one client sending requests to a collection of eight bricks. For reads, all the protocols saturate the network link between the client and the storage cluster at the same load. For small writes, the protocols that perform RAID calculations at the client saturate the client's network link quickly, while the **seq** and **amongst** protocols saturate at a higher IO rate, when the brick performing the coordination saturates.

Figure 10 shows the effect of multiple clients performing IO to a storage cluster, when each client issues IOs at 200 op/s, well below saturation, but the aggregate number of clients increases to saturate the bricks. In this case, the brick-based protocols saturate earlier, while **direct** protocols can support about 50% greater total IOs.

4.3 Network performance sensitivity

Network RAID protocols are sensitive to network behavior and connectivity, unlike centralized RAID approaches. In particular, while the storage bricks are likely to be located in a machine room with a high-performance interconnect, the network connection between a client and storage can vary widely. Moreover, the performance can vary dynamically as interfering traffic comes and goes.

For this investigation, we modeled one client connected to 8 bricks, to ensure that the bricks are not the bottleneck, and varied the latency on the link from the client to the switch while maintaining 1 Gb/s, 0.01 ms links between bricks. The experiment used RAID 4 layout, with a mixture of 2/3 small writes and 1/3 reads with the client generating 100 requests/second. We performed this experiment both with RAM disk and modeling rotating magnetic media. The results are similar for both, though more exaggerated with RAM disk. We show the results for rotating media here.

From this experiment we see that, as expected, a slow client network link penalizes those protocols that perform the most communication from the client (Figure 11). In particular, the **(direct,slock,none)** protocol is slowest at long latencies because it requires communication with both the lock server and the storage bricks, while the **(direct,ts,2pcw)** protocol does better because it avoids the synchronous message exchange with the lock server, and it matches the performance of the idealised **(direct,none,none)** protocol, which is a lower bound on the performance of client-based protocols . The **(coord,seq,none)** protocol does best on slow networks because the operation mixture includes RAID 4 small writes, which requires a read-modify-write message exchange. The coordinator can perform that exchange over the fast links between bricks rather than over the client's link.

4.4 Effect of serialization

Now we turn our attention to the effects of the choices for serialization and atomicity on performance.

Serialization can effect operation latency when the protocol must communicate with a lock server. We saw in Figure 11 that the **slock** protocols had a higher latency than the protocols that performed no messaging for serialization (**seq** and **ts**), and that this difference is sensitive to network performance.

Contention can also affect performance. Contention is a

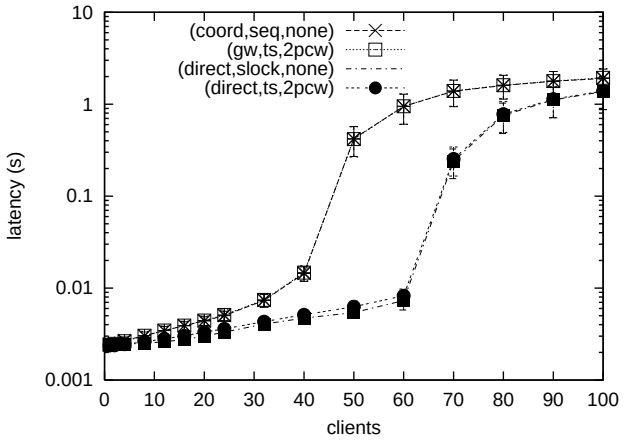

Figure 10: Response time for RAID 4 small write as the number of clients increases, using RAMDisk. The curves for (coord,seq,none) and (gw,ts,2pcw) overlap as the upper curve. Direct protocols saturate bricks at a higher aggregate load because the place work on the clients.

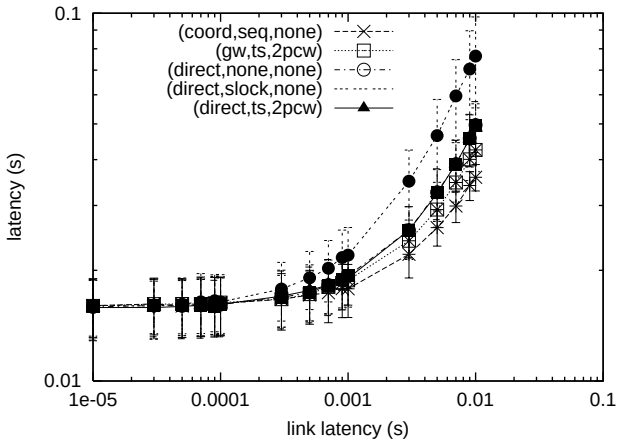

Figure 11: Effect of the latency on the client's network link on response time with rotating magnetic disk model for RAID 4 layout.

side-effect of serialization, and shows itself as an increase in latency: a lock server delaying lock requests, or an optimistic method rejecting and retrying an operation. Optimistic methods are optimized for low contention, while pessimistic approaches do the same work whether contention is high or low.

To compare how these approaches behave we simulated a system with four clients connected to 27 bricks, with data organized in RAID 4 stripes of 4 data plus 1 parity. Each client generated uniform 4 KB, write-dominated random requests at 125 requests/sec. All clients randomly accessed the same data, and so we varied the amount of the shared data to vary the probability of getting conflicting operations.

Figure 12 shows latency as a function of the size of data shared. The **(direct,none,none)** protocol is the lower bound and remains nearly constant regardless of the data size since it does no concurrency control, indicating that the delay

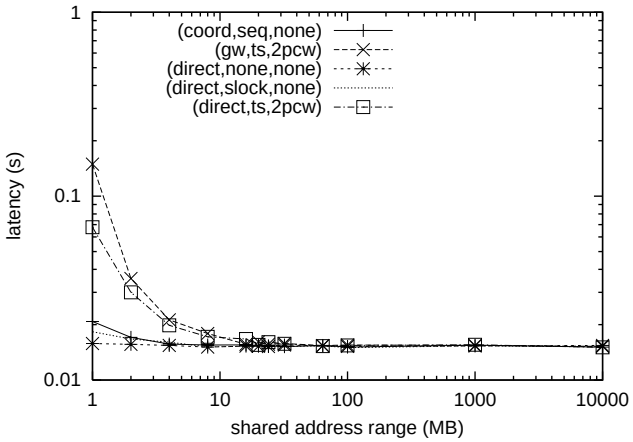

Figure 12: Effect of contention on response time with RAID 4 layout using rotating magnetic disks. Confidence intervals are omitted to improve clarity of presentation. A small shared data area increases the contention, so contention decreases from left to right in this graph. Optimistic protocols have higher response time under high contention than do pessimistic protocols.

came from serialization rather than a saturated brick. Server locking and centralized sequencers both yield significantly better latency than timestamp-based serialization under high contention. Figure 12 also shows that when the data set is larger than about 16MB, there is little difference in contention between the protocols.

4.5 Effect of atomicity

From the experiments we have conducted, there is no difference in latency or saturation throughput that depends on whether the protocol provided no particular atomicity or used two-phase commits on write (**2pcw**). Surprisingly, we found that topology and serialization explained all the performance differences we observed amongst the protocols we implemented.

5. CONCLUSIONS

A network RAID protocol provides for consistent reading and writing of redundant data in a distributed storage system. Such a protocol ensures that redundant data copies are consistent, possibly even when there are concurrent operations and failures. Network RAID protocols are separate from RAID layouts. A RAID layout defines what the replication strategy is to be for some set of data, while the protocol defines how to access that data.

There are many possible network RAID protocols, and many different ones have been used in various systems. We have developed a taxonomy that includes all the protocols we have encountered, and that suggests some new combinations. The taxonomy classifies protocols according to their data path structure, serialization, atomicity and failure tolerance. This allows systematic comparison of the protocols used in different systems, and is the basis for a choosing which protocol to use in new systems.

To implement a system, one must choose a protocol from all these possible feature combinations. The choice depends

partly on the features that the combination provides: if the system only needs single-block atomicity and is designed around crash recovery, simple atomicity mechanisms are likely appropriate.

Protocols are also chosen based on performance. The key factors determining performance are the relative scarcity (or abundance) of memory, I/O, and network bandwidth between clients and bricks, the RAID layout(s) involved, and the mix of operations performed on the data.

We evaluated several protocols for the load they place on a system and the resulting performance, with the goal of understanding the effects of each independent feature choice. We found that the choice of data flow structure and serialization/atomicity are largely independent of each other.

In low to moderate contention environments, serializing updates using pessimistic locking and optimistic two-phase commit plus timestamps give similar performance and load. Locking involves an extra synchronous network round-trip on IOs unless locks can be amortized over multiple operations. However, optimistic approaches can be more expensive at high contention because they may retry operations many times.

These results suggest that there is little performance penalty in most circumstances for providing multi-block atomicity.

6. REFERENCES

[1] Dlock proposal. T10 draft, 2003. 98-224r0.

[2] A. E. Abaddi and S. Toueg. Maintaining availability in partitioned replicated databases. *ACM Trans. on Database Sys.*, 14(2):264–90, June 1989.

[3] K. Amiri, G. Gibson, and R.Golding. Highly concurrent shared storage. In *20th Intl. Conf. on Distributed Computing Sys.*, April 2000.

[4] A. Azagury, R. Canetti, M. Factor, S. Halevi, E. Henis, D. Naor, N. Rinetzky, O. Rodeh, and J. Satran. A two-layered approach for securing an object store network. In *IEEE Intl. Security In Storage Workshop*, 2002.

[5] J. Bucy, G. Ganger, and Contributors. The disksim simulation environment version 3.0 reference manual. Technical report, CMU, 2003.

[6] P. Cao, S. B. Lim, S. Venkataraman, and J. Wilkes. The TickerTAIP parallel RAID architecture. *ACM Trans. on Comp. Sys.*, 12(3):236–67, August 1994.

[7] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. on Comp. Sys.*, 20(4):398–461, Nov. 2002.

[8] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Trans. on Par. and Distrib. Sys.*, 10(6), 1999.

[9] S. Frølund, A. Merchant, Y. Saito, S. Spence, and A. Veitch. FAB: Enterprise storage systems on a shoestring. In *Proc. 9th Workshop on Hot Topics in Op. Sys.*, May 2003.

[10] R. Golding. The Palladio access protocol. Technical Report HPL–SSP–99–2, HP Laboratories, Storage Systems Program, November 1999.

[11] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. Efficient consistency for erasure-coded data via versioning servers. Technical Report CMU-CS-03-127, CMU, March 2003.

[12] J. H. Hartman and J. K. Ousterhout. The Zebra striped network file system. *ACM Trans. on Comp. Sys.*, 13(3):279–310, Aug. 1995.

[13] H. Kung and J. Robinson. On optimistic methods of concurrency control. *ACM Trans. on Database Sys.*, 6(2):213–226, June 1981.

[14] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. ACM*, 21(7):556–65, Jul. 1978.

[15] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proc. 7th Intl. Conf. on Architectural Support for Programming Languages and Op. Sys.*, pages 84–92, 1996.

[16] D. D. E. Long, B. R. Montague, and L.-F. Cabrera. Swift/RAID: A distributed RAID system. *Computing Systems*, 7(3):333–359, 1994.

[17] D. L. Mills. Network time protocol: specification and implementation. Internet RFC 1059, July 1988.

[18] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (raid). In *ACM SIGMOD Intl. Conf. on Management of Data*, pages 109–16, June 1988.

[19] The VINT Project. *The* ns *manual*, October 2003. http://www.isi.edu/nsnam/ns/doc/ns_doc.pdf.