Cuckoo: Layered clustering for NFS

Andrew J. Klosterman, Gregory Ganger

October 2002
CMU-CS-02-183

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Layered clustering allows unmodified distributed file systems to enjoy many of the benefits of cluster-based file services. By interposing between clients and servers, layered clustering requires no changes to clients, servers, or the client-server protocol. Cuckoo demonstrates one particular use of layered clustering: spreading load among a set of otherwise independent NFS servers. Specifically, Cuckoo replicates frequently-read, rarely-updated files from each server onto others. When one server has a queue of requests, read requests to its replicated files are offloaded to other servers. No client-server protocol changes are involved. Sitting between clients and servers, the Cuckoo interposer simply modifies selected fields of NFS requests and responses. Cuckoo provides this load shedding with about 2000 semicolons of C code. Further, analyses of NFS traces [7, 8] indicate that replicating only 1000–10,000 objects allows 42–77% of all operations to be offloaded.

We thank the members and companies of the PDL Consortium (including EMC, Hewlett-Packard, Hitachi, IBM, Intel, Microsoft, Network Appliance, Oracle, Panasas, Seagate, Sun, and Veritas) for their interest, insights, feedback, and support. We thank IBM and Intel for hardware grants supporting our research efforts. This material is based on research sponsored in part by the Air Force Research Laboratory, under agreement number F49620-01-1-0433.

Keywords: Cuckoo, NFS, replication, overload, load shedding, file systems, trace analysis

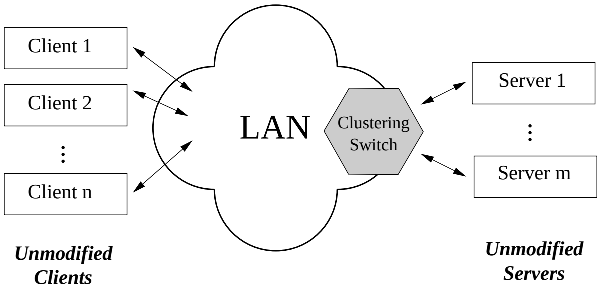


Figure 1: **Layered clustering architecture.** Clients, servers, and the client-server protocol are unmodified. The "clustering switch" is the only change, and it's role is to add the clustering functionality by transparently translating some client requests into redirected server requests. The same role can be played by a collection of small intermediaries at the front-ends of the servers.

1 Introduction

File services implemented as cooperating clusters of servers would be great [2, 10, 12, 13, 24]. They offer incremental scalability of storage capacity and performance. They can spread data amongst themselves so as to balance the workload. They can keep redundant data for fault tolerance. They can provide high-end features with commodity components. For many years, the community has known their superiority to today's common file service architectures.

Unfortunately, though, their market penetration is minimal. Particularly in mid-sized environments, it remains common to have one large server or a small set of stand-alone servers. Perhaps it is the client-side changes required for most cluster designs that hampers their deployment. Perhaps it is the "different-ness" or the desire to leverage existing investments. Perhaps it is a chicken-and-egg problem: cluster file services are not common because there is no popular cluster file system protocol. Whatever the reason, the well-known advantages are not being enjoyed. Worse, the particular architectures of popular distributed file systems (notably, NFS and CIFS) make scalability and load balancing difficult and time-consuming.

This paper explores an alternate architecture, *layered clustering*, which seeks to achieve a large fraction of the benefits of cluster file services with minimal change to existing systems. Specifically, layered clustering leaves clients, servers, and the client-server protocol unchanged; it interposes a small piece of software in front of servers, either at the front-end of each server or in a network component to which they are attached. (See Figure 1.) This small piece of software can selectively forward requests addressed to any given server to another, potentially achieving many of the cost-effectiveness, scalability, and load balancing benefits of a full cluster file service.

We believe that full clustering is clearly the correct solution, if one has a clean slate. But, given the inertia of installed bases, it is interesting to explore how far one can get with minor changes to existing setups. If able to achieve a significant fraction of clustering benefits, layered clustering is a compelling and practical option.

This paper makes a case for layered clustering of traditional file servers. In particular, it argues for read-only replication of popular files and selective shedding of requests from busy servers to replica holders. Analysis of several NFS traces indicates significant potential for spreading load among departmental servers. In particular, 45–83% of the requests to such servers are read-only.

Further, replicating only 1000 – 10,000 data objects (files or directories) allows 42–77% of requests to be offloaded with no need for updates and no consistency issues.

As a proof of concept, we describe Cuckoo[1], which provides request offloading for NFS in about 2000 new lines of code in each server. An offline planner selects and replicates files, and the new server code uses the replicas as needed. Only read operations can be offloaded; all write operations are handled by the corresponding file's home server. No replica updating is done, and a home server simply stops using replicas that become out of date. Cuckoo's very simple design demonstrates the promise of layered clustering.

We further describe how the same Cuckoo functionality could be repackaged into a smart switch, providing the same benefits with no changes to existing NFS clients or servers. Such an architecture could do for distributed file systems much of what load-balancing switches have done for multi-machine web services.

The remainder of this paper is organized as follows. Section 2 motivates layered clustering, provides background, and discusses related work. Section 3 discusses layered clustering designs for load balancing. Section 4 describes Cuckoo. Section 5 evaluates the Cuckoo approach to load balancing for NFS services. Section 6 discusses how Cuckoo could be embedded in a switch. Section 7 summarizes the paper's contributions.

2 A Case for Layered Clustering

Layered clustering converts a set of stand-alone traditional servers into a cooperating cluster. In layered clustering, unmodified clients continue to use traditional protocols (e.g., NFS or CIFS), and servers continue to own and export independent file systems. Intermediary software translates selected requests into other requests of the same protocol, whose responses are transparently delivered back to the original client. For example, read requests to files stored on busy servers could be offloaded to alternate servers holding replicas, which would then respond to clients on behalf of the busy server.

This section argues the case for layered clustering. It overviews some benefits of cluster-based file services, speculates on why they are not yet prevalent, discusses an example of layered clustering, identifies some assumptions and limitations, and discusses related work.

2.1 Benefits of clustering

In traditional distributed file systems, such as NFS or CIFS, a server exports one or more file systems to clients. Clients mount exported file systems into their local namespace, allowing applications to use them transparently. Multiple servers can be used, with each one independently exporting file systems. Clients send requests for any given remote file to the server that exports the file system containing it.

In cluster-based file systems, a set of servers collectively export a file system to clients. Files and metadata are spread across the servers, via striping [5, 10] or mapping tables [2], and clients send requests to the appropriate server(s). Cluster file systems have a number of advantages over

[1] Cuckoo hens lay their eggs in the nests of other species, leaving those birds to incubate and raise the chicks as surrogate parents [4].

the traditional model, including load balancing, fault tolerance, incremental scalability, and graceful degradation.

Most cluster file systems automatically balance load across the set of servers. This is a known property of striping [14]. Alternatively, files can be assigned to servers based on observed access patterns. A balanced load benefits from the CPU, memory, disk, and network bandwidth capabilities of all servers, without one server becoming a bottleneck. With traditional servers, on the other hand, each server's load is dictated by the portion of the namespace that it exports. Imbalance is common, and significant effort and downtime are needed to better balance loads. Specifically, portions of the namespace must be redistributed amongst the servers, and all client mountpoints must be modified. Conventional wisdom says that such balancing is rarely performed.

Many cluster file system designs also maintain data redundancy, either via replication [6, 24] or parity [2, 10, 15], to provide fault tolerance. With traditional servers, a single failure makes a portion of the namespace unavailable.

Many cluster file system designs support both incremental growth and graceful degradation. As servers are added, files are migrated or replicated to them and their resources contribute to the cluster. As servers fail, redundancy allows requests to be served, and the load can be redistributed evenly over the remaining servers. Neither of these properties are present in the traditional model.

2.2 Slow market penetration

The benefits of cluster-based file services are compelling and have been known for many years. We can only speculate as to why cluster file systems are not yet widely used. We suspect that the two main hindrances are the need for client-side changes and the desire to leverage existing server investments.

Quite likely, the main hold-up is a chicken-and-egg problem. To work well, cluster file systems usually involve client-side functionality. Since there is no standard cluster file system protocol, the corresponding software is not a default component of desktop operating systems. Thus, implementers and administrators must put some effort into integration and testing with each new release of the operating system.[2] On the other hand, the incumbent traditional file systems (e.g., NFS and CIFS) just work, giving them a substantial advantage.

Another hurdle, in mid-sized environments, is the inertia of an installed base. As an organization gets started, system administrators can conveniently install a single file server using a traditional file system protocol; going with a cluster file system from the beginning requires forethought and extra investment. When user requirements grow beyond the single server, the administrators can either replace the existing server (and the associated expertise) with an entirely new cluster file system or buy a second standalone server and manage both. This same decision is incrementally faced, and the expedient decision often made, until the server farm reaches a size where the administrative load becomes unacceptable.

Another issue, of unknown practical significance, is that many system administrators are unfamiliar with cluster-based file services. This lack of awareness makes clustering a tougher sell

[2] This speculation is based on anecdotal experiences with AFS [11], as reported by system administrators at Carnegie Mellon University [3]. AFS remains popular at Carnegie Mellon, because it was invented there and because it has superior administrative features, but it does not have market share anywhere close to its competitors. As such, support for it is not integrated into desktop operating system code bases. Some effort is involved with verifying and patching the AFS client software into each supported version of Linux and Windows.

and also creates some negative word-of-mouth, because of new configuration tasks and new failure modes. Re-education of administrators is an additional hurdle faced.

2.3 Load balancing via layered clustering

Layered clustering offers some of the benefits of full cluster file systems without the issues slowing their deployment. Particularly for mid-sized environments, layered clustering offers a low-investment alternative to client-side changes and standalone server replacement. By simply sliding an intermediary component into place, existing servers are made to work together with no other changes.

One very promising use of layered clustering is load balancing across a set of servers. If a subset of popular, read-mostly files are replicated onto other servers, intermediary software can shed read requests from busy servers to other servers. As described in Section 4, this requires very little code. The practical value of this approach depends mainly on how effective it is at providing the desired load balancing features. For request offloading to be effective, two things must be true of the environment: (1) a significant fraction of requests must be offloadable, and (2) the process of offloading must be efficient.

The simplest use of layered clustering allows only read operations, including directory lookups and attribute reads, to be offloaded. Fortunately, previous studies indicate that such operations make up a large percentage of the requests in real environments. For example, Gibson et al. report breakdowns for substantial NFS and AFS environments in which read operations comprise 85–94% of all requests and 78–79% of all server CPU cycles [9]. Other file system workload studies [18, 25] report similar dominance of read activity. For read offloading to help, it is also necessary for a substantial fraction of these read requests to go to files that are modified infrequently—otherwise, the replicas will rarely be up-to-date when needed. Section 5 shows that 45–83% of all requests are reads of rarely-modified files in departmental NFS environments.

Implemented properly, request offloading in layered clustering should also be efficient. If implemented in a smart switch, offloading at peak request rate should be possible, since it requires only minimal processing of RPC headers; Section 6 describes this further. If implemented with server-side intermediaries, all offloaded requests are sent by the client to the home server of the given file, which can then take action. Thus, offloading efficiency has two parts: the work required of the home server and the network latency between servers. Again, rapid decision-making should be possible at a low level of the protocol stack, minimizing processing overhead. As for inter-server communication, we expect that servers participating in layered clustering would be well-connected in a machine room, which should minimize network delays.

2.4 Assumptions and limitations

Layered clustering assumes a “machine room” model of multi-server environment, in which the servers are well-connected and equally trusted. Such a configuration allows a clustering switch to be inserted between the servers and all clients or, at least, high-bandwidth, low-latency communication between software components running on the servers.

Although we do not know the upper bounds, we believe that layered clustering makes sense mainly for relatively small numbers of servers (e.g., 2–20). Beyond such numbers, true clustering is probably too superior an approach.

Layered clustering assumes that the only access to served files is via the client-server protocol passing through the intermediaries. If there is a local access path, the clustering infrastructure may act on out-of-date information because the server is not involved.

The read offloading use of layered clustering has two main limitations relative to true clustering. First, all operations on files that change must be performed at their home server, bounding the set of offloadable operations. Second, no fault-tolerance benefits are provided, unlike many true clustering solutions.

2.5 Related work

There has been a huge amount of work in distributed file systems. Here, we focus on particularly relevant categories of related work.

Layered clustering builds on the proxy concept [19], using interpositioning to add clustering to an existing client-server protocol. Slice [1] uses this same concept to transparently provide unmodified clients access to scalable network storage. Layered clustering seeks to change neither the clients nor the servers, aiming for a subset of the design space with minimal changes to existing infrastructure. The fundamental consequence of this choice is less flexibility. A practical consequence is that the intermediaries are placed at the server side rather than the client side.

Some recent work in academia and in startups share Cuckoo's layered clustering architecture. Anypoint aggregates an entire ensemble of servers into a single virtual server by redirecting requests to the appropriate server [26]. Similarly, ZForce offers "file switches" that aggregate standard file servers into a single name space [27]. They allow file types (based on file extension) to be stored in stripes or mirrors. Our Cuckoo prototype demonstrates a more incremental usage of layered clustering, but these are all forms of the general architecture.

AFS [11] provides several administrative features that would make layered clustering less necessary. Most notably, a set of AFS servers provide a uniform namespace broken up into directory subtrees called volumes. The client's view of the file system is independent of which servers serve which volumes, and transparent volume migration is explicitly supported. Further, read-only volumes can be replicated on multiple servers and the clients can send their requests to any replica. Read offloading goes beyond this by allowing servers to shed load for read-write volumes as well, but is of greater value to less sophisticated systems.

Layered clustering is analogous to web server clustering support offered in some network switches, which can improve load balancing in Internet server farms [17]. Doing this for file servers does require more effort, because of frequent updates and long-lived interactions (either via sessions or via server-specific filehandles), but should provide similar benefits.

Another approach, sometimes used in large installations, is to have unmodified clients interact with a set of front-end file servers that share a collection of back-end storage components. Caching and consistency issues could be substantial, but are usually avoided by having different servers use different portions of each storage component. This approach can provide load balancing for the storage components as well as fault tolerance via "fail over."

3 Cuckoo Overload Shedding

Cuckoo uses an interposition agent to transparently redirect client requests from busy servers to servers holding replicas of popular data. The interposition agent resides on the network path between the client issuing a request and the server to which the request is directed. Some network packets containing requests are forwarded to other servers holding replicas of popular data. Servers satisfy redirected requests and replies are directed back to the client.

Many design points exist within the general Cuckoo architecture. The server components are the home server (where a data item is authoritatively stored) and the surrogate server (possibly multiple servers that store replicas of a data item). The interposition agent, that redirects requests and responses, can reside anywhere on the network path between the clients and servers. As such, it could be a part of the client protocol stack [1], a component of an intervening network element (e.g., the "clustering switch" of Figure 1), or a part of the server protocol stack. The forwarding of requests to surrogate servers could be done by re-writing the original packet and placing it on the network, or by encapsulating the original request in a new RPC and sending it to an available server. Most existing NFS protocol implementations (e.g., NFS on IP/UDP) require that responses come from the server to which they were directed. The manner in which Cuckoo enforces this will depend on other design choices.

Furthermore, the type of offloaded requests effects a Cuckoo implementation, especially its complexity. Read-type operations[3] are the easiest to offload because the data returned by either the home server or any replica site is identical as long as the replicas are valid. Replica sites would maintain a mapping of the original filehandle for a file or directory to the file handle being used to store it locally. This mapping would then be consulted when satisfying forwarded requests. Write-type operations[4] introduce additional complexity with consistency and failure-mode issues. Since cluster file systems deal with these same issues, there are well known techniques that Cuckoo could use to support write offloading.

3.1 Components

There are eight components in a Cuckoo load shedding system. The first of these components are the replicas of storage objects (directories, links, and files) spread throughout the system. The second and third are the decision-making modules needed to determine which objects should be replicated and make load-shedding decisions. The fourth is an access trace necessary to provide data for the decision makers to use. The fifth and sixth components are data structures needed to indicate which storage objects are replicated on which servers and to translate requests for those objects. The seventh and eighth are packet rewrite abilities to satisfy requests remotely and to convince clients that their requests are being serviced by the home servers.

3.1.1 Replicas

Object replicas are stored on surrogate servers as normal objects of the same type (i.e., file, directory, or link). Once created, replicas of storage objects can exist indefinitely within the storage

[3] In NFS version 2, the read operations are GETATTR, LOOKUP, READLINK, READ and READDIR.

[4] In NFS version 2, the write operations are SETATTR, WRITECACHE, WRITE, CREATE, REMOVE, RENAME, LINK, SYMLINK, MKDIR and RMDIR.

system. They exist solely to benefit the home server from which they originated whenever requests are offloaded from it. Storing replicas is not much different from storing any other object within the system. The only additional requirement is that when created, every reasonable effort is made to set the attributes of the replica to be identical to those of the original. Any attributes that cannot be copied will be compensated for by substituting data and rewriting packets as they are in transit to the clients that originated a request.

3.1.2 Replication decisions

The creation of replicas can be performed during system idle time by a planning entity outside of the critical path of the client-server protocol. With no knowledge of the server implementation, and the correct access permissions, this “planner” can simply use existing file system commands (e.g., READ, WRITE, GETATTR, SETATTR) to create replicas between servers. During this process, the planner can learn of any device-specific file information and save it to be included in the data structures for request translation.

3.1.3 Load-shedding decisions

The component that decides whether or not requests should be offloaded must be able to quickly reach a decision. It should not add a significant amount of work to the critical path of a request when the system is already very busy. Furthermore, once it has decided to offload a request, it should quickly decide to which replica to redirect the request.

3.1.4 Trace of accesses

A straightforward way for the active system to provide information to the planner is by recording a trace of accesses. Trace information can be scanned for patterns and used to make determinations on which files should be replicated. Information available in the trace should include timestamp, server identity, operation, arguments (at least one filehandle), and whether or not the operation was offloaded.

3.1.5 Replica table

In order for requests to be redirected, a table identifying replicated files must reside in a decision-making element located along the protocol path between clients and servers. This “replica table” must contain information about each replicated object and the location(s) where it is available. This list would be consulted on every write-type operation to make sure that any replicated object is removed from the list as soon as its replicas are made inconsistent with the home server’s copy of the object. It would also be consulted on read-type operations whenever the home server for the object is busy and requests are being offloaded.

3.1.6 Attribute table

A second table of information associated with each replicated file must also be maintained. This table is consulted as data is returned from replica servers and used to “correct” the responses being returned to clients. Corrections are needed to completely cozen the clients, since there are some attribute fields that are server-dependent and must be changed to match the values that the home server would respond with.

3.1.7 Request re-write

In order to remotely satisfy a request using a replica, any offloaded requests must appear to be directed to a file on a surrogate server. This can be done by simply changing the filehandle that refers to the object on the home server to reflect the filehandle that the replica is stored under on the surrogate server. A “filehandle translation table” is maintained for this purpose and contains pairs of filehandles: the original filehandle for an object and the filehandle of its replica. This table is separate from the “replica table” (while records the filehandles of replicated objects) but could be merged if desired.

3.1.8 Response re-write

When the response for a forwarded request is sent from a surrogate server, it must be modified to be of use. The packet must be made to appear to originate at the home server so that it will be accepted by the client. Furthermore, any irreproducible attributes for the object must be replaced by the values that the home server would return to the client, which are stored in the attribute table.

3.2 Decisions

The decision components make choices as to which storage system objects are replicated and whether or not (and which of) those replicas are available for use.

3.2.1 What to replicate?

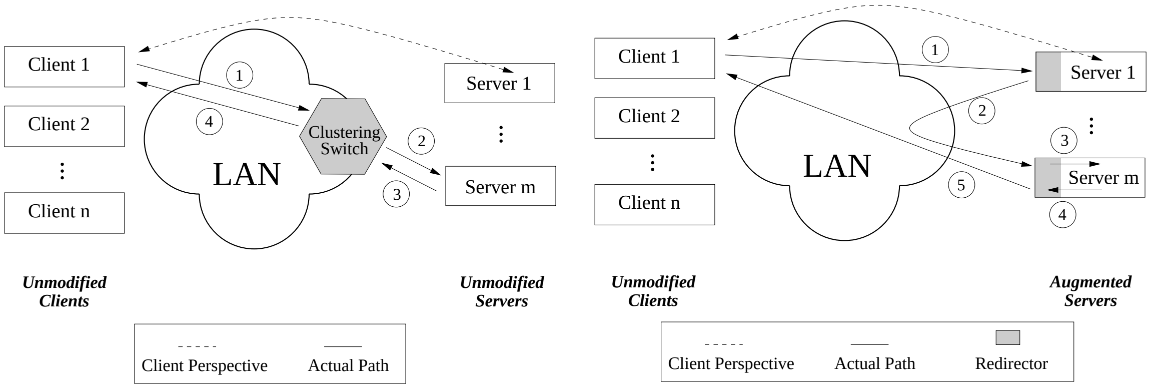
The process of deciding which objects to replicate is dependent upon the access pattern of system activity. Using the trace, an entirely offline algorithm can be run by the planner to decide upon replicas, create them, and then integrate those replicas into the running Cuckoo system.

The ideal files to replicate are those that are frequently referenced and read-only. Generating this list of objects is as simple as ranking all filehandles from most accesses to fewest and discarding all filehandles that see write-type operations performed on them.

Objects that are infrequently written may also be good candidates for replication as the replicas may be expected to be valid for the greater part of the time between planner executions (the “replication window”). Again, a rank ordering of the filehandles would suffice. The twist with “read-mostly” objects then becomes identifying whether or not a replica would be invalidated rapidly or could be expected to survive for most of a replication window.

3.2.2 When to use replicas?

The process of deciding when to use a replica is dependent upon current system activity. Whenever a home server is deemed to be overloaded, all opportunities for load shedding should be embraced.



(a) switch-embedded redirection

(b) server-side redirection

Figure 2: **Possible Cuckoo implementations:** These pictures show two possible implementation schemes for Cuckoo. Both maintain the same client perspective of communicating entirely with a home server (Server 1) for all operations. However, each has a different mechanism for handling operation offloading, as explained in Section 3.3.

3.2.3 Which replica to use?

The process of deciding which replica to use is dependent upon recent system activity. Whenever multiple replicas are available, an operation should be offloaded to whichever surrogate server last saw a request for the filehandle in question. This increases the likelihood of the file actually being in cache on the surrogate. Whether or not the file is actually cached, of course, is conditional upon the workload presented to the surrogate server and how recently the the file was last referenced on the surrogate.

3.3 Location of functionality

Of the eight Cuckoo components, only one must reside on the servers: the replicas, stored as ordinary files, directories and links. The remaining seven components reside along the client-server protocol path, for example, inside the “clustering switch” shown in Figure 2(a), or distributed as a thin layer of software on each server as shown in Figure 2(b). In both cases, the positioning of the components allows them to act upon requests sent to the servers and upon responses directed toward the clients.

With components concentrated in a clustering switch, load-shedding decisions can be made at the same time as packet forwarding decisions for requests. The access trace records can also be generated at this time, with the planning component reading the trace at its leisure. The clustering switch would have the replica table available and would perform any necessary packet rewriting to update a filehandle and address a request to a surrogate server. This clustering switch would also rewrite responses sent from surrogate servers to clients to properly forge the identity of the home server to which the request was originally directed.

Figure 2(a) shows requests first passing into a clustering switch where they are rewritten and then forwarded in the second step to a surrogate server holding a replica. The reponse is then sent back through the switch in step three which finally rewrites the response so the client's perspective is maintained.

With components distributed across each server, as in Figure 2(b), an additional network hop in incurred for each offloaded operation. Each server is responsible for tracing accesses to itself and making decisions about load shedding. Each server also knows which of its objects are replicated and where those replicas lie. Handling of offloaded requests would have to happen outside of regular NFS RPC channels so that a surrogate can translate a request's filehandle to a local filehandle and know to which client to respond.

Figure 2(b) shows redirection handled by the home servers for replicated data. When a client request for replicated data first arrives at a busy server, the server forwards that request to another server (the second step). That surrogate server translates the request to act upon a local filehandle in the third step and it is serviced in the fourth step. Before returning a response to the original client, the response must be modified to appear as if it came from the home server (the fifth step).

Both of these scenarios share issues of where replication decisions are made and handling of the network transport protocol. The decisions on what to replicate, and the replica creation, are performed through an external means in both cases. A planning system pores over the access traces, creates replicas, and updates the replica tables. In order to cozen the client systems into believing that they are receiving data from a home server, rather than the surrogate server that actually services the request, packets must appear to come from the home server. This means that the network and transport layer headers appear to the client as if they originated from a home server. Using raw sockets to forge packets trivializes this problem for UDP/IP. While the TCP/IP combination presents a more difficult challenge, it is successfully done in HTTP-redirecting switches via TCP splicing [16] and recently for NFS over TCP.

4 Implementation of Cuckoo

The current implementation of Cuckoo follows the model illustrated in Figure 2(b). A modified user-level NFS server provides storage to the system. Creation of replicas and population of the replica list is performed via a command line tool. Core components of Cuckoo are inserted into the server's routines for handling requests. The remainder of this section describes the Cuckoo NFS server and describes how it offloads operations.

4.1 The Cuckoo NFS server

The prototype Cuckoo NFS server is built into an updated version of the S^4 user-level NFS server [21] and supports NFS version 2 [23]. Cuckoo has no dependency on the particular features of S^4 and could be implemented, in the fashion described here, in most NFS servers. This particular user-level NFS server implementation was chosen primarily for ready access to the authors.

Cuckoo was implemented by adding additional data structures, conditional function calls in most NFS service routines, and a new set of RPCs. The data structures hold information about what replicas are located on what servers and how to translate offloaded operations into local requests and extra attributes. The additional code in the NFS RPC service routines allows operations to be offloaded when the server is busy and watches for writes to replicated data. A new set of RPCs populates the replica lists and request translation tables as well as providing a means for offloading operations.

4.1.1 Data structures

There exist tables within the Cuckoo NFS server prototype for handling inbound requests and for servicing offloaded operations. These are the “replica table” and “filehandle translation table” described in Section 3. Furthermore, the replica table is augmented with a “replica site table” to reduce its memory footprint.

The replica table is maintained by the home server for data objects and it contains a mapping of which objects are replicated on which servers. This table has one entry for each replicated object. These entries contain the following information: the filehandle of the replicated object, a bitfield of replica sites, and a cached access time (“atime”) attribute. The bitfield of replica sites has a bit set for each of up to 32 replica sites that this server is in contact with and which hold replicas of this particular object. These bit positions correspond to indices in a statically configured “replica site table,” having one entry for each replica site, that holds information on how to contact that replica site (e.g., an IP address and port number).

The filehandle translation table is used when servicing offloaded requests to locate the local copy of a replicated object and to correctly impersonate the home server for the data when responding to a client. This table has one entry for each replica being stored by the local NFS server. These entries contain the following information: the filehandle for the object on its home server, the local filehandle, and copies of irreproducible attributes. The irreproducible attributes are those attributes that cannot be set in the metadata of a replica to exactly match the values that would be returned by the home server for the data. Such attributes are found in the `fattr` NFS version 2 structure (e.g., `fsid`, and `fileid`).

These structures are updated when new replicas are made or existing replicas are written to. As new replicas are created, new entries are inserted by the home server into its replica table and by the replica sites into their filehandle translation tables. As replicated objects are written to, entries are removed from the home server’s replica table, effectively invalidating all replicas. After such a write, the replica sites still hold their copies of the data, and they will continue to do so until they are told that the replica is no longer valid. The planning component of Cuckoo knows which files have been replicated and can decide to re-replicate or delete replicas that have been written to.

4.1.2 Additional code

Code is added to the NFS RPC service routines to cover two distinct cases: writes and reads to replicated filehandles. Any observed writes to replicated filehandles immediately cause replicas to be invalidated by removing the entry for the filehandle from the replica table. During times of high load, reads of replicated filehandles are watched for so that they can be offloaded.

Each of the write-type NFS RPC service routines checks that the filehandle it is operating on is not replicated before proceeding. This check is performed with a lookup by filehandle in the replica table. If found in the replica table, the filehandle’s entry is removed so that subsequent operations cannot access the replicas, which are now inconsistent with the data on the home server. Then, the operation proceeds.

When the load on the home server is high, lookups into the replica table are performed as a part of servicing each read-type operation in hopes that the operation can be offloaded. Any time a lookup is successful, the operation is offloaded to a replica site for the specified filehandle with a Cuckoo RPC call.

There is one piece of S^4 that Cuckoo leverages: its audit log entries. The S^4 NFS server makes a note of every access and Cuckoo uses this as its access trace. The Cuckoo planner analyzes this trace to make replication decisions and to refresh replicas whose primary copy has been modified. Such functionality could be implemented in Cuckoo with a simple log file that records each operation and its arguments.

4.1.3 New RPCs

Cuckoo introduces an additional set of remote procedure calls in order to successfully offload operations and to populate its replica tables and filehandle translation tables. One RPC is introduced for each offloadable operation to encapsulate the arguments for that operation and to forward the dynamic information (e.g., `atime`) necessary for the replica site to successfully forge the home server's response. There is another RPC that updates the replica table with new locations of replicas of a particular filehandle. Updating the filehandle translation table involves an RPC that conveys the home server filehandle, the replica site filehandle, and the irreproducible file attributes.

4.2 Making replicas

In the Cuckoo prototype, replicas are not automatically created after scanning the access trace. Currently, a command line utility is used to copy objects between home servers and replica sites. In the process of doing so, this program learns the filehandles for the object on both servers, can set the attributes on the replica server to match the home server, and knows the irreproducible attributes. Once the file is copied, the replica site has its filehandle translation table updated, and then the home server's replica table is updated. This makes the replica known on both servers, and available for use.

4.3 Inbound requests

As NFS requests arrive at the server, they are dispatched to the service routines that have been modified to implement the Cuckoo functionality.

Operations are offloaded if there are more than a threshold number of operations currently being processed. The first replica site for a filehandle is forwarded the operation in the current implementation. Implementing smarter choices of replica sites is future work.

To offload an operation, the operation's arguments and the information necessary for a replica site to correctly forge a response back to the client are packaged into a RPC and sent to the Cuckoo service running on the replica site. Necessary information includes the home server's IP address and port number, the SunRPC transaction ID [22] of the operation (available in a reserved field of the Sun RPC structure in some versions of `glibc`), and the client's IP address and port number.

With the operation forwarded, an additional step must be taken: the normal response from the home server to the client must be suppressed. This is to prevent the client from getting multiple responses to its request. To prevent the home server from responding, a slight modification is made to the SunRPC generated dispatch routine to suppress the sending of a response when a flag is observed upon return from the NFS service routine.

4.4 Offloaded operations

When replica sites receive RPCs to perform offloaded operations, they commit to performing them whenever their load does not exceed their own threshold for offloading operations. The operation is simply dropped if their threshold is exceeded and it becomes the client's responsibility to retry the operation.

Once committed to servicing an offloaded operation, the filehandle encoded in the operation's argument is looked up in the replica site's filehandle translation table. The local filehandle is inserted into the request and then submitted to the local NFS server which retrieves the desired information. When the request completes, a network packet is formed with information forwarded from the home server and sent onto the network via a raw IP socket. In all regards other than Ethernet address, the client thinks that it is receiving information from the home server of the originally requested data.

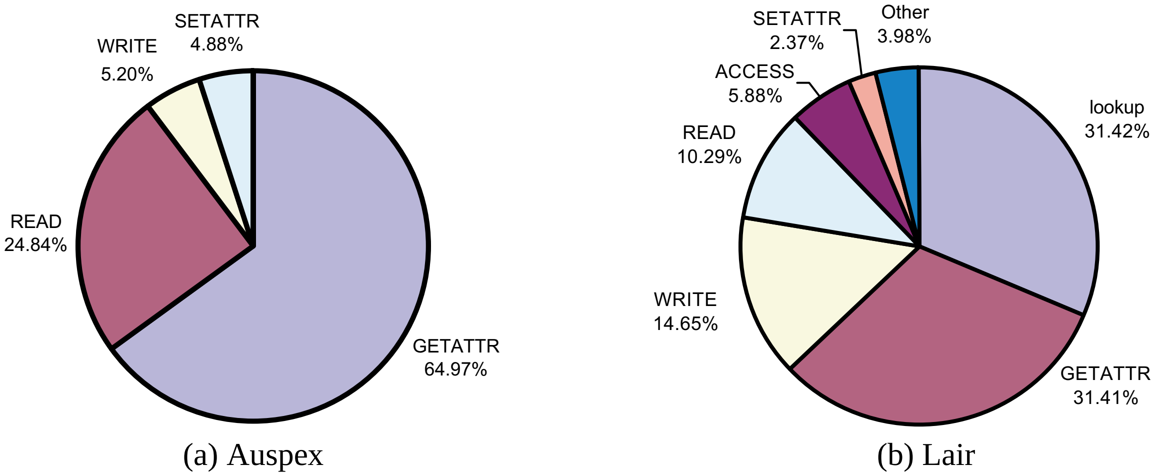
Some NFS operations, however, must be treated specially by Cuckoo when they are serviced: READDIR and LOOKUP. The NFS READDIR operation contains a "cookie" indicating where it should begin in a request and where it left off in a response (so that a multiple command READDIR can get the listing of a large directory). Cuckoo can offload such requests whenever the entire directory listing can be returned to the client in one response packet. This situation removes the possibility that a client would use the returned directory cookie to try to read further into a directory. Since a Cuckoo cluster may be heterogeneous, and server implementations might use cookies in different ways, no guarantees can be made that cookies would be useful in the case where two sequential READDIR operations were serviced by different servers. Furthermore, for offloaded READDIR operations, the entire directory must be replicated so that the surrogate server would have all pertinent information to return to the client.

The NFS LOOKUP operation can only be offloaded if the entire directory has been replicated on a surrogate server. As usual, satisfying such an offloaded request involves first translating the parent directory filehandle into a local filehandle using the filehandle translation table. Then, the local directory can be found and the requested filename located. Once again, the filehandle translation table is consulted to translate the local filehandle back into the original filehandle. This value can then be returned to the client in the NFS LOOKUP response.

5 Evaluation

In this section, we examine two week-long network-level traces of NFS client-server interactions. Investigation of characteristics of these traces will help us determine whether Cuckoo is a useful instance of layered clustering for load-shedding NFS servers. Each workload is of an academic department with shared storage systems being used for research, class, and project development purposes. Both traces were gathered at the network level. The first trace, hereafter referred to as the Auspex trace, is described by Dahlin, et al. [7] and captures the NFS activity generated by 236 clients on four Ethernets connected to an Auspex server. The second trace, hereafter referred to as the Lair trace, as described by Ellard, et al. [8], captures the NFS activity of the Harvard University EECS Department accessing a Network Appliance filer.

5.1 Operation breakdown



(a) Auspex (b) Lair

Figure 3: **Operation Breakdown:** Components (a) and (b) of this figure show the breakdown of operations observed in each of the analyzed traces. In the Lair trace, NFS version 3 operations dominated and are shown in uppercase letters; NFS version 2 LOOKUP operations are indicated in lowercase. All Auspex operations are considered to have been performed on files – there was no explicit directory access information available to us in the trace. A small fraction (0.11%) of Auspex operations could not be accurately classified. For the Lair trace, the "Other" category represents the contribution of all other operations to the total.

Cuckoo's design is predicated on an expectation that at least some important workloads are read-dominant, meaning that the majority of operation performed are reads of data and metadata.

Figure 3 shows the breakdown of operations performed in the two traces. As we see, read activity does dominate, with metadata reads accounting for over 62% of accesses in both cases. Overall, read-type operations comprise almost 90% of operations for Auspex and 79% for Lair.

The Auspex trace has some peculiarities, however, that reduce its fidelity. The form of the trace available for analysis was sanitized to the point that it no longer held RPC calls and arguments, but less specific descriptions of operations. For instance, the "READ" classification contains the NFS operations for file reads and directory reads (READDIR). The directory reads were not broken out into their own classification because they accounted for a very small portion of the trace. Similarly, the "GETATTR" classification contains the attribute accesses made by the NFS LOOKUP operation, but these could not be distinguished.

The Lair trace provided excellent information about each operation captured. Most operations recorded in the trace were NFS version 3 operations; the only NFS version 2 operation that appears on the pie chart is the LOOKUP operation, which was the only version of LOOKUP in the trace.

Overall, these traces confirm the intuition that read operations are a useful target.

	Auspex Trace				Lair Trace			
# writes	# fh	% fh	# ops(K)	% ops	# fh	% fh	# ops(K)	% ops
0	115477	59.77	13936	81.45	244640	52.80	15331	50.14
1	120045	62.14	13988	81.75	292818	63.20	15494	50.68
2	124689	64.54	14039	82.05	332253	71.71	16069	52.56
3	126709	65.59	14092	82.36	365767	78.94	16398	53.63
4	128148	66.33	14119	82.52	376190	81.19	16586	54.25
5	129124	66.84	14129	82.64	395153	85.28	17144	56.08
10	136260	70.53	14240	83.23	406906	87.82	17446	57.06
Total writes	1724152				5971965			
Total ops(K)	17110				30573			
Total fh	193191				463355			

Table 1: **Read-mostly filehandles and their operations:** This table shows the number of filehandles present in the traces, along with the number of operations issued to those filehandles, for a given number of write operations observed. A large percentage of the overall operations are issued to many read-only filehandles. In this table, “write” refers to any write-type operation (e.g. WRITE, SETATTR, etc.).

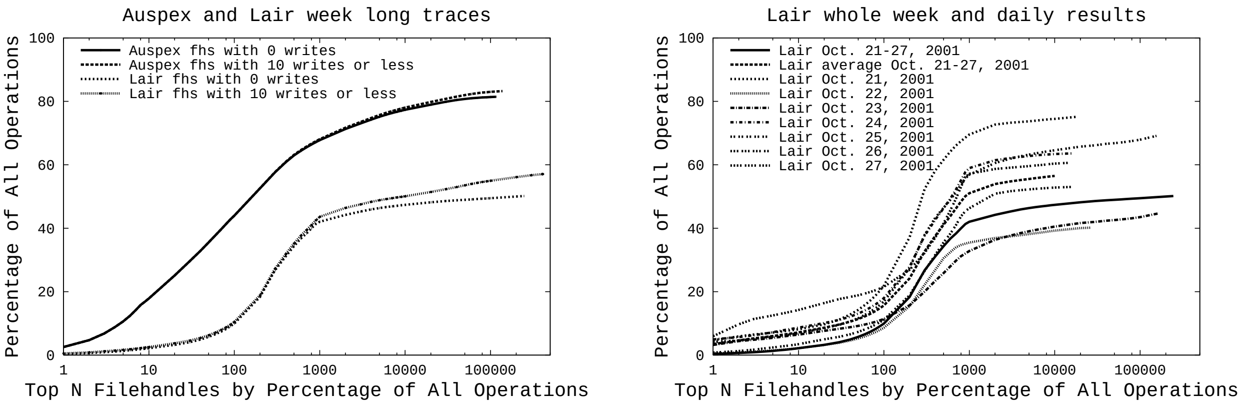
5.2 Replicating objects

Deciding which objects to replicate is the job of the Cuckoo planner. Our operating assumption, that a large number of operations are issued to files that infrequently change, indicates that replicas should be made of read-mostly files. Furthermore, we felt that it was a relatively small number of read-mostly files that accounted for a large number of read accesses. To confirm these intuitions, we examined the number of NFS filehandles that saw few writes and the number of operations issued to those filehandles. We also examined the cumulative distribution of operations issued to the most active read-only files to determine the number of operations that we might be able to offload.

5.2.1 Read-mostly objects

Table 1 shows our observations from the traces. For each trace we counted the number of filehandles that saw 0, less than or equal to one write operation, less than or equal to two write operations, etc. all the way up to ten write operations, and the operations performed on all of those filehandles in the category. This cumulative distribution of number of filehandles, versus the count of operations performed on those filehandles, provides us with an indication of the number of writes that our planner should tolerate in determining read-mostly files worthy of replication.

Upon examination of the traces, we see that tolerating up to ten writes to an object represented by a filehandle does not capture significantly more operations than are issued to filehandles which see no write activity (read-only filehandles). For the Auspex trace, this shows that, as a definition of “read-mostly” extends from zero writes up to ten writes, 10% more filehandles are included as candidates for replication, but that only 2% more operations would be issued to those extra 20,000 filehandles. Similarly, in the Lair trace we see that, between those filehandles with zero writes and



(a) Auspex and Lair

(b) Lair: read-only

Figure 4: **Operations on read-mostly filehandles:** The graph in (a) shows that as more of the most popular filehandles are replicated, a greater percentage of all operations are captured in requests to those filehandles. For comparison, (a) shows lines for filehandles with zero writes as well as for all filehandles with ten or fewer writes. The difference between the lines is minimal, indicating that the additional work required for handling writes (updates to replicas) might not be worth the effort. The graph in (b) shows the variability in the number of read-only filehandles over each day for one-week of the Lair trace. The sweet spot appears to be at the replication of the top 1000 filehandles throughout the week, by day and in aggregate.

those with up to ten writes, 35% more filehandles are candidates for replication but only 7% more operations are captured.

Observing such trends, one may conclude that there is little benefit in considering any but the read-only filehandles for replication, since they account for 50-81% of all operations performed in the traces.

5.2.2 Operations on read-only objects

There are over 100,000 read-only filehandles present in each trace, and it may not be feasible to replicate all of them. It is useful to consider how many filehandles must be replicated to allow a given percentage of all operations to be offloaded.

To explore this question, we determined the percentage of all operations issued to each of the read-mostly filehandles. Ordering these filehandles from most active to least active and plotting their contribution to the overall trace activity produces curves as show in Figure 4. The x-axis, therein, indicates the most active filehandles, ordered from most active to least active, in each of the categories that were plotted. The y-axis indicates the contribution to the overall percentage of operations that those filehandles made. The points on each line shown in the plot tell us that if the most active *x-position* filehandles in the trace had been replicated, the operations issued to those filehandles would count for *y-position* percentage of all operations present in the trace. The values reported in these plots represents the optimal replication strategy based on full knowledge of which files will be accessed and in what manner (read or write). The knee in the curves, around the 1000 most popular filehandles mark, indicates a sweet-spot for maximum benefit at minimum replication of files and directories.

The lines plotted in Figure 4(b) represent a further breakdown, by day, of the Lair trace. The plot also shows the average over each day of the week, and the same week-long curve shown in Figure 4(a) for the read-only filehandles. By breaking up the trace into its daily components, we are effectively evaluating seven separate traces. Each curve (with the exception of the daily average) is plotting a line over a separate, one-day set of operations. Note that some filehandles may show up on only some days' traces, being read-only for only part of the week. Repeating our earlier observation, we see a knee of each curve at approximately 1000 filehandles. So, for each of the days of the week, had the 1000 most active read-only filehandles for that day been replicated, between 32.84% and 69.45% of operations could have been offloaded that day.

The fact that the curve for the average of all days is higher than the week-long trace can be attributed to the fact that more filehandles, which are read-only for at least one entire day, are factored into that curve. Furthermore, this can be seen to indicate that, consistent with intuition, more frequent replication can yield greater opportunities for offloading operations. In the case of the average of all days, there is an opportunity for a file that was written on one day to be replicated on another day when it is accessed in a read-only fashion.

5.3 Analysis summary

In summary, the charts in Figure 3 show that we are examining two read-dominated workloads, suggesting that there is room for Cuckoo to help. The data presented in Table 1 show that replicating the read-only objects in these traces leads to a large number of offloading opportunities (50% to 81% of all operations). The knees in the curves presented in Figure 4 show us that, by replicating just the most popular 1000 read-only objects, 42% to 67% of all operations can be offloaded. Of course, the likelihood of being able to offload an operation depends on the workload during overload, but certainly a sweet spot for replication can be seen.

6 Going Cuckoo in a Switch

A natural place for layered clustering intermediaries is in a smart switch. Such an architecture can provide clustering benefits with zero changes to clients, servers, and the client-server protocol. Additionally, it eliminates any extra work for busy parent servers, allowing effective offloading even of cache hits. It also eliminates inter-intermediary communication and simplifies proxying of network traffic. This section expands on the reasons for, and methods of, putting Cuckoo in a switch.

Although our Cuckoo read offloading was prototyped as server-side intermediaries, its components have been designed to work in switch-based configurations as well. In particular, the Cuckoo algorithms assume no access to, or knowledge of, server internals; Cuckoo decisions require little state beyond the current request and the filehandle tables. Identifying replicated files by filehandle eliminates any dependence on server specifics, such as how a filehandle is interpreted. Likewise, using table lookups to map a parent filehandle to the corresponding surrogate filehandle eliminates the need for coordinating file names or other structures between the parent and surrogate (unless the file is a part of a replicated directory). Finally, having the replica planner execute offline allows it to be outside the switch, reducing switch resource requirements.

Embedding Cuckoo in a switch would eliminate extra network traffic and extra latency beyond the packet processing overhead within the switch. The effort needed to process packets is not large and not dissimilar to routing efforts—the values of a few byte offsets in packets are examined to determine that the destination server (identified by IP address) is busy, the packet contains the header of an offloadable NFS request (one of the five read operations), and that a replica exists (which is a lookup of the filehandle in the replica table). If the request is offloaded, the IP address and filehandle are modified and the packet is sent to the surrogate server. If the request mutates a replicated file, the corresponding replica table entry is invalidated.

Embedding Cuckoo in a switch would also simplify proxying of network traffic. In particular, whereas Cuckoo's current implementation is limited to NFS-over-UDP, switch-based proxying makes feasible read offloading for NFS-over-TCP as well. Doing so requires the ability to splice one TCP connection into another, which has been demonstrated by several researchers [16, 20], and the ability to create, manage, and shutdown TCP connections. Switch support for doing this for NFSv3 has recently been described.

One downside of switch-embedded Cuckoo would be collocation in one component of the replica and mapping tables for all servers. Although the Cuckoo structures are relatively compact, this will increase the memory requirements. The obvious way to address this issue is to reduce the number of replicas, since the sizes of the tables grow linearly with this number. For example, having two replicas of 10,000 files requires 640 KB in the current Cuckoo prototype (assuming 32 bytes per filehandle). The total space for ten servers with the same setup is 6.4 MB. However, the trace analysis in Section 5 indicates that reducing the number of replicated files per server from 10,000 to 1000 would only reduce the percentage of offloadable requests from 47–77% to 42–67%.

7 Summary

Eventually, cluster-based file systems should become common, but layered clustering offers a low-investment transition path—it provides some of the benefits with minimal change to existing infrastructure. Trace analysis and experiments with Cuckoo illustrate the feasibility of layered clustering for load balancing among NFS servers. With about 2000 lines of C code, read offloading with 1000 replicas can allow 42–67% of requests to a busy server to be shed to other servers.

Acknowledgments

We thank the members and companies of the PDL Consortium (including EMC, Hewlett-Packard, Hitachi, IBM, Intel, Microsoft, Network Appliance, Oracle, Panasas, Seagate, Sun, and Veritas) for their interest, insights, feedback, and support. We thank IBM and Intel for hardware grants supporting our research efforts. This material is based on research sponsored in part by the Air Force Research Laboratory, under agreement number F49620-01-1-0433.[5]

[5] The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the

We also thank Chris Costa, Mike Dahlin (for the Auspex traces), Dan Ellard (for the Lair traces), Mike Mesnier, Craig Soules, John Strunk, and Ken Tew.

References

[1] Darrell C. Anderson, Jeffrey S. Chase, and Amin M. Vahdat. Interposed request routing for scalable network storage. *Symposium on Operating Systems Design and Implementation* (San Diego, CA, 22–25 October 2000), 2000.

[2] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless network file systems. *ACM Transactions on Computer Systems*, **14**(1):41–79. ACM Press, February 1996.

[3] Lou Anscheutz and Jim McKinney. Personal communication with Computing Facilities managment of the Electrical and Computer Engineering Department at Carnegie Mellon University, August 2002.

[4] Michael Brooke and T. R. Birkhead. *The Cambridge encyclopedia of ornithology*. Cambridge University Press, 1991.

[5] Luis-Felipe Cabrera and Darrell D. E. Long. Swift: using distributed disk striping to provide high I/O data rates. *Computing Systems*, **4**(4):405–436, Fall 1991.

[6] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. *Symposium on Operating Systems Design and Implementation* (New Orleans, LA, 22–25 February 1999), pages 173–186. ACM, 1998.

[7] Michael D. Dahlin, Clifford J. Mather, Randolph Y. Yang, Thomas E. Anderson, and David A. Patterson. A quantitative analysis of cache policies for scalable network file systems. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Nashville, TN, 16–20 May 1994), pages 150–160. ACM, 1994.

[8] Daniel Ellard, Jonathan Ledlie, Pia Malkani, and Margo Seltzer. Passive NFS tracing of an email and research workload. *Conference on File and Storage Technologies* (San Francisco, CA, 31 March–2 April 2003), pages 203–217. USENIX Association, 2003.

[9] Garth A. Gibson, David F. Nagle, Khalil Amiri, Fay W. Chang, Eugene M. Feinberg, Howard Gobioff, Chen Lee, Berend Ozceri, Erik Riedel, David Rochberg, and Jim Zelenka. File server scaling with network-attached secure disks. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Seattle, WA, 15–18 June 1997). Published as *Performance Evaluation Review*, **25**(1):272–284. ACM, June 1997.

[10] John H. Hartman and John K. Ousterhout. The Zebra striped network file system. *ACM Symposium on Operating System Principles* (Asheville, NC), pages 29–43, 5–8 December 1993.

[11] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, **6**(1):51–81, February 1988.

[12] Edward K. Lee and Chandramohan A. Thekkath. Petal: distributed virtual disks. *Architectural Support for Programming Languages and Operating Systems* (Cambridge, MA, 1–5 October 1996). Published as *SIGPLAN Notices*, **31**(9):84–92, 1996.

[13] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. Replication in the Harp file system. *ACM Symposium on Operating System Principles* (Pacific Grove, CA, 13–16 October 1991). Published as *Operating Systems Review*, **25**(5):226–238, 1991.

[14] Miron Livny, Setrag Khoshafian, and Haran Boral. Multi-disk management algorithms. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 69–77, 1987.

[15] Darrell D. E. Long, Bruce R. Montague, and Luis-Felipe Cabrera. Swift/RAID: a distributed RAID system. *Computing Systems*, **7**(3):333–359. Usenix, Summer 1994.

[16] David A. Maltz and Pravin Bhagwat. TCP Splice for application layer proxy performance. *Journal of High Speed Networks*, **8**(3):225–240, 1999.

[17] Vivek S. Pai, Mohit Aron, Gaurav Banga, Michael Svendsen, Peter Druschel, Willy Zwaenepoel, and Erich Nahum. Locality-aware request distribution in cluster-based network servers. *Architectural Support for Programming Languages and Operating Systems* (San Jose, CA, 3–7 October 1998). Published as *SIGPLAN Notices*, **33**(11):205–216, November 1998.

[18] Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson. A comparison of file system workloads. *USENIX Annual Technical Conference* (San Diego, CA, 18–23 June 2000), pages 41–54. USENIX Association, 2000.

Air Force Research Laboratory or the U.S. Government.

[19] Marc Shapiro. Structure and encapsulation in distributed systems: the proxy principle. *International Conference on Distributed Computing Systems* (Cambridge, Mass), pages 198–204. IEEE Computer Society Press, Catalog number 86CH22293-9, May 1986.

[20] Oliver Spatscheck, Jorgen Hansen, John Hartman, and Larry Peterson. Optimizing TCP forwarder performance. *Transactions on Networking*, **8**(2):146–157. IEEE; ACM, April 2000.

[21] John D. Strunk, Garth R. Goodson, Michael L. Scheinholtz, Craig A. N. Soules, and Gregory R. Ganger. Self-securing storage: protecting data in compromised systems. *Symposium on Operating Systems Design and Implementation* (San Diego, CA, 23–25 October 2000), pages 165–180. USENIX Association, 2000.

[22] Sun Microsystems, Inc. RPC: remote procedure call protocol specification. Network Working Group, RFC–1050, April 1988.

[23] Sun Microsystems, Inc. NFS: network file system protocol specification. Network Working Group, RFC–1094, March 1989.

[24] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: a scalable distributed file system. *ACM Symposium on Operating System Principles* (Saint-Malo, France, 5–8 October 1997). Published as *Operating Systems Review*, **31**(5):224–237. ACM, 1997.

[25] Werner Vogels. File system usage in Windows NT 4.0. *ACM Symposium on Operating System Principles* (Kiawah Island Resort, Charleston, South Carolina, 12–15 December 1999). Published as *Operating System Review*, **33**(5):93–109. ACM, December 1999.

[26] Kenneth G. Yocum, Darrell C. Anderson, Jeffrey S. Chase, and Amin M. Vahdat. Anypoint: extensible transport switching on the edge. *USENIX Symposium on Internet Technologies and Systems* (Seattle, WA, 26–28 March 2003), 2003.

[27] Z-force, Inc. www.zforce.com.