Load Management in a Large-Scale Decentralized File System

Atul Adya, William J. Bolosky, Ronnie Chaiken, John R. Douceur,
Jon Howell, and Jacob R. Lorch
Microsoft Research

Abstract

This paper discusses our general approach to load management in a distributed system, as well as its application to a particular system, Farsite. Farsite is a peer-to-peer distributed file system that uses its constituent machines to maintain consistency of file system metadata and replicated file content. We argue that control theory is inappropriate for load management in this and other similar systems, and give alternative techniques for preventing overload of limited resources such as CPU and disk. We describe our method of workflow graphs*, which allows a system designer to describe the potential sources of overload and ensure all are managed properly, and we apply this method to Farsite. We also describe novel techniques for load management, including* clown-car compression *and a scheme for achieving approximately even file replication without central coordination or global knowledge.*

1. Introduction

This paper addresses the problem of load management for distributed systems in general, and for our Farsite [1] system in particular. The goal of load management is to prevent the system from placing more load on system resources, such as disk space or network bandwidth, than these resources can handle. This is particularly difficult for distributed systems, in which each machine has an incomplete view of the system and thus may easily create more load than the system can handle.

Farsite is a large-scale file system implemented entirely without centralized servers. Its directories are maintained using Byzantine-fault-tolerant (BFT) replicated state machines [5], and its files are replicated and distributed among the machines that use the file system. It is easy for a client to apply substantial load to this system, e.g., by simply untarring or recursively copying a large directory tree. Under stresses of this sort, things tend to break: Work backs up, queues overflow, timeouts expire, messages get lost, and resources get depleted.

Most system designers focus on correctness and performance issues and tend to ignore load management. However, expecting that underlying subsystems such as the operating system scheduler and TCP will prevent overload is dangerous. Fig. 1 (discussed in detail in section 5) shows the importance of explicit load management. We ran a simple experiment in which we created a number of directories in two different versions of Farsite, one with all our load management techniques enabled, and one with a *single* load manager turned off. Removing just this one instance of load management caused the system's use of memory to perpetually grow, eventually exhausting memory and causing a system crash. Our experiences with Farsite show that there are many such crashes waiting to happen if designers ignore explicit load management.

Feedback control theory provides one way to deal with load. When machines begin to get overwhelmed with load, they send feedback signals to the machines generating it; these machines slow down their operation and relieve the overload. However, conventional control does not work well for Farsite, for at least three reasons. First, there is substantial lag between the generation of workload and when it is noticed, so feedback arrives much later than needed. Second, since every file creation leads to several replicas being made, there is a multiplicative effect of activity on workload, amplifying the effect of inaccurate feedback. Third, machines are heterogeneous and are not dedicated solely to Farsite, so it is difficult to predict the effect on the system of a given control.

Therefore, instead of using conventional control for Farsite and other similar systems, we recommend the following approach. First, *reduce* the effective workload by various standard techniques including caching and compression, or by a novel technique called *squelching* that stops unnecessary load from being produced. Next, prevent possible overload of bandwidth-limited resources such as CPU and network bandwidth by *queuing* or *shedding* requests that find

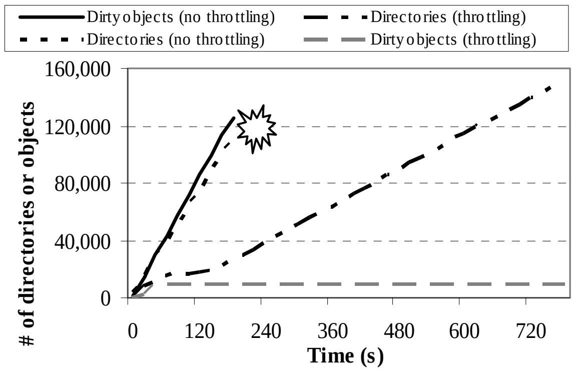

Fig. 1: Effect of throttling--crash occurs without it

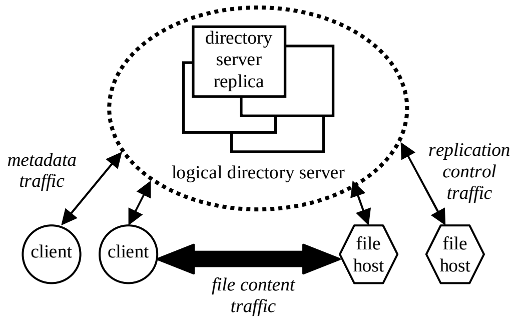

Fig. 2: Network interactions in Farsite

such resources saturated. Finally, prevent possible overload of resources with fixed amounts of space, such as memory and disk, with either *throttling* or *infinite-load management*, terms which we now describe in more detail.

Throttling stops the flow of load to a component whenever its load reaches a certain threshold, and infinite-load management allows a component to deal with an arbitrary amount of offered load. Infinite-load management can be achieved with various techniques, including *premature release*, *shedding*, and our novel *clown-car compression*. Premature release means a component releases load early to a downstream component if it reaches its limit. Shedding means a component simply drops load, relying on rectification processes to correct any resultant inaccuracies later. Clown-car compression means compressing an arbitrary amount of load into a fixed amount of space.

Large-scale systems may contain many sites that are susceptible to overload. To help a system designer identify such sites, and to assist in determining whether each potential overload has been appropriately addressed, we develop the method of *workflow graphs*. After we describe this method in general, we demonstrate its use in Farsite as we apply our workload-management techniques to that system.

Our work has yielded several contributions, including: (1) a successful demonstration of applying workload management in a working distributed system; (2) the method of workflow graphs for methodically identifying and eliminating potential overloads; (3) the technique of clown-car compression to accumulate an arbitrary amount of load; and (4) a mechanism for achieving approximately even file replication without central coordination or global knowledge.

This paper is organized as follows. Section 2 briefly describes Farsite. Section 3 discusses load management: what it is, why we believe conventional control theory is inapplicable, the techniques we suggest, and our method of workflow graphs. Section 4 shows how Farsite uses our load management techniques and applies our method of workflow graphs to that system. Section 5 shows experimental results demonstrating the effectiveness of our load management techniques in Farsite. Section 6 surveys related work, and section 7 summarizes and concludes.

2. Background: the Farsite file system

Farsite [1] is a distributed file system under development at Microsoft Research. Its goal is to run a file system entirely on untrusted client workstation machines, with minimal administrative control or trust, in well-connected environments on the scale of 100,000 nodes. Rather than relying on administrators to anticipate and provision for growth, it exploits unused storage space on client computers, which grows in aggregate as the computing installation size grows.

Farsite assumes a cooperative environment in which users allow others to use the spare disk capacity of their workstations. This assumption is most reasonable within an organization that owns all the machines on which users work, such as a university or corporation. As a consequence of this assumption, Farsite views disk space as a shared resource that needs to be managed from a global perspective.

Fig. 2 shows a stylized view of a Farsite file system. Every machine in the system has three roles: client, directory server group member, and file host. As a *client*, the machine provides an interface to the file system for local applications. As a *directory server* group member, the machine serves as one member of a replicated state machine managing file system metadata. As a *file host*, the machine stores replicas of file data on behalf of the overall system.

Although we expect most users of Farsite to cooperate, we allow for the possibility that some act maliciously. Thus, the replicated state machines used for directory groups are Byzantine-fault-tolerant [5], i.e., they tolerate arbitrary failure of participants, including failure to act appropriately. BFT state machines require a high degree of replication: $3f$+1 machines tolerate f failures. Therefore, storing actual file data in BFT state machines is impractical. Instead, Farsite replicates file contents on file hosts, where simple replication on f+1 machines tolerates f failures.

To scale to 100,000 nodes, a single Farsite installation will contain many independent directory server groups, each implementing a small subset of the file system namespace. Each client may interact with many such directory server groups.

2.1 System architecture

Fig. 3 illustrates the software architecture running on each host. Three managers implement the three roles a host assumes in the system: The *client manager* manages locally-used metadata and file data. The *file host manager* manages file replicas stored on the local disk. The *directory manager*, serving as a single replica in a BFT state machine, manages metadata.

In addition to these three *role managers*, Farsite includes two other main software components: A kernel-level *file-system driver* exports a file-system interface; it services data-intensive read and write operations directly, and relays most operations to the user-level client manager. The *object manager*, which serves all three role managers, manages the on-disk and in-memory storage of data other than file contents.

Persistent disk storage is used for several purposes, in addition to its direct use by applications and the operating system. The client manager and file host manager store encrypted file contents in NTFS files [16]. These files are exported remotely to enable access by other clients and replication by other file hosts. The object manager is a front-end to a conventional database system that also uses local disk.

2.2 System operation

When the Farsite kernel driver receives an application request to open a file, it handles the request as follows. If a copy of the file is in the local cache, it opens the file and returns a handle to the application. If the file is not present locally, it calls up to the user-level component of the client manager, which asks the directory server for a list of file hosts with replicas of the file. The client contacts one of these file hosts and copies the file to its cache. It then notifies the driver, which opens a handle to the file and returns it to the application. The file remains cached locally until it has gone unused for several days; while it is cached, open and read operations can be serviced by the driver alone.

When applications modify directories or files, the driver logs the updates and passes them up to the client manager at user level. The client manager sends the updates to the directory server in batches. These update messages include changes to directory information, such as created or deleted files, and notifications of writes to files. They do not include updated file contents, just a hash of them.

After the directory server learns of a new file having been written, it randomly selects a set of file hosts and tells them to make copies of the new file. When the server learns of an update to an existing file, it tells the file hosts that store old versions of the file to make up-to-date copies. The server inserts a *write-absorption delay* of one hour before telling file hosts to copy a recently written file, to provide an opportunity to absorb subsequent writes to the same file. Studies have shown that files are often overwritten or deleted soon after they are written [4, 20].

When a file is created or modified, the client marks the local copy of the file *dirty*, meaning the file is the only copy in the system and must not be ejected. Once another copy is made, the directory server notifies the client, and the client clears the dirty flag.

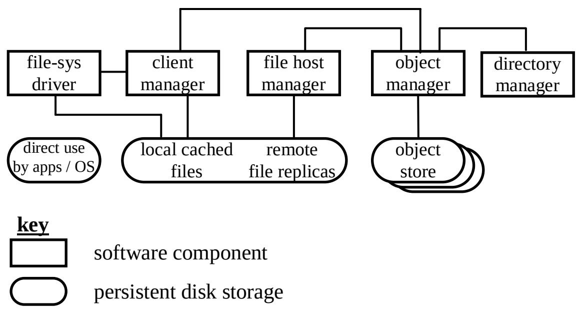

Fig. 3: Software architecture

2.3 Omissions

This system requires a mechanism for ensuring file consistency. Farsite employs a pessimistic strategy in which a server locks a file when a client requests it for write access. Lock management is an involved and orthogonal topic, so we do not discuss it further herein.

Similarly, there are obvious security concerns with storing files on clients. Encryption can protect against inappropriate reading of files, but there are more involved issues regarding destruction or damage of file copies. As with consistency, security is beyond the scope of this paper but is discussed elsewhere [1].

3. Load management

In this section, we discuss load management in systems such as Farsite. Our goal is to prevent system resources from becoming overloaded. Some of these resources, such as disk space, have limited size; we call them *static* resources. Others, such as CPU and network bandwidth, have limited bandwidth, i.e., their capacity is measured in units that include time, such as MB/s. We call these *dynamic* resources.

3.1 Problems with conventional control

One approach to load management is to use a conventional control system [12], which applies an adjustment to a process to limit the deviation of one or more *controlled variables* from their desired values. Control systems operate by adjusting the values of one or more *manipulated variables* so as to indirectly affect the values of the controlled variables. For example, in Farsite, the load on each resource is a variable we would like to control and the rate at which clients are permitted to perform file system operations is a variable we can manipulate.

Control systems are either *open-loop* or *closed-loop*. Open-loop controllers base their manipulations on a *predictive model* of the controlled systems' response; closed-loop controllers base manipulations on *observations* of the controlled systems' response. Open-loop controllers need accurate models of system behavior and precise measurement of system input to choose the magnitude and direction of their control. Closed-loop controllers can use simpler models to inform the direction of their control, allowing feedback to progressively refine the magnitude.

Open-loop control is not practical for Farsite, or most other distributed systems, since for such systems accurate models are infeasible. Machines cannot independently predict the load effect of their activity, since load is a function of all machines' activity. We would need a scalable, high-speed mechanism to gather and distribute load information among machines, which would add great complexity to the system if it were even possible. Furthermore, in Farsite, machines are randomly selected, heterogeneously configured, and externally perturbed, making accurate load estimation especially difficult.

A closed-loop controller also will not work for Farsite, or any other system in which activity has an effect on load far in the future. In Farsite, the hour-long write-absorption delay between the time files are updated and the time file replicas are made means that feedback for a closed-loop control system would arrive long after it was most needed. This long feedback delay also means that, to be even somewhat helpful in limiting load spikes, the delayed backpressure would have to be very aggressive, drastically retarding the completion of file I/O operations on the client and making the user experience very unpleasant.

Another problem with using a control system for Farsite is that the cost of generating future workload is far lower than the cost of processing that workload. Every file a client creates is replicated three to four times system-wide, so the bandwidth used for replication is many times the disk bandwidth used by the client. Also, workload generation uses local disk bandwidth, but replication uses remote disk bandwidth and network bandwidth, and the latter is far scarcer. The difference in relative cost gives the control loop a high forward gain, which magnifies any inaccuracy in the value of the applied control signal and necessitates aggressive feedback.

3.2 Load reduction

The first step in controlling load on resources is load *reduction*, i.e., lessening it. This does not prevent overload, but merely reduces its probability. There are countless examples of reduction techniques. Here, we give three illustrative examples that are used frequently in Farsite: *caching*, *compression*, and *squelching*.

Caching is storing results so later operations can reuse them and thereby place less load on the system.

Compression is combining multiple operations into one aggregate operation when performing the aggregate operation will produce less load than performing the operations separately. For example, in Farsite, file system updates are delayed in the hope that they can be combined with subsequent ones. Coda [14] also uses compression, replacing log entries with smaller, semantically equivalent sequences of entries.

Squelching is the prevention of unnecessary load production. For example, if a file host knows it will reject a certain subset of file replica requests, it can tell the server this and thereby prevent the server from unnecessarily introducing work items into the system. Squelching is especially useful when the rejecter and the item producer are on different machines, since the squelching reduces network load.

3.3 Preventing dynamic overload

There are two methods for preventing overload of dynamic resources: *shedding* and *queuing*. Shedding drops a work item when there are insufficient resources to handle it; this requires that the system designer add *rectification processes* to correct for deficiencies due to dropped work items. Queuing delays work until there are sufficient resources to handle it. This converts dynamic load temporarily into static load, namely the disk or memory space taken by the queued work item.

Often, queuing is implicit and requires no special implementation effort. For example, if one creates more jobs than processors can currently handle, the operating system scheduler will automatically block jobs until CPU bandwidth is available. However, this is not necessarily beneficial, since the resulting static load may not be apparent. For instance, if a system is loaded with far more jobs than it can handle, the large number of blocked thread stacks may cause memory overload. The system designer must foresee such scenarios and make the static load more explicit so that she can apply the techniques we will describe in later subsections for preventing static overload. Since not all such scenarios can be foreseen, stress testing the system is useful for identifying where static load caused by queuing is a problem.

3.4 Preventing static overload

3.4.1 Identifying potential static overloads

We have developed the *workflow graph methodology* to aid system designers in enumerating potential static overloads and ensuring there are sufficient mechanisms for preventing them all. In this subsection, we discuss the first step in this methodology: identifying and graphing the locations where static overload can occur in the system.

We call a place in the system where static load can accumulate a *reservoir*. Examples of reservoirs in Farsite are the queue of file replications needing to be made but for which bandwidth is currently unavailable, the set of file updates waiting for an hour before they are processed in the hope they can be compressed, and the set of file replicas stored on the local disk.

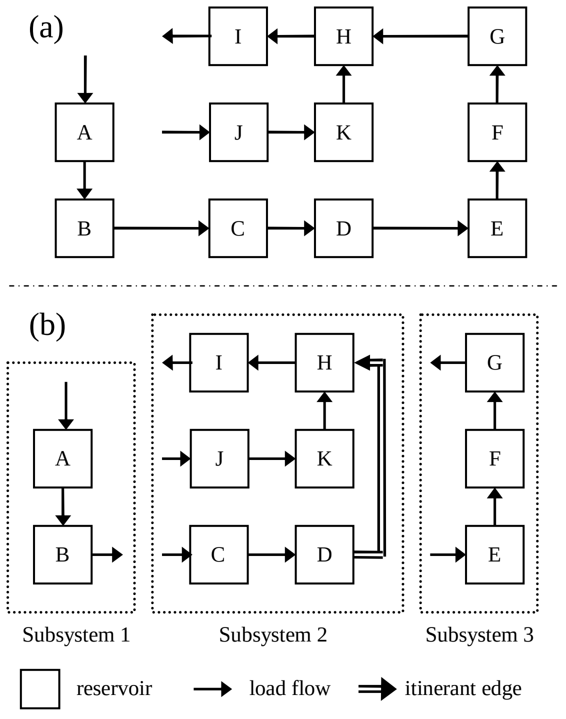

Fig. 4: Sample workflow graphs: (a) and (b) are the same system, with (b) divided into subsystems

The first step in our methodology is to graph the reservoirs and load flows in the system. A sample workflow graph is shown in Fig. 4. In such a graph, a box denotes a reservoir and an arrow denotes a load flow. An arrow between boxes shows where load can come out of one reservoir and enter another. The load removed from the first may be different than the load added to the second; for instance, a reservoir of replica creation requests may pass load into a reservoir of file replicas on disk, reflecting the fact that after making a file replica one deletes the replica request. An arrow with a sink but no source shows where load enters the system; e.g., this might represent an application modifying a file, thereby increasing load on the reservoir of updates to send to a server. An arrow with a source but no sink shows where load exits the system; e.g., an arrow out of a reservoir of local file copies could represent those files getting deleted.

Since most systems are large and complex, it is useful to break the system into possibly-overlapping subsystems and graph each separately. Furthermore, breaking the system into subsystems allows each graph to consist solely of reservoirs and load flows within a single machine, which is important for reasons we will describe later. Wherever dividing into subsystems separates the source and sink of a load-flow arrow, we depict this as an arrow getting rid of load in one subsystem and another arrow creating load in the other subsystem. For instance, when we divide the workflow graph of Fig. 4a into subsystems in Fig. 4b, the arrow from B to C becomes an out-arrow from B and an in-arrow to C. As a special case, if load can travel out of a subsystem, through external reservoirs, then back into the subsystem, we need to depict this flow. We call this an itinerant edge and depict it with a double-arrow. For instance, going from Fig. 4a to Fig. 4b, we needed to add an itinerant edge from D to H.

Every reservoir in the graph is a potential source of static overload that must be addressed with one of the methods in the next subsection. By annotating the graph as described in that subsection, we can ensure that these reservoirs are immune to static overload.

3.4.2 Static overload prevention techniques

We prevent static overload by ensuring that each reservoir is incapable of becoming overloaded. We have two ways to prevent overload of a reservoir: *throttling,* which prevents load in excess of its capacity from ever reaching it, or *infinite-load management*, which allows it to deal with arbitrary amounts of load.

Throttling means setting a limit on a reservoir's load and temporarily cutting off the flow of load to that reservoir when the reservoir reaches that limit. We call this limit the *throttling threshold*. For example, in Farsite, one of our reservoirs is the set of client updates waiting to be sent to the server. If it reaches a certain threshold, it tells the file system it temporarily cannot submit updates, and the file system then temporarily disables completion of certain file system operations.

A reservoir's throttle need not stop a load flow directly into that reservoir; instead, it may stop load flow at an earlier point. For instance, in Fig. 4, reservoir G might throttle the flow from E to F, ultimately preventing flow to G. However, this can be problematic if the *throttling delay,* i.e., the time it takes load to flow from the throttled link to the reservoir applying the throttle, is significant. The problem arises because the reservoir decides to engage or disengage the throttle based on its own load, so these decisions are late by the throttling delay. When the throttle is engaged late, load has already flowed arbitrarily quickly across it, potentially causing a load spike; when the throttle is disengaged late, work is unnecessarily halted waiting for it. In particular, we do not throttle across Farsite's write-absorption delay because the system would oscillate too much. Initially, applied work would not be regulated at all; once the threshold was reached, work would be stopped for some arbitrarily long period while the system handled the load generated during the hour-long delay. This type of behavior is very unpleasant to users.

The throttling threshold is an important parameter of a throttle. To set it, first determine the reservoir's *capacity,* i.e., the most load it should ever contain. Set the threshold equal to this capacity minus the total capacity of all reservoirs between the throttling

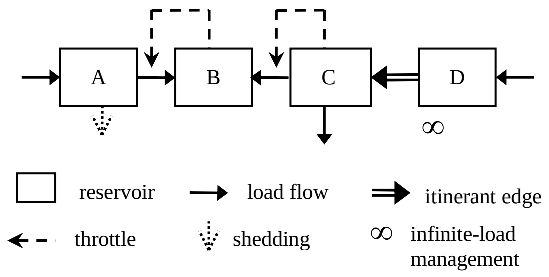

Fig. 5: Sample workflow graph with static overload prevention symbols

reservoir and the throttled link. The rationale is that even with the throttled link stopped, load may still flow from reservoirs after the link to the throttling reservoir, and it must be able to store that load in addition to what it already has. Note that since the threshold must be positive, this constrains how far back in a load flow one can throttle: one cannot throttle across reservoirs unless their total capacity is less than the throttling reservoir's capacity. Note also that in some cases a reservoir's capacity may change with time; for instance, the space available for Farsite files will change as unrelated applications use and release space.

Although we noted earlier that control theory is generally inappropriate for distributed systems like Farsite, throttling is actually a degenerate instance of a discrete, closed-loop control system. This implies that one should apply throttling only when there is minimal feedback delay, low forward gain, and deterministic responsiveness. These conditions all hold in the areas in Farsite where we use throttling.

Infinite-load management is the alternative to throttling, and refers to any technique that allows a reservoir to deal with an arbitrary amount of load. Examples of such techniques are *premature release*, *clown-car compression*, and *shedding*.

Premature release means that whenever the load on a reservoir reaches a certain limit, it sends load on its normal course out of the reservoir immediately rather than waiting for the normal time it would release load in this way. This ensures that its load never exceeds that limit. For instance, one reservoir in Farsite is the set of file system updates held in the file system kernel buffer. These updates are normally sent to user level periodically to avoid crossing the kernel-user boundary for every update. However, if the kernel buffer grows beyond a certain limit, it sends the updates to user level immediately, rather than when the periodic process next would; this means the reservoir cannot overload.

Premature release is infeasible if load cannot always be released faster than it is accrued, e.g., if releasing load in some cases requires a disk access. If releasing load requires remote processing, one cannot count on arbitrary speed from the network or a remote node, and thus cannot use premature release. Throttling can also preclude the use of premature release, as follows. Making a link throttled means the source of that link cannot always count on being able to dispatch load across it; thus, that source cannot use premature release to prevent overload.

Clown-car compression manages arbitrary load via compression that can fit an arbitrarily large load in a fixed amount of space (much as one can stuff in a clown car as many clowns as show up). In Farsite, we do this by mapping a set of work items onto a bounded-size set of objects, thus compressing arbitrary amounts of work into a finite amount of space. Specifically, the set of objects is the set of files in the file system, guaranteed by quotas and other mechanisms to be within a bounded size. We discuss our use of clown-car compression in more detail in subsection 4.3.

Shedding is simply discarding load. We have already seen how shedding can prevent dynamic overload; we can also prevent static overload by discarding items taking up space beyond a reservoir's capacity. As with shedding for dynamic resources, shedding for static resources also requires rectification processes, to correct deficiencies due to dropped load.

We use symbols in workflow graphs to indicate static overload prevention. A throttle is depicted by a dashed arrow from the throttling reservoir to the throttled flow. Infinite-load management is indicated by an ∞ symbol beneath the reservoir, unless it uses shedding. For shedding, we highlight the fact that items are dropped and a rectification process is needed by using a different symbol: a dotted arrow pointing downward from the shedding reservoir. Fig. 5 shows a sample workflow graph with such symbols.

As stated earlier, a reservoir cannot overload if either (1) it can deal with an arbitrary amount of load or (2) load in excess of its capacity can never reach it. From this, we deduce the following method for determining whether a reservoir in a workflow graph is immune to overload: it is so if either (1) it uses infinite-load management, or (2) every path along which load can enter the system and eventually reach that reservoir contains a link throttled by that reservoir. Furthermore, if no static overload exists in a set of subsystems, and together those subsystems cover all reservoirs in the system, then the system is immune to static overload.

3.4.3 Choosing overload prevention techniques

In a distributed system, a reservoir on one machine should not rely on another machine to prevent overload since it may be slow, connected by a slow link, disconnected entirely, or off. In Farsite, it may even be malicious. Thus, we consider it an important general principle that overload prevention in a distributed system must not use the network. Load *reduction* may use the network, but overload *prevention* must not. For this reason, as discussed earlier, we suggest that all subsystems depicted in a workflow graph be contained within a single machine.

Choosing which overload prevention technique to use for each reservoir is an art. Here, we offer some general principles from experience. Shedding should be a last resort, since it requires a rectification process and furthermore this process itself can add load. Throttling is good for matching workload creation rate to workload processing rate, but can only be used when throttling delay is short, there is a mechanism for halting load flow across the throttled link, and the throttling reservoir's capacity exceeds the total capacity of reservoirs between it and the throttled link. Premature release, clown-car compression, and infinite-load management techniques are generally preferable to throttling, since they do not block the flow of load in the system; however, such techniques are not always applicable. Premature release, for instance, is only possible when load can always be released faster than it accrues. Also, be aware that premature release can sometimes increase load; for example, ejecting a cached file prematurely can increase load on the system if the file is accessed again.

In some cases, using one of our overload prevention techniques necessitates complex system changes with far-reaching consequences. Examples of this will come up in the next section, where we apply our load-management techniques to Farsite.

4. Load management in Farsite

4.1 Disk space management

We begin with a discussion of Farsite's management of disk capacity, since the local disk is a critical limited resource used by several of its subsystems. The main technique Farsite uses for this is premature release, i.e., deleting cached files or file replicas earlier than they normally would be deleted. However, even after choosing the general technique to use, there are still many details of implementation to resolve. In this subsection, we describe the approach Farsite uses to decide which files to delete and when to delete them. Its goals are (1) prioritizing local over public usage of a machine's disk, and (2) making optimal use of the public portion of a machine's disk.

The first goal is readily achieved by dividing disk usage into five priority categories, as shown in Table 1. As described in section 2, Farsite's files are maintained on the local disk, and its object store uses databases also on the local disk, so Farsite essentially competes with the operating system and applications for use of this space. Since Farsite is able to operate, albeit inefficiently, with little local storage, highest priority is given to non-Farsite uses. Next-highest priority is maintaining a *free-space target* as a buffer and to reduce disk fragmentation. Next is the set of cached files, which are copied locally on first access and cached for several days so access by local applications is efficient. Next is the set of file replicas, since public use is given lower priority than local use. The lowest priority is cached files that have become *cold*, i.e., not accessed for several days. There is no reason to waste available space, so we keep these if there is room.

Table 1: Disk space-usage priorities

Priority	Category	Sorted by
1 (max)	non-Farsite use	n/a
2	free-space target	n/a
3	cached files (hot)	last-use time
4	file replicas	copy rank
5 (min)	cached files (cold)	last-use time

The second goal, making best global use of space for file replicas, is more complicated. We would like to have an equal number of replicas of each file in the system, so we employ a decentralized approximation of the following centralized solution: Keep track of the copy count of each file. Also, keep track of the global ideal copy count, equal to the number of storage bytes available divided by the number of file bytes. Wherever possible, eject replicas of files whose copy counts exceed the ideal copy count by 1.0 or more, and replace them with replicas of files whose copy counts are less than the ideal by 1.0 or more.

Since Farsite is decentralized, neither file hosts nor servers have enough information to compute an ideal copy count. Therefore, Farsite manages file replicas as follows. When a server tells a file host to make a copy of a file, it tells it the *rank* of the copy. The first copy of each file has rank 1; the second has rank 2; etc. Replicas on a machine are prioritized according to their rank, so file hosts will eject rank 4 copies before ejecting rank 3 ones, etc. Also, each server prioritizes creating new replicas according to the lowest non-existing rank of each file.

Clearly, this technique uses no global knowledge or central coordination. To see that it achieves our goal, consider a system with enough space for two copies of each file. File A has only one copy, and file B has three. Therefore, some file host has a rank 3 (or possibly greater) copy of file B. The server of file A will contact file hosts and try to make a rank 2 copy of file A. Once it contacts the file host with the rank 3 copy of file B, that file host will eject that file to make room for the new rank 2 copy of file A.

A complication is that there may be few file hosts willing to accept copies of a particular rank that a server would like to make. In this case, a server may have to make the request many times before finding a file host that will accept it. Therefore, Farsite uses squelching to manage the network load of copy-request messages, as follows. Each file host knows its *rank*

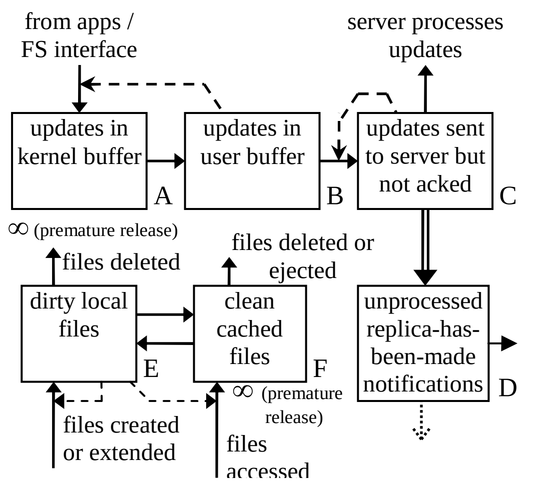

Fig. 6: Client subsystem

ceiling, the highest-rank copy it is willing to make. When a file host replies to a server's copy request, or when the file host's rank ceiling increases, it sends the server its rank ceiling value. Servers will not send copy requests for ranks that exceed a host's ceiling.

The file host maintains its rank ceiling as follows: Once per hour, if the disk free space is at least double the free-space target, the ceiling is incremented, because more space is available for replicas. It also increments the ceiling every 24 hours, because the passage of time makes cache files cold. If a file host ever rejects or ejects a copy, it sets the ceiling to one less than the copy's rank, since this rank is no longer within the acceptable range. On a new, empty machine, we do not initialize the ceiling to a large value just because there is plenty of space, since this would allow the host to accept high-rank copies which would likely be ejected as the disk fills up. To prevent this waste of effort, we initialize the ceiling to zero and allow it to climb gradually.

We extend the above technique by using probabilistic squelching to avoid hotspots. If a small number of file hosts have higher rank ceilings than most other file hosts, they may be disproportionately besieged by copy requests. For example, if many servers want to make rank 4 copies and only a few file hosts have rank ceilings above 3, these servers will send all their copy requests to these few file hosts. In large installations, this problem can be significant.

To address this issue, servers probabilistically limit their requests to match the rank-acceptance criteria of file hosts, as follows. As described above, servers know the reported rank ceilings of all file hosts on which they store files. Each server maintains a rank ceiling of its own, which it sets every 10 minutes to the rank ceiling of a randomly selected file host. This way, the rank ceiling follows the distribution of the file hosts' rank ceilings. Each server restricts its copy requests to copy ranks no higher than its current rank ceiling. Thus, if a small number of file hosts have high ceiling values, a proportionally small number of servers will make high-rank requests at any given time.

Incidentally, the above algorithm requires periodic communication between every file host and every server, which will not scale to large systems. The following algorithm, which we have not yet implemented, solves this problem: File hosts report their rank ceilings only when asked by servers. Every 10 minutes, each server asks a single random host for its rank ceiling and sets its own ceiling to that value.

4.2 Client subsystem

The first Farsite subsystem we discuss is the client subsystem, which exports a file system interface to applications on the local machine so they can perform operations on the Farsite namespace. Fig. 6 shows the workflow graph for this subsystem, which we will now describe in detail. To distinguish the reservoirs labeled A, B, C, etc. in Fig. 6 from reservoirs in other figures, we refer to them in the text as 6A, 6B, 6C, etc.

The client subsystem manages metadata using reservoirs 6A – 6D, as follows. Applications using the file system interface generate file system updates in the kernel driver. These are stored in reservoir 6A, the set of updates held in the kernel buffer. Periodically, these updates are sent across the kernel-user boundary to reservoir 6B, the user-level update set, since most of the client subsystem runs in user level. Periodically, these updates are sent to the directory server. While the directory server is processing the updates, they are held locally in reservoir 6C, the list of updates waiting to be acknowledged as received by the directory server. Eventually, the server processes the updates and notifies the client that a replica was made. These notifications are held in reservoir 6D until they can be processed; processing involves persistently marking the local file clean, i.e., no longer the only valid copy of the file in the system.

The client subsystem manages file content using reservoirs 6E and 6F, as follows. When an application creates a new Farsite file, the client subsystem stores its contents in a new local NTFS file. This file is dirty, i.e., it cannot be deleted until a replica is made elsewhere. Creating such a file thus adds load to reservoir 6E, the set of dirty local files; load also enters the system into this reservoir when a file is extended. When the server notifies the client that an updated file has been replicated, it becomes clean and moves to reservoir 6F, the set of clean cached files. Load also enters the system into this reservoir when a client accesses a non-local file, causing it to become cached locally. Files can leave this reservoir and enter reservoir 6E if a local application modifies the file.

The client subsystem has many mechanisms for load reduction. Files are cached upon access so they need not be fetched every time they are subsequently accessed. Updates gathered in the user buffer are not immediately passed on so they can be combined and compressed. Updates generated in the kernel are buffered to reduce the frequency of expensive kernel-user crossings. These mechanisms reduce load, but do not guarantee prevention of static overload, so other techniques are used for this, as follows.

Reservoir 6C, which holds requests sent to the directory server but not yet acknowledged, cannot use premature release since it cannot release load until a remote node processes it. So, it throttles load coming in from reservoir 6B, the user-level update buffer. This, in turn, prevents reservoir 6B from using premature release, so it uses throttling instead. When it gets too full, it throttles the driver by signaling it to temporarily stop completing application I/O requests. Together, these throttles limit client workload generation to the rate of server workload acceptance (albeit not the rate of workload processing). This throttle is engaged only when the applied load is very high, and its throttling delay is only a few seconds, so it will not create the unpleasant user experience we would get by using feedback control across the entire update/replication path, which takes at least an hour.

Reservoir 6A, the kernel update buffer, is amenable to premature release, so we use it. Whenever the buffer reaches a certain size, it is immediately sent across the kernel-user boundary instead of waiting for the next periodic send.

Reservoir 6D stores notifications from the server that files are clean. It cannot use premature release since it performs a disk operation to release load. Also, it cannot throttle any earlier flow in the subsystem without incurring a high throttling delay including all remote processing along the itinerant edge. Thus, we reluctantly use shedding: it drops notifications when its load exceeds a limit. This necessitates a rectification process: the client periodically contacts the server and queries file clean/dirty status. Without this process, some files might never be marked clean and then could never be deleted.

Reservoir 6F, the set of clean cached files, uses premature release, as described in subsection 4.1: if space needs to be made available, cold cache files are released prematurely, and if the disk space shortage is severe, hot cache files are also released. If there is so little disk space that there is no room for additional files in the cache, Farsite will begin giving applications out-of-space errors whenever they perform file system operations that would increase the size of the local cache. Essentially, this is a throttle from reservoir 6E, the set of dirty files, to all paths that allow file cache load to enter the system.

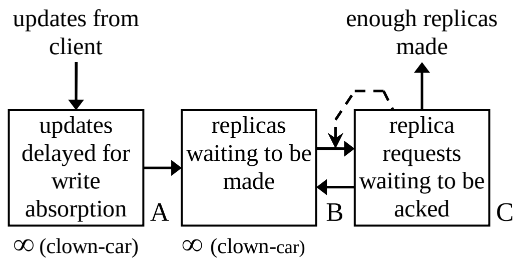
updates from client
enough replicas made
updates delayed for write absorption
A
replicas waiting to be made
B
replica requests waiting to be acked
C
∞ (clown-car)
∞ (clown-car)

Fig. 7: Directory server subsystem

4.3 Directory server subsystem

Next, we discuss the directory server subsystem, which is responsible for managing file system metadata, including the location of file replicas on file hosts. Fig. 7 shows the workflow graph for this subsystem, which we now describe in detail.

The directory server subsystem receives file system updates from clients, which it stores in reservoir 7A, the set of updates waiting for the write-absorption delay before they will be processed. Once this delay passes, the updates become work items reflecting the need to make initial replicas of the updated files. If the server cannot make replica requests immediately due to dynamic load constraints, it queues them in reservoir 7B, the set of such requests waiting to be sent. As described earlier, replica requests with a lower rank have priority over requests with a higher rank, so reservoir 7B is actually a multi-queue, one queue for each rank. When resources are available to send replication requests, they are sent, and they are placed in reservoir 7C, the set of replication requests sent but not acknowledged as completed by the remote file host. When a request is acknowledged as completed, the directory server decides whether another replica needs to be made, and if so adds load back to reservoir 7B. If no further replicas need to be made, the load is removed from the system.

The main reduction the directory server performs is compression in reservoir 7A, delaying updates so they can be combined with later updates for more efficient processing. The rest of this subsection discusses our techniques for preventing static overload.

Reservoir 7C cannot use premature release, since its outgoing link involves processing by a remote machine. Thus, it uses throttling: once a threshold number of replica requests outstanding is reached, requests must wait in reservoir 7B to be dispatched.

Reservoirs 7A and 7B use our novel clown-car compression technique to avoid static overload. There are three factors that allow us to use clown-car compression here: (1) the semantics of file update, (2) the dynamic partitioning of workload among directory server groups, and (3) quotas that limit total file-system space usage.

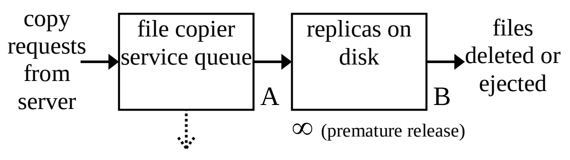

Fig. 8: File host subsystem

The relevant semantics of file update is that when successive updates are made to a single file's metadata (including its content hash), all updates except the last are obsolete. Thus, all metadata updates except the last can be discarded, and reservoirs 7A and 7B need to hold at most one update per file in the system. Furthermore, each server only receives updates to files that it manages, so the maximum size of these two reservoirs is proportional to the count of files managed by the server group. The actual clown-car compression technique is to maintain these reservoirs merely as annotations in the server's database, such that an update to a managed file is "in the reservoir" if a particular field of the file's database record has a value indicating so. There is thus no overload as long as the server can support the files it supposed to manage.

To ensure this latter condition, the Farsite architecture includes a mechanism (designed but not yet implemented) for dynamically partitioning file management among all server groups in the system, so each group's load will not grow beyond what it can manage. This mechanism prevents overloading of any particular server group as long as the entire system can support the count of files stored in it by clients. This condition, in turn, is ensured by quotas. The semantics of a file system include a limitation on the amount of available space: if a client attempts to store more files that the system is willing to support, the client's file creation requests fail with an out-of-space error.

4.4 File host subsystem

Next, we discuss the file host subsystem, responsible for storing file replicas and making them available to clients who need to access them. Fig. 8 shows the workflow graph, which we now discuss.

The file host subsystem receives requests from directory servers to make file replicas, and services them as soon as dynamic load constraints allow. While they wait to be serviced, they sit in reservoir 8A, the queue of replication requests waiting to be serviced. We found it important to make this queue explicit, since if we simply issued every request as soon as it arrived and relied on the scheduler to implicitly queue requests for which CPU or disk bandwidth was unavailable, the set of threads in the system grew without bound. When a replica is made, a file is added to reservoir 8B, the set of replicas on the local disk.

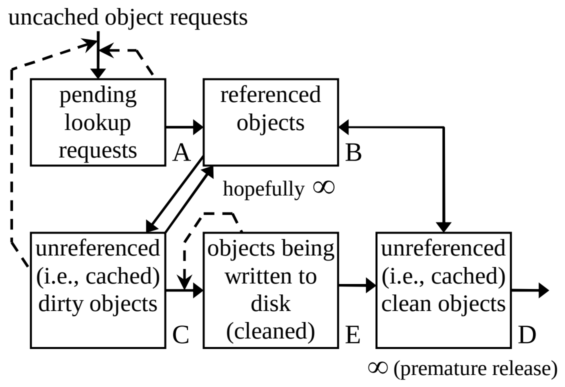
uncached object requests
pending lookup requests
A
referenced objects
B
hopefully ∞
unreferenced (i.e., cached) dirty objects
C
objects being written to disk (cleaned)
E
unreferenced (i.e., cached) clean objects
D
∞ (premature release)

Fig. 9: Object pager subsystem

Reservoir 8A cannot use premature release since it relies on disk bandwidth, and it cannot use throttling since load comes from another machine. Thus, we use shedding to prevent static overload: the file host drops any requests that arrive when 60 are outstanding. This requires a rectification process, which is that the file host tells the server to have another file host make the replica, and even if this message is lost the server eventually times out the request and does this anyway. Note that the server informs the file host of its timeout value when it sends the original replication request; if the file host finds that the timeout has expired when it dequeues the request and is about to initiate it, it drops the request since executing it would be pointless.

Reservoir 8B, the set of replicas stored on the local disk, uses premature release. We discussed such disk space management earlier in subsection 4.1.

4.5 Object pager subsystem

The last Farsite subsystem we discuss is the object pager subsystem, responsible for caching persistent objects in memory and paging them to and from disk. The pager manages memory consumed by directory data and meta-level file data; memory consumed by file contents is managed by the operating system's cache manager [16] and so is out of our control. A typical pager prevents overload by slowing down the rate at which objects are written [11, pp. 300–303], but, as we discuss below, our pager achieves this goal by throttling the object lookup rate. Fig. 9 shows the workflow graph for the object pager.

When system code requests an object that is not cached, a lookup request is placed in reservoir 9A, the set of pending lookup requests. When the request completes, the fetched object X is kept in memory in reservoir 9B as long as X remains referenced. When X is no longer referenced, it is not immediately discarded; instead, it is retained in the cache in the hope of satisfying imminent requests. If X is dirty, it cannot be discarded anyway until it is written out to disk and thereby cleaned. Cached dirty objects are held in reservoir 9C. Cached dirty objects are written out, i.e., cleaned, after having remained unreferenced for three minutes. While an object is being cleaned, it is placed in reservoir 9E; it is moved to reservoir 9D, the set of cached clean objects, after cleaning is complete. Eventually, if an object in reservoir 9D has not been accessed for a while, it is discarded.

When system code requests a cached object, it is moved from reservoir 9C or 9D to reservoir 9B, the set of referenced objects. Note that no static load is actually added to the subsystem on such a cache hit.

We now discuss the static overload prevention techniques the object pager uses. Reservoirs 9A and 9E prevent overload by simply throttling their inputs.

We also use throttling for reservoir 9C, but, unlike most pagers, we do not throttle this reservoir's incoming link directly. Instead of blocking operations that mark an object dirty when the count of dirty unreferenced objects exceeds a threshold, we instead block object lookups in this case, for the following reason. We use an event-driven programming model that frequently relies on the atomicity of non-blocking calls [2]. We already have to mark object lookups as non-atomic, since they may involve a disk operation on a cache miss. To avoid also marking set-object-dirty calls as blocking, we do not throttle these calls. Note that stopping object lookups while waiting for the cleaner to clean an object cannot cause deadlock, because the cleaner simply writes objects to disk and thus never requires object lookups.

The throttle applied by reservoir 9C is not a typical throttle in that it does not simply stop flow across the link as long as its size exceeds a threshold. Our design accounts for the fact that we expect Farsite to sometimes be more constrained in CPU bandwidth than disk bandwidth, and sometimes the other way around. This is because Farsite runs on a collection of heterogeneous machines with heterogeneous CPU and disk bandwidths, and also because those machines are running other applications, which can dynamically cause Farsite to become CPU-limited or disk-limited. When Farsite is disk-limited, it will tend to dirty objects faster than it cleans them, so in this case we want to throttle object lookup so that it is rate-matched with the cleaner. However, when Farsite is CPU-limited, it will tend to clean objects faster than it dirties them and such rate-matching is not needed. Thus, although we want to apply hysteresis before switching to the rate-matched state, we want to promptly switch back to the non-rate-matched state when we can. We now describe the algorithm we use to achieve this goal.

When the count of dirty unreferenced objects exceeds the *dirty-object threshold* (2000 objects), lookups are disallowed and the cleaner starts cleaning dirty unreferenced objects even before their three-minute timers have expired. When the count of dirty unreferenced objects is between one and the dirty-object threshold, the lookup dequeues a single waiting request each time the cleaner cleans an object, i.e., it rate-matches lookup to cleaning. When the cleaner removes the last object from the cleaner queue, the system dequeues a batch of waiting lookup requests, where the batch size is limited to the dirty-object threshold. This last case is the transition from the rate-matched to the non-rate-matched state.

Static overload in reservoir 9D is prevented with premature release: if its size reaches a threshold, the least-recently-referenced object is prematurely discarded. Note that this is not shedding since the normal way to release load from reservoir 9D is to discard it. Thus, it requires no rectification process.

The workflow graph indicates a potential static overload in reservoir 9B, the set of referenced objects. We cannot throttle the lookup queue as we do for reservoir 9C, because this throttle would get passed back to the code making uncached object requests. If this code is blocked, it will not have an opportunity to release its references, resulting in a deadlock. We cannot use premature release since there is no mechanism to force code to release a reference. Also, we cannot use shedding because this would break code that relies on referenced objects remaining in memory until explicitly dereferenced.

Therefore, our only recourse to prevent overload is to ensure that the load placed in this reservoir never exceeds a reasonable bound via careful code design. We limit the size of work batches; we process work in a FIFO fashion; and we cache unreferenced objects in the object pager, which is sensitive to its memory usage, rather than caching these objects in an ad-hoc manner (which could result in holding unneeded references in the system code). It is somewhat unsatisfying to rely on this approach since we cannot prove that this reservoir is completely free of possible static overload. However, this is no different than the way programmers commonly write code to keep memory footprints low.

5. Experimental results

In this section, we describe the results of experiments we performed on Farsite to demonstrate the efficacy of our load management techniques.

5.1 Effect of throttling

The first experiment we run tests the efficacy of the throttle we use to limit the set of dirty unreferenced objects, i.e., reservoir C in Fig. 9. We ran Farsite twice with the same set of four machines and the same client workload, once with the throttle enabled and once with it disabled. The workload was simply a breadth-first creation of a tree of directories, each directory of which

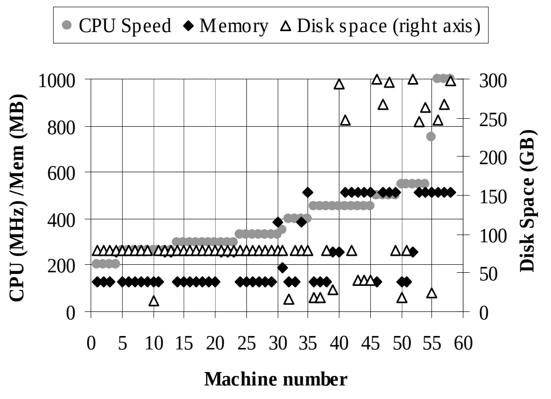

Fig. 10: Characteristics of 58 machines in experiment; disk space is space available to Farsite

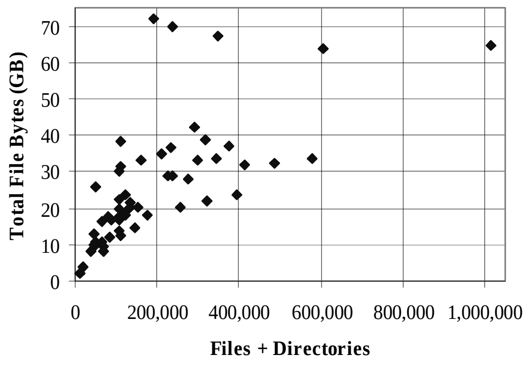

Fig. 11: Sizes of 58 file systems for applied load

had 100 subdirectories. Fig. 1, presented in the introduction, shows the count of directories created by all four machines, and the count of dirty unreferenced objects on a particular machine. With the throttle disabled, the count of dirty unreferenced objects kept increasing, until memory was exhausted and Farsite was unable to proceed due to a memory allocation failure. With the throttle enabled, the count of dirty unreferenced objects eventually stabilized thanks to the throttle on object lookups. This throttle slowed down file system operation, so directories were created less quickly, but since it did not crash it was eventually able to create more directories than without the throttle.

5.2 Long-term stability

To demonstrate the effectiveness of our load-management techniques, we ran a stressful experiment on a 58-machine Farsite system. We used machines several years out of date partly because these were what we could scrounge up but also because using slow machines increases the relative load on the system. Fig. 10 illustrates the characteristics of the machines and shows that they are quite heterogeneous. A system of this small size does not require decentralized servers, but to test our techniques, we configured all machines identically. Thus, we had 58 clients and 58 server groups. Since each server is a replicated state machine running on four machines, each machine participated in four directory server groups.

We drove Farsite with an enormous offered load by copying images of whole file systems from normal users' desktop machines onto the Farsite clients. The copied file systems were taken from machines at Microsoft, and their high-level size characteristics are graphed in Fig. 11. Altogether, we created roughly 1.5 TB of file data in 10.5 million files. Since we made 5.25 TB total available to Farsite, there was enough room for a single local cached copy, plus an average of 2.5 remote replicas, of each file.

This experiment places tremendous load on the system; the mere fact that Farsite can make it through this workload without crashing is a significant testament to our load-management techniques. The heterogeneity of machines presents many opportunities for slow machines to be swamped by load from faster machines, and for machines with large disks to be overwhelmed by requests to create replicas. Furthermore, the heterogeneity of the traces made it difficult to ensure each file received its fair share of global public storage.

At time 0, we began the experiment by starting the file-system copies on all clients at once. The copy operations completed in a mean of 8 hours. The range was 16 minutes to 100 hours with a median of 4 hours and a standard deviation of 15 hours. The large skew is due both to the nearly three-order-of-magnitude difference in input size and the speed difference of the machines in the experiment. This extremely skewed distribution presents a challenge to Farsite, because the small systems had an opportunity to make high rank replicas when free disk space was plentiful, which then had to be removed as the larger, slower ones completed. We ran the experiment for a total of 100 hours, at which point we stopped to submit this paper.

Farsite replicates files much more slowly than it creates them for several reasons. First, disk bandwidth is greater than network bandwidth. Second, the application that created the load did not actually write all of the bytes (it just set the end of file), but this optimization was not used for network copies. Third, because the file host component of Farsite is designed to be nice to the owner of the workstation on which it runs, it limits file replication operations to at most 80% of real time. Our lab has an over-provisioned switched Ethernet infrastructure, so network congestion was unlikely to have had much effect.

Fig. 12 shows the evolution of the reservoirs on a single representative directory server. Tier *n* refers to requests in the replication multi-queue to make rank *n* replicas of a file. For efficiency, the implementation

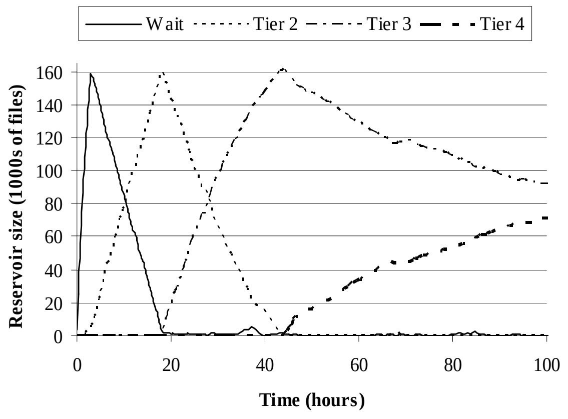

Fig. 12: Evolution of one server's reservoirs; tier *n* contains requests to make rank-*n* file replicas

makes no distinction between (a) files waiting to have an initial replica made because they are waiting for an hour and (b) files waiting to have an initial replica made because the replication multi-queue output link is throttled; thus, the line depicting the number of files waiting for their first replica includes both these types of files. The main point of this graph is that the system works as we intended. The write-absorption wait queue grows steadily for the first hour as file updates accumulate. An hour later, when rank 1 replicas begin to be made, work items begin moving from the wait queue to the second tier of the replication multi-queue. At hour 3, the copy completes and files stop arriving in the wait queue. When all rank 1 replicas are made at about hour 18, rank 2 file replicas begin being made, so the second tier shrinks and the third tier grows. When a directory server sees several consecutive failures to replicate a file, it considers it possible that the failures may be at the data source(s) rather than the destinations, so it returns the file to the wait queue; this is the source of the small number of files that reappear from time to time in that queue (e.g., around hours 37 and 80–86). As file hosts fill up with low-rank replicas, they will eject higher-rank replicas to make space, and these will return to the replication multi-queue. This is (barely) visible on the tier 2 queue around hour 67.

As rank 3 copies are made for files in the third tier, disks begin to fill up, so probabilistic squelching slows down the copying rate. Eventually, rank 3 copies begin to be ejected from disks, causing servers to make new requests on the third tier, and so the third tier grows occasionally, such as at hour 72.

Fig. 13 shows files' copy count versus time. In our experimental configuration, there is room for an average of 2.5 replicas per file, so eventually all files should have at least two copies, half of all files should have three copies, and no files should have more than three. Unfortunately, our experiment stopped before we reached steady state, but the trend toward the

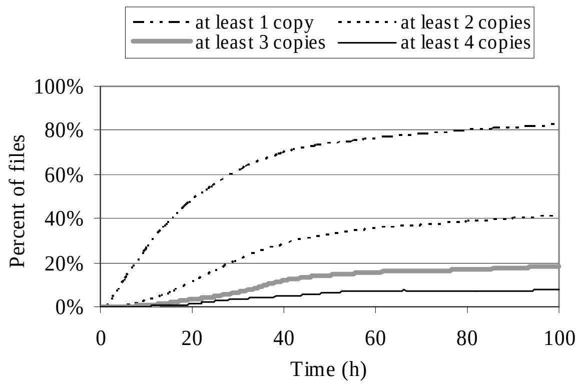

Fig. 13: File copy counts over time

correct values is visible near the right edge of the graph: The percentage of files with at least one copy (beyond the one cached on the writing machine) is rising slowly toward 100%. The percentage of files with at least two copies is rising faster. The percentages with at least three or four copies have leveled off, and given more time the four-copy line would drop as rank 4 replicas were ejected to make space for rank 2 replicas of the remaining files that have only one copy.

6. Related work

Other researchers have designed and built decentralized file systems [3, 18], but to the best of our knowledge, Farsite is the first to address the problem of load management.

Several researchers have applied feedback control to managing load in software systems. Some [17, 23] have proposed a general architecture of exposing the queues between components to enable an external controller to adjust resource allocations; others [24] advocate more general interfaces. Other systems researchers [6, 7, 10, 21, 22] have employed adaptive control, wherein long-term data is gathered to refine a system model that is in turn used to tune the parameters of a control system. Neither feedback nor adaptive control is responsive enough to overcome our long loop delay without painfully aggressive throttling.

Farsite's distributed disk prioritization scheme gives highest priority to local applications and then attempts to share remaining capacity in an egalitarian fashion. This has been done before in other contexts, including global memory [9], content distribution [19], and file-system caching [8].

Throttling pagers is a standard technique in operating systems [11, 15]. Farsite's pager is new only in that it has a non-blocking write path and a mechanism to switch between rate-matching and non-rate matching modes.

7. Summary and conclusions

Farsite is a decentralized file system intended to scale to tens of thousands of machines. Yet, even in small installations, machines can be overwhelmed by unmanaged load from other machines, causing backups, overflows, timeouts, and depleted resources. For this reason, we developed methods for load management for distributed systems in general, and applied them to Farsite.

Managing load using feedback control theory is impractical because of non-locality, long delay, and high forward gain; and using open-loop control is infeasible because of machines' physical distribution, heterogeneity, and unpredictability. Thus, we propose an alternate method for managing load in Farsite and other similar distributed systems. (1) Reduce load to the extent possible using techniques such as caching, compression, and squelching; (2) make clear where implicit queuing can cause dynamic load to be converted to static load; and (3) prevent overload of static resources.

To prevent static overload, each reservoir where static load can accumulate must use either throttling or infinite-load management. Our techniques for infinite-load management include premature release, clown-car compression, and shedding. Clown-car compression is a novel extension of standard compression techniques that, by mapping an arbitrarily large set of operations onto a fixed-size set, enables the system to handle an arbitrary applied load.

We developed the method of workflow graphs to help system designers enumerate the potential sources of overload and ensure each is managed appropriately. A workflow graph of a subsystem shows the reservoirs in that subsystem, the load flows to and from those reservoirs, and the load management techniques used on each reservoir.

Some techniques for load management require complex system design. For instance, in Farsite we manage disk space in part using a novel mechanism for achieving approximately even file replication without central coordination or global knowledge.

Two experiments demonstrate the efficacy of our load-management techniques. In one, we show that one of our throttles makes the difference between crashing and running stably. In the other, a long-running 58-machine experiment, we demonstrate that the system is able to run without overloading despite an enormous offered load.

References

[1] A. Adya, W. Bolosky, M. Castro, R. Chaiken, G. Cermak, J. Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer. "FARSITE: Federated, available, and reliable storage for an incompletely trusted environment", *5th OSDI*, Dec 2002.

[2] A. Adya, J. Howell, M. Theimer, W. Bolosky, and J. Douceur. "Cooperative task management without manual stack management", *USENIX Technical Conference*, Jun 2002.

[3] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. "Serverless network file systems", *15th SOSP*, Dec 1995.

[4] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. "Measurements of a distributed file system", *13th SOSP*, Oct 1991.

[5] M. Castro and B. Liskov. "Practical Byzantine fault tolerance", *3rd OSDI*, Feb 1999.

[6] J. S. Chase, D. C. Anderson, P. N. Thakar, A M. Vahdat, and R. P. Doyle, "Managing energy and server resources in hosting centers", *18th SOSP*, Oct 2001.

[7] C. M. Chen and N. Roussopoulos. "Adaptive database buffer allocation using query feedback", *19th VLDB*, Aug 1993.

[8] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson. "Cooperative caching: using remote client memory to improve file system performance." *1st OSDI*, Nov 1994.

[9] M. J. Feeley, W. E. Morgan, F. H. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. "Implementing global memory management in a workstation cluster", *15th SOSP*, Dec 1995.

[10] J. N. Matthews, D. Roselli, A. M. Costello, R. Y. Wang, and T. E. Anderson. "Improving the performance of log-structured file systems with adaptive methods." *16th SOSP*, Oct 1997.

[11] R. Nagar. *Windows NT File System Internals*. O'Reilly, 1997.

[12] K. Ogata. *Modern Control Engineering*, 3rd Ed. Prentice Hall, 1997.

[13] D. Saha, S. Mukherjee, and S. Tripathi. "Multi-rate traffic shaping and end-to-end performance guarantees in ATM networks." *ICNP*, Oct 1994.

[14] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. "Coda: A highly available file system for a distributed workstation environment", *IEEE Trans Computers* 39(4), 1990.

[15] A. Silberschatz and P. B. Galvin. *Operating System Concepts*, 4th ed. Addison-Wesley, 1994.

[16] D. A. Solomon and M. Russinovich. *Inside Windows 2000*, 3rd Ed. Microsoft Press, 2000.

[17] D.C. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole. "A feedback-driven proportion allocator for real-rate scheduling", *3rd OSDI*, Feb 1999.

[18] C. A. Thekkath, T. Mann, and E. K. Lee. "Frangipani: a scalable distributed file system", *16th SOSP*, Oct 1997.

[19] D. Villela and D. Rubenstein. "A queuing analysis of server sharing collectives for content distribution", *Columbia Univ. Electrical Engineering Tech Report* EE2002-04-121, 2002.

[20] W. Vogels. "File system usage in Windows NT 4.0", *17th SOSP*, Dec 1999.

[21] C. A. Waldspurger. "Memory resource management in VMware ESX Server." *5th OSDI*, Dec 2002.

[22] G. Weikum, C. Hasse, A. Mönkeberg, and P. Zabback. "The Comfort automatic tuning project", *Information Systems* 19(5), pp. 381-432, 1994.

[23] M. Welsh, D. Culler, and E. Brewer. "SEDA: An architecture for well-conditioned, scalable Internet services", *18th SOSP*, Oct 2001.

[24] R. Zhang, C. Lu, T. F. Abdelzaher, and J. A. Stankovic. "ControlWare: A middleware architecture for feedback control of software performance", *ICDCS*, Jul 2002.