Activating Storage Systems with Agents[1]

John H. Hartman   Scott Baker   Ian Murdock

TR 02-01

Abstract

Swarm is a scalable, modular storage system that allows high-level services to influence low-level storage functions such as data layout, metadata management, and crash recovery via *agents*. An agent is a program that is attached to data in the storage system and invoked when particular events occur during the data's lifetime. For example, when Swarm needs to write data to disk, agents attached to the data are invoked to determine a layout policy. Agents can be persistent, so that they remain 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. Swarm and its agent mechanism are implemented as a Linux kernel module. 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.

June 18, 2002

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

[1]This research was supported in part by DARPA Contract F30602-00-2-0560 and NSF grant EIA-0080123.

1 Introduction

Traditional storage systems are inflexible, providing fixed storage abstractions, access protocols, and data management policies. In contrast, the Swarm storage system [5] may be configured to support multiple storage services simultaneously, each implementing its own abstractions, access protocols, and data management policies. 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 allow 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 particular events occur during the data's lifetime. For example, when Swarm needs to write data to disk, agents attached to the data are invoked to determine a layout policy. Agents are stored alongside the data they manage and are persistent, allowing the agents to continue managing the data even after the applications or file systems that created them exit or are unmounted.

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. For example, 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 current system state to determine a policy that is optimized for a particular situation.

Swarm is implemented as a loadable module for the Linux 2.2 kernel. We have developed and experimented with several Swarm-based services that use agents, including a local file system called Sting that stores files and directories in Swarm, a cleaner service that reclaims unused portions of Swarm's log, a simple logical disk that presents a virtual disk abstraction on top of Swarm's log abstraction, a web layout agent that clusters web pages and their embedded images in the log, and a read-ordered layout agent that organizes file blocks according to previous read access patterns. The different agents employed by these services and the resulting performance improvements demonstrate the usefulness of the agent infrastructure. The overhead of invoking an agent is less than 1us, and a simple agent requires about 4us per block of computation, making the agent mechanism a viable way of implementing high-level policy decisions in low-level storage functions.

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. The agent section also includes a discussion of the performance overhead incurred, and security considerations. Finally, we describe the Swarm-based services we developed that use agents, and the benefits they derive from doing so.

2 Swarm Overview

Swarm [5] 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 that 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 [4]. Individual storage devices are optimized for cost-performance and aggregated to provide the desired absolute performance.

Swarm clients use a *striped log* abstraction [6] 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 allows missing portions of the log to be reconstructed when a storage device fails. Clients cache blocks in memory and write them to the log in batches, allowing blocks within the batch to be ordered to improve read performance, and also 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

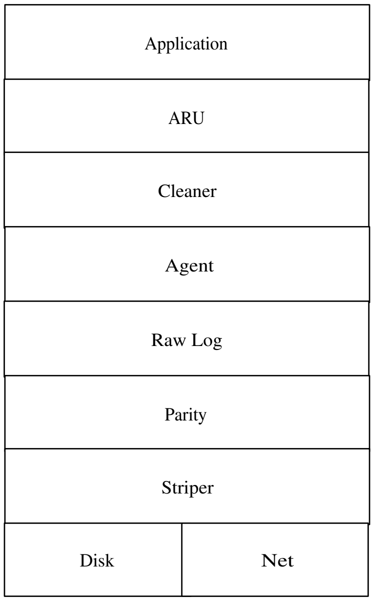

Figure 1: **Swarm Architecture**. A particular instance of Swarm is constructed by layering Swarm modules to obtain the desired functionality for the storage system. Each layer augments, extends, and/or hides the functionality of the layers below it. The agent layer is responsible for implementing the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

perform these functions; this results in improved scalability, reliability, and performance over centralized file servers.

Swarm is a storage system, not a file system, because it can be configured to support a variety of storage abstractions and access protocols. For example, a Swarm cluster could simultaneously support Sun's Network File System (NFS) [14], HTTP, a parallel file system, and a specialized database interface. Swarm accomplishes this by decoupling high-level abstractions and functionality from low-level storage. Rather than providing these abstractions directly, Swarm provides an infrastructure that allows high-level functionality to be implemented above the underlying log abstraction easily and efficiently. This infrastructure is based on layered modules that can be combined together to implement the desired functionality (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.

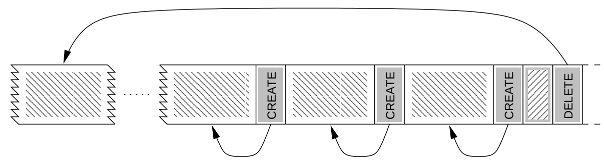

Figure 2: **Log Format**. The light objects are blocks, and the dark objects are records. Each CREATE record indicates the creation of a block, and each DELETE record indicates a deletion; the arrows show which block is affected by each record and represent references visible to the log layer. Note that the contents of the blocks themselves are uninterpreted by the log layer.

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.

2.1.1 Log Format

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.

Block contents are service-defined and are not interpreted by the log layer. Once written, blocks persist until explicitly deleted, though their physical locations in the log may change as a result of cleaning or other reorganization. New blocks are always appended to the end of the log, allowing the log layer to batch together small writes into fragments that may be efficiently written to the storage devices. Once written, a block may be accessed using its *log address*, which consists of a unique fragment identifier and an offset within the fragment. Given a block's log address and length, the log layer retrieves the block from the appropriate storage device and returns it to the calling service. When a service stores a block in the log, the log layer responds with its log address so that the service may update its metadata appropriately.

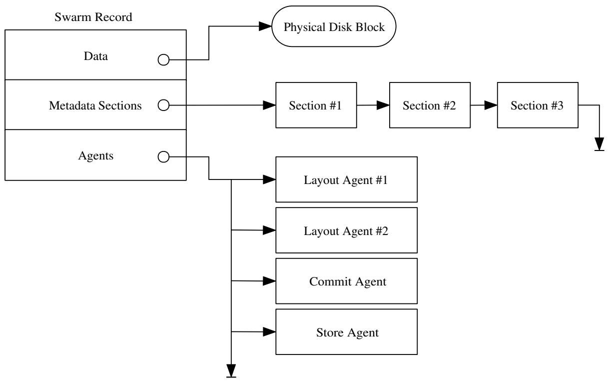

Figure 3: **Record Format**. Each record contains a pointer to an associated data block (if any), a variable number of sections in which each service stores service-specific recovery information, and references to each agent that has been attached to the record.

Records are used to recover from client crashes. A crash causes the log to end abruptly, potentially leaving a service's data structures in the log inconsistent. 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 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. Records are implicitly deleted by *checkpoints*, special records that denote consistent states. The log layer guarantees atomicity of record writes and preserves the order of records in the log, so that services are guaranteed to replay them in the correct order.

2.1.2 Log Layout

As applications and storage services write data to Swarm, the log layer caches the blocks in memory and writes them to the log in batches. As the log layer 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 log accesses. Blocks that are accessed together 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.

The log layer 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, the log layer provides an *ordered sets* abstraction that allows services to express block layout preferences. Each set contains a list of blocks that should be clustered together in-order in the log. The services that are using the log create the sets and assign blocks to them. A service can create as many sets as it likes, and assign an arbitrary number of blocks to each set.

To store blocks in the log, the service submits the sets containing the 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, blocks will be members 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, the set priorities are used to decide which set a block is clustered with, and which sets are split. 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 for placing blocks in the log based on set membership and priorities, and splitting sets when necessary. First, the sets are ordered from lowest priority (1) to highest (N). 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 priorty ensures that the log layer favors splitting lower-priority sets over higher-priority. 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. If the service has a preference, it should use the set priority mechanism to express it.

3 Agent Infrastructure

Swarm provides an infrastructure for building storage services on top of the striped log, allowing applications to tailor the storage system to their exact needs. Swarm is implemented as a collection of modules that are layered to build storage systems in much the same way that protocols may be layered to build network communications subsystems [7]. 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 the appropriate modules such that all interfaces between modules are compatible.

To provide a clean separation between layers, Swarm allows storage services to attach *agents* to the records that move up and down the service stack. Agents are programs that are invoked at various points in the record's lifetime to influence or control how the record and its associated data block are managed in the storage system. Agents allow services to inject service-specific functionality into the lower levels of the storage system, and to do so in a way that does not require the lower levels to have any knowledge about how the storage service works. For example, agents allow Swarm to implement application-specific data layout policies without knowledge of application access patterns, and to update application metadata without knowledge of application metadata structures.

Agents allow mechanism to be effectively and efficiently decoupled from policy in the implementation of storage services. The mechanism for organizing blocks in the log is the ordered sets abstraction. A set tells the log that the blocks it contains should be stored consecutively

```
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 4: **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 Swarm stack.

and contiguously. It does not tell the log layer why they should be stored that way, so the log layer has no idea under what conditions the set memberships remain valid. Instead of augmenting the set abstraction with attributes that communicate these sort of policies to the log layer, Swarm uses agents that codify the policies for placing blocks into sets.

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.

The agent infrastructure is implemented in the *agent layer*. The agent layer is responsible for flushing the cache and invoking service-provided agents to assign the blocks and records in the cache to ordered sets. The agent layer provides an interface for services to create agents and attach them to records (Figure 4). Although each record could have its own unique agents, typically a single agent will be attached to multiple records that should be handled similarly.

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.

3.1 Agent Types

The agent layer implements four native types of agents: *layout*, *commit*, *store*, and *replay*. Layout agents are invoked when the agent layer flushes the cache, allowing services to specify a layout policy for the records and blocks being flushed. Commit agents are invoked after the log layer has assigned a log address to a record and its associated block, allowing the service to update its metadata to reflect the new address. Store agents are invoked after the record and its associated block have been written to the log, allowing the service to clean up any record state. Replay agents are invoked when replaying records after a crash, allowing services to take actions appropriate for crash recovery.

The agent layer 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.

3.1.1 Layout Agents

Layout agents are responsible for deciding how blocks and records should be laid out in the log. A layout agent is invoked when blocks are flushed from the cache and written to the log. When it is invoked, the layout agent is provided a list of all records to be written that have the agent attached to them. The agent processes the list and puts the records into ordered sets. The log layer uses the ordered sets to determine where blocks are placed in the log; it attempts to store the blocks in each set contiguously and in-order. The layout agent can use whatever method it chooses to allocate blocks to sets. For example, a layout agent for a file system may assign blocks from each file to a different set, in the order in which they appear in the file. This ensures that file blocks are laid out contiguously and in-order.

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

3.1.2 Commit Agents

The commit agent for a record is invoked once the raw log layer 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 block's record and the log address where it will be written. The commit agent typically 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.

The agent layer also provides a default commit agent to simplify service implementation. This agent requires that the recordData be a list of log addresses, and it updates them to contain the block's address. This is adequate for services with simple metadata.

3.1.3 Store Agents

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, for example, by removing the block from the cache. This cannot be done until the block has been stored. As another example, a synchronous block write can be implemented by registering a store agent on the block before submitting it to the log layer. When the agent is invoked, it wakes the thread that is writing the block.

3.1.4 Replay Agents

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 handled.

3.2 Agent Interface

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.

3.3 Agent Persistence

Typically, an agent is *persistent*, in that it remains attached to a record until the record is deleted. A persistent agent is stored in the log by the agent layer 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.

The agent layer also supports *transient* agents, agents that are invoked only once and not retained 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 that is used for synchronous log writes. This agent is attached to a record when it is submitted, after which the submitting thread blocks. The agent is invoked once the block is stored in the log, and it resumes the waiting thread. Since it is transient, it is only invoked once, as desired, and it does not survive machine reboots, which is also desirable since the waiting thread will not either.

3.4 Overhead

The agent layer does add overhead to the storage functions provided by Swarm. 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 intent is that the overall system performance improvements that agents enable offsets the overhead of running the agents. Section 4 describes the different agents developed and how much they improved system performance.

An agent is invoked when a particular event occurs to a record. The agent is expected to respond to the event by manipulating the state of the system, e.g. by adding the record to a set, or updating metadata. For this reason, agents are invoked synchronously by Swarm. The overhead of invoking an agent consists of the cost of a procedure call, plus the cost of packaging up the records on which the agent should act. We measured the cost of invoking a null agent (one that does no work) at less than 1 microsecond.[1]

We also measured the overhead of the default layout and commit agents described in Section 3.1.1 and Section 3.1.2, respectively. These default agents are probably the minimal useful agents for those agent types. The default layout agent requires 21 microseconds per block, of which memory allocation consumes 16 microseconds, and manipulating the set data structures 4 microseconds. 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 around 5 microseconds. The default commit agent does much less work than the layout agent, and therefore requires only 4 microseconds per block.

[1] All performance numbers presented in this paper were measured on a 166Mhz Intel Pentium Pro PC with 64MB of RAM, running Linux version 2.2.16. The Swarm log is stored on a Quantum Fireball SE4.3 SCSI disk connected to an Adaptec 2940W SCSI host adapter.

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 [10] to verify the agents' correctness, or using software fault isolation [16, 15] to isolate the agents. All of these should be acceptable, although running agents in a separate process will likely have high overheads.

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. Software fault isolation is probably the best match for our current prototype as it allows the agents to be written in C, but still isolate them. Software fault isolation has the added advantage that the Vino 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.

4 Examples

We have implemented several types of 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.

4.1 Cleaner

As in other log-structured storage systems, Swarm uses a *cleaner* that periodically garbage-collects unused blocks in the log to make room for new segments [13]. In Swarm, the cleaner is implemented as a layer above the log, hiding the log's finite capacity from higher-level services. The cleaner monitors the blocks and records written to the log, allowing it to track which portions of the log are unused. 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.

In Swarm, the cleaner operates by attaching agents to records as they are submitted to the log. The cleaner uses store agents to track which blocks in the log contain live data and which have been deleted. The store agent attached to creation records updates the cleaner's data structures to indicate that the associated blocks contain live data; conversely, the store agent attached to deletion records marks the associated blocks as deleted, and also deletes stripes that become empty as a result.

The cleaner also creates a new type of agent called a *cleaning* agent that is used by the upper layers to clean blocks. The cleaner invokes a record's cleaning agent when it decides the block must be cleaned. The cleaning agent takes whatever actions are necessary to clean the block. For example, 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.

4.2 Sting

Sting is a local file system that we have implemented as part of Swarm (Figure 5). When loaded into the Linux kernel, it allows application programs to access standard UNIX files and directories that are stored in Swarm. 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 [9]. Sting uses two layout

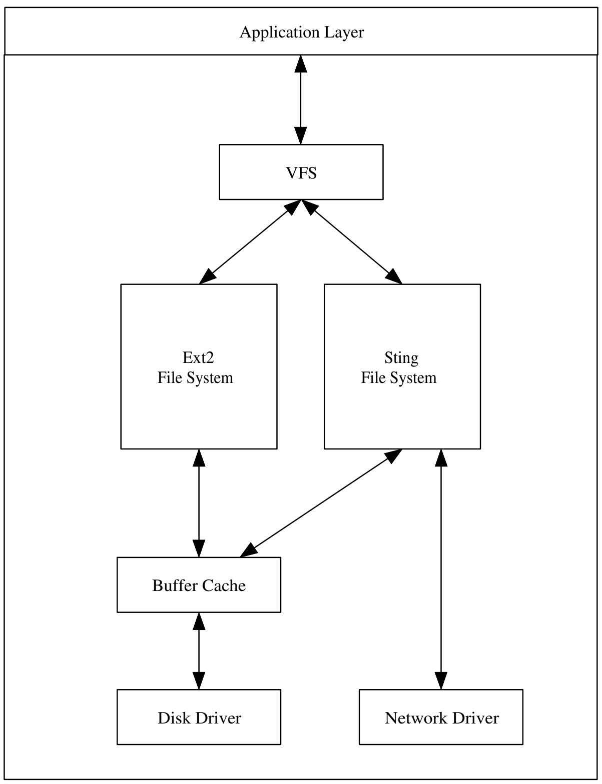

Figure 5: **Sting**. Sting is implemented as a Swarm module. The entire Sting/Swarm system is loaded into the Linux kernel below the VFS layer and above the buffer cache and network drivers. Sting uses the buffer cache to access local disks, and the network driver to access remote Swarm storage servers.

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.

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 as necessary.

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.

Sting's cleaning agent cleans blocks by reading them into the cache and marking them as dirty. The file cache will then write 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.

4.3 Simple Logical Disk

The simple logical disk (SLD) service presents the abstraction of a logical disk, one in which blocks are accessed via fixed addresses. The SLD insulates higher-level services from the log by maintaining a mapping from SLD addresses to log addresses. When a block is moved within the log, this mapping changes, but not the

SLD address. This allows traditional file systems, such as ext2, to run on Swarm without modification.

The SLD agents are responsible for maintaining the mapping table. SLD uses two commit agents to accomplish this. The BlockCommit agent is attached to data blocks, and is used to update the block's log address in the mapping table. The TableCommit agent is attached to blocks that contain the mapping table itself, and is used to store the table block's address in the SLD superblock. This is a good example of a service that has two types of metadata (the mapping table and the superblock), and that 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 mapping table from the disk and updates them with the log addresses of the blocks being replayed.

4.4 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.

4.4.1 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 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 the log layer 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 two HTML files, each containing four embedded images. This simulates a web browser viewing the two pages. The pages and their images are stored in the same directory. The default Sting agents will cluster all of the blocks together because the files are in a single directory, but in an unspecified order. Ext2 will store the blocks similarly.

With 4KB images, the pages and images are read a factor of 1.7 times faster using the web-layout agent than Sting alone, and 7.7 times faster than ext2. Larger images reduce the benefit of the smaller seek times the agent provides, but with 64KB images the read time was still 1.3 times faster than Sting alone, and 1.9 times faster than ext2. These experiments are not intended to be definitive on how to organize web pages on disk, but do demonstrate that agents allow applications to deviate from the default layout policies, and that doing so can result in substantial performance gains.

4.4.2 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 startup times of three applications, emacs, gdb, and jikes (a Java compiler). Using the read-ordered agent improved the emacs start-up time by a factor of 1.7 (from 1.2 to 0.7 seconds). Similarly, gdb improved from by a factor of 1.67 (0.5 to 0.3 seconds), and jikes by a factor of 1.5 (0.3 to 0.2 seconds). 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

The agent infrastructure and services described in this paper have been implemented in the Swarm prototype, with the exception of persistent agents. The reference to a persistent agent is stored in the records, but 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. This solution assumes that agents' identifiers and functionality doesn't change between reboots, which may not be reasonable. Requiring agents to be re-registered is not a problem for services that are initialized on startup, such as Sting, but doesn't 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 doesn't 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 how Sting is architected to reach this goal.

6 Related Work

Swarm is log-based, and as such is heavily influenced by the Log-Structured File System (LFS) [13]. 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 [6].

Organizing data on disk to improve access performance has a long history and many examples. Probably the closest to our work are the layout policies of the Fast File System (FFS) [9]. Sting's layout agents' policies are inspired by FFS, in that both attempt to lay out files contiguously and cluster files from the same directory together. Sting differs from FFS slightly in that FFS has an upper limit on the number of file blocks it will store contiguously before moving to a different area of the disk. For simplicity, we did not implement such a limit in Sting.

Other file systems have allowed applications to specify data layout, typically through small, notational programming languages. MPI-IO [3], 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 [1] 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 [11], 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 [8] 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) [2] 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 [12] makes use of processing power on the disk drive to run application code. This can dramatically improve application performance by moving processing closer to the disk, avoiding I/O bus bottlenecks, and by taking advantage of the inherent parallelism in running application code on multiple disks. The active disk work thus far has been confined to running application algorithms on the disk drives; Swarm's agent technology focuses on using agents to influence the functioning of the storage system itself.

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 have 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 have 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. Both of these agents demonstrate the value of agents in file system design.

Acknowledgments

We would like to thank Tammo Spalink and Rajesh Sundaram for their help in designing and implementing Swarm.

References

[1] Peter F. Corbett, Jean-Pierre Prost, Chris Demetriou, Garth Gibson, Erik Reidel, Jim Zelenka, Yuqun Chen, Ed Felten, Kai Li, John Hartman, Larry Peterson, Brian Bershad, Alec Wolman, and Ruth Aydt. Proposal for a common parallel file system programming interface. http://www.cs.arizona.edu/sio/api1.0.ps, September 1996. Version 1.0.

[2] Wiebren de Jonge, M. Frans Kaashoek, and Wilson C. Hsieh. The logical disk: a new approach

to improving file systems. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93)*, pages 15–28, Asheville, North Carolina, December 1993.

[3] Message Passing Interface Forum. Mpi-2: Extensions to the message-passing interface. http://www.mpi-forum.org/docs/mpi-20.ps.Z.

[4] 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. In *Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, June 1997.

[5] John H. Hartman, Ian Murdock, and Tammo Spalink. The Swarm scalable storage system. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems (ICDCS '99)*, June 1999.

[6] John H. Hartman and John K. Ousterhout. The Zebra striped network file system. *ACM Transactions on Computer Systems*, 13(3):274–310, August 1995.

[7] Norman C. Hutchinson and Larry L. Peterson. The *x*-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[8] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 52–65, Saint-Malô, France, October 1997.

[9] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[10] George Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation (OSDI '96)*, October 1996.

[11] R. Rashid, A. Tevanian, M Young, D. Golub, R. Baron, D. Balck, W. J. Bolosky, and J. Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. *IEEE Transactions on Computers*, 37(8):896–908, August 1988.

[12] Erik Riedel. *Active Disks - Remote Execution for Network-Attached Storage*. PhD thesis, Carnegie Mellon University, November 1999. Available as Technical Report CMU-CS-99-177.

[13] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[14] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the Sun Network File System. In *Proceedings of the Summer 1985 USENIX Conference*, June 1985.

[15] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation (OSDI '96)*, October 1996.

[16] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93)*, December 1993.