SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2006; **36**:117–137
Published online 15 September 2005 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.685

Customizing the swarm storage system using agents

John H. Hartman[1], Scott Baker[1,*,†] and Ian Murdock[2]

[1]*Department of Computer Science, University of Arizona, 1040 E. 4th Street, Tucson, AZ 85721, U.S.A.*
[2]*Progeny Linux Systems, 8335 Allison Pointe Trail, Suite 160, Indianapolis, IN 46250, U.S.A.*

SUMMARY

Swarm is a scalable, modular storage system that uses agents to customize low-level storage functions to meet the needs of high-level services. Agents influence low-level storage functions such as data layout, metadata management, and crash recovery. An agent is a program that is attached to data in the storage system and invoked when events occur during the data's lifetime. For example, before Swarm writes data to disk, agents attached to the data are invoked to determine a layout policy. Agents are typically persistent, remaining attached to the data they manage until the data are deleted; this allows agents to continue to affect how the data are handled long after the application or storage service that created the data has terminated. In this paper, we present Swarm's agent architecture, describe the types of agents that Swarm supports and the infrastructure used to support them, and discuss their performance overhead and security implications. We describe how several storage services and applications use agents, and the benefits they derive from doing so. Copyright © 2005 John Wiley & Sons, Ltd.

KEY WORDS: log structured; LFS; file system; storage system; agents

1. INTRODUCTION

Traditional storage systems are inflexible, providing fixed storage abstractions, access protocols, and data management policies. In contrast, the Swarm storage system [1] allows these to be customized. Swarm accomplishes this by decoupling high-level abstractions and functionality from low-level data storage. Rather than providing high-level abstractions directly, Swarm provides an extensible, layered infrastructure that allows high-level storage functionality to be composed in a modular fashion, with each layer augmenting, extending, or hiding the functionality of the layers below it.

Swarm employs *agents* to enable applications, file systems, and other storage services to influence and control key storage functions such as data layout, metadata management, and crash recovery. An agent is a program that is attached to data in the storage system and invoked when specific events

*Correspondence to: Scott Baker, Department of Computer Science, University of Arizona, 1040 E. 4th Street, Tucson, AZ 85721, U.S.A.
†E-mail: bakers@cs.arizona.edu

Received 17 August 2004
Revised 11 March 2005
Accepted 11 March 2005

occur during the data's lifetime. For example, before Swarm writes data to disk, agents attached to the data are invoked to determine a layout policy. Agents are persistent and are stored alongside the data they manage, allowing the agents to manage the data even after the applications or file systems that originally created the data no longer exist.

Agents add a new dimension of flexibility, extensibility, and power to storage systems. Agents allow applications and storage services to extend Swarm in application-specific ways, without requiring Swarm to have any implicit knowledge about how the application or storage service works. Agents allow Swarm to update metadata without knowledge of the metadata structures, and to implement application-specific data layout policies without knowledge of or assumptions about future access patterns. Furthermore, because agents are programs, they are inherently more powerful than static policies; agents can take advantage of the current system state to determine a policy that is optimized for a particular situation.

For example, consider the process of reorganizing or *cleaning* a storage system. As new data are written and old data are removed, the layout of files and directories may become fragmented. Cleaning returns the storage system to a more optimal layout. Traditional storage systems perform cleaning without any knowledge as to how applications will use the data. The organization chosen by the storage system may differ from what the applications would choose. However, if agents are deployed within the storage system, then the agents can tailor the reorganization to the specific requirements of each application.

Swarm is implemented as a loadable module for the Linux 2.2 kernel. We developed and experimented with several Swarm-based services that use agents, including a local file system called Sting, a cleaner service that reclaims unused portions of Swarm's log, and a simple logical disk that presents a virtual disk abstraction on top of Swarm's log abstraction. We also implemented several different layout strategies using agents that demonstrate the ability of an agent to tailor the storage system to match the characteristics of the data. Our purpose in doing so was not to show that an agent-based implementation of a particular policy outperforms a hard-coded implementation; rather, that agents can customize Swarm to implement a variety of layout policies. The Web layout agent clusters Web pages and their embedded images, improving locality. The read-ordered layout agent records the read access patterns of files and then organizes the file blocks to speed up similar reads in the future. The different agents employed by these services and the resulting performance improvements demonstrate the usefulness of the agent infrastructure.

This paper describes Swarm's agent mechanism and how it is used to improve the performance and flexibility of applications and storage services that run on Swarm. We first provide an overview of Swarm, then describe Swarm's agent mechanism and the infrastructure that supports it. Finally, we describe the Swarm-based services we developed that use agents, and the benefits they derive from doing so.

2. SWARM ARCHITECTURE

Swarm is a storage system that provides scalable, reliable, and cost-effective data storage. At its lowest level, Swarm provides a log-structured interface to a cluster of storage devices. These devices act as repositories for fixed-sized pieces of the log called *fragments*. The storage devices have relatively simple functionality, so they are easily implemented using inexpensive commodity hardware or network-attached disks [2]. Individual storage devices are optimized for cost-performance and aggregated to provide the desired absolute performance.

Swarm clients use a *striped log* abstraction [3] to store data on the storage devices. This abstraction simplifies storage allocation, improves file access performance, balances server loads, provides fault tolerance through computed redundancy, and simplifies crash recovery. Each Swarm client creates its own log, appending new data to the log and forming the log into fragments that are striped across the storage devices; RAID-style parity enables reconstruction of missing portions of the log when a storage device fails. Clients cache blocks in memory and write them to the log in batches, improving read performance by allowing blocks to be organized within the batch, and improving write performance by writing multiple blocks to the log in a single operation. Each client maintains its own log and parity, and therefore does not need to coordinate with other clients to perform these functions; this results in improved scalability, reliability, and performance over centralized file servers.

Swarm provides an infrastructure for building storage services on top of the striped log, allowing applications to tailor the storage system to their particular needs. Swarm is implemented as a collection of modules that are layered to build storage systems in much the same way that protocols are layered to build network communications subsystems [4]. Each module in Swarm implements a storage service that communicates with the lower levels of the storage system through a well-defined interface, and exports its own well-defined interface to higher levels. Storage systems are constructed by layering modules with compatible interfaces (Figure 1). Each layer can augment, extend, or hide the functionality of the layers below it. For example, an atomicity service can layer above the log, providing atomicity across multiple block writes. In turn, a logical disk service can layer above this extended log abstraction, providing a disk-like interface to the log and hiding its append-only nature.

2.1. Log layer

The striped log is the central abstraction in Swarm. The striped log abstraction and corresponding interface are implemented in the *log layer*. The log layer is responsible for forming data written by higher levels into an append-only log and striping the log across the underlying storage devices. The layers above the log are called *storage services* (*services* for short) and are responsible for implementing high-level storage abstractions and functionality. The log layer's main function is to multiplex the underlying storage devices among multiple services, allowing storage system resources to be shared easily and efficiently.

2.2. Log format

A single Swarm client may support a variety of services that use the log to store their data, such as file systems, databases, and virtual disks. The log itself is an ordered stream of *blocks* and *records* (Figure 2). It is append-only: blocks and records are written to the end of the log and are immutable. Blocks contain data belonging to high-level services, and are not interpreted by the log. For example, a file system stores its data and metadata in blocks. Records are used to recover from client crashes. A crash causes the log to end abruptly, potentially creating inconsistencies in any data structures it contains. A record contains information necessary to recover from the crash, enabling services to repair inconsistencies by re-applying the state changes indicated by the records (Figure 3). For example, a file system might append records to the log as it performs high-level operations that involve changing

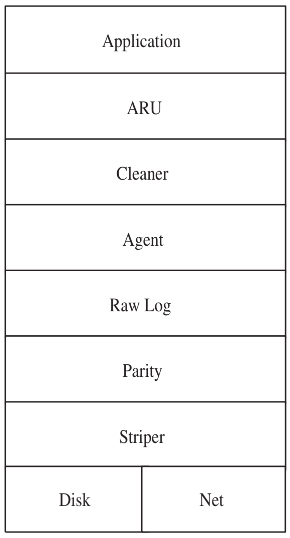

Figure 1. Swarm architecture. A particular instance of Swarm is constructed by layering Swarm modules to obtain the desired functionality for the storage system. Each layer augments, extends, and/or hides the functionality of the layers below it. The agent layer is responsible for implementing the agent infrastructure described in this paper.

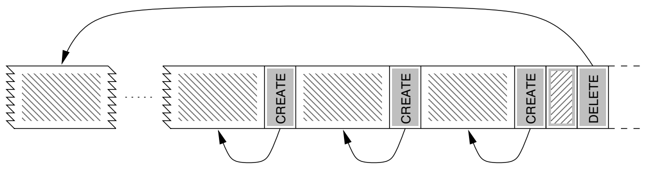

Figure 2. Log format. The light objects are blocks, and the dark objects are records. Each CREATE record indicates the creation of a block, and each DELETE record indicates a deletion; the arrows show which block is affected by each record and represent references visible to the log.

several data structures (e.g. as happens during file creation and deletion). During replay, these records allow the file system to easily redo (or undo) the high-level operations. Swarm provides a set of standard record types, including create block and delete block, to which higher-level services can add their own service-specific types of records.

Records are implicitly deleted by *checkpoints*, special records that denote points in the log at its data structures are consistent. Checkpoint records are issued at the behest of services. For example, a file system service may decide to checkpoint when the amount of data written exceeds a threshold, or when the data has aged a certain amount of time. The log layer guarantees atomicity of record writes and

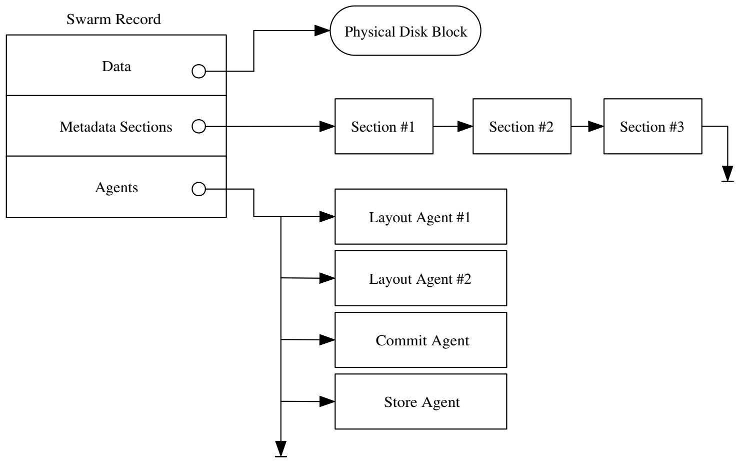

Figure 3. Record format. Each record contains a pointer to an associated data block (if any), a variable number of sections in which each service stores service-specific recovery information, and references to each agent that has been attached to the record.

preserves the order of records in the log, so that services are guaranteed to replay them in the correct order.

2.3. Log layout

Swarm caches blocks in memory as services write data and stores them to the log in batches. As Swarm creates the log from the blocks in the cache, it must make decisions about how the blocks are organized in the log. Proper data layout is important, since it affects the performance of subsequent reads. Blocks that are read at the same time but distributed throughout the log are much slower to access than if they were clustered together, due to the high cost of disk seeks and lost opportunities to perform large data transfers.

Swarm has no implicit knowledge about the contents of the blocks that it stores, so without help from the services that created the blocks, it does not know how to organize them in the log. To address this problem, Swarm provides an *ordered sets* abstraction that allows services to express block layout preferences. Each set contains a list of blocks that should be contiguous and consecutive in the log. Sets are created by the services that write the blocks. A service can create as many sets as it likes, and assign to each as many blocks as it likes.

A service submits sets containing blocks to the log layer. The log layer packs the sets into log fragments so that no set spans a fragment boundary. If a set is too large to fit into a fragment it must obviously span a boundary; the log layer simply splits an oversized set arbitrarily into sets that fit into fragments. A service that wishes to avoid this can do so by ensuring that every set fits into a fragment.

In some cases, a service may want to express layout policies that require blocks to appear in multiple sets. For example, a file system might use a set to specify that the blocks of a file should be laid out consecutively and contiguously, and use another set to specify that all files in the same directory should be clustered together. In this situation, each file block is a member of a file set and a directory set. Swarm attempts to pack all sets with blocks in common into the same log fragment, thus ensuring that blocks are clustered properly. If the sets do not all fit into a fragment, then Swarm is forced to split the blocks of some sets across fragment boundaries. In this case, priorities are used to decide which sets are split and which are not. For example, giving a file set higher priority than a directory set indicates that it is more important to cluster the blocks of a file than it is to cluster files in the same directory. If all the files in a directory cannot fit in the same fragment, then the directory set is split so that some of the files are stored in different fragments.

The log layer uses the following algorithm (Figure 4) for placing blocks in the log based on set membership and priorities, and splitting sets when necessary. First, the sets are ordered from highest priority (N) to lowest (1). The sets with priority N are packed into log fragments so that two sets are placed in the same fragment if there is a priority $N-1$ set that contains blocks from both of the priority N sets. Once the blocks in the priority N sets have been packed, the blocks in the priority $N-1$ sets are packed by considering common membership in priority $N-2$ sets, and so on. For example, if each priority N set contains blocks from the same file, and each priority $N-1$ set contains blocks from files in the same directory, then the algorithm packs file sets into the same fragment if they have blocks belonging to the same directory set.

Packing sets into fragments according to priority ensures that the log layer favors splitting lower-priority sets over higher-priority sets. If the same block appears in multiple sets with the same priority, then one of the sets is arbitrarily chosen and the others ignored. Intuitively, this indicates that it is equally important that the block be clustered with the other blocks in the different sets, so the log layer is free to choose any set that it wants. The set priority mechanism should be used to express any preference the service might have.

3. AGENT INFRASTRUCTURE

Sets allow services to specify relationships between blocks to Swarm; Swarm uses the sets to organize blocks in the log. Sets do not, however, tell Swarm why the relationships between blocks exist, or how long they will continue to exist. For example, a file system may put all the blocks of a file into a set, but Swarm does not know that this set represents a file and therefore the relationship should persist until the file is deleted.

Rather than hard-code high-level abstractions such as files, directories, indices, etc., into Swarm, Swarm allows storage services to attach agents to records. Agents implement policies, and express them by creating sets. Sets are only used to place the block in the log once, after which they are discarded. If a block needs to be rewritten to the log (e.g. because it was modified), its agents are again invoked to assign the block to sets. Agents not only provide a convenient decoupling of mechanism and policy, but also provide a much more powerful mechanism for specifying policy than the ordered sets themselves, since an agent can take into account the current state of the system each time a block is written.

```
/* CollapseSet: Produces buildSet, the union
of all higher priority sets that intersect
with lowSet */

CollapseSet (lowSet)
{
  buildSet = {}
restart:
  for each record R in lowSet do {
    for each set, highSet, with priority
    higher than lowSet do {
      if R is a member of highSet then {
        lowSet += record
        buildSet = Union (buildSet, highSet)
        highSet = {}
        goto restart
      }
    }
  }
  /* put any remaining records from lowSet
  into buildSet */

  buildSet = Union (buildSet, lowSet)
  lowSet = {}
}

CollapseAllSets(setList)
{
  Sort setList by descending priority
  For each set, S, in setList do
    CollapseSet(S)
}
```

Figure 4. Set packing algorithm. The set packing algorithm is used to sort sets by priority, combine them, and produce a linear order that can be used to write the data to disk.

Agents introduce a potential security hole, since they run inside the Swarm environment and affect Swarm's functionality. Without proper precautions, a buggy or malicious agent could corrupt data structures belonging to other services or Swarm itself. Swarm must be able to protect itself from agents, and agents must be able to protect themselves from each other. Swarm must also ensure that agents do not consume an undue amount of resources. These concerns are addressed in Section 3.5.

An agent is typically associated with a particular layer, allowing the agent to access the layer's internal data structures. For example, imagine a collection of agents that need to share persistent state. This state could be stored in the layer to which the agents belong, along with routines to access and update the state.

```
typedef Status (AgentFunc) (Interface *iPtr,
    RecordRef *recordList, void *agentData);

/* register an agent */
Status
RegisterAgent(Agent_Interface *agentPtr,
    char *name, AgentFunc *func,
    int agentType, int flags,
    void *agentData, AgentId *id);

/* attach an agent to a record */
Status
AttachAgent(Agent_Interface *agentPtr,
    Record *record, int level, AgentId id,
    int flags, void *recordData);

/* invoke all agents of a given type */
Status
InvokeAgents(Agent_Interface *agentPtr,
    Record *record, int agentType);
```

Figure 5. Agent routines. The `agentData` is an opaque data field that is specified to `RegisterAgent` when the agent is created, and passed to the agent when it is subsequently invoked. The `recordData` is specified when the agent is attached to a record, and is available to the agent when it processes the record. The `level` parameter specifies the service's level in the service stack.

3.1. Agent interface

Swarm provides an interface for services to create agents and attach them to records (Figure 5). Although each record could have its own unique agents, typically a single agent will be attached to multiple records that should be handled similarly.

When the agent is invoked, it is passed a list of records to which it was attached. Furthermore, when an agent is attached to a record, a fixed-size opaque data field (called `recordData`) can be provided that is stored in the record and available to the agent when it processes the record. The agent is also passed an `agentData` parameter that was provided when the agent was created. The `agentData` contains agent-specific information that also helps the agent perform its function.

Agents are invoked beginning with the lowest-level service and working toward the highest (i.e. in the reverse order in which they were attached to the record). Conceptually, agents are attached to records as they pass down through the layers, and the agents are invoked as the response passes back up through the layers. Swarm does not have provisions for allowing different agent orderings, perhaps specified when the agents are created or when they are attached to records. A general facility for this would require inter-layer knowledge to allow their agents to be ordered properly. Instead, Swarm invokes the agents in layer order.

3.2. Agent types

Swarm implements four default types of agents: *layout*, *commit*, *store*, and *replay*. A layout agent is invoked when Swarm flushes the cache, allowing the service to specify a layout policy for the records and blocks being flushed. A commit agent is invoked after Swarm has assigned a log address to a record and its associated block, so that the service can update its metadata to reflect the new address. A store agent is invoked after the record and its associated block have been written to the log, allowing the service to clean up any record state. A replay agent is invoked when replaying records after a crash, allowing a service to recover from the crash.

Swarm also provides facilities for services to define new types of agents and cause them to be invoked when appropriate. This functionality is used by the cleaner, for example, to create a new type of agent that handles cleaning a block.

3.2.1. Layout agent

A layout agent is responsible for deciding how blocks and records should be stored in the log. The agent is invoked before dirty blocks are flushed from the cache, and it is provided a list of all records to which the agent is attached. The agent puts these records into ordered sets. Swarm uses the ordered sets to determine where blocks are placed in the log; it attempts to store the blocks in each set contiguously and consecutively. The layout agent uses whatever method it chooses to allocate blocks to sets. For example, a file layout agent can put each file's blocks in a different set, in the order in which they appear in the file. This ensures that file blocks are laid out contiguously and consecutively.

Multiple layout agents can be attached to the same record, either by the same layer or different layers. Consider, for example, an agent that clustered blocks according to file, and another agent that clustered blocks by directory. The set priority mechanism is used to prioritize the sets produced by multiple agents, as described in Section 2.3.

To simplify the implementation of higher-level services that do not care about layout, Swarm provides a default layout agent (Figure 6). This agent simply assigns all records to the same set, creating a new set when the current one reaches the size of a fragment.

3.2.2. Commit agent

The commit agent for a record is invoked after Swarm has placed a record's set in the log, and has therefore committed to writing the record's block at a particular log address. When it is invoked, the agent is provided with the record and the log address where it will be written. The commit agent uses this information to update any metadata that refers to the block. For example, a commit agent for a file system would update the file's inode and indirect block metadata to contain the new log address for the data block.

Swarm also provides a default commit agent to simplify service implementation. This agent requires that the recordData contain a list of log addresses, and it updates them to contain the block's address.

```
Status DefaultLayoutAgent (
  Agent_Interface *iPtr,
  RecordRef *recordList,
  void *agentData)
{
  RecordRef *tmpPtr;
  Set *curSet = NULL;

  LIST_FORALL(&recordList->links, tmpPtr) {
    while (agentPtr->AddRecord(agentPtr,
      curSet, &tmpPtr->record) != SUCCESS) {
      /* fails if curSet is full */
      if (curSet!=NULL)
        agentPtr->SubmitSet(agentPtr,
          AGENT_SERVICE, curSet);
      agentPtr->CreateSet(agentPtr, 0,
        FALSE, AGENT_SERVICE, &curSet);
    }
  }
  if (curSet!=NULL)
    agentPtr->SubmitSet(agentPtr,
      AGENT_SERVICE, curSet);

  return SUCCESS;
}
```

Figure 6. Default layout agent. The default layout agent puts all of the records into the current set. When the current set is full it is submitted to the log layer and a new set created.

3.2.3. Store agent

The store agent is invoked after a record and its associated data block have been successfully stored in the log. Typically, a store agent is responsible for cleaning up the block's state, such as by removing the block from the cache. Another use is to write a block synchronously by registering a store agent on the block and suspending the thread. When the agent is invoked, it wakes the waiting thread.

3.2.4. Replay agent

The replay agent is invoked when replaying the log during server recovery. The agent is given records from the log in the order in which they appear in the log. The replay agent is often similar to the commit agent in that it updates the block's metadata to reflect its position in the log. Processing isn't exactly the same because the server may have crashed, causing the log to be truncated, which in turn may affect how the records are processed.

3.3. Agent lifetime

Typically, an agent is *persistent*, remaining attached to a record until the record is deleted. A persistent agent (and any associated state) is stored in the log by Swarm so that it is not affected by machine crashes, and can be invoked after the machine recovers. For example, replay agents are always persistent because they are only invoked after a crash and therefore must survive the crash. Layout agents are also usually persistent since they are invoked throughout the block's lifetime each time it is cleaned.

Swarm also supports *transient* agents, agents that are invoked only once and do not persist across machine reboots. These agents are used for processing that should not be done after a reboot, such as cleaning up in-memory data structures. The best example of a transient agent is the one Swarm uses to implement synchronous log writes. This agent is attached to a record when it is submitted, after which the submitting thread waits. The agent is invoked after the block is stored in the log, and it resumes the waiting thread. Since it is transient, it is only invoked once and it does not survive machine reboots, which is desirable since the waiting thread will not either.

A concern with persistent agents is backward compatibility. Persistent agents should continue to work properly if the operating system and/or Swarm are upgraded. Ensuring an agent's backward compatibility is the responsibility of the agent designer, much as ensuring an operating system's backward compatibility is a responsibility of the operating system designer. Swarm's agents are named by text strings, and a convention can be adopted to include a version number as part of the agent's name.

3.4. Overhead

Swarm does add overhead to the storage functions. Agents are invoked when records are laid out, committed, stored, replayed, and cleaned. Of course, different agents perform different amounts of computation, so it is impossible to characterize the overall performance effect of agents. The goal is that the overall system performance improvements agents provide more than offsets the overhead of running them. Section 4 describes the different agents we have developed and how much they improved system performance.

We measured the overhead of the default layout and commit agents described in Sections 3.2.1 and 3.2.2, respectively. These default agents are probably the minimal useful agents for those agent types. On a 1.7 GHz Pentium 4 CPU, the default layout agent requires 4 μs per block, of which memory allocation consumes 2.3 μs, and manipulating the set data structures 1.7 μs. The memory allocation overhead is clearly too high, and is something we plan to rectify. Once that is fixed, the cost per block for the default layout agent should be less than 2 μs. The default commit agent does much less work than the layout agent, and therefore requires only 0.4 μs per block.

3.5. Protection and security

Swarm must ensure that agents do not interfere with each other, or the proper functioning of Swarm itself. It must also ensure that they do not consume an inordinate amount of resources. There are many possible solutions to these problems, since these same issues arise in many contexts. One is to write the agents in a type-safe language, such as Java. The use of such a language would limit the agents to accessing only those data structures to which they are granted access; this would prevent an agent from accessing anything but its own blocks. The use of Java will likely reduce agent performance, but this is probably acceptable since the agents are invoked as part of a relatively slow I/O operation. Another downside of this approach is that it requires a Java Virtual Machine inside of the Swarm infrastructure, which increases Swarm's resource requirements and complexity.

Other possible protection mechanisms include running the agents in a separate process, using proof-carrying code [5] to verify the agents' correctness, or using software fault isolation [6,7] to isolate the agents. All of these should be acceptable, although running agents in a separate process will likely have high overheads.

Proof-carrying code and software fault isolation have both been demonstrated in an operating system kernel, and therefore both solutions are a good match for our current prototype. Software fault isolation would allow the agents to be written in C, but still isolate them. The Vino [7] project has already used it to isolate untrusted code inside an operating system kernel, allowing us to leverage that body of work when applying it to Swarm. The disadvantage of software fault isolation is runtime overhead, which can range from 5 to 200%. Proof-carrying code has the benefit that it places the burden of creating the proof on the code producer, while allowing the code consumer to perform a fast and simple proof verification process. Verification is a one-time process; once an agent is verified and admitted to swarm, it could be executed with no runtime penalty.

Our current prototype does not protect against malicious or buggy agents; for expediency, the agents are written in C and no mechanisms are employed to isolate them. When an agent is invoked it is passed a list of blocks to which it has been attached. The agent has no direct access to blocks belonging to other services and agents, preventing it from doing so trivially. Nonetheless, a deployed version of Swarm's agent infrastructure would require protection mechanisms.

4. EXAMPLES

We implemented several different agents in the Swarm prototype. These agents are linked into the Linux kernel module, and are attached to records by services as part of each service's processing of the record. This section describes the services to which we added agents, how they use agents, and what benefits they derive from their use. All performance numbers in this section were collected on a system with a 1.7 GHz Pentium 4 CPU, 768 MB of main memory, a Seagate Cheetah 9.0 GB SCSI-3 disk, and running Linux 2.2.16 and Redhat 7.2.

4.1. Cleaner

Like other log-structured storage systems, Swarm uses a *cleaner* that periodically garbage-collects unused blocks in the log to make room for new segments [8]. The cleaner continuously monitors the utilization of stripes in the system. When the utilization exceeds a high water mark, the cleaner cleans stripes until a low water mark is reached. During the cleaning process, the cleaner selects the most underutilized stripes as candidates for cleaning. The cleaner maintains an in-memory *stripe table* that monitors the blocks and records written to the log, allowing it to track which portions of the log are underutilized. The stripe table is periodically checkpointed to disk so that the accounting information is available in case swarm is restarted. In the event of a crash, the cleaner uses a replay agent to update the stripe table from the most recent checkpoint. The cleaner is also responsible for free space management, enforcing quotas on higher-level services, initiating cleaning to move live data out of underutilized stripes so that the space they occupy can be used for new log data, and reserving the appropriate number of stripes so that cleaning always makes progress.

The cleaner is an example of a service that defines a new type of agent: a cleaning agent. A cleaning agent is responsible for moving live data out of an underutilized stripe so the stripe can be deleted and reused. Cleaning agents are usually attached to the create record for a block. For each stripe that is to be cleaned, the cleaner enumerates the create records in the stripe and invokes the cleaning agents attached to those records. A cleaning agent takes whatever actions are necessary to clean a block. For example, a very simple cleaning agent reads in the live block, and then writes out the live block and a delete record for the old instance of the block. The cleaning agent for the Sting file system cleans a block by reading it into the file cache and marking it as dirty, causing it to be written back out to the log the next time the cache is flushed. When the cache is flushed, the Sting layout agent has the opportunity to reorganize the cleaned blocks for the most efficient layout. If a block does not have a cleaning agent, or the cleaning agent does not relocate the block within a reasonable amount of time, then the cleaner has the option of deleting the block outright.

With the exception of cleaning agents, the cleaner is largely transparent to the services above it. When a service submits a create record to the cleaner, the cleaner attaches its own store agent to the create record. When the block is stored this agent adjusts the stripe table to reflect the increased utilization.

When a service submits a delete record, the cleaner attaches a store agent to the delete record. When the delete record has been written to the log, Swarm invokes the cleaner's store agent and the utilization of the stripe is decreased to reflect the deleted block. When a stripe becomes empty, the cleaner automatically deletes the stripe, allowing the log layer to reuse it.

One complication with delete records is that a misbehaving application might issue multiple delete records for the same block. If the cleaner were to keep a simple running total of the live bytes in a stripe, multiple delete records would cause it to underestimate the stripe's utilization. The cleaner avoids this problem by maintaining a bitmap for each stripe that records which blocks are live and which are deleted. The bitmap ensures that multiple deletes of the same block are only counted once. The bitmap is part of the stripe table, and is written to the log when the stripe table is checkpointed.

4.2. Atomic Recovery Units

The Atomic Recovery Unit (ARU) service provides atomicity across multiple log writes, and is modeled after the ARU facility originally provided by the Logical Disk system developed at MIT [9,10]. An ARU is a form of weak transaction, guaranteeing only that operations in the same ARU either all complete or all fail.

The ARU service allows higher-level services to create an ARU, perform writes within the context of the ARU, and close the ARU. When an ARU is created, the ARU service writes a Begin record to the log and returns a unique ARU identifier to the service that created it. Subsequent records that are part of the ARU are tagged with the ARU identifier. When the ARU is closed, an End record is written. If the machine crashes before all of the blocks in the ARU are written to the log, then upon reboot all of the modified blocks are deleted from the log and they revert to their contents before the ARU.

Thus, if an ARU is created and a failure occurs before the end of the ARU, none of the operations within the ARU have any effect.

The ARU layer uses its own store agent that it attaches to every record submitted. When a record is stored, the ARU store agent is invoked before any other store agents. The agent queues the records and blocks invocation of any higher-level store agents until the an End record is seen. In this way the higher-level store agents are not invoked until the entire ARU is stored. Similarly, a replay agent is attached to all records that runs first and queues all other replay agents until the End record is seen. If the End is never found, the queued records are deleted and their replay agents not run.

The ARU layer violates the layered ordering of agents because its agents must run before all other agents, even those belonging to the cleaner below it. The ARU layer may sit above the cleaner, but the cleaner cannot be oblivious to its existence because the cleaner must not process records within an ARU until the entire ARU has been successfully stored. For example, if a service deletes a block within an ARU, the cleaner must not denote the block as deleted until the ARU completes successfully. This could be handled by putting ARU-specific knowledge in the cleaner, but defeats the purpose of layering and is unnecessary since simply running the ARU agents first solves the problem. The ARU layer specifies that its agents should run first by providing a flag to that effect to the agent layer when the agents are created. The agent layer allows one of each type of agent to have this flag set; it is an error to specify that two agents of the same type should run first.

An alternative approach would be to put the ARU layer below the cleaner. This would cause the ARU agents to run before the cleaner agents without special handling, but has the serious downside of requiring the ARU layer to run on a finite log. In essence, the ARU layer would have to perform its own cleaning, which is a more onerous solution than the one we have chosen.

4.3. Sting

Sting is a local file system that we implemented as part of Swarm. When loaded into the Linux kernel, it allows application programs to access standard UNIX files and directories that are stored in Swarm (Figure 7). Sting is log-structured, and uses a variety of agents to ensure that data are stored in the log efficiently, and that metadata are kept up to date.

Sting uses layout agents to implement a data layout policy similar to that of FFS [11]. Sting uses two layout agents: FileLayout and DirectoryLayout. The FileLayout agent creates a set for each file, putting the blocks of the file into the set in the order in which they appear in the file; this tells the log layer that a file's blocks should be laid out in the log contiguously and in order. The DirectoryLayout agent creates a set for each directory, putting all blocks belonging to files in the directory into the set; this tells the log layer that files from the same directory should be clustered together in the log. The sets created by the DirectoryLayout agent have lower priority than the FileLayout agent; this tells the log layer that it is more important to keep the blocks of a file together than it is to cluster files from the same directory.

Sting uses two commit agents for metadata management: a DataCommit agent for data blocks and indirect blocks, and an InodeCommit agent for blocks that contain inodes. The DataCommit agent stores the address of the block in the proper inode or indirect block, reading it into the cache if necessary. The InodeCommit agent stores the inode's log address in the inode map.

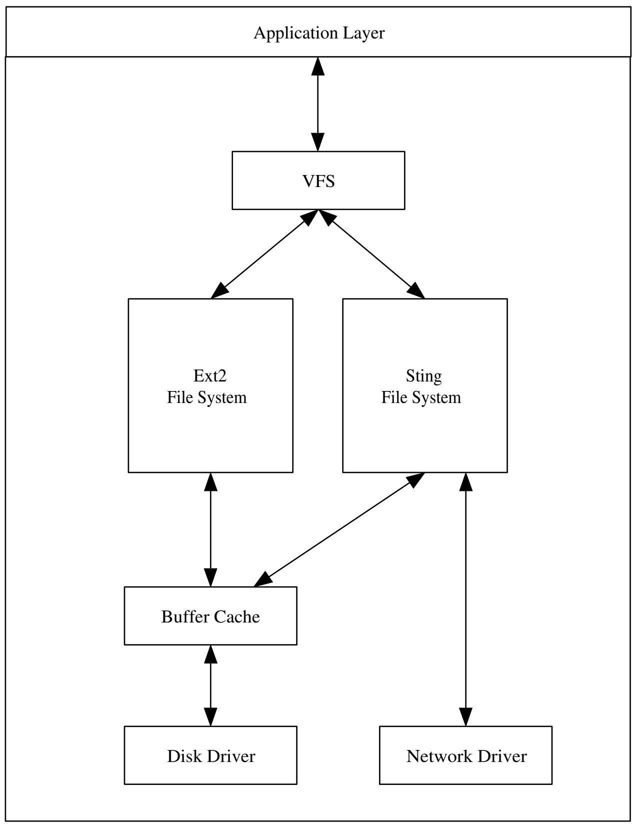

Figure 7. Sting. Sting is implemented as a Swarm module. The entire Sting/Swarm system is loaded into the Linux kernel below the VFS layer and above the buffer cache and network drivers. Sting uses the buffer cache to access local disks, and the network driver to access remote Swarm storage servers.

The Sting store agent is responsible for cleaning up after a dirty block has been written, by releasing all relevant locks and marking the block as clean. The Linux page cache is then free to replace the block.

The Sting replay agent performs much the same function as the traditional Unix `fsck` program that fixes file system metadata after a crash. During normal operation, file namespace operations such as creating a file or directory, creating a hard link, or unlinking a file or directory generate records that are stored in the log. During replay, the replay agent processes Sting's records from the log in order, using them to reconstruct the correct namespace. Data operations, such as writes, are logged in a similar function, allowing the file's contents to be reconstructed during replay.

Sting's cleaning agent cleans blocks by reading them into the cache and marking them as dirty. The file cache then writes them back out to the log at a later time. As a sanity check, the cleaning agent first cross-references the block with the file metadata to verify that it is still in use. If it isn't, it is simply deleted.

We measured the effectiveness of Sting's layout agents by comparing the performance of Sting and ext2 when running a benchmark that unpacks and compiles the Linux kernel. Both Sting and ext2 were configured to use a single disk; Swarm's striping mechanism was not used. Sting completes the task in 4:53 min, and ext2 in 4:59 min. These results are the average of five trials, and from them we conclude that Sting's layout agents allow Sting to achieve comparable performance to ext2, without hardcoding Sting's layout policies or metadata management into the lower layers of the storage system.

4.4. FAT file system

We have also implemented a FAT file system service in Swarm. The FAT file system uses a File Allocation Table (FAT) to control the layout of files. The FAT table is implemented as an array with one entry for each disk block. Each entry contains the log address of the data block and a pointer to another FAT entry, allowing blocks to be linked together into a linked list. A file is implemented as a linked list of FAT entries.

Layout in the FAT file system uses the default layout agent that is supplied by Swarm's agent layer. The FAT file system uses Swarm's agents to keep its metadata up to date. Its commit agent updates the block addresses in the FAT table, and its store agent frees all locks associated with the block and marks the block as clean. The FAT replay agent loads the FAT superblock from the disk and processes the FAT records from the log to update the FAT table. Finally, the FAT cleaning agent reads the block from disk and immediately writes it back out to the log. The FAT file system does not make use of the Linux page cache, and therefore it writes the block to the log itself. The FAT file system was implemented in approximately 1200 lines of C code.

4.5. Simple logical disk

The simple logical disk (SLD) service presents the abstraction of a virtual disk that can be used by higher-level services to store fixed-size blocks of data at fixed addresses. The SLD insulates higher-level services from the log by maintaining a *block address table* (BAT) that maps SLD addresses to log addresses. When a block is moved within the log its entry in the BAT changes, but not its index into the BAT. This allows traditional file systems, such as ext2, to run on Swarm without modification.

The SLD agents are responsible for maintaining the BAT. SLD uses two commit agents to accomplish this. The BlockCommit agent is attached to data blocks, and is used to update the block's entry in the BAT. The TableCommit agent is attached to the blocks of the BAT itself, and is used to store the BAT block's address in the SLD superblock. This is a good example of a service that has two types of metadata (the BAT and the superblock) and uses different agents to keep the two up to date.

SLD also attaches a replay agent to all records that reads the SLD superblock and BAT from the disk and updates them with the log addresses of the blocks being replayed. SLD was implemented in approximately 600 lines of C code.

4.6. Application layout agents

Swarm's agent mechanism is also available to application programs. This is useful, for example, to application programs that store files in Sting but want to influence how Sting organizes blocks in the log. By attaching its own layout agent to records, an application can implement block layouts that differ from Sting's. In this section, we present two sample application layout agents: a Web page agent and a read-ordered agent. The Web page agent clusters Web pages with their embedded images, and the read-ordered agent lays out blocks according to previously observed read access patterns.

4.6.1. Web page layout

HTML pages often contain embedded images. These images are referred to by URL in the HTML document, and are stored in a separate file in the Web server's file system. If a browser reads a page, it is almost certain to read the embedded images too. The Web page layout agent attempts to cluster pages together with the embedded images they contain. The pages and images are stored in Sting, so the Web layout agent is invoked after Sting's layout agents, allowing the former to override the layout specified by the latter.

The simplest way to determine the images embedded in a page is to parse the page's HTML. The Web page layout agent relies on a user-level program to parse the Web pages and present the image information to the agent in an easily processed form. The layout agent is attached to all the records for the Web pages and images, and when it is invoked, it creates a set that contains all the blocks for the Web page and its images. The blocks of the Web page are put into the set first, followed by the blocks of the images, in the order in which the links to the images appear in the page. This causes Swarm to cluster the blocks on the disk in the order in which they are likely to be accessed.

We performed a simple experiment to demonstrate such an agent is easily implemented, and can result is significant performance gains. The agent consists of only about 300 lines of C code. For the experiment, a process reads 350 HTML pages, each containing 10 embedded images. The pages and images are stored in separate directories. Pages are each approximately 300 bytes, and images vary from 2 to 32 KB.

The benchmark completed in 29.7 s when using Sting and the Web layout agent vs. 34.9 s on ext2, thus demonstrating the benefit of an application-specific layout agent. We were able to obtain a modest performance gain using a very simple agent. This experiment is not intended to be definitive on how to organize Web pages on disk, but does demonstrate that agents allow applications to deviate from the default layout policies, and that doing so can result in performance gains.

4.6.2. Read-ordered layout

The read-ordered layout agent puts blocks into sets in the order in which they were previously read. Most files are read sequentially and in their entirety, so this agent might seem uninteresting, but it does improve start-up performance for executables, whose pages are typically not read in order.

The read-ordered agent has two components: a facility that records the read pattern, and the layout agent itself. In our current prototype, the recording is turned on and off by the user. The recorded access pattern is then used by the layout agent to order the blocks in the file the next time they are cleaned. The layout agent itself is relatively simple, consisting of about 250 lines of C code. It reads the recorded access pattern for a file and puts the file's blocks into sets in the order in which they were read.

This causes the log layer to store the blocks in the same order. We measured the improvement in start-up times of two applications, emacs and gdb. Using the read-ordered agent improved emacs start-up time by a factor of 1.4 (from 534 to 373 ms). Similarly, gdb improved by a factor of 1.3 (from 240 to 183 ms). We consider these respectable performance improvements from such a simple agent. Swarm's agent infrastructure makes this possible, by allowing the agent to organize the blocks according to past access patterns.

5. STATUS AND FUTURE WORK

The agent infrastructure and services described in this paper are implemented in the Swarm prototype, except for storing persistent agents in the log. The reference to a persistent agent is stored in the records, but the code for the agent itself is not stored in the log. Instead, the system relies on the service or application to re-register the agent with the agent layer on system startup. Requiring agents to be re-registered is not a problem for services that are initialized on startup, such as Sting, but does not work well for agents that were created by applications. We are currently working on adding the functionality to store persistent agents in the log.

On a related note, there is a tradeoff between how much functionality should be encapsulated in the agent, and how much the agent can get from its environment. Encapsulating all functionality in an agent makes the agent self-sufficient, but increases the size of the agent and may complicate the design and implementation of the service. On the other hand, a minimal agent is smaller and probably does not affect the service's organization as much, but it requires a richer environment in which to run. As an example, the Sting agents interpret and modify Sting's metadata, such as inodes, indirect blocks, and directories. In the current implementation, the Sting agents rely on routines in the Sting service to perform much of this work. This reduces the agent complexity, but requires that the Sting service exist in order for them to run. This violates the premise of persistent agents, that they will continue to do their work after the service that created them ceases to exist. Ideally, one should be able to configure a system without Sting, yet continue to have the cleaning agents attached to Sting's records function. This does not work in the current system. We might be able to simply reorganize functions so that the agents are self-contained, but it may take refactoring Sting to do so.

Although we demonstrated Swarm's flexibility by implementing several types of agents, there are obviously many more possible types. Security is one aspect of a file system that is typically inflexible; existing solutions range from simple ownership and permission bit schemes to more complex access control lists. An access control agent could implement an arbitrarily complex policy, allowing security to take into account the context in which files are accessed. Agents could also be used for instrumentation and statistic collection. For example, an agent could perform security monitoring or intrusion detection.

6. RELATED WORK

Swarm is log based, and as such is heavily influenced by the Log-Structured File System (LFS) [8]. Swarm's use of a log as the only storage abstraction mirrors LFS, and Sting's use of inodes and an inode map are also borrowed from LFS. Swarm differs from LFS in the use of agents to affect log layout, metadata management, and cleaning. This allows the file system, Sting, to be decoupled from the storage system, Swarm. In LFS, these two functions are tightly coupled. This decoupling is also one of the features that distinguishes Swarm from Zebra [3].

Other file systems have allowed applications to specify data layout, typically through small, notational programming languages. MPI-IO [12], for example, allows each file to have layout attributes (info) such as the stripe width, size of each striping unit, and the size of each array element for files that store arrays. This information allows the underlying storage system to store the file efficiently, but has limited semantics. The Scalable I/O File System [13] has similar functionality and limitations.

Extensible operating systems allow entire subsystems to be added and replaced, including file systems. Typically, the entire file system is installed as a whole, which does allow file system functions such as layout to be tailored to an application's needs, but is a very heavy-weight mechanism for doing so. Linux provides loadable kernel modules that allow entire file systems to be loaded in this fashion. Mach provides for external pagers [14], which are user-level daemons that move virtual memory pages between memory and disk. This mechanism could also be exploited by an application to affect layout policies, but is also a heavy-weight solution. The Xok exokernel [15] supports user-level library file systems (libFSes). The underlying disk storage is multiplexed among libFSes via XN, the exokernel's in-kernel storage system. Each libFS is responsible for managing its portion of the underlying storage, allowing it to implement its own metadata and layout policies. XN provides protection between libFSes using untrusted deterministic functions, which interpret libFS-specific metadata for XN. These functions allow XN to determine which blocks belong to which libFSes. Swarm uses ACLs for protection, although a discussion of this topic is outside the scope of this paper. Xok is similar to Swarm in that it multiplexes the underlying storage among multiple storage services, but has very different mechanisms for doing so.

The Logical Disk (LD) [9] aggregates multiple physical disks into a single virtual disk, thus hiding the storage system's organization from the file system that is using it. LD provides a list abstraction that helps accomplish this. LD attempts to cluster blocks on the same list together, allowing the file system to express relationships between blocks and how they should be stored. Similarly, the block lists themselves can be placed in a larger list, expressing locality between lists. LD attempts to store lists that are near one another in the meta-list close together on the disks. Swarm's set abstraction is similar to LD's lists, but Swarm's agent abstraction has no parallel in LD. LD has no inherent mechanism for creating and changing lists.

Active disk technology [16] makes use of processing power on the disk drive to run application code. Similarly, Scriptable RPC [17] and ABACUS [18] both allow executable content to be run on a remote storage system. These techniques can dramatically improve application performance by moving processing closer to the disk, avoiding I/O bus and network bottlenecks, and by taking advantage of the inherent parallelism in running application code on multiple disks or servers. Swarm's agent system differs in that it is targeted at the functioning of the storage system itself, allowing higher-level services to affect low-level policies. Swarm agents are attached to data and are thus able to continue to function even with the service that created them ceases to exist.

7. CONCLUSION

Agents provide a flexible mechanism for services and applications to implement policies that affect low-level storage system functions, such as data layout, metadata management, and crash recovery.

Swarm must multiplex a single log between multiple services efficiently, and do so without understanding the internals of those services. Agents provide the means of doing this. A service can attach a service-specific agent to a record when it is passed to the log for storage. The agent will be invoked when its associated event occurs (e.g. the block is assigned a log address), allowing the service to take service-specific actions in response. In this way, Swarm can be organized in layers, such that the higher layers augment the functionality of the lower layers, without the lower layers having to know anything about the higher layers. For example, the cleaning layer can clean the blocks belonging to higher layers without knowing implicitly how the blocks should be organized on disk, or the format of the block's metadata.

We implemented several services that use agents. The cleaner not only uses agents to implement cleaning, but also creates a new type of agent, a cleaning agent, that higher-level services can use to influence the cleaner. The Sting file system uses agents to implement basic file system functionality, including laying out a file's blocks contiguously, and updating metadata to record block locations. The SLD service uses agents to implement a simple logical disk. Finally, we implemented several application-level layout agents to demonstrate that applications that use a file system can use agents to influence how the file system organizes blocks on disk. The Web agent clusters a Web page and its included images on the disk, improving Web server performance, and the read-ordered agent organizes blocks in the order in which they are accessed, improving read performance. These examples demonstrate the value of agents in file system design.

REFERENCES

1. Hartman JH, Murdock I, Spalink T. The Swarm scalable storage system. *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems (ICDCS '99)*, June 1999. IEEE Computer Society Press: Los Alamitos, CA, 1999; 74–81.
2. Gibson GA *et al.* File server scaling with network-attached secure disks. *Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, June 1997. ACM Press, 1997; 272–282.
3. Hartman JH, Ousterhout JK. The Zebra striped network file system. *ACM Transactions on Computer Systems* 1995; **13**(3):274–310.
4. Hutchinson NC, Peterson LL. The *x*-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering* 1991; **17**(1):64–76.
5. Necula G, Lee P. Safe kernel extensions without run-time checking. *Proceedings of the Second Symposium on Operating Systems Design and Implementation (OSDI '96)*, October 1996. USENIX Association, 1996; 229–243.
6. Wahbe R, Lucco S, Anderson TE, Graham SL. Efficient software-based fault isolation. *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93)*, December 1993. ACM Press, 1993; 203–216.
7. Seltzer MI, Endo Y, Small C, Smith KA. Dealing with disaster: Surviving misbehaved kernel extensions. *Proceedings of the Second Symposium on Operating Systems Design and Implementation (OSDI '96)*, October 1996. USENIX Association, 1996; 213–227.
8. Rosenblum M, Ousterhout JK. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems* 1992; **10**(1):26–52.
9. de Jonge W, Kaashoek MF, Hsieh WC. The logical disk: A new approach to improving file systems. *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93)*, Asheville, NC, December 1993. ACM Press, 1993; 15–28.
10. Grimm R, Hsieh WC, de Jonge W, Kaashoek MF. Atomic recovery units: Failure atomicity for logical disks. *Proceedings of the 16th International Conference on Distributed Computing Systems*, Hong Kong, May 1996, IEEE Computer Society Press: Los Alamitos, CA, 1996; 26–37.
11. McKusick MK, Joy WN, Leffler SJ, Fabry RS. A fast file system for UNIX. *ACM Transactions on Computer Systems* 1984; **2**(3):181–197.
12. Message Passing Interface Forum. Mpi-2: Extensions to the message-passing interface. http://www.mpi-forum.org/docs/mpi-20.ps.Z [17 May 2002].

13. Corbett PF *et al.* Proposal for a common parallel file system programming interface, September 1996. Version 1.0. http://www.cs.arizona.edu/sio/api1.0.ps [17 May 2002].
14. Rashid R, Tevanian A, Young M, Golub D, Baron R, Balck D, Bolosky WJ, Chew J. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. *IEEE Transactions on Computers* 1988; **37**(8):896–908.
15. Kaashoek MF, Engler DR, Ganger GR, Briceño HM, Hunt R, Mazières D, Pinckney T, Grimm R, Jannotti J, Mackenzie K. Application performance and flexibility on exokernel systems. *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, Saint-Malô, France, October 1997. ACM Press, 1997; 52–65.
16. Riedel E. Active disks—remote execution for network-attached storage. *PhD Thesis*, Carnegie Mellon University, November 1999. Available as *Technical Report CMU-CS-99-177*.
17. Sivathanu M, Arpaci-Dusseau AC, Arpaci-Dusseau RH. Evolving RPC for active storage. *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, San Jose, CA, October 2002. ACM Press, 2002; 264–276.
18. Amiri K, Petrou D, Ganger GR, Gibson GA. Dynamic function placement for data-intensive cluster computing. *Proceedings of the USENIX 2000 Annual Technical Conference*, June 2000. USENIX Association, 2000; 307–322.