Persistent storage for distributed applications

Richard Golding and John Wilkes

Storage Systems Program
Computer Systems Laboratory
Hewlett-Packard Laboratories, Palo Alto, CA

Research work in supporting distributed applications has traditionally focussed on the dynamic part of their interactions—that is, network communication paths. It's our belief that although these are important, their very transience means that they are much less valuable in the long term than the persistent state that these applications manipulate and leave behind. Such state can be enormous—tens of terabytes are not uncommon for large-scale commercial applications, which are frequently constructed from suites of federated, distributed applications. Such systems are themselves classic examples of a distributed application composed from a set of cooperating pieces. As with network communications, the persistent storage medium must be well-behaved, in the sense of providing predictable behavior, so that applications do not interfere with each other. We believe that quality of service (QoS) guarantees, and ways to automatically reason about resource provision to meet them, is the key to building effective and useful storage services.

1 Introduction

Composing distributed applications requires media through which the composed applications can communicate. Most research in this area has focussed on dynamic interactions, such as network connections and object invocation. This work has resulted in a preliminary understanding of what is required for the interactions to work well: things like network quality of service (QoS) and compositional correctness matter.

Although dynamic interactions are important, their very transience means that they are much less valuable in the long term than the persistent state that these applications manipulate and leave behind. The persistent storage can be provided by a database system, a file system, or a low-level storage service—and we are concentrating on the latter.

Persistent storage is, in our opinion, best provided as a general external storage service, rather than as part of other specialized services. Building one general system means that we can solve many hard problems in one place, rather than re-implementing them many times. A general service also makes it possible to connect many different kinds of applications using it.

The challenge is to make a general storage service useful. This requires that the value of the compositions it enables outweighs the costs of using it—including the difficulty of writing components that use it, the cost of the hardware required, and the cost of maintenance or management while it is being used.

The system must exhibit predictable behavior to meet these requirements. It is essential that each component can be built without having to reason about the behavior of every other component that might be in the system, now or in the future. This becomes especially important in large distributed systems, where the number of components can amplify small application misbehaviors into serious problems. Predictability also means that the system needs to be continuously available, secure, and incrementally expandable. The lack of support for any one of these introduces potential exception situations that components must account for on their own.

Our research work focuses on storage systems that provide predictable performance, with special emphasis on the provisioning of the persistent storage. We emphasize three aspects:

- a well-defined notion of service level to support specification and measurement of QoS guarantees, and the ability to reason about provisioning to meet them;
- design and configuration systems to support that provisioning; and
- runtime systems to enforce, measure, and improve it.

We use abstract, application-specified QoS requirements, along with measurements of how they are being met, to drive resource allocation decisions. In this way we automate many

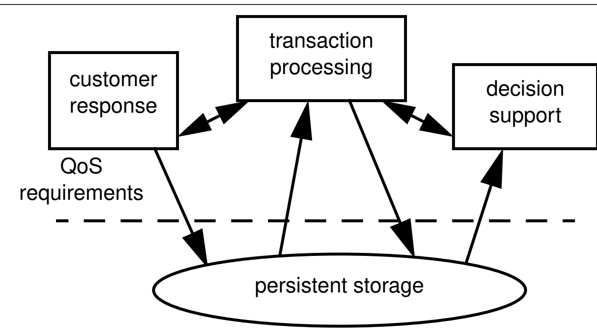

Figure 1 — a suite of applications communicating through persistent storage.

of the management activities that are required to use current storage systems, thus reducing its cost of use. This also insulates application components from the particular behaviors of specific devices, and makes the system capable of supporting the needs of a wide range of applications.

We believe that the design, construction, and maintenance of such storage services is a complex, difficult problem well worthy of attention. From a research perspective, they offer exciting challenges in continuous availability; providing hard quality of service (QoS) guarantees (and the associated real time design issues); coping with enormous scale—both physical and logical; handling gradual evolution as its component parts are replaced or upgraded.

Further, we believe that the internals of large-scale storage systems are themselves classic examples of a distributed application composed from a set of cooperating pieces—and a commercially extremely important one at that.

We discuss some of those research issues in this paper, and present the tack we are taking to address them.

2 A storage service model

We advocate a simple, well-behaved abstraction based on a block-level interface to storage with QoS guarantees. The abstraction is simple so that it can be implemented easily with good cost and performance. The service is well-behaved in the sense that applications negotiate QoS levels for their communication with storage, and the service does not allow other applications to compromise agreed-upon service levels.

Figure 3 shows the abstraction. In it, storage is organized into *virtual stores,* large-granularity (several megabytes) chunks of byte-addressable storage. Each virtual store has a set of QoS requirements attached to it, defining the aggregate performance expected of the data, along with reliability, security, and capacity requirements. When an application goes to use the virtual store, it opens a *connection* to the store, specifying the performance that will be required of the particular session. The connection carries a *contract,* which specifies both the performance needed from the storage system and the behavior that the application will follow. Once a connection has been established, the application issues individual read and write requests.

Virtual stores have limited operational semantics: reading and writing blocks of data, with simple, powerful consistency and atomicity guarantees when the store is concurrently used by multiple hosts. This is the common denominator model on which more complex services, such as databases or file system, can be built, and it matches the semantics provided by storage devices today. We have excluded features such as locking and caching, which we believe can be better implemented by other system layers. We are developing a formal model to specify these semantics more exactly.

Both the virtual store and connection abstractions hide from the application the actual resources that are used to provide for QoS requirements. This decouples the application from details of the storage service, and gives the service the freedom to automatically manage resources—which makes the provisioning problem tractable, and frees the application component designer from device-specific concerns. The connection model also provides the basis on which the system can ensure non-interference between applications using storage: each application need only be concerned about its own performance contract with the storage system, and not be concerned with the behaviors of other applications or with interference from management tasks internal to the service.

The requirements attached to virtual stores and connections form a hierarchy. Each individual IO operation is associated with a particular connection, and uses resources allocated to that connection. The operation is admitted if it falls within the application behavior specified in the connection's contract. If the operation is outside that behavior, they are processed on a best-effort basis, or may be rejected. Likewise when a connection is opened, it uses resources allocated to the associated store, and is admitted if the aggregate performance of all connections to its target store, after adding the new connection, is within the overall requirements for the store. New connections that do not fit within those requirements can be admitted on a best-effort basis—

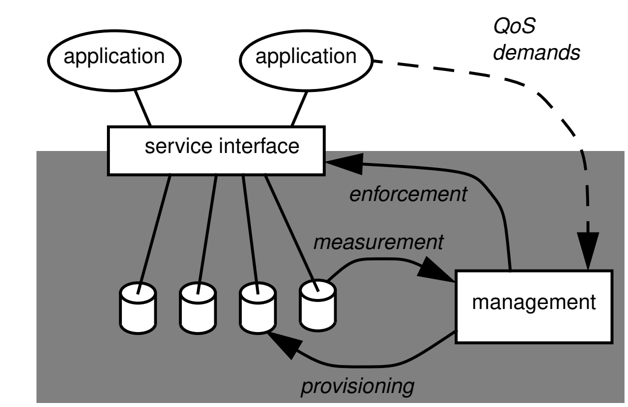

Figure 2 — using QoS requirements and measurements to drive and effect provisioning decisions.

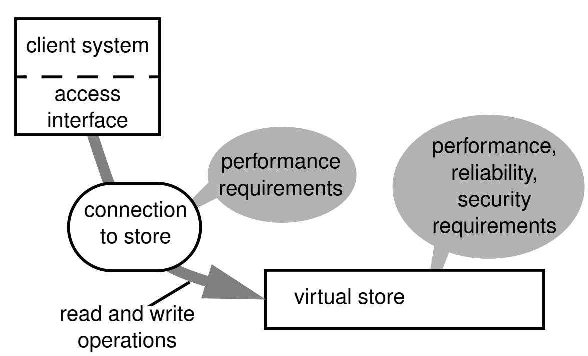

Figure 3 — abstract view of storage interconnection.

meaning that the admission will occur if resources can be found to support the stream, though once the stream is admitted the IO operations in it will get guaranteed performance.

The QoS for a store can specify reliability and security aspects as well as just performance.

3 Our research

There are many areas that need to be developed in order to build a storage service with QoS guarantees. We picked three areas to focus our attention on. They are: how to specify QoS levels; techniques for reasoning about providing resources to meet requested QoS levels; and runtime mechanisms that support resource provisioning.

3.1 Specifying QoS levels

Connections and stores, in our model, both have QoS requirements specified for them. The kinds of requirements that can be expressed must be broad enough to cover a wide range of applications, simple enough that they can actually be determined for real applications by real programmers, and specific enough that resource needs can be accurately determined.

We have developed a method of QoS specification that meets these needs. In it, applications' QoS levels are expressed as a list of named attributes; these can be written in a form like

{{capacity 10MB} {requestRate 21.5}}

We separate the static parts of service specification from the dynamic part. The static part is true of the data whether or not it is being used; this includes attributes such as reliability and capacity. The dynamic part specifies how the data will be used, and is in turn separated into application behaviors, such as request rate and reference locality, and data performance, such as latency.

The dynamic usage of a store varies over time, both in the long term as different collections of connections become active, and in the short term as the load on a single connection varies, perhaps because of burstiness. Meeting performance guarantees in this environment is most easily done by overprovisioning: allocating enough resources to a store or connection that any transient load can be satisfied. This, however, often leads to seriously underutilized resources when bursts are rare.

The QoS specification we use provides mechanisms that enable efficient resource utilization. The first, *phasing*, addresses long-term behaviors: it indicates what connections can be active at the same time, and which cannot. This reflects the way many applications go through phases of behavior (hence the name). The relationship can be given as a probability, in which case the QoS guarantees will be met probabilistically.

The second mechanism addresses short-term behaviors, such as bursts. The application behaviors and performance expectations can be specified as probability distributions, which are in turn used to provide probabilistic performance guarantees.

Our prior work on this area also discusses how device capabilities can be specified in a similar way [Borowsky97].

There is a final part of our specification that gives requirements for the system as a whole rather than for individual components. These typically are expressed as goal functions that compute a numeric goodness metric from the state of the system. Examples include the cost of the system and the balance of load across devices.

To date we have defined a set of QoS attributes that appear to model several complex database workloads well. (Note that many workloads for which QoS has been traditionally specified—such as continuous media streams—have very simple specifications.) We have built tools that semi-automatically extract these specifications by measuring a running copy of the application.

3.2 Reasoning about provisioning

We use requested QoS levels, device capabilities, and system goals to drive automatic resource allocation (provisioning) mechanisms.

We decided to frame the provisioning problem as a constrained optimization problem: determine an assignment of work to devices so that the system-wide objectives, such as cost and load balance, are as good as possible while respecting the constraints that each store's or connection's QoS must be met. This approach has allowed us to use standard optimization heuristics based on bin-packing.

We have had to extend these heuristics in a few ways. First, we had to adapt the techniques to work well with a large number of constraints, many of which are poorly behaved. For example, the response latency for a particular connection does not necessarily increase when more work is added to a disk. Second, we are adapting some standard heuristics to handle on-line resource allocation, especially that caused by incremental changes in the system or workload. Finally, we are investigating how to make the heuristics scale well up to the sizes needed for petabyte-scale storage services.

We have produced a static capacity-planning tool, and demonstrated that it can configure systems as well as humans can, and in a small fraction of the time. Experiments have

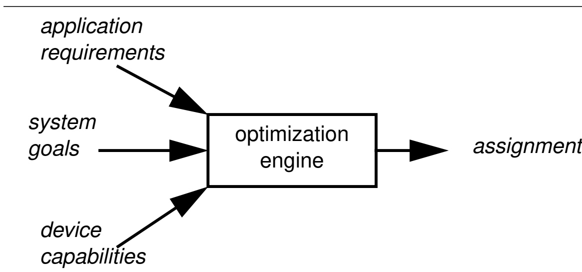

Figure 4—using an optimization engine for provisioning.

shown that the configured systems properly provide enough resources to each application to meet performance requirements, and that even slight decreases in resources allocated degrade application performance below the specified levels, which argues that we are getting good allocations. We also have preliminary heuristics for handling on-line resource allocation and scaling—though much remains to be done.

3.3 Runtime support

We use runtime support mechanisms to effect the provisioning decisions made by the optimization engine.

There are two separate parts to the runtime support, as shown in Figure 5. One part consists of the access path from the application to storage devices, and the second are the management services, including the optimization engine. The management services make provisioning decisions and record them in layout metadata; the access path uses the metadata to determine what resources to use in processing an IO request. The hosts use an optimistic cache coherence mechanism, based on version tags, to ensure that they are using accurate metadata even while the management services are migrating data from device to device.

Another way to look at this separation of concerns is that the management services implements admission control and resource allocation policies for stores and connections, while the rest of the access path handles only individual requests. This means that the access path is concerned with scheduling requests, enforcing the behavior that the application has stated it will follow, and enforcing security.

The management services include the optimization engine, a migration engine, and a metadata repository. The optimization engine makes decisions about what resources should be allocated to different stores. These decisions are expressed as requests to the migration engine, which determines when to execute these requests. When the migration engine determines that a connection from one device to another can be admitted without interfering with other guarantees, it arranges for the two devices to communicate directly to rearrange data. The metadata repository keeps track of the actual current state of the system, including the transient state as data is being copied.

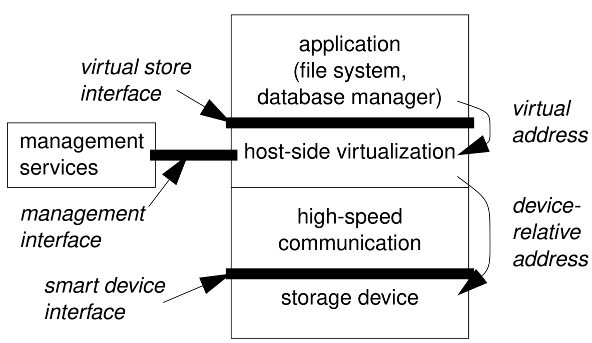

Figure 5 — the layers in the Palladio architecture, including major interfaces and address translation.

The optimization engine and migration engine are lightweight and distributed: parts can fail and restart without loss of accuracy or function. The metadata repository, on the other hand, is stateful, and so it is highly replicated for fault tolerance and scalability, using standard replication, commitment, and failure suspicion protocols.

The access path includes the host-side virtualization layer, which uses and caches metadata that it gets from the metadata repository to map virtual store address onto chunk addresses; the network, which provides high-speed, flow-controlled data movement, borrowing from work including Hamlyn, VIA, and disk-directed IO; and the storage device, which is a disk or disk array, but with a smart device interface that provides scheduling mechanisms to support QoS requirements and support for cross-device update atomicity.

There is a third part to the runtime system not shown in Figure 5: the feedback loop by which devices and host inform the management services about important events that it should react to.

4 Distributed storage as a composed application

The components in our system—the parts of the management services, plus hosts and storage devices—are composed to form the overall system. The composition is achieved using a few basic services.

Network communication. They use a connection-oriented transport that provides QoS guarantees, and layer higher-level flow-control and presentation protocols on top of it.

Transactions and atomic commitment. Resources must be allocated in a fault-tolerant way. The metadata repository must likewise support agreement among multiple replicated copies. We are building these high-level activities on top of a simple transaction mechanism, which uses a distributed atomic commit protocol internally.

Failure detection. The atomic commit protocol, among other things, is built in turn upon a failure detection (actually suspicion) mechanism.

Loosely-synchronized clocks. We use timestamps throughout the system to allow components to reason about temporal relationships among events.

Cache coherence. Each host caches a copy of the metadata it needs, as mentioned earlier. The data can be moved on the fly, however, and so we are using an optimistic cache coherence mechanism, based on version tags stored with data on disk, to ensure that the application is always reading from or writing to the right place.

Naming and location. Each component of the system is separately nameable and addressable. Since we are implementing this system using FibreChannel, we are basing naming and addressing on the FibreChannel world-wide unique name (WWN) mechanism. Components inside the management services use a distributed location service for finding mobile metadata objects.

Event propagation. Each system component can report about significant events to the management services, so that the management services can choose how to react. This is built on top of an event distribution and filtering service.

5 Related work

Many systems have separated storage into its own service.

Distributed database systems provide the richest semantics for persistent storage. Examples include the myriad of commercial client-server databases including Oracle, Sybase, Informix, and IBM products, and also a large number of research systems. Database services tend to be specific to their model of data, and hard to use for other purposes.

File-level services provide distributed access at a file-level granularity. Examples of these systems include the CMU NASD work, NFS, AFS, Echo, CIFS, and the Amoeba file services, among others. These systems work in terms of small-granularity entities owned by one user, and usually provide comparatively rich semantics. Specificity is the cost of this approach: the distributed file system is often appropriate only for a narrow range of workloads. This specificity has not prevented some of these systems from becoming widespread.

Distributed block services provide large-granularity block-addressed storage areas, with the simplest reasonable interface to the storage so that it is useful to as many kinds of applications as possible (at the cost of requiring intelligence in the applications.) Examples include Datamesh, Cambridge storage servers, and the DEC SRC Petal system. Centralized large disk array systems such as those from EMC and StorageTek provide shared block-level storage services.

Many other researchers have investigated how to provide QoS guarantees in networks, in process scheduling, and in storage. We have drawn on a very large body of this work.

6 Summary

The system we are building is ambitious in its scope, and hence there are a number of significant open research areas. (Collaboration invited!) Some of them include:

- What kind of QoS is appropriate for mixed workloads? While continuous media have simple rate and jitter requirements, suites of transaction processing and scientific applications can have complex phasing behaviors, and there can be the requirement that resources be best-effort over a certain minimum, but fairly distributed among multiple connections.
- How can rich QoS requirements be achieved in host interfaces, networks, and devices? That is, what are the specific admission control and scheduling algorithms at each level? This becomes challenging when fair-best-effort traffic is to be supported, for example.
- How can storage provisioning best be connected to outside resource management policies?
- How should security be implemented?
- Are shared file and database systems in fact easy to build on this interface? Is there a need for additional function (e.g. data synchronization operations)?
- How can optimization algorithms best be scaled up to handle very large problems?
- How can optimization algorithms best be made reactive?

We believe that a distributed block storage interface is a good building block for distributed applications; it's also an interesting application in its own right, which must be composed of subparts. Our experience has been that QoS is the key to a good storage system, and we look forward to an exciting debate on this.

References

[Borowsky97] E. Borowsky, R. Golding, A. Merchant, L. Schreier, E. Shriver, M. Spasojevic, and J. Wilkes. *Using attribute-managed storage to achieve QoS.* 5th Intl. Workshop on Quality of Service, Columbia Univ., New York, June 1997.