Filing system interfaces to support distributed multimedia applications

Stephen Childs
University of Cambridge Computer Laboratory
New Museums Site, Pembroke Street,
Cambridge CB2 3QG UK
Stephen.Childs@cl.cam.ac.uk

Abstract

As multimedia applications become part of mainstream computing, storage systems have to deal with many different file types, each with their own usage patterns and resource requirements. However, conventional file systems do not recognise this heterogeneity, and treat all stored data alike. By using file classes and interfaces to describe files of different types, useful information can be provided by the application and used by the file system to choose appropriate storage policies.

An architecture is proposed by which file systems can provide support for different file classes in a flexible and extensible manner. This architecture is based on the Multi-Service Storage Architecture (MSSA) and will be implemented on the Nemesis operating system, which provides the resource guarantees necessary for multimedia applications.

1 Introduction

Everything is a file. Or at least if you use UNIX it is. The file abstraction has been one of the most widely used concepts in the design and implementation of operating systems and applications. We contend that the "everything is a file" model has had its day. Filing systems need a more flexible model for representing the objects they store to applications. This model should be based on well-defined interfaces which allow salient features of the object class to be exposed to higher levels, while hiding implementation details that are not relevant.

All storage objects are **not** the same. They differ from each other in both the semantic and physical requirements they make on a storage system. For example, a user's document file needs to be easily locatable by a plain-text name and should be accessible as quickly as possible. On the other hand, stored e-mails do not need to appear to the user as normal files in a directory as long as an e-mail application can locate and display them. On the physical side, video and audio clips need to be stored in a manner which facilitates their timely play-back and user files must have a fast response time.

The use of a well-defined interface for each file type will allow applications to specify their exact needs and enable the system to make policy decisions which take into account their unique properties.

However, requiring applications to know all the implementation details for their particular file type would make it very difficult to compose applications in a straightforward way. But this may not be necessary. It is possible to distinguish certain generic file classes e.g. flat files, structured files, continuous-medium (CM) files. And at a finer level, applications use many standard file formats (e.g. MPEG for video and AU for audio), so it makes sense to provide system facilities to enable many applications to use these file types.

In conventional systems, each application provides full support for their own file type on top of the limited services provided by the file system. It would be much more appropriate to support file types as part of the file system, using managers to control each generic type e.g. CM, structured files, and implementing translators to provide any format-specific processing necessary.

This leads to a layered approach to file system design, with a clear division of responsibilities between different layers. Value-adding services such as indexing, directory services and filters can be added to the system and associated with new file classes. These value adding services use the functions provided by the system, and extend them with their own extra facilities.

These are issues which have to be addressed today more than ever before. The continuing increase in the power of desktop computers, coupled with the growth of the Internet and the rise in the expectations of ordinary users is generating a demand for new types of applications. These applications must process video and audio data of different types, and will be expected to be able to interact in a distributed system, over a LAN or the Internet. The issue of system support for the many different file types which result needs to be dealt with in a consistent way, rather than on an *ad hoc* basis from application to application.

In this paper, I identify a number of desirable features for a "modern" file system (2) and briefly describe an operating system which can support them (3). I then describe the benefits of providing information about file types at a system level (4), and present an architecture which allows this (5). An implementation framework (6) and sample application (7) are presented and access control (8) issues briefly discussed.

2 Layered approach and QoS parameters

Two very desirable properties for a modern, extensible filing system which can support existing and future file types are as follows:

- A layered structure
- Provision for Quality of Service

A layered approach to file system design provides the flexibility needed to support the diverse storage needs of applications. In the past, filing systems have been vertically integrated, incorporating policy about access patterns and usage implicitly and providing a very restrictive interface to clients. In the layered model, common functions are grouped together into layers and exported via interfaces.

This approach allows higher-level services to be created which can then use these interfaces to access the lower-level functions in a simple manner without needing to concern themselves with potentially complex implementation details.

The lowest levels of the system are concerned with the physical storage of data on the media. They implement mechanisms for disk scheduling, block allocation and other low-level functions. They would then provide abstractions such as byte streams to the higher levels. The protection mechanisms to implement security policies of higher layers must also be located in this layer.

Directory services, indexing, security and other logical features are built on top of the physical storage layer. The layered approach means that multiple services of this type can coexist, as long as they use the features of the physical layer to perform storage.

High-level, type-specific translators take responsibility for accepting high-level requests such as open, close, play, record, from applications and translating them into the parameterised requests needed by the lower layers. For example, when an application makes a request to play a video file, the translator would negotiate a guarantee from the storage subsystem for a session delivering one frame every 1/30s and then initiate playback.

This introduces the second requirement for a modern file system: the need for a mechanism to specify and guarantee resource allocation. This is usually referred to as Quality of Service (QoS) and is increasingly important as CM files become more integrated into standard applications.

CM files differ from conventional files not only in their large size and sequential access patterns, but also in their need for timely delivery of data at a specified rate. Users are sensitive to glitches in the data they view, and so streams which are in progress must be guaranteed the resources they need to continue transferring data at the required rate.

In a QoS-aware system, mechanisms are provided which allow applications to specify the amount of resources they need. Based on their requests, they negotiate a contract with the system which specifies these resource needs in detail. Once a contract has been negotiated, the system guarantees the application that it will continue to receive these resources for as long as it needs, or until a specified termination time.

The progress of technology and the increased processing power and network bandwidth it has delivered have led to a growing interest in multimedia applications. This in turn has brought Quality of Service issues to the fore, as these applications require resource guarantees to deliver acceptable performance to users.

For guarantees to be effective, they need to apply end-to-end, incorporating network, disk, CPU and memory. Previously these functions were not part of operating systems, but a new breed of operating system is emerging which incorporates primitives to specify QoS requirements, and mechanisms to enforce them.

3 Nemesis: a suitable platform

The Nemesis operating system [5], currently under development in Cambridge and at other sites, has been designed with support for QoS as a primary system function. It provides primitives for specifying resource requirements and a mechanism for accounting them. This is realised as follows:

- As much functionality as possible is implemented at application-level in shared libraries.
- A highly modular system design allows resource usage to be correctly charged to individual applications, rather than to device drivers or kernel code.
- Resource guarantees given to applications can be honoured because the system structure prevents crosstalk between applications.

This integrated support for QoS is already important to provide reliable multimedia performance, and will be increasingly important in distributed systems, where users may be using resources on more than one machine.

Another factor that makes Nemesis a suitable platform for a storage system of this type is that it is based on the concepts of interfaces and objects. All Nemesis system components are described in an interface definition language known as MIDDL. This language allows the state and methods which define objects to be described in a standard, platform-independent way. The fact that this is integrated into the system, rather than only at a programming-language level, makes it easy to integrate new components closely with existing ones.

4 System support for file-types

Currently, support for different file types is implemented on an *ad hoc* basis by the applications that need them, or in libraries. The only abstraction provided by the filing system is that of a flat file (or stream of bytes), and each application must do all the extra work necessary on top of this very limited abstraction. So, in conventional systems, CM facilities are implemented over the flat-file facilities provided by the file system. Those trying to build specialised CM systems have realised the limitations of this approach and often use file systems which are optimised for the periodic and sequential access patterns of CM files and are unsuitable for traditional files. ([1], [7])

When CM support is built on top of a conventional file system, some very undesirable side-effects result. In the event of contention between file accesses, the file system has no policy of arbitrating between applications that takes account of their specific characteristics. For example, at higher levels, an application accessing a short text file obviously has very different needs from one playing back a long video file, but at the file system level, these are treated exactly the same by caching, scheduling, block allocation and other facilities.

The system simply has no way of knowing that the text file needs a quick response time or that the video file has a periodicity which must be maintained. These are properties of a specific kind of file, and as far as it is concerned, there only *is* one kind of file. There are many reasons why the file system should have access to information about the type of files it is dealing with. For example, in the case of overload, if the file system "knew" about the various file types it was serving, the system would be able to use information about specific files in use to degrade service to applications gracefully in a manner that produces as little disruption to clients as possible.

Another example of the problems this indiscriminating approach can cause is seen when general policies for caching are implemented which apply across file types with very different access patterns. Typically, the file system caches all data that passes through it. This is appropriate for traditional, small files, as there is a good chance they will be accessed again soon. However, when large CM files are cached in this way, their size means that they repeatedly fill and overwrite the entire cache, thus rendering it useless. This implies the need for multiple different mechanisms and policies to coexist in the storage system, which can deal with individual file-types in suitable ways.

Existing systems are almost all optimised for either "traditional" file access patterns (usually derived from a limited set of traces) or continuous media files. They do not cope well with a mix of heterogeneous file types, and in some cases do not even permit this. It is clear that policies and mechanisms differ widely between file types, and it is not possible to provide a general solution which can deal efficiently with all file types.

What is needed is an *integrated file system*, in which strategies for multiple file types coexist, and requests are treated differently depending on their file class. The Symphony file system [8] provides a good example of this approach.

5 The Multi-Service Storage Architecture (MSSA)

The MSSA [6] is a storage system which has been designed with a layered model to provide extensibility and flexibility. The basic feature taken to be common to all file classes is the low-level storage of data. This function is provided by the Physical Storage (PS) layer. Higher-level, logical features such as file classes, naming and location are provided by the Logical Storage (LS) layer. Support for a number of generic file classes is provided in the system.

The basic low-level unit of allocation is the Byte Segment (BS). This is a logical sequence of bytes, maintained on disk as a list of extents. The interface to all byte segments is the same, but they may be implemented in different ways. For example, the underlying medium for one BS may be a disk, while another may be implemented on a tape, CD-ROM, etc. Thus, low-level implementation details are dealt with by the storage service, and the client accesses the facility through a simple interface. Byte Segment Containers group byte segments, and provide a way of classifying byte segments implemented in different ways.

File classes are a logical function (LS layer). Each file class exports a different interface to allow type-specific features to be controlled. However, **all** storage of file data is done through the byte segment interface. There are file classes which are considered part of MSSA, such as Flat Files, Structured Files and Continuous Media Files, and developers can also create new file classes, which are managed by their own value-adding services.

File classes are supported by custodes. A custode can be thought of as a server which manages objects. Each custode manages only one class of objects and each object is managed by only one custode at a time. A custode is the smallest unit of distribution and different custodes may be on the same machine or on different networked machines.

Each custode manages one file class. These are generic classes. For example, text files and data files are examples of flat files, while MPEG video and CD audio are CM files. Each custode provides a number of high-level operations suitable to the file class it supports. These might include such operations as Play and Record for CM files, SelectMember for structured files, etc.

Translators are the means by which the system supports different encoding formats such as MPEG, AVI, etc. When an application starts a session with a Continuous Medium File Custode (CFC), it specifies the name of the translator for its data type. The translator is responsible for performing any format-specific processing and for setting up and coordinating the session with the (low-level) BS custode. audio, effects for video, etc.

The MSSA's built-in classes may be extended in two ways to create Value Adding Services. Firstly, new classes may be created, and secondly, operations of existing classes may be specialised. This may be described as providing the following object-oriented functions:

- Abstraction: New file classes, providing new file service interfaces and implementations, may be built on top of the existing MSSA custodes. (For an example of this, see section 7).
- Specialisation: A value-adding service provider (or custode) may specialise selected operations of an MSSA custode or other value-adding custodes below it in the hierarchy.

6 Design issues

Although MSSA was originally designed as a network storage service, it has many features which are also suitable for a workstation file system. It is proposed to build a native file system for Nemesis, based on the concepts of the MSSA. This file system will provide a flexible service to applications, which will allow them to manipulate many file types in a simple and efficient way.

6.1 Nemesis file system concepts

In Nemesis, the disk is represented to clients by an abstraction known as the User-Safe Disk (described in [2]). The basic idea behind the USD, as behind all Nemesis device drivers, is to provide a safe way for clients to transfer data directly to the device, without needing to go through a server. (Incidentally, this approach is similar to that taken in many other file system projects, such those on Network Attached Secure Disks (NASD)[3], and seems to be widely recognised as a good way to achieve low-latency transfers and remove the server bottleneck.) The motivation behind this approach is to allow accounting to be done on a per-client basis, by avoiding the need to interact with other modules.

The USD is the representation of the disk seen by the client, and as such exports a very basic, low-level interface. Transactions are performed on USD Extents, which are just ranges of blocks on the disk. When a file system binds to a disk, it must register a callback routine. This routine normally provides the protection and translation information for the file system.

When a client attempts to access an extent on the disk for the first time, a fault occurs, and the USD invokes the file system's callback routine. This routine checks the client's permissions for these blocks, and returns a success value to the USD. The USD caches this information, so that on subsequent accesses to the same blocks by this client, no interaction with the file system server is necessary.

Interaction between the client and the USD is done through streams. A client that wishes to read or write to the disk requests a stream. Once this stream has been set up, the client uses it for all its reads and writes, which are accomplished by sending packet descriptors describing the blocks to be read/written down the stream.

6.2 MSSA on Nemesis

In the MSSA, all storage is performed through the Byte Segment Custode. The byte segment is the most basic abstraction available to clients. This may be rate-based, in the case of segments used to store CM files. In this case, the BSC will perform the necessary read-ahead and buffer management to guarantee that the data is transferred at the rate requested. Thus the BSC controls all physical storage, acting as a storage manager for the whole system.

The proposed implementation for Nemesis is to have the BSC acting as a file system above the USD. Clients which wish only to use the BS services, i.e. for byte addressable, uniquely identified segments, will talk directly to this module. Other modules will be built to provide functions such as directory services, indexing, etc. These are the value-adding services of MSSA. Also, modules supporting different file types will be constructed. The function of these is to accept simple requests from applications (like OpenMPEGFile) and translate them into the more specific information needed by the BSC to set up the appropriate rate-based session and reserve resources.

The aim of this is to make it easier to construct applications which use continuous media files. By providing support for rate-based storage and retrieval in system modules, applications can use straightforward constructs to manipulate multimedia.

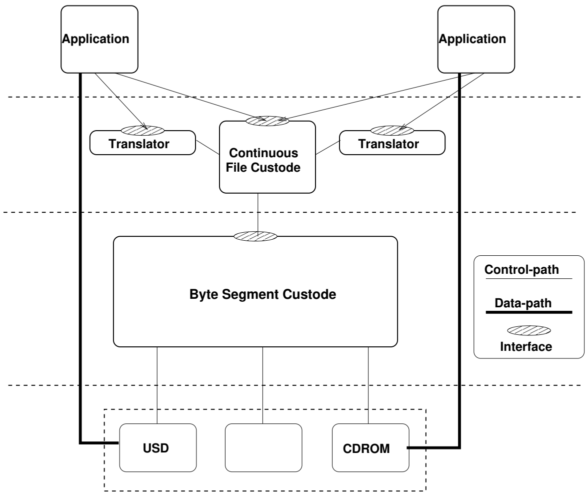

Figure 1: Architecture of MSSA for Nemesis

Figure 1 shows the proposed architecture. The normal entry point for applications is through the translator for the file format they will be using. However, if they do wish to implement a custom file format, they can access the service through the generic Continuous File Custode. In this case, they will have to provide the specific parameters for their file type.

The CFC sets up a control connection with the BSC to negotiate the resources necessary to guarantee timely playback. Once suitable parameters have been established, the BSC creates a stream between the application and the disk. This stream is then used by the application for data transfer. The stream maintains its guaranteed resource level for as long as it is open. This is ensured by the USD.

Information about access rights to segments is held in the BSC, and is translated into low-level information on extents. When an application tries to access a particular area for the first time, the USD will invoke the callback function registered by the BSC. This will perform the necessary access control checks, and return the extent permissions which apply. This is then cached by the USD, and used when validating subsequent accesses to those extents by the same client.

6.3 Use of interfaces in Nemesis

Nemesis system components are defined using an interface definition language known as MIDDL. (While similar in functionality to other IDLs, MIDDL offers a number of extra constructs to deal with low-level and operating system interfaces. Its type-system is unfortunately different from those of now-standard IDLs such as OMG IDL.) This makes a clear mapping from the MSSA constructs onto a Nemesis implementation straightforward. The interface exported by the BSC is defined in MIDDL, and then the methods defined are implemented in C. Stub files for RPC can also be automatically generated from the interface definition.

File classes are also represented by interfaces, and the server which manages them implements their methods. Thus different implementations of translators and custodes are possible, as long as they conform to the defined interfaces.

6.4 Status of implementation

The implementation of MSSA on Nemesis is currently in its early stages. The previous implementation used specialised NVRAM hardware to permit efficient atomic reads and writes. The version currently being implemented will run on standard PCs, and so must provide similar functionality by other means. A number of other design issues also need to be addressed.

7 Sample application

As an example of how this approach may be used to compose a distributed application, take an application for recording a conference seminar. A seminar is represented as a structured file, consisting of video and audio of the speaker and any slides and meta-data such as the speaker's name, the date and location.

A new file class, Seminar is defined. Each recording of a seminar will be an instance of this class. The new file class in this case is composed of existing MSSA file classes, i.e. Continuous Medium Files for the audio and video, and Flat Files for textual data.

The class provides operations for manipulating Seminar objects. These might include operations like ShowSlides, etc. If another application wishes to use a Seminar object, it does not have to worry about how the class is implemented, and can use simple abstractions like Seminar.Slides to manipulate components of the class.

The server which implements the class is responsible for providing the mapping between operations invoked on Seminar objects and the MSSA file class operations. When a Seminar file is opened, the appropriate files will be opened via their managers, and their segments retrieved from the BSC as necessary.

Figure 2 shows the situation when a Seminar file is opened for recording. The application sends a request for a new file to be opened in Record mode to the Seminar custode. The Seminar custode then sends the appropriate cf_open calls to the CFC, passing the names of the audio and video translators as parameters.

The format-specific translators are then called by the CFC. They set up rate-based sessions with the BSC with parameters based on the timing requirements of their specific format. This creates a data stream between the source and the BSC. A session ID is also returned to the Seminar custode which allows it to interact directly with the appropriate translators.

To enable distributed applications to be constructed, control interfaces may be exported using

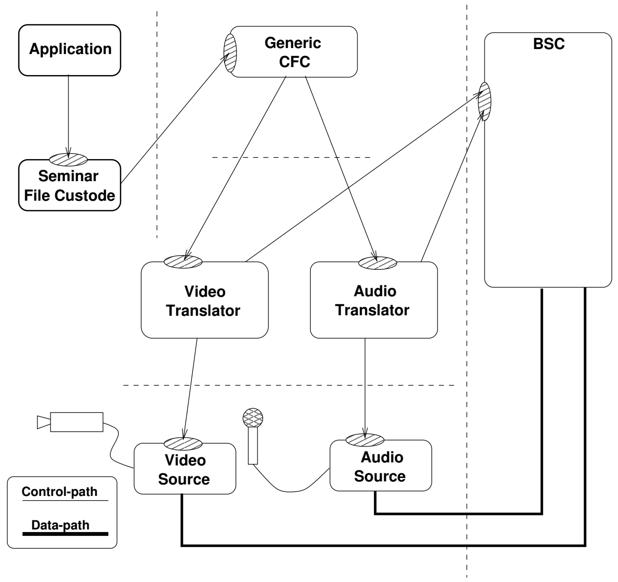

Figure 2: Seminar application

RPC. The application may run on one node, and communicate with sources and custodes on other nodes. The dashed lines in Figure 2 indicate points at which network partitions may occur. However, synchronisation issues may lead to stricter controls on distribution.

If at a later date, new features are required, for example an indexing function which indexes the seminar based on the slide being displayed, these can be implemented as value-adding clients which are inserted between the application and the MSSA. All file data would then pass through the value-adding client on storage and retrieval.

8 Access control

An issue of primary concern in the design of distributed applications and their storage services is that of access control. When many applications are running on different hosts in an open system, the limitations of standard security systems are exposed.

In order to define security policy for many applications, each of which may represent one or more users, traditional approaches which insist that access control is determined purely in terms of user identity are inadequate. They do not provide the flexibility necessary to describe the complex roles necessary to define delegation, grouping and other relationships between applications.

For this reason, it is planned to use a role based access control model. This allows a two-level approach, where each process has a unique identifier which can be authenticated and then gains other roles by interacting with other services [4].

This approach also allows value-adding services to provide protection for the objects they control. If directories (for example) are managed by a directory server, we only want to allow applications to be able to access directory objects through that server. This can easily be modelled by ensuring that only that server may issue the capabilities necessary to enter roles which can access the objects.

9 Conclusion

An approach to building distributed storage systems using strongly typed interfaces for files has been presented. In the context of a layered storage system, this allows the specific properties of each file class to be supported by the system and allows new types to be easily integrated. It also allows the system to select appropriate policies based on the type of requests it sees.

Any such system should also incorporate a means of specifying and guaranteeing resources in order to allow continuous media files to be stored and retrieved in a timely fashion. It is proposed to provide this by building the system on top of the Nemesis operating system, which provides low-level QoS functions and uses interfaces to define system components.

References

[1] David P. Anderson, Yoshitomo Osawa, and Ramesh Govindan. A file system for continuous media. *ACM Transactions on Computer Systems*, 10(4):311–337, November 1992.

[2] Paul Barham. *Devices in a Multi-Service Operating System.* PhD thesis, University of Cambridge Computer Laboratory, October 1996.

[3] National Storage Industry Consortium. Network Attached Storage Devices. `http://www.nsic.org/nasd`.

[4] Richard Hayton. *OASIS: An Open Architecture for Secure Interworking Services*. PhD thesis, University of Cambridge Computer Laboratory, 1996. Technical Report No. TR399.

[5] Ian Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal on Selected Areas in Communication*, 14(7):1280–1297, September 1996.

[6] Sai-Lai Lo. *A modular and extensible network storage architecture*. PhD thesis, University of Cambridge Computer Laboratory, 1994. Technical Report No. TR326. Distinguished Dissertations in Computer Science, CUP 1995.

[7] P. Venkat Rangan and Harrick M. Vin. Designing file systems for digital video and audio. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 81–94, 1991.

[8] Prashant J. Shenoy, Pawan Goyal, Sriram S. Rao, and Harrick M. Vin. Symphony: An integrated multimedia file system. In *Proceedings of SPIE/ACM Conference on Multimedia Computing and Networking*, January 1998.