A Cost-Effective Distributed File Service with QoS Guarantees

Kien Le, Ricardo Bianchini, and Thu D. Nguyen

Department of Computer Science, Rutgers University
{lekien, ricardob, tdnguyen}@cs.rutgers.edu

Abstract. Large-scale, value-added Internet services composed of independent cooperating or competing services will soon become common place. Several groups have addressed the performance, communication, discovery, and description aspects of these services. However, little work has been done on effectively composing paid services and the quality-of-service (QoS) guarantees that they provide. We address these issues in the context of distributed file storage in this paper. In particular, we propose, implement, and evaluate a cost-effective, QoS-aware distributed file service comprising a front-end file service and back-end (third-party) storage services. Our front-end service uses mathematical modeling and optimization to provide performance and availability guarantees at low cost by carefully orchestrating the accesses to the back-end services. Experimental results from our prototype implementation validate our modeling and optimization. We conclude that our approach for providing QoS at low cost should be useful to future composite Internet services.

Keywords: Distributed storage, quality of service, cost optimization.

1 Introduction

Large-scale, value-added Internet services composed of independent cooperating or competing services will soon become common place. We refer to these services as *composite services*. Two technology trends suggest this new class of services: the progress toward ubiquitous Internet connectivity even from devices with limited resources, and the increasing adoption of service communication, discovery, and description standards, such as the Simple Object Access Protocol (SOAP), the Universal Description, Discovery and Integration Service (UDDI), and the Web Service Definition Language (WSDL). Together, these trends are forcing functionality and data into the network infrastructure in the form of remotely accessible services.

Composite services promise anytime, anywhere access to powerful services and vast data sets. A composite service may use constituent services that provide complementary functionality or data. For example, a composite stock service might use a service that provides stock quotes in some currency and a service that translates an amount of money (e.g., a stock quote) in one currency into another. In contrast, a composite service may use services that provide the same functionality or data. For example, a composite job-scheduler service might use multiple job-execution services. Regardless of type, we expect that composite services and their constituent services will provide service-level agreements (SLAs) for a monetary charge.

In terms of structure, composite services are organized into a front-end service and multiple independent back-end services. The front-end service monitors and aggregates the back-end services, whereas the back-end services communicate with the front-end service but not with each other. In the above examples, the stock and job-scheduler services are called front-end services, whereas the stock-quote, currency-exchange, and job-execution services are called the back-end services.

For several years, researchers have been studying composite services in one form or another in the CORBA, Grid, and Web Service communities. These works have mostly focused on the performance, communication protocols, discovery mechanisms, and description of these composite services. Little work has been done on effectively composing paid services and the quality-of-service (QoS) guarantees that they provide.

In this paper, we address these issues in the context of distributed file storage. In particular, we propose, implement, and evaluate a cost-effective, QoS-aware composite file service comprising a front-end file service and back-end (third-party) storage services. The composite file service is intended to support soft real-time applications that involve large data files, such as the visualization of large-scale scientific data (e.g., [1]). For these applications, it is important to guarantee that data files will be available a large fraction of the time, and that a large percentage of file accesses will be served within a certain amount of time.

The composite service provides "soft" availability and performance guarantees, i.e. in extreme scenarios, such as a network partition separating front-end and back-end services, the guarantees may be violated. When these violations occur, the service compensates users for the violations.

Our front-end service allows users to choose the performance and availability guarantees that they desire on a per-file basis. Based on the chosen availability guarantee, the front-end service replicates the file across the back-end services. Based on both chosen guarantees, the back-end services' behaviors, and their SLAs, the front-end service intelligently distributes the requests across the back-end services to provide the chosen guarantees at low cost.

The front-end service uses mathematical modeling and optimization to carefully orchestrate the accesses to the back-end services. More specifically, the front-end service combines two algorithms: Base and OptWait. Base is reminiscent of traditional job scheduling. It sends each request to one of the back-end services that replicate the corresponding file, according to a ratio determined by the mathematical machinery to meet the file's performance guarantees while minimizing access cost. In contrast, OptWait is more sophisticated. It may actually send each request to multiple back-end services in turn (starting with the cheaper ones) until the request is satisfied. The amount of time it waits for each service to respond is determined mathematically and depends on the probability that the service will return a reply during that time and on the file's performance guarantee. Because we can mathematically decide on the best algorithm, our composite service picks the best algorithm for each file.

Because our initial focus (and the focus of this paper) is on the request-distribution aspect of our work, we have implemented a prototype of our composite service with a single front-end file server. The server implements the NFS protocol and executes our mathematical machinery. It communicates with client machines using a standard NFS protocol over UDP, whereas it communicates with back-end services using XML over HTTP. Several Internet storage services, e.g. Amazon.com's S3 [2], could implement the back-end services. However, for greater control of our experiments, we implemented our own back-end services, which provide data blocks named by absolute number.

Experimental results from our prototype implementation validate our modeling and optimization approach. Our analysis of the algorithms studies several different parameters, including the performance and availability guarantees, and the characteristics and behavior of the back-end services. Our most important results show that our composite service is successful at providing the guarantees that it promises. The results also show that, independently, Base and OptWait provide the lowest cost in different parts of the parameter space, whereas our combined system always produces the lowest cost.

2 Related Work

Our work builds upon previous research on service composition, QoS-aware resource management, and distributed file and storage systems.

Service composition. This has been an important research topic in the Web Services community, e.g. [3,4]. These works typically consider the QoS-aware composition of services from constituent services that provide complementary computational functionality. For this reason, they do not consider request-distribution policies across the services. Our work differs from these efforts as we study request-distribution policies that are both QoS- and cost-aware, across functionally-equivalent constituent services.

QoS-aware resource management. A large body of work has been done on this topic, especially in the context of networks, server clusters, and grid environments, e.g. [5,6,7]. These works consider resource allocation, provisioning, reservation, and negotiation, as well as admission-control policies in guaranteeing QoS (and sometimes optimizing costs) for the systems' users.

The extent of the performance guarantees provided by our composite service is limited to the front-end and back-end services' behaviors, as well as the communication between front-end and back-end services; the composite service cannot provide performance guarantees about the communication between clients and the front-end service. All other works on server-side QoS guarantees have this same limitation. We envision combining our QoS guarantees with those of future networks to completely eliminate this limitation. Nevertheless, an easy approach to tackle this problem with current network technology is to place front-end servers on the same local-area network as clients. In this approach, the front-end server could be an appliance, like today's load balancing or storage appliances.

Although we can benefit from previous QoS works in managing the resources of our front-end service and by leveraging network QoS, this paper focuses on request distribution across the black-box back-end services, which allow us no control over their resource allocation. In fact, the back-end services can themselves be distributed. The only information about them that we rely upon is their SLAs.

Distributed file and storage systems. Most of the research in distributed file and storage systems has been focused on cluster or local-area network environments, in which resources are dedicated to the system and owned by the same administrative entity, e.g. [8,9,10]. Due to their low communication latencies, these systems are amenable to small data and meta-data transfers. In contrast, peer-to-peer file and storage systems have also become prominent in recent years, e.g. [11,12,13]. These works have typically concentrated on achieving extreme performance scalability and availability in the presence of high churn in the online membership of constituent nodes.

Although our composite file service can be seen as a peer-to-peer system in the strictest sense, it lacks a few defining characteristics of previous systems, such as peers that often become unavailable. Further, we are interested in pushing the boundaries of traditional distributed file systems, such as NFS, by using them across the wide area. Two papers have addressed the effect of high latencies on file system traffic [14,15], but neither of them considered QoS or costs. We expect Internet block-storage services to become widespread in the future, as protocols such as iSCSI become more popular.

Summary of contributions. As far as we know, this paper is unique in a few respects. First, our work seems to be the first to focus on cost- and QoS-aware request distribution across third-party services. Second, our OptWait request-distribution algorithm departs from traditional scheduling policies by potentially assigning a request to multiple back-end services in turn. Finally, our approach of considering the entire set of recent response times from each back-end service, rather than using a single metric such as the recent average response time or the maximum recent response time, in mathematically determining request distributions is also novel.

3 Our Composite File Service

In this section, we discuss the basic principles behind our composite file service, our request-distribution algorithms, and our current implementation.

3.1 Basic Principles

Overview. As already mentioned, our composite file service comprises a front-end file service and a number of back-end block-storage services. The front-end service translates the file system API, e.g. create, read, write, unlink, into block accesses that are forwarded to one or more back-end services. The front-end service composes the user-requested guarantees from the back-end services at low cost. In fact, even if a single storage service could provide the required guarantees directly to the user (who could use a local file system and iSCSI, for example, bypassing the front-end service), the composite file service could still provide them for a lower cost, e.g. by forwarding some of the requests to a back-end service with lower cost per access whenever possible.

In our design, the front-end service is implemented by a number of distributed servers for both performance and availability. Each user mounts the file system through one of the front-end servers, which is chosen using a separate Web interface listing all available front-end servers and their geographical locations. The same file system can be mounted concurrently at different front-end servers. However, the front-end service provides no consistency guarantees when read-write and write-write file sharing is not done on the same front-end server. When the same front-end server is used, strong consistency is

Table 1. Notation and definitions

Notation	**Definition**
A_{front}	Availability of the front-end service
A_i	Availability guarantee provided by back-end service i
(P_i, L_i)	Performance guarantee provided by back-end service i: When service is available, P_i% of requests should be served in L_i time
(c_i^r, c_i^w, c_i^s)	Read, write, and storage costs of back-end service i
A_f	Availability requested by the creator of file f
(P_f, L_f)	Performance requested by the creator of file f: When service is available, P_f% of requests should be served in L_f time
H_f	Set of back-end services that store file f
S_f	Size of file f
r_f, w_f	Expected percentage of reads and writes to file f
R_f, W_f	Actual percentage of reads and writes to file f
P_f^r, P_f^w	Percentage of reads and writes to file f that complete in L_f time
$CDF_i(L)$	Percentage of requests served by back-end service i in L time
p_i	Probability of sending a request to back-end service i (optimized by Base)
(l_i, p_i)	Length of wait at back-end service i and expected percentage of requests served by i during the wait (optimized by OptWait)
$Cost(f)$	Expected monetary cost of serving file f
$AccessCost_t(f)$	Actual monetary cost of serving file f during interval t
$TotalCost(f)$	Actual monetary cost of serving file f over all intervals

guaranteed. To guarantee high availability and fault tolerance, all data and meta-data are replicated across several back-end services. Furthermore, the front-end servers only store soft state, such as a disk cache of meta-data, and keep write-ahead logs of updates in the back-end. All files are accessible from an inode-map stored at a few specific back-end services (and cached on the disks of the front-end servers). Thus, if a front-end server fails, the user can mount the file system through another front-end server, which can take over for the failed server using its write-ahead log.

The back-end block-storage services may be provided by different service providers. Although our front-end service treats the back-end services as "black boxes", we do assume that each back-end service is bounded by an SLA with the front-end file service. In particular, each back-end service i promises to meet an availability guarantee of A_i and a performance guarantee of (P_i, L_i) at a cost of (c_i^r, c_i^w, c_i^s). The two guarantees specify that service i will be servicing access requests A_i% of the time and, when it is available, P_i% of the accesses will complete within time L_i. The SLAs are defined over a long period of time, say one month, so that short-lived performance anomalies do not cause SLA violations. The cost tuple (c_i^r, c_i^w, c_i^s) specifies that each read access costs c_i^r, each write access costs c_i^w, and each unit of storage per unit of time costs c_i^s. Table 1 summarizes the notation used in our modeling.

In computing request distributions, the front-end service uses the availability and cost information from the SLAs with the back-end services. Instead of relying on the performance guarantees provided by the back-end services in computing distributions, we use the latency of requests as observed at the front-end service to encompass the latency of the wide-area network. Specifically, the front-end service monitors the latency of block accesses to each back-end service over two periods of 12 hours per day. The request distributions computed during a period of 12 hours are based on the cumulative distribution function (CDF) of the latencies observed during the same period of the day before. For example, the request distributions computed during the afternoon on Wednesday are based on the latencies observed during the afternoon on Tuesday. This approach is motivated by the cyclical workloads of many Internet services [5]. We plan to investigate more sophisticated approaches for considering block access latencies as future work.

File creation and access. When a file f is first created, the user can specify a desired availability guarantee of A_f and a performance guarantee of (P_f, L_f). (Files for which the user requests no guarantees are stored at a single back-end service and served on a best-effort basis.) These desired characteristics, if accepted by the front-end service, determine that it must be able to serve access requests to f $A_f\%$ of the time and that $P_f\%$ of the requests must complete within time L_f, when the service is available. If a file access request involves $n > 1$ blocks, the target latency for the request becomes nL_f. Again, these guarantees are defined over a long period of time, e.g. one month.

Obviously, we can only meet the requested availability if the front-end service itself is more available than A_f. If that is the case, it will choose a set of back-end services H_f to host f that meets (or exceeds) A_f. The front-end service randomly selects back-end services from three classes – inexpensive, medium, and expensive – one at a time in round-robin fashion. These classes are likely to correspond to services with generally high, medium, and low response times, respectively, although that is not a requirement. Assuming that failures are independent, the front-end service will select a set of back-end services that satisfies the following inequality:

$$A_{front} \times (1 - \prod_{i \in H_f} (1 - A_i)) \geq A_f \quad (1)$$

where A_{front} is the availability of the front-end service. This formulation assumes that the back-end services are always reachable from the front-end service across the network. However, it can be easily replaced by more sophisticated formulations without affecting the rest of the system.

The front-end will choose a minimal set H_f in the sense that, if any back-end service is removed from H_f, the remaining set would no longer be able to meet A_f. Once H_f has been chosen, the front-end service will solve a cost-optimization problem for the two algorithms and choose the one that produces the lowest cost for f.

At this point, file f can be accessed by clients. On a read to f, the front-end service will forward a request to a subset of H_f for each needed block according to the chosen algorithm. On a write, the front-end will forward the request to all back-end services in H_f to maintain the target data availability, while concurrently writing to the write-ahead log if necessary. The front-end service only waits for the possible write ahead and one back-end service to process the write before responding to the client. In the background, the front-end service will ensure that the write is processed by the other back-end services in H_f as well. When write sharing is done through the same front-end server, this approach to processing writes favors lower latency without compromising strong consistency; the pending writes can be checked before a subsequent read is forwarded to the back-end.

Optimizing costs. Our request-distribution algorithms, Base and OptWait, are run by the front-end service to minimize the cost of accessing the back-end services in H_f. As mentioned above, their respective optimization problems are solved at first during file creation, but they may need to be solved again multiple times over the file's lifetime. In particular, whenever the file is opened, a new distribution is computed but only if the current distribution is stale, i.e. it was not computed based on the same period of the day before. After the back-end services are selected and the request distribution is computed, the front-end service can inform the client about the cost of each byte of storage and the (initial) average cost of each block access, given the requested guarantees. Note that the cost of accessing the write-ahead logs is not included in the cost computations; this cost is covered by our service fees (discussed below).

Because we select the H_f back-end services randomly from three classes of services, our cost optimization produces a "locally" optimal cost; it is possible that this cost will not be the lowest possible cost (i.e., the "globally" optimal cost) for a system with a large number of back-end services. Attempting to produce the lowest possible cost would involve searching an exponentially large space of back-end service groupings, which could take hours/days of compute time to explore meaningfully, even if a heuristic algorithm were to be used. We plan to explore this issue in our future work.

The front-end accumulates the access costs accrued during the periods of stable request distribution, i.e. in between consecutive changes to the request distribution. The overall cost of the composite service is then the sum of the costs for each stable period. Periodically, say every month, the front-end service charges each of its users based on how many accesses and how much storage the front-end service required of its back-end services on behalf of the user. Formally, the total cost to be charged is:

$$\mathit{TotalCost}(f) = \sum_{\forall_t} \mathit{AccessCost}_t(f) + S_f \sum_{i \in H_f} c_i^s \qquad (2)$$

where $AccessCost_t(f)$ is the access cost of each period t of stable request distributions since the last calculation of $TotalCost(f)$ and S_f is the maximum size of the file since the last calculation of $TotalCost(f)$. We define $AccessCost_t(f)$ exactly below.

Service fees and compensation. Finally, note that the costs incurred by the front-end service are actually higher than the sum of $\mathit{TotalCost}(f)$ for all files. As mentioned above, the cost of accessing the write-ahead logs is not included in $\mathit{TotalCost}(f)$. In addition, when the client load is low, the front-end service may need to send additional accesses to the back-end services to properly assess their current performance (and availability). These extra accesses increase costs for the front-end service; the extra cost can be amortized across the set of users as a "service fee".

Further, there may be situations in which the guarantees provided by the front-end service are violated. For example, the network between the front-end service and some of the back-end services may become unusually slow or back-end services may start violating their SLAs. As mentioned above, the front-end service responds to these situations by recomputing its request distributions accordingly, but the recomputations may not occur early enough. Nevertheless, in case of back-end SLA violations, the front-end service will be compensated for them and the compensations can be passed on to its users. In case of network problems, the front-end service can use its service fees to compensate users.

3.2 Base

In Base, a read request to a file f is forwarded to a single back-end service $i \in H_f$ with probability p_i. (Writes are sent to all back-end services in H_f.) Base computes these probabilities so as to minimize the cost of servicing accesses to f while respecting the performance guarantees requested for the file. Formally, Base needs to minimize:

$$\mathit{Cost}(f) = r_f \sum_{i \in H_f} p_i c_i^r + w_f \sum_{i \in H_f} c_i^w \tag{3}$$

subject to the following two constraints:

1. $\forall i \in H_f, p_i \geq 0$ and $\sum p_i = 1$ 2. $r_f P_f^r + w_f P_f^w \geq P_f$

where r_f is the fraction of read block accesses to f, w_f is the fraction of write block accesses to f, P_f^r is the percentage of read accesses that complete within L_f, and P_f^w is the percentage of write accesses that complete within L_f.

Equation 3 computes the average cost of reads and writes, reflecting the read-to-write ratio ($r_f : w_f$), and the fact that each read incurs the cost of only 1 back-end access according to the probabilities p_i (hence $p_i c_i^r$), while each write incurs the cost of accessing all back-end services. Constraint 1 states that the probabilities of accessing each back-end service in H_f have to be non-negative and add up to 1. Constraint 2 requires that the percentage of reads and writes that complete within L_f time must be at least P_f to meet the guarantees requested by the user.

We then define P_f^r and P_f^w as:

$$P_f^r = \sum_{i \in H_f} p_i \mathit{CDF}_i(L_f) \qquad P_f^w = \max_{i \in H_f}(\mathit{CDF}_i(L_f)) \tag{4}$$

where the $\mathit{CDF}_i(L)$ operator produces the percentage of requests satisfied within L time by back-end service i, as observed at the front-end service. P_f^w is determined by the best performing back-end service because the front-end forwards each write in parallel to all back-end services and replies to the client when the first one completes.

Equations 3 and 4 together with the two constraints completely define Base's optimization problem, except for how to determine r_f and w_f. The user can optionally estimate r_f and w_f and pass them as parameters at file creation time. If the user does not provide this information, we split constraint 2 above into two parts, $P_f^r \geq P_f$ and $P_f^w \geq P_f$, and instantiate Equation 3 with the assumption that $r_f = 1$ and $w_f = 0$. This approach correctly but conservatively ensures that the solution to the optimization problem provides the required guarantees for f. For details on this point, please refer to the longer, technical report version of this paper [16].

After each period t of stable request distributions computed by Base, we compute the cost of accessing the H_f back-end services during the period as:

$$AccessCost_t(f) = R_f \sum_{i \in H_f} p_i c_i^r + W_f \sum_{i \in H_f} c_i^w \tag{5}$$

where R_f is the number of read requests and W_f is the number of write requests serviced during period t.

Finally, note that a malicious client is not able to lower its access costs by providing fake values for r_f and w_f, since these costs are computed based on the actual requests made by the client during each period of time.

3.3 OptWait

In OptWait, the front-end service takes the different approach of possibly forwarding a read request to more than one back-end service. In particular, the front-end service forwards each read request to the back-end services in sequence, from least to most expensive, waiting for a bounded amount of time for each service to respond before trying the next service.

The basic idea behind OptWait is illustrated in Figure 1, which shows three performance CDFs for three back-end services. Let us assume that the left-most curve represents the most expensive service, whereas the right-most curve represents the least expensive service. OptWait would first forward a request to the least expensive service, waiting for an amount of time l_1. This would allow OptWait to take advantage of the percentage of requests (p_1) that complete fairly quickly. If the request did not complete within l_1 time, OptWait would then forward the request to the medium-cost service and wait for some wait time l_2. Again, the goal would be to leverage the steep part of the medium-cost service's CDF. If, after $l_1 + l_2$ time, the request still had not completed at either back-end service, OptWait would then forward the request to the most expensive service and wait for the request to complete at any of the three back-end services.

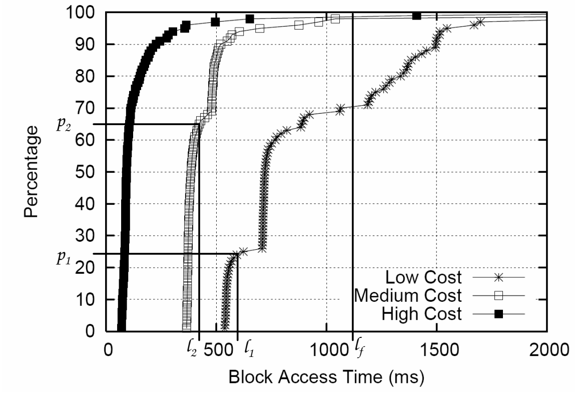


Fig. 1. Performance CDFs for three services. An OptWait distribution might specify that a request should be forwarded to multiple back-end services in turn.

The key to OptWait is setting appropriate l_i times. Like in Base, we do so by optimizing the access cost under the performance constraints imposed by the guarantees requested by the user. Assuming H_f with 3 back-end services, our problem is to minimize the following equation:

$$\begin{aligned} Cost(f) = r_f[&p_1C_1 \\ &+((1-\mathit{CDF}_1(l_1+l_2))p_2+\mathit{CDF}_1(l_1+l_2)-p_1)(C_1+C_2) \\ &+(1-(1-\mathit{CDF}_1(l_1+l_2))p_2-\mathit{CDF}_1(l_1+l_2))(C_1+C_2+C_3)] \\ +w_f &\textstyle\sum_{i\in H_f} c_i^w \end{aligned} \tag{6}$$

where $p_i = \mathit{CDF}_i(l_i)$, $\mathit{CDF}_i(l) = 0$ when service i is not being used for reads (i.e., $l_i = 0$), $C_i = 0$ when service i is not being used for reads and $C_i = c_i^r$ when it is, and $l_i = \infty$ when i is the last service being used for reads. (We only present the equation for the restricted case of 3 back-end services for clarity and because of space constraints. We refer the interested reader to [16] for the general formulation.)

Equation 6 computes the cost of writes in the same manner as the Base cost function (Equation 3), as the two algorithms treat writes in the same way. More interestingly, it computes the cost of reads by summing up the multiplication of the probability that each back-end service will need to be accessed by the cost of doing so. For example, if services 1 and 2 are used for reads, the first two lines of the equation compute the cost, whereas the third line becomes 0. The first line multiplies the probability that service 1 replies within l_1 time (p_1) by the cost of accessing service 1. For the requests that are not serviced by service 1 within l_1, service 2 would be activated. Thus, the second line of the equation sums up the probability that service 1 does not reply within $l_1 + l_2$ time but service 2 does reply within l_2 time ($(1-\mathit{CDF}_1(l_1+l_2))p_2$), and the probability that service 1 replies after l_1 but before $l_1 + l_2$ time ($\mathit{CDF}_1(l_1+l_2) - p_1$). The second part of the cost is obtained by multiplying this probability by the cost of making one access to service 1 and one access to service 2.

Equation 6 should be minimized subject to the following constraints:

$$1.\ \forall i \in H_f, l_i \geq 0 \qquad 2.\ r_f P_f^r + w_f P_f^w \geq P_f$$

where constraint 1 simply states that times have to be non-negative and constraint 2 is the same as that for Base. (Just as for Base, the front-end service can break constraint 2 into two parts and compute costs for $r_f = 1$ and $w_f = 0$, if the user does not provide information about r_f and w_f as a parameter.) We define P_f^w just the same as for Base, since the two algorithms handle writes in the same way. In contrast, P_f^r is defined as:

$$\begin{aligned} P_f^r = &\mathit{CDF}_1(L_f) \\ &+(1-\mathit{CDF}_1(L_f))\mathit{CDF}_2(L_f-l_1) \\ &+(1-\mathit{CDF}_1(L_f))(1-\mathit{CDF}_2(L_f-l_1))\mathit{CDF}_3(L_f-l_1-l_2) \end{aligned} \tag{7}$$

where again $\mathit{CDF}_i(l) = 0$ when service i is not being used for reads.

In plain English, the first additive component of Equation 7 represents the probability that the least-expensive service will reply in a timely manner (within L_f time) if it is used, the second component is the probability that service 2, if used, will reply in a timely manner (given that a request is only forwarded to it after l_1 time) but not service 1, and so on. (Again, because the general formulation and its closed form [16] are hard to read, we only present the equation for a system with exactly 3 back-end services.)

After each period t of stable request distributions computed by OptWait, we compute the cost of accessing the H_f back-end services during the period by replacing r_f and w_f in Equation 6 by R_f and W_f, respectively.

3.4 Implementation

We have implemented a prototype front-end file service called Figurehead to explore our request-distribution algorithms in real systems with real workloads. Although Figurehead should be supported by multiple geographically distributed servers in practice, it is currently based on a single node as a proof-of-concept implementation.

Figurehead consists of four components: an NFS version 2 facade that allows the file service to be accessed through standard NFS clients, a file system that supports the NFS facade and uses remote back-end block services for storage, an optimization module that computes the best request distribution strategy, and a module that constantly monitors the performance of the back-end services. All components were written in Java and run in user space. Relevant details about these four components are as follows.

NFS facade. The multi-threaded NFS facade accepts NFS remote procedure calls via UDP. It implements the NFS version 2 protocol almost completely; the only calls that have not been implemented are those dealing with symbolic links.

The one complication that the NFS protocol poses for Figurehead is that opens and closes are not sent through to the server. Thus, whenever the NFS facade receives a create or the first access to an unopened file, it opens the file and caches the opened-file object returned by the file system. A cached opened-file object is closed and discarded after it has not been accessed for 5 minutes.

File system. The file system behind our NFS facade uses the same meta-data scheme to represent a file as the Linux ext2 file system. The inode was changed to include information about the availability and performance guarantees requested by the creator of a file. An inode-map maps each inode to the set of back-end services that is hosting the file. All data and meta-data except for the inode-map are stored at the back-end services in 8-KByte blocks. The file system communicates with the back-end services over a Web Service interface, namely the RPC implementation from Apache Axis [17].

When a file is first created, the file system chooses a set of back-end services to host the file as described in Section 3.1. It then allocates an inode, saves the availability and performance guarantees for the file in the inode (along with other traditional file-system information, such as owner and time of creation), enters the mapping of *inode-number* $\rightarrow H_f$ into its inode-map, and writes the inode to the appropriate back-end services. The file system also opens the file.

When a file is opened, the file system extracts the set of back-end services that is hosting the file (H_f) from the inode-map, obtains their access time CDFs from the monitoring module, reads the inode to obtain the performance guarantees, and asks the request distribution module to compute the best request distribution strategy for the file. This last step is not necessary when the file is being re-opened and the current request distribution was computed based on the same period of the day before. To determine whether to recompute a request distribution, Figurehead maintains information about when each distribution is computed. When a previous request distribution exists but a new computation is required, the computation is performed in the background and adopted when completed. When client requests arrive, the file system uses the file meta-data to identify the corresponding blocks and forwards the appropriate block operations to the back-end services. Reads are handled according to the current request distribution, whereas writes are forwarded to all back-end services in H_f.

The file system maintains a write buffer to ensure that each write to a file f eventually reaches all of the nodes in H_f. When a write request arrives, the file system assigns a thread per back-end service in H_f the task of ensuring that the write eventually reaches a particular back-end. Each write is then discarded from the write buffer once it has propagated to all back-ends in H_f. We assume that the back-end services can handle small "overwrites;" that is, a write that only partially overwrites a previously written block can be sent directly to the back-end services without having to read the old data and compose a new complete-block write. This avoids making small overwrites more expensive than a complete-block write because of the need to read the block.

The file system implements two levels of meta-data caching. First, all meta-data is currently cached on a local disk (and is never evicted) using a Berkeley database [18]. This cache reduces the number of accesses to the back-end services by eliminating repeated remote meta-data accesses. In fact, the cache makes the meta-data accesses to the back-end services relatively infrequent for the large-file applications we target (dominated by reads and/or overwrites), so these accesses are not currently reflected in our mathematical machinery. Second, file-specific meta-data, i.e. inodes and indirect blocks, are cached in memory for open files as the meta-data is accessed. This avoids repeatedly accessing the cache on disk for a stream of accesses to the same file. Meta-data of an open file that is cached in memory is evicted when the file is closed. Our policy of holding a file opened in the NFS facade for 5 minutes beyond its last access implies that meta-data for an open file is also cached in memory by the file system for the same amount of time.

Finally, since the NFS clients cache data themselves, our file system (in fact, the entire front-end service) does not cache data at all.

Request-distribution module. This module solves the optimization problems posed by Base and OptWait, and chooses the algorithm that produces the lowest cost. The Base optimization problem is solved using the linear programming solver lp_solve [19] and produces the p_i probabilities with a precision of a few decimal places. Unfortunately, minimizing cost in OptWait is not a linear programming problem. To solve it, we consider all feasible combinations of the probabilities p_i's (in steps of 1% in our current implementation) for the back-end services in H_f to compute the best l_i's wait times. Even though this is essentially a brute force approach, it does not take long to compute as the size of H_f is small (typically two or three), even for high P_f requirements. We report running times for this module in Section 4.

Monitoring module. This module is responsible for monitoring each back-end service in terms of its performance as seen at the front-end service. Specifically, this module probes each back-end service periodically with regular block accesses (every 5 seconds in our current implementation). With the access times measured from these accesses, this module constructs the performance CDF for the service.

Figurehead limitations. Currently, Figurehead has three limitations. First, as we mentioned above, it is implemented by a single server, rather than a collection of geographically distributed servers. Second, we have not yet implemented the write-ahead log for crash recovery. Third, the monitoring module currently does not use information from regular accesses to the back-end services, always issuing additional block accesses to assess their performance (and availability). These extra accesses increase costs and would not be required when the regular load on the back-end services is high enough. We are currently addressing these limitations.

4 Evaluation

In this section, we first explore and compare the two request distribution algorithms over the space of different costs and back-end service behaviors. We then study the impact of using past access time data to predict current behaviors of the back-end servers. Finally, we evaluate our prototype Figurehead implementation, and validate that it provides the performance guarantees computed by the mathematical machinery.

Ideally, we would like to study our system using actual back-end services on the Internet. However, at this point, there are not enough of them to provide a large range of data. Thus, we have collected access times over a period of close to one month from 50 PlanetLab machines to support our evaluation. These data were collected by running a simple block-storage service on each machine, populating each service with 5120 blocks, and randomly accessing a block according to a Poisson process with mean inter-access time of 1 second from a client machine located at our site.

4.1 Base vs. OptWait

We first compare Base and OptWait mathematically assuming fixed access time CDFs for the back-end services. In particular, we chose data from three PlanetLab nodes, `planetlab2.cs.umass.edu`, `planetlab1.cs.unibo.it`, and `planet-lab.iki.rssi.ru`, whose CDFs are shown in Figure 1. We study a set of three nodes because they provide a sufficiently rich space to understand the behaviors of the two algorithms, yet is not overly complicated to explain.

Overall results. Figure 2 plots the average cost (*Cost*(f)) achieved by Base and OptWait for a read-only workload as a function of the per-file guaranteed latency (L_f), with a per-file percentage guarantee (P_f) of 95%. (The results are similar for other P_f values.) Each of the curves represents a different combination of algorithm and per-access cost for each back-end service. For example, the curve labeled OptWait [5,10,15] represents the cost computed by OptWait

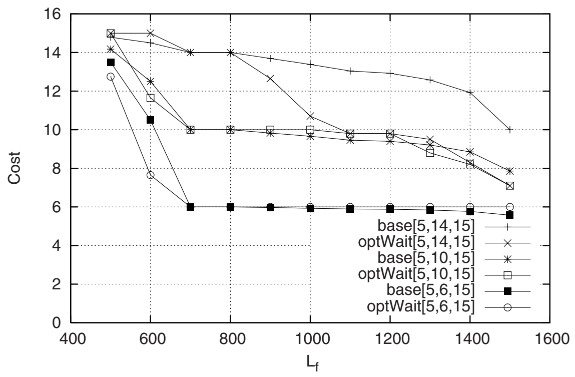


Fig. 2. Costs achieved by Base and OptWait vs. L_f, assuming a read-only workload and $P_f = 95\%$

Table 2. Costs and distributions with back-end service costs = [5,10,15] and $P_f = 95\%$. The Base distributions are listed as [p_1, p_2, p_3], whereas the OptWait distributions are listed as [(l_1, p_1), (l_2, p_2), (l_3, p_3)]. l_1, l_2, l_3 are given in ms.

L_f (ms)	Base Cost	Base Dist	OptWait Cost	OptWait Dist
500	14.17	[0,17,83]	15.00	[(0,0),(0,0),(∞,100)]
600	12.50	[0,50,50]	11.65	[(0,0),(511,89),(∞,100)]
700	10.00	[0,100,0]	10.00	[(0,0),(∞,100),(0,0)]
800	10.00	[0,100,0]	10.00	[(0,0),(∞,100),(0,0)]
900	9.83	[3,97,0]	10.00	[(0,0),(∞,100),(0,0)]
1000	9.66	[7,93,0]	10.00	[(0,0),(∞,100),(0,0)]
1100	9.46	[11,89,0]	9.80	[(923,68),(0,0),(∞,100)]
1200	9.40	[12,88,0]	9.80	[(923,68),(0,0),(∞,100)]
1300	9.21	[16,84,0]	8.80	[(794,62),(∞,100),(0,0)]
1400	8.85	[23,77,0]	8.20	[(923,68),(∞,100),(0,0)]
1500	7.86	[43,57,0]	7.10	[(1404,86),(0,0),(∞,100)]
1600	5.00	[100,0,0]	5.00	[(∞,100),(0,0),(0,0)]

when $c_1^r = 5, c_2^r = 10$, and $c_3^r = 15$ fractions of dollar per access (what fraction exactly is irrelevant to our study). Table 2 lists the optimized costs and request distributions for Base and OptWait for costs [5,10,15].

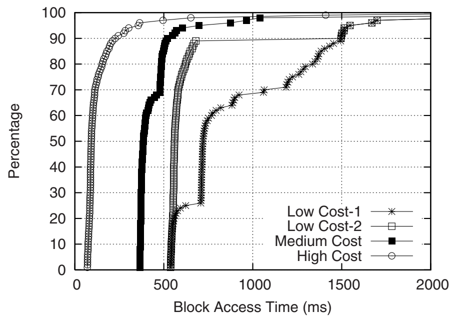
From these figures, we can see that neither Base nor OptWait is always better than the other. At the extremes, i.e. at very low or very high latency guarantees, the two algorithms behave the same because there is no room for optimization. For very low latency guarantees, the only choice is to use the most expensive service all the time (if it is possible to meet the guarantee at all). For very high latency guarantees, the obvious best choice is to use the cheapest service all the time.

In between these extremes, the relative behavior of Base and OptWait depends on the shapes of the access time CDFs of the back-end services, as well as their costs. For example, consider the costs achieved by Base and OptWait for cost [5, 10, 15] at latency guarantees of 500ms and 600ms. At 500ms, Base achieves lower cost than OptWait because it is able to use the medium-cost service 17% of the time, whereas OptWait cannot yet use the medium-cost service (see Table 2). In this case, for p_2 in OptWait to be greater than 0, l_2 would have to be at least 365ms, leaving insufficient time for accessing the high-cost service should the request fail to complete at the medium-cost service within l_2. At 600ms, OptWait does better than Base because its greater use of the medium-cost service, 89% vs 50%, more than offsets the 11% of the time that it has to use both the medium-cost and high-cost service.

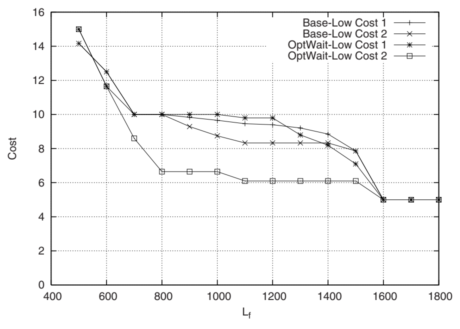
In general, we observe that Base can typically start using a lower-cost back-end service before OptWait as the guaranteed response time increases. This is because Base never resends requests. However, eventually, OptWait can use the lower-cost service more aggressively because it can avoid the tail of the CDF by re-sending requests to the more expensive services as needed.

Table 3. Costs and distributions with $P_f = 95\%$, as a function of L_f and back-end service costs. The Base distributions are listed as $[p_1, p_2, p_3]$, whereas the OptWait distributions are listed as $[(l_1, p_1), (l_2, p_2), (l_3, p_3)]$. l_1, l_2, l_3 are given in ms.

L_f (ms)	Back-End Costs	Base Cost	Base Dist	OptWait Cost	OptWait Distribution
1200	[5,10,15]	9.40	[12,88,0]	9.80	[(923,68),(0,0),(∞,100)]
1200	[5,6,15]	5.88	[12,88,0]	6.00	[(0,0),(∞,100),(0,0)]
1200	[5,14,15]	12.92	[12,88,0]	9.80	[(923,68),(0,0),(∞,100)]
1300	[5,10,15]	9.21	[15.79,84.21,0]	8.80	[(794,62),(∞,100),(0,0)]
1300	[5,6,15]	5.84	[15.79,84.21,0]	6.00	[(0,0),(∞,100),(0,0)]
1300	[5,14,15]	12.58	[15.79,84.21,0]	9.50	[(1064,70),(0,0),(∞,100)]
1400	[5,10,15]	8.85	[23.08,76.92,0]	8.20	[(923,68),(∞,100),(0,0)]
1400	[5,6,15]	5.77	[23.08,76.92,0]	6.00	[(0,0),(∞,100),(0,0)]
1400	[5,14,15]	11.92	[23.08,76.92,0]	8.30	[(1285,78),(0,0),(∞,100)]



(a)



(b)

Fig. 3. (a) CDFs for 4 back-end services. (b) Access cost achieved by Base and OptWait when using two different sets of three back-end services {low-cost-1, medium-cost, high-cost} and {low-cost-2, medium-cost, high-cost}. Both with cost [5,10,15] and $P_f = 95\%$.

Impact of the back-end service costs. Observe that Base's distribution of requests is independent of the ratio between the costs of the three back-end services. That is, as long as $c_3^r > c_2^r > c_1^r$, Base will choose the same set of distribution probabilities (p_1, p_2, p_3) regardless of the ratios c_1:c_2:c_3. OptWait, on the other hand, may alter its distribution strategy based on the cost ratios. For example, consider in Table 3 the distributions computed for L_f within the interval [1200ms, 1400ms] for costs [5, 6, 15] vs. [5, 10, 15]. For [5, 10, 15], OptWait chooses to use either the low- and medium-cost or low- and high-cost services. For [5, 6, 15], OptWait only chooses to use the medium-cost service. This is because the medium-cost service is only slightly more expensive than the low-cost service; immediately choosing it is less costly than potentially having to forward the request to two services.

Impact of the shape of the CDFs. Base and OptWait also behave differently with respect to the shapes of the CDFs. In general, Base's behavior depends on the three key points $CDF_1(L_f)$, $CDF_2(L_f)$, and $CDF_3(L_f)$, whereas OptWait's behavior depends on the shape of all CDFs between 0% and $CDF_i(L_f)$. These dependencies can be seen clearly in Figures 3(a) and (b). Figure 3(a) shows the CDFs for 4 back-end services from which we derived two sets of three services {low-cost-1, medium-cost, high-cost} and {low-cost-2, medium-cost, high-cost}.

Figure 3(b) shows that OptWait behaves significantly better when using low-cost-2 in the interval [600ms, 1600ms] because low-cost-2 is substantially "steeper" than low-cost-1. Base is also able to leverage low-cost-2's better behavior to improve its cost, but less so than OptWait. The reason is that Base only leverages the fact that low-cost-2 gives a better $CDF_1(L_f)$ than low-cost-1, rather than the fact that low-cost-2 gives an additional 30% of requests completing under 700ms over low-cost-1 in this interval.

4.2 Validating the Mathematical Machinery

We now validate our mathematical approach when servicing actual file system workloads. We also validate that the prediction of back-end service behaviors using past access time data do not significantly degrade our QoS guarantees. First, we use simulation to analyze the mathematical approach independent of the details of an actual implementation. Next, we evaluate our prototype implementation.

Workloads. We use two realistic workloads. The first models an interactive visualization application, where the user is navigating through a large amount of data–for example, a large rendering model or large scientific data set. This application is exactly the type of soft real-time application that Figurehead is designed to support.

This workload is constructed based on publications on visualization systems [20,21,22], and has the following attributes: a random Poisson read access stream with a mean interarrival time of 50ms on a large data file. It currently does not make a difference to Figurehead whether a read stream is random or sequential, since Figurehead does not currently do any prefetching or caching. We assume a random read access stream because these accesses are dependent on the user's interactive navigation.

The second workload models a scientific application running on a grid environment. Although this is not a classical soft real-time application, it still constitutes an interesting workload because predictability of data access can significantly reduce the burden of resource management and coordination of the stages of a multi-stage application such as the one described in [23].

This workload is constructed based on data extracted from [1,23,24,25], and has the following attributes: a sequential read access stream from a single large file followed by a sequential write access stream to the same file. This read/write access stream represents a multi-phase application with an initial read phase to load input data and a final write phase that saves the computed results. We assume that intermediate results generated between the initial and final phases are stored on local storage rather than a file system such as Figurehead. We further assume that the initial input data and the final results have the same size; thus, the read-to-write ratio is 1:1. Finally, both the read and write access streams are Poisson processes with mean interarrival times of 50ms.

Because the WAN latencies we consider are larger than 50ms, we assume that the access streams of both applications are generated by a number of concurrent threads.

Table 4. Simulation results with $(P_f, L_f) = (95\%, 600\text{ms})$ and costs [5,10,15]. **V** denotes the visualization workload, **S** the scientific workload, **B** the Base algorithm, and **O** the OptWait algorithm. **Expected** is the percentage of requests expected to complete before L_f as computed by the algorithm. **Simulated** is the actual percentage of requests that completed before L_f in a simulation run. **Min**, **Max**, **Avg** are the minimum, maximum, and average values across 18 runs using 18 half-day traces from the PlanetLab machines.

	V-B		**S-B**		**V-O**		**S-O**	
	Expected	**Simulated**	**Expected**	**Simulated**	**Expected**	**Simulated**	**Expected**	**Simulated**
Min	95	95.06	95	95.28	95.2	95.61	95	95.56
Max	95	95.89	95	97.28	97.36	97.89	97.33	98.56
Avg	95	95.48	95	96.07	96.22	96.57	95.7	96.78

Simulation using *a priori* knowledge of back-end service behaviors. Our first experiment is as follows. Take a trace of the three machines whose overall behaviors are shown in Figure 1 over a period of 9 days. Construct a CDF for each back-end service for each 12-hour period of the 9 days. For each 12-hour period, use the corresponding CDF to compute the distribution using Base and OptWait for $P_f = 95\%$, $L_f = 600ms$, costs [5,10,15], and $c^r = c^w$ for all back-end services. Then, simulate Figurehead's response time for 18000 accesses for each workload using the 12-hour traces that were used to construct the CDFs. This corresponds to statistical oracular knowledge of the behaviors of the back-end services.

Table 4 shows the results for 18 runs of each application/distribution algorithm pair, where each run was performed using a distinct half-day period of the 9-day trace. For both workloads under Base and OptWait, the simulation always leads to exceeding the QoS guarantee. This is because we construct and use the CDFs in a conservative manner. In particular, each CDF is represented by a set of 100 discrete points, representing the latency corresponding to each percentage point on the CDF. Now suppose that the mathematical engine needs a percentage value corresponding to the latency 1000ms. If our CDF has the points (999ms, 95%) and (1001ms, 96%), then we would return 95%, rather than an interpolated value between 95% and 96%. We choose this conservative approach because an interpolated value would be optimistic sometimes but pessimistic other times, making the mathematical machinery less predictable.

An additional interesting observation to make is that mathematically, Base always achieves a distribution that should theoretically give the exact P_f required (in this case, 95%). OptWait, on the other hand, because of our discrete approach for computing the best distribution, typically overachieves compared to the required P_f. (Note that, for $L_f = 600ms$, OptWait achieves lower cost than Base despite this overachievement.) As shall be seen, this overachievement makes OptWait more robust when the CDF is computed based on past data.

Impact of using past access times to predict current back-end service behaviors. We now consider the impact of not having *a priori* information on the expected behaviors of the back-end services. In particular, as mentioned in Section 3.1, we run the same experiments as above but use a CDF constructed from the response times observed in the same 12-hour period 1 day ago to predict each back-end service's behavior in the

Table 5. Simulated results for $(P_f, L_f) = (95\%, 600\text{ms})$ and costs [5,10,15] when using data access times from 12 hours ago to predict the current behaviors of back-end services. The notation is the same as in Table 4. **Failures** is the number of 12-hour simulation runs that did not meet the QoS guarantee.

	V-B		**S-B**		**V-O**		**S-O**	
	Expected	**Simulated**	**Expected**	**Simulated**	**Expected**	**Simulated**	**Expected**	**Simulated**
Min	95	92.72	95	94.28	95.2	93.83	95	95.67
Max	95	97.72	95	98.33	97.36	98.83	97.33	98.61
Avg	95	95.35	95	96.11	96.24	96.42	95.76	96.8
Failures	0	5	0	6	0	5	0	0

current 12-hour period (e.g., 8am-8pm from Tuesday to predict behavior for 8am-8pm Wednesday). Table 5 shows the results for 16 12-hour runs (we could not use the first two half-day periods because they did not have any past history for prediction).

As expected, past data is not a perfect predictor of current behavior. This leads to a number of 12-hour simulation runs where Figurehead would not be able to achieve the QoS guarantee. In fact, approximately 35% of the runs missed the QoS guarantee under Base. OptWait has a comparable failure rate for the Visualization workload but was perfect for the Scientific workload. As already mentioned, OptWait is somewhat more resilient to the imperfect predictor because it typically overachieves compared to the required P_f. On the other hand, the imperfect predictor can also lead the 12-hour runs to achieve more than the QoS requirement, i.e. more than P_f of the requests complete within L_f time. In fact, the **Max** values for both Base and OptWait are larger in Table 5 than in Table 4.

However, the most important observation here is that *both request-distribution algorithms provide the performance guarantees that they promise* when the entire 8 day period is considered (see the simulated **Avg** entries). (Recall that QoS guarantees are defined over long periods of time, such as one month.) The reason for this result is that the QoS requirement is exceeded during the majority of the 12-hour periods, which more than compensates for the many fewer periods when the requirement is not met.

4.3 Prototype Behavior

We now validate that our prototype, Figurehead, actually provides the performance guarantees computed by the mathematical machinery. All results reported below were obtained by running on 5 PCs connected by a Gb/s Ethernet switch. Each PC is configured with 1 hyper-threading Intel Xeon 2.8 GHz processor, 2 GBytes of main memory, and 25 GBytes of disk space. Three of the machines were used as back-end block servers and one as the client. The other machine ran Figurehead. We always assume that the three back-end services are needed to meet the client's specified availability requirement. Again, all the experiments assume $P_f = 95\%$, $L_f = 600ms$, costs [5,10,15], and $c^r = c^w$ for all back-end services. To mimic a wide-area network, we inserted delays to the completion times of accesses to the back-end services. We use the same 9-day trace as in the last subsection; the delays were randomly chosen from the appropriate half-day period. (We used the traces instead of running the back-end services themselves on PlanetLab nodes for repeatability.)

Microbenchmarks. We first present results from microbenchmarks to illustrate the performance of Figurehead. For these microbenchmarks, we did not inject any network delays so that performance reflects what is achievable over a LAN. We also assume that r_f and w_f are known ahead of time; i.e., r_f is 1 when measuring read performance and 0 when measuring write performance. We measured write performance for appends (rather than overwrites) to a file.

Using these microbenchmarks, we find that the times required to read and write 1 byte of data are approximately 30ms and 66ms, respectively. Appends are more expensive than reads because they require writing meta-data. Overall, Figurehead reads and writes are about one order of magnitude slower than on a local disk. The higher access latency of Figurehead arises mainly from using a Berkeley database as disk cache and the Web Services interface to access the back-end block servers. These inefficiencies can be easily eliminated in a production-grade implementation. However, the fairer comparison is between accessing a back-end service through Figurehead and accessing it directly, both on a WAN. Because network trips dominate in this scenario, Figurehead would impose a much lower overhead. For example, the lowest average latency we measured for the PlanetLab nodes is 165ms. Given this latency, Figurehead would impose roughly a 30% degradation when all accesses are appends.

Another important issue is the overhead of computing request distributions. The time to solve a Base and OptWait optimization problem is approximately 710us and 14ms, respectively. We found that, while the time to solve OptWait does increase with L_f, it does so quite modestly. The reason for the slight time increase is that a higher L_f tends to generate a larger search space in OptWait. Finally, these optimization times do not change significantly with changing P_f and so we do not show those results here.

Macrobenchmarks. Finally, we ran the two workloads described in the last section concurrently against a running instance of our Figurehead prototype. We ran each workload/distribution algorithm pair 4 times, each time for a distinct half-day period from the 9-day trace (the first 4 half-day periods). Overheads from the system (e.g, computing time inside the Figurehead front-end) led to a degradation in meeting the QoS requirement P_f by almost nothing to at most 1%. Detailed measurements show that the main sources of overheads were synchronization delays, inaccuracies in the sleep function used to emulate WAN latencies, and accessing the Berkeley DB. Despite these overheads, *the prototype consistently provides the proper guarantees when all the periods are considered.*

5 Conclusions

In this paper, we addressed the issue of composing functionally-equivalent, third-party services into higher level, value-added services by developing a distributed file service. In this context, we proposed two request-distribution algorithms that optimize costs at the same time as providing performance and availability guarantees. To achieve this goal, both algorithms rely on information about the behavior of the third-party services to mathematically determine the request distributions. While one algorithm is reminiscent of traditional scheduling policies, the other departs significantly from these policies, as it may schedule the same request at multiple third-party services in turn.

We found that both algorithms provide the guarantees that they promise. Comparing the algorithms, we found that neither is consistently the best. Nevertheless, using our mathematical modeling, the system can actually select the best algorithm for each file a priori. Experimental results from our prototype implementation characterized its performance and the optimized access costs under the two algorithms.

Composite services such as the one we studied are in the horizon. Based on our experience and results, we believe that these services can benefit from our modeling and optimization approach for guaranteeing quality-of-service at low cost.

References

1. Shasharina, S.G., Wang, N., Cary, J.R.: Grid Service for Visualization and Analysis of Remote Fusion Data. In: Proceedings of the International Workshop on Challenges of Large Applications in Distributed Environments (June 2004)
2. Amazon: Amazon Simple Storage Service, `http://aws.amazon.com/s3`
3. Gu, X., Nahrstedt, K.: Distributed Multimedia Service Composition with Statistical QoS Assurances. IEEE Transactions on Multimedia 8(1) (February 2005)
4. Zeng, L., Benatallah, B., Ngu, A., Dumas, M., Kalagnanam, J., Chang, H.: QoS-Aware Middleware for Web Services Composition. IEEE Transactions on Software Engineering 30(5) (May 2004)
5. Chase, J., Anderson, D., Thackar, P., Vahdat, A., Boyle, R.: Managing Energy and Server Resources in Hosting Centers. In: Proceedings of the Symposium on Operating Systems Principles (October 2001)
6. Krauter, K., Buyya, R., Maheswaran, M.: A Taxonomy and Survey of Grid Resource Management Systems for Distributed Computing. Software–Practice and Experience 32(2) (February 2002)
7. Subramanian, L., Stoica, I., Balakrishnan, H., Katz, R.: OverQoS: An Overlay Based Architecture for Enhancing Internet QoS. In: Proceedings of the Symposium on Networked Systems Design and Implementation (March 2004)
8. Gibson, G.A., Nagle, D.F., Amiri, K., Butler, J., Chang, F.W., Gobioff, H., Hardin, C., Riedel, E., Rochberg, D., Zelenka, J.: A Cost-Effective, High-Bandwidth Storage Architecture. In: Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (October 1998)
9. Radkov, P., Yin, L., Goyal, P., Sarkar, P., Shenoy, P.: A Performance Comparison of NFS and iSCSI for IP-Networked Storage. In: Proceedings of the USENIX Conference on File and Storage Technologies (March 2004)
10. Thekkath, C.A., Mann, T.P., Lee, E.K.: Frangipani: A Scalable Distributed File System. In: Proceedings of the Symposium on Operating Systems Principles (October 1997)
11. Bhagwan, R., Tati, K., Cheng, Y.C., Savage, S., Voelker, G.M.: Total Recall: System Support for Automated Availability Management. In: Proceedings of the Symposium on Networked Systems Design and Implementation (March 2004)
12. Dabek, F., Kaashoek, M.F., Karger, D., Morris, R., Stoica, I.: Wide-Area Cooperative Storage with CFS. In: Proceedings of the Symposium on Operating Systems Principles (October 2001)

13. Rowstron, A., Druschel, P.: Storage Management and Caching in PAST, a Large-Scale, Persistent Peer-to-Peer Storage Utility. In: Proceedings of the Symposium on Operating Systems Principles (October 2001)
14. Martin, R., Culler, D.: NFS Sensitivity to High Performance Networks. In: Proceedings of the International Conference on the Measurement and Modeling of Computer Systems (May 1999)
15. Ng, W.T., Hillyer, B., Shriver, E., Gabber, E., Ozden, B.: Obtaining High Performance for Storage Outsourcing. In: Proceedings of the USENIX Conference on File and Storage Technologies (January 2002)
16. Le, K., Bianchini, R., Nguyen, T.D.: A Cost-Effective Distributed File Service with QoS Guarantees. Technical Report DCS-TR-615, Department of Computer Science, Rutgers University (August 2007)
17. Apache: Apache Axis, `http://ws.apache.org/axis/`
18. Olson, M.A., Bostic, K., Seltzer, M.I.: Berkeley DB. In: Proceedings of the USENIX Annual Technical Conference, FREENIX Track (June 1999)
19. Berkelaar, M.: LP_Solve, `ftp://ftp.es.ele.tue.nl/pub/lp_solve/`
20. Almeida, J.M., Krueger, J., Eager, D.L., Vernon, M.K.: Analysis of Educational Media Server Workloads. In: Proceedings of the International Workshop on Network and Operating Systems Support for Digital Audio and Video (June 2001)
21. Crandall, P.E., Aydt, R.A., Chien, A.A., Reed, D.A.: Input/Output Characteristics of Scalable Parallel Applications. In: Proceedings of the ACM/IEEE conference on Supercomputing, IEEE Computer Society Press, Los Alamitos (1995)
22. Wong, W.M.R., Muntz, R.R.: Providing Guaranteed Quality of Service for Interactive Visualization Applications (poster). In: International Conference on Measurement and Modeling of Computer Systems (June 2000)
23. Thain, D., Bent, J., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H., Livny, M.: Pipeline and Batch Sharing in Grid Workloads. In: Proceedings of the IEEE Symposum on High Performance Distributed Computing, IEEE Computer Society Press, Los Alamitos (2003)
24. Nieuwejaar, N., Kotz, D.: The Galley Parallel File System. In: Proceedings of the ACM International Conference on Supercomputing, ACM Press, New York (1996)
25. Wang, F., Xin, Q., Hong, B., Brandt, S., Miller, E., Long, D., McLarty, T.: File System Workload Analysis for Large-Scale Scientific Computing Applications. In: Proceedings of the IEEE/NASA Goddard Conference (April 2004)