An Extensible Integration Framework for the Capture of Multimedia Assets

by

Jeffrey Hu

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1999
[June 1999]

Author ....[illegible]..........
Department of Electrical Engineering and Computer Science
May 21, 1999

Certified by ....[illegible]..........
David Karger
Associate Professor
Thesis Supervisor

Accepted by ....[illegible]..........
Arthur C. Smith
Chairman, Department Committee on Graduate Students

An Extensible Integration Framework for the Capture of Multimedia Assets

by

Jeffrey Hu

Submitted to the Department of Electrical Engineering and Computer Science
on May 21, 1999, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

We present an extensible integration framework in the spirit of the purchase and integrate approach to building asset management systems. Applying the concept of layers, this framework provides a systematic way to integrate heterogeneous software components. Our approach differs from other approaches in that we add a resource management layer to mediate between applications and services. This mediation allows computationally intense, distributed services to be managed by the system. Using this framework, we integrated a set of content services that support a system that captures video streams.

Thesis Supervisor: David Karger
Title: Associate Professor

Acknowledgments

I would like to thank Tryg Ager and Robin Williams for giving me a place and a project for my thesis at the IBM Almaden Research Center. Without their support and encouragement, this thesis would not have been possible.

While I was at IBM, Bill Graham also provided many valuable technical insights which have inevitably made their way into this thesis.

Aaron Van Devender and Ryan Jones began the work on the video capture system. I had the luxury of continuing where they left off at the end of the summer.

I would like to thank my brothers Roger and Stanley who read parts of this thesis. I also could not have done this thesis without the support of my parents.

Lastly, I would like to thank David Karger for being my thesis advisor and providing valuable guidance as I was starting my work on the thesis.

Contents

1 Introduction **10**

1.1 Problem 10

1.2 The Integration Framework 12

1.3 Video Capture System 13

1.4 Thesis Overview 14

2 Background **15**

2.1 Remote Procedure Call 15

2.2 Distributed Object technology 16

2.2.1 CORBA 16

2.2.2 Java RMI 18

2.2.3 Critique of Distributed Object Technologies 19

3 The Integration Framework **21**

3.1 Design Philosophy 21

3.2 Services 23

3.3 Resource Management 23

3.4 Applications 26

3.5 Infrastructure 28

4 Services **29**

4.1 Basic Description 29

4.2 Components of a Service 31

4.2.1 Service Interface 32

4.2.2 Event Adapter 32

4.2.3 Wrapper Adapter 33

4.3 Service Operations 34

4.3.1 Service Invocation 34

4.3.2 Service Registration 36

5 Resource Management **39**

5.1 Basic Description 39

5.2 Handling Application Requests 40

5.2.1 Request Acceptance 40

5.2.2 Job Scheduling 42

5.3 Resource Manager Registration 45

6 Video Capture System **46**

6.1 System Model 46

6.1.1 Data Model 47

6.1.2 Task Model 49

6.2 Content Services 50

6.2.1 MPEG Encoding 50

6.2.2 Key Frame Extraction 51

6.2.3 File Conversion Service 52

6.2.4 VDF-to-XML Conversion Service 52

6.2.5 XML-to-SQL Conversion Service 53

6.3 Service Integration 53

6.4 Applications 54

6.4.1 Work Order 54

6.4.2 Task Completion 55

6.5 State of the System 56

7 Conclusion **57**

7.1 Future Work 58

A Service Specifications of the Video Capture System 59

B System Infrastructure 63

B.1 Universal Unique Identifier Generators 63

B.2 Distributed File System 64

B.2.1 Construction 65

B.2.2 Uniform Naming 65

B.3 Event Channel 66

B.4 Socket Copy Service 67

List of Figures

2-1 Sample IDL for a Bank object 17

2-2 ATM invoking a withdraw method on a remote Bank object (CORBA version) 18

2-3 ATM invoking a withdraw method on a remote Bank object (Java RMI version) 19

3-1 Layers of services, resource management nodes, and applications . . . 22

3-2 Components of the Resource Management Layer 24

3-3 Larger System with Multiple Resource Domains 26

3-4 Service Invocation Process. Steps required for invocation are numbered in order of operation. 27

4-1 Components of a Service 32

4-2 Two examples of XML elements 35

4-3 Elements demonstrating the uniqueness restriction of XML elements . 36

4-4 Sample Service Return Value (includes Invocation Parameters) 37

5-1 Sample Request Acceptance 41

5-2 Waiting for a Request On A Job Handle 42

5-3 Resource Manager State for a FIFO Scheduling Policy 43

5-4 Resource Manager State for an Actual Time Scheduling Policy 44

6-1 Simplified Version of Video Capture System Data Model 47

6-2 Task Model of Video Capture System 49

6-3 System Diagram of the Video Capture System 54

B-1 Sample 128 Bit Universal Unique Identifier 64

List of Tables

A.1 MPEG Capture Service Specification 59
A.2 File Conversion Service Specification 60
A.3 Key Frame Extraction Service Specification 61
A.4 VDF-to-XML Conversion Service Specification 61
A.5 XML-to-SQL Conversion Service Specification 62

B.1 Comparison of Names per Naming System 66
B.2 Sample VDFSFileMap file 67

Chapter 1

Introduction

In constructing a computer system, we should take advantage of the current wealth of existing hardware and software components. Instead of developing systems from scratch, a more cost effective approach would be to purchase components from outside vendors and integrate them into a complete system. With this approach in mind, we have created an extensible integration framework for building multimedia asset management systems.[1]

This thesis discusses the integration framework through which we can systematically build multimedia asset management systems. Within this framework, multimedia services can be easily integrated into a distributed system. In particular, we focus on how these services are managed as shared resources. We used this framework to construct a system which automatically captures and stores video streams.

1.1 Problem

In traditional software engineering, a system is developed by breaking it into functional and specified modules. These modules are either implemented or broken into subsequently smaller modules. As the modules are implemented, they are tested and integrated into larger modules until the complete system is formed.

Object-oriented programming languages provide semantics that allow systems to

[1]This framework was developed at the IBM Almaden Research Center.

be constructed in a modular way. By associating methods with data, objects can be constructed and manipulated behind a well defined interface. Implementation details are hidden behind the object interface from the rest of the system. This encapsulation simplifies interaction with other software modules, reducing complexity and the likelihood of bugs being introduced in the integration process.

Distributed object technology applies the concepts of object-oriented programming languages to a distributed environment. In object-oriented languages, operations are performed on objects by calling a method of an object. The execution of the method occurs in the same thread and address space of the calling procedure. However, the semantics of distributed objects are slightly different. When a procedure invokes a method of a distributed object, the calling procedure is a client and the callee object is a server. The client-server distinction between the caller and the callee must be made because they execute in different threads and address spaces.

In a distributed system, clients make requests on servers. The server can be single threaded, or it can be multithreaded. If the server is single threaded, only one client request can be handled at a time. For a multithreaded server, the number of available threads determines the number of requests that can be handled simultaneously. A multithreaded server often has upper limits on the number of threads it can have. This server may change the number of active threads up to this limit depending on the number of requests it sees. Thus, a server can manage how its own resources are being used.

Because distributed objects are also clients and servers, the issue of resource management inevitably arises. When a client invokes a method of a remote object, it must communicate its request to a thread of the server where the object is located. The server thread is responsible for handling the request and invoking the method on the object. Deciding how threads are used in the server is a resource management decision that cannot be avoided.

Given existing distributed object technology, we have developed an integration framework. We chose to use a distributed object technology since it provides a well developed base on which we can integrate components. The integration framework is developed for constructing systems in the domain of multimedia asset management.

As the name suggests, the purpose of an asset management system is to manage assets. Assets are persistent information objects that are composed of a set of smaller information objects. From the point of view of a user, an asset is a unit of information. A common example of an asset management system is a library. A library manages many assets in the form of books. A book is an asset that is composed of smaller information objects such as words, pictures, author names, and titles.

An asset management system also has a set of services that can be performed on assets. In multimedia asset management systems, services are computationally intensive because multimedia data tends to be very large. Certain multimedia tasks such as the compressing of video files can take minutes or hours depending on the length of the video. Thus in the domain of multimedia asset management systems, services must be managed or else the performance of the system will degrade poorly if more than a few requests for a single service arrive.

To deal with the resource management issues, we add a third layer between clients and servers. This layer removes the resource management issues from the servers and places them in the middle layer. Moving responsibility for resource management out of servers also simplifies the implementation of servers.

In the context of the integration framework, this layer is called the resource management layer. It provides a flexible mechanism to manage the various types of services that might be used in an asset management system.

1.2 The Integration Framework

The integration framework provides a systematic way to integrate new services into an asset management system. The framework is divided into three layers. In ascending order, the three layers are the service, resource management, and application layers. At each layer, the framework provides a set of common mechanisms that assist in functions such as resource discovery and resource scheduling. Given these mechanisms, applications can be easily created from a distributed set of services.

Looking at the requirements for a computer system, we think the system should be easy to extend. For example, we might decide to add replicas of a service to handle larger workloads. We might also add entirely new types of services to the system. Another thing we could do is upgrade service components to take advantage of features available in newer versions of the component. In any case, these changes should be easy to make.

Simple client-server architecture fails to completely meet all these extensibility requirements. If a server is replicated, we must change the clients to be aware of the replica in order to take advantage of the extra server. Similarly, if a server is removed from the system, all clients using that server will have to be modified.

The mediation of requests between applications and services by the resource management layer provides a layer of indirection that allows services to be location independent. When an application invokes a service, it does not need to know the location of the service at the time of invocation. Services and applications can thus be modified without affecting each other's operation. This decoupling allows a systems that is constructed under the integration framework to be extensible.

To summarize, the integration framework provides a set of common mechanisms that support a systematic way of building a distributed multimedia asset management system. The framework describes three layers which allow applications and services to be modified with minimal impact on each other. A system constructed from the framework can thus be easily extended by adding new services or applications.

1.3 Video Capture System

We built a video capture system using this integration framework. Providing a concrete scenario with real requirements, the video capture system automatically captures broadcasted video streams and performs operations needed for a multimedia asset management system. These operations include the generation of metadata, the creation of compressed formats of the video stream, and the storing of this data and metadata.

1.4 Thesis Overview

In this thesis, we present the architecture and implementation of the extensible integration framework, including the resource management layer. We discuss the video capture system as a case study for the framework.

In the next chapter, we present some of the current approaches to software integration. Chapter 3 presents a general overview of the framework. Chapters 4 and 5 discuss the service and resource management layers. Chapter 6 discusses the video capture system. We conclude in chapter 7.

Chapter 2

Background

In this chapter, we explore a few existing technologies and approaches used in integrating distributed applications. These technologies will illustrate some of the common issues that an integration framework for a distributed system must support.

2.1 Remote Procedure Call

The Remote Procedure Call (RPC for short) is a simple mechanism which allows a program to invoke an operation on a remote server. To the programmer, an RPC has semantics similar to that of a normal procedure call. Arguments can be passed to the remote procedure when it is invoked. Return values are received after the remote procedure has completed. RPC thus provides convenient and familiar semantics for programmers who wish to write programs that perform remote operations [1].

RPC hides most of the inherent networking issues involved in performing remote operations. Issues such as reliable network transport and messaging are handled through stubs at the client and server. These stubs ensure that operations are performed at most once and that the proper parameters are sent between the client and server. The type of the parameters is also preserved between the client and server by transmitting the data in a way that includes the type. This method of transmitting typed data is called *marshaling*.

2.2 Distributed Object technology

Distributed object technology applies the concepts of object-oriented programming to a distributed environment. The semantics for using objects in distributed object technologies are analogous to those of RPC. In distributed object technologies, methods can be invoked on a remote object similar to how remote procedures are invoked. Although analogous, the semantics of distributed objects do not perfectly match those of RPC or object-oriented languages. However by containing semantics for both RPC and object oriented languages, distributed object technologies provide a useful way to integrate software applications in a distributed environment.

We now briefly explore two existing distributed object technologies and focus on how they support software integration. These two technologies are Java RMI and CORBA.[1]

2.2.1 CORBA

CORBA is a distributed object technology created as a collaborative effort of many software vendors. Collectively known as the Object Management Group (or OMG for short), this consortion developed a specification for semantics to create and use distributed objects. This specification can be found in [3].

The basic idea behind CORBA is to have every distributed object associated with an intermediary called an ORB.[2] When a distributed object needs to interact with another distributed object, it communicates through this intermediate ORB. The ORB acts on behalf of the distributed object by communicating directly with the target object or another ORB.

A system implementer uses CORBA by defining an interface of an object. The interface is defined in IDL, a declarative language for specifying interfaces to an object. The interface of an object includes the types and names of arguments and the return values of a method. The object may also contain typed fields which can be accessed

[1]A more complete survey and comparison of distributed object technologies can be found in [2].
[2]ORB stands for Object Request Broker. CORBA stands for Common Object Request Broker Architecture.

```
module BankModule {
    interface Bank {
        void withdraw(in short account, in float amount);
        void deposit(in short account, in float amount);
        float balance(in short account);
    }
}
```

Figure 2-1: Sample IDL for a Bank object

by name. Figure 2-1 shows a sample specification for a `Bank` object in IDL.

This interface described in IDL is compiled by an IDL compiler into stubs, skeletons, header files, and other runtime structures needed by CORBA. These structures provide mechanisms that allow operations to be performed on remote objects. The actual behavior of the remote object is left to be implemented in some programming language by a programmer.

Like the stubs in RPC, stub and skeletons generated by the IDL compiler ensure that methods are invoked at most once and parameters are properly marshaled between a client and a server. Unlike RPC, CORBA stubs and skeletons communicate with each other using an ORB as an intermediary instead of contacting each other directly.

The reason an ORB is used is to allow greater flexibility in specifying how an object can be called. One such feature allowed by an ORB is dynamic invocation of remote methods. Ordinarily a remote method invocation is statically created. In this case, the method invocation is compiled as a call to a specific method. CORBA also allows dynamic invocations. In this case, the ORB receives a request that is interpretted to determine which method of the object should be called. Dynamic invocations thus require an intermediary such as an ORB.

Figure 2-2 shows the sequence of events in which an `ATM` object invokes the `withdraw` method on the `Bank` object. The `ATM` invokes the `withdraw` method on the stub of the `Bank` object. The stub then uses the ORB to forward the call to the skeleton of the `Bank` object. The `withdraw` method is finally invoked when the

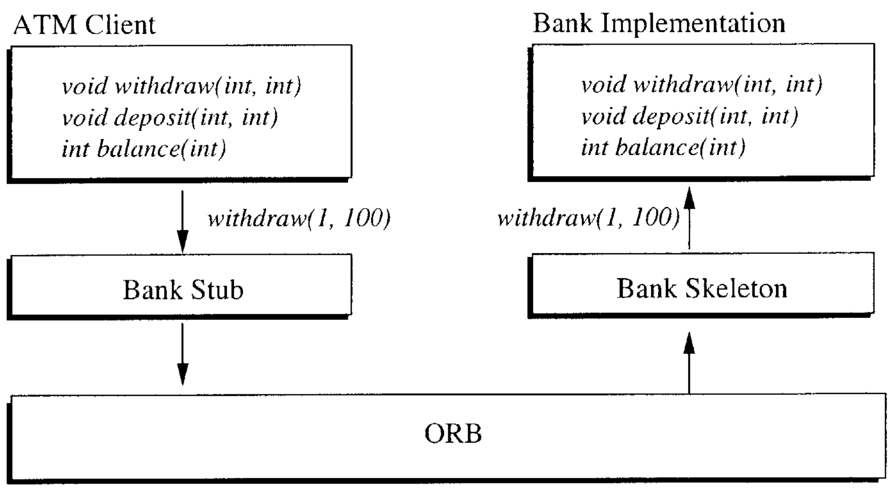


Figure 2-2: ATM invoking a withdraw method on a remote Bank object (CORBA version)

`withdraw` method of the implementation is called by the skeleton.

2.2.2 Java RMI

Java RMI[3] provides a lightweight mechanism for using distributed objects [4]. Like CORBA, Java RMI stubs and skeletons enforce at most once semantics and the proper transmission of parameters between the client and server. Unlike CORBA, Java RMI does not use an intermediary to perform remote invocations. Java RMI stubs communicate directly with skeletons.

When using Java RMI, an implementer first decides which methods of an object to export to remote clients. The exported methods are specified as a Java language construct known as an interface. An RMI compiler is then used to process the Java class, producing a stub and skeleton for the class.

The same `ATM` and `Bank` example is shown in figure 2-3 using Java RMI as the distributed object technology. To invoke the `withdraw` method of the `Bank`, the `ATM` invokes the `withdraw` method of the stub. The stub then connects directly to the skeleton of the `Bank` object. The skeleton calls the corresponding `withdraw` function in the running implementation of the `Bank` object.

[3]RMI stands for Remote Method Invocation

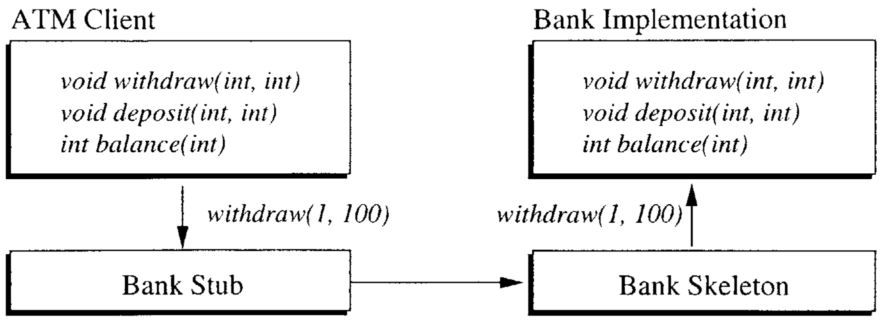


Figure 2-3: ATM invoking a withdraw method on a remote Bank object (Java RMI version)

2.2.3 Critique of Distributed Object Technologies

Both distributed object technologies support encapsulation by forcing implementers to use well defined interfaces. With CORBA and Java RMI, object-oriented semantics provide a useful abstraction mechanism through which software may be integrated.

As mentioned earlier however, the semantics of an object-oriented programming language do not map perfectly to those of distributed objects. In an object-oriented programming language, objects are invoked by procedures or objects through the use of local methods. When a method of an object is called, the caller is assured that the invocation occurs. However, since a remote method call on a distributed object involves a client making a request to a server object, the client may have to block until the server object can handle its request.

CORBA provides a mechanism through which a server may create a number of *servants* to handle multiple requests. When a request arrives for an server object, the ORB uses a structure that is called an *object adapter* to dispatch the request to one of the servants. By controlling the number of servants in the system, resources can be managed. However, all servants by definition are on the same server. As a result, resources that are distributed cannot be managed with this mechanism.

Java RMI objects, on the other hand, do not even have the ability to handle more than one remote method invocation at a time. When a Java object is created, it can handle only one invocation at a time. Multiple Java objects could be created, but without an intermediary between the client and the server object, the client has no way of knowing whether the object is already handling a request. In this case, a client can do no better than picking one of the objects at random to perform a remote invocation and hope that the object is not already handling a request.

In the domain of multimedia asset management systems, the resource management mechanisms in CORBA are inadequate and are nonexistent for Java RMI. Since services in a multimedia asset management system are generally computationally intensive, we need to distribute services across multiple machines and manage them as resources. As a result, we need a resource management layer which can manage distributed resources. The resource management layer is thus the primary extension this thesis presents to distributed object technology.

Our integration framework is implemented on top of distributed object technology. Specifically, we implemented the framework using Java RMI. The reason we chose Java RMI over CORBA is due mostly to practical reasons. Being a lightweight mechanism, Java RMI is widely available and the interfaces are standard across all platforms. CORBA, on the other hand, has a large set of features, but each implementation varies slightly from the others. We elected to use Java RMI in creating the integration framework to avoid the implementation complexities of CORBA.

Chapter 3

The Integration Framework

In this chapter, we present an overview of the integration framework. The framework is composed of three layers: the service layer, the resource management layer, and the application layer. First, we begin with the design philosophy of the framework by describing how the layers relates to one another. We then describe how each of the three layers in the framework work in general, leaving later chapters to cover it in greater detail.

3.1 Design Philosophy

When automating the management of assets, three types of operations are performed. The first is to determine what tasks, if any, need to be undertaken. The second is to determine how to perform the tasks. The third is to actually perform the tasks.

In building an extensible integration framework, we have kept these three types of operations separated amongst the three separate layers. At the top layer are applications which are written by programmers to perform some set of *tasks*. We define a task to be an operation that requires a service to be invoked in order to be completed. When a task needs to be performed, an application submits a request for a service to the resource management layer.

In the resource management layer, agents accepts requests and determine how to perform the requested tasks. They make decisions determining what machines

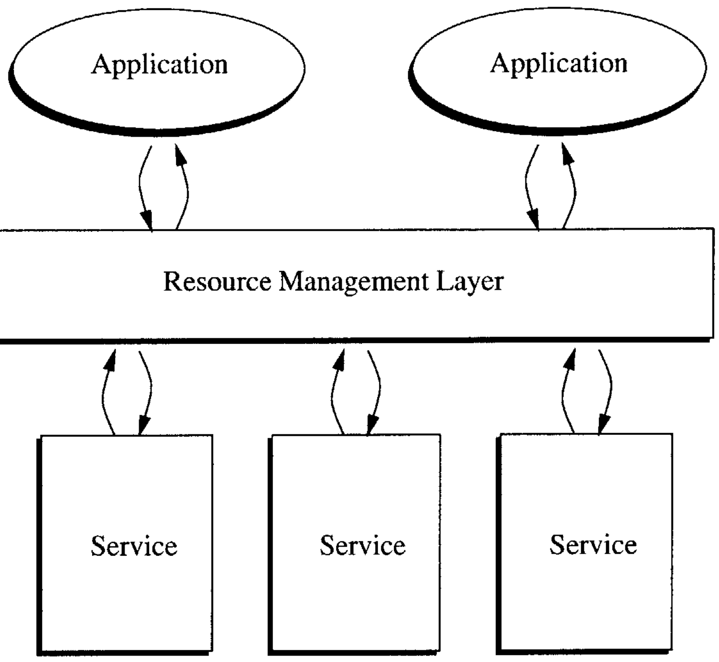


Figure 3-1: Layers of services, resource management nodes, and applications

to use and when to perform a task. An application could specify requirements on a request. These requirements are taken into consideration. Once an agent in the resource management layer determines when and which services to use, the services are invoked.

For example, consider a system that has a service that captures video streams. Suppose also that two instances of the service exist. If a user wanted to capture a television broadcast occurring from 2:00 P.M. to 2:30 P.M., he could use an application to request that the broadcast be captured. The application would then send the request to the resource management layer. As long as one of the services is available for the requested time interval, the request can be satisfied. A resource management agent is responsible for checking to see that the request can be satisfied and accepting the request if the check succeeds. The request is rejected if the check fails.

A number of nice properties arise with this layering of functions. The first advantage is that the set of issues that must be considered is limited to a particular layer. Implementers of services need only concern themselves with making sure that a service performs its task properly. Application implementers need not worry about how services are implemented nor how they are invoked. They need only to concern themselves with the high level logic determining what services should be invoked and submitting the requests to a resource management agent.

A second advantage of this layered approach is that the interaction between components is more flexible. The resource management agents provide a layer of indirection between applications and services. Applications are not hard coded to using specific instances of a service. We can add services dynamically and make them instantly available to applications. This flexibility is important in creating an extensible system.

3.2 Services

We define *services* to be typed, computational resources. In order for a service to be invoked, an application must submit a request matching the type of the service. An instance of a service can handle just one request at a time. If more than one request arrives, the instance cannot handle the request until it completes the its current request. To allow for better performance and reliability, multiple instances of a service may be added to the system.

Services are added by registering them with the resource management layer. Service registration is accomplished by using an event mechanism that is included with every service in the framework. Services are discussed in greater detail in chapter 5.

3.3 Resource Management

The purpose of the resource management layer is to serve as an intermediary between applications and services. Applications submit requests for services to the resource management layer. The resource management layer determines how to fulfill these requests and invokes the appropriate services.

In addition to moderating requests from applications, the resource management layer keeps track of services in the system. When services are ready to announce their

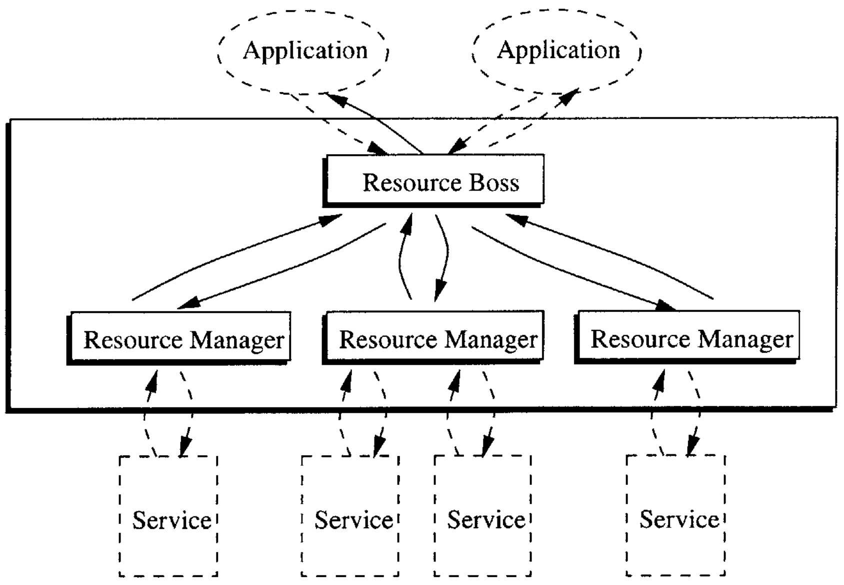


Figure 3-2: Components of the Resource Management Layer

availability to applications, they register themselves with the resource management layer.

When requests for services exceed the number of available instances of a service, they are put in a queue and scheduled for a later time. Many different types of services can be used in the system, and thus we saw a need to support different scheduling policies. The simplest scheduling policy that we support in the system is FIFO.[1] As this name suggests, requests are handled in the order that they arrive.

We also support a scheduling policy that we call *actual time*. This policy is useful for scheduling requests that need to be performed at a specific time of day. For example, a request may require that a video encoding service record the nightly news from 11:00 PM to 11:30 PM each day. Such a time constraint is specified in what we would call actual time. The scheduling policies are described in greater detail in chapter 5.

Two types of agents make up the resource management layer. These agents are *resource managers* and *resource bosses*. Figure 3-2 shows how these components are organized topologically in the resource management layer.

[1]First In, First Out

A resource manager keeps track of services and schedules requests for its services according to some scheduling policy. Each resource manager embodies a single scheduling policy and can manage multiple types of services. When a resource manager receives a request, it handles the request by invoking services according to its stated scheduling policy.

A resource boss performs the function of a directory service in our framework. A resource boss keeps track of the resource managers and directs requests from applications to a resource manager. The resource boss only directs requests to a resource manager that has a service matching the type of the request. After the request is routed to a valid resource manager, it is handled according to the resource manager's scheduling policy.

Services are registered with a single resource manager. Resource managers are registered with a single resource boss. We assume that services of the same type register with the same resource manager although this is not a hard requirement. Resource managers should be added depending on how many different scheduling policies are needed. Only one resource boss is usually needed except in certain cases where the number of services is so large that multiple administrative domains are needed.

Resource bosses can be used to implement administrative *resource domains*. A resource boss defines a resource domain since it has a list of registered resource managers. Each resource manager in turn has a list of registered service instances. By this chaining of registrations, the resource boss defines a set of resource managers and service instances. This set determines a resource domain. When an application contacts a resource boss for a service request, the request must be satisfied by a service in the resource domain.

If two resource domains exist, they can be joined together into a larger resource domain by registering one resource boss with the other. The registration of resource bosses is restricted to avoid overlapping resource domains. The simple rule when registering a resource boss to another resource boss is to make sure that their defined resource domains do not intersect.

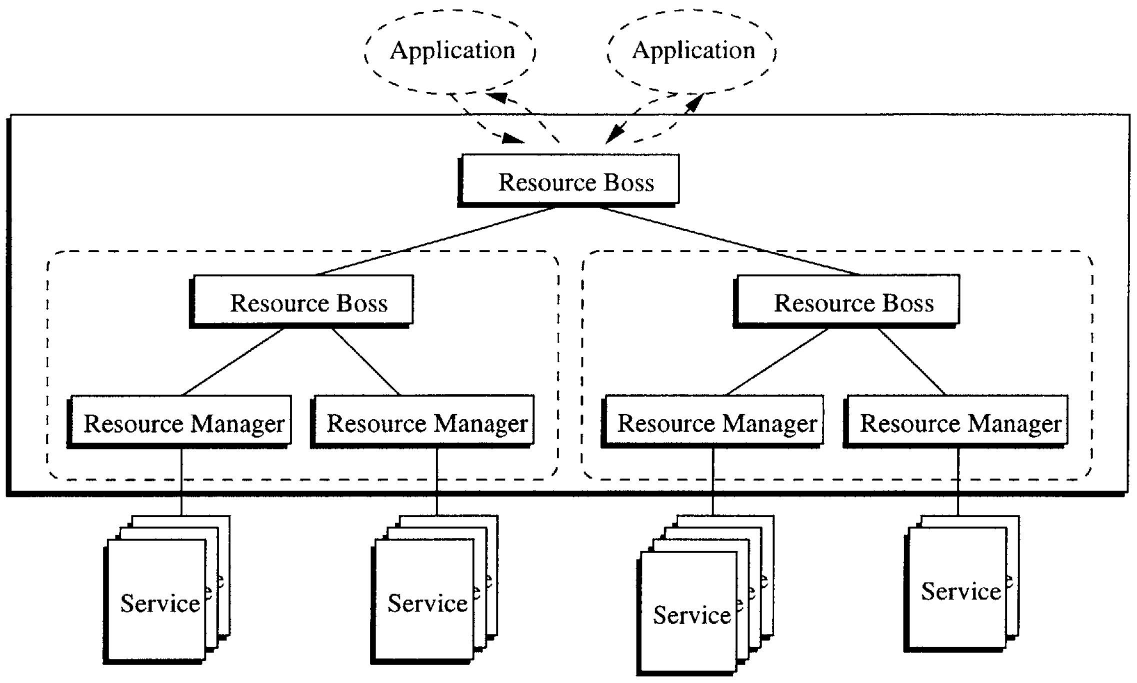


Figure 3-3: Larger System with Multiple Resource Domains

Resource domains are useful for large computer systems where an owning organization may want to partition administrative access to services. One reason why such a partitioning might be desired is to allow different departments in an organization to maintain a set of services. The services would be needed by the entire organization, but each department would be responsible for maintaining the services specific to their domain of work.

Figure 3-3 shows an example of a larger system with services that are arranged into multiple administrative *resource domains*. The resource management layer is covered in greater detail in chapter 5.

3.4 Applications

Composing the highest layer of the integration framework, applications are written by programmers to perform useful tasks for end users. When building an application, a programmer decides *what* services need to be performed. Questions such as *which* instance of a service and *when* is left to the resource management layer.

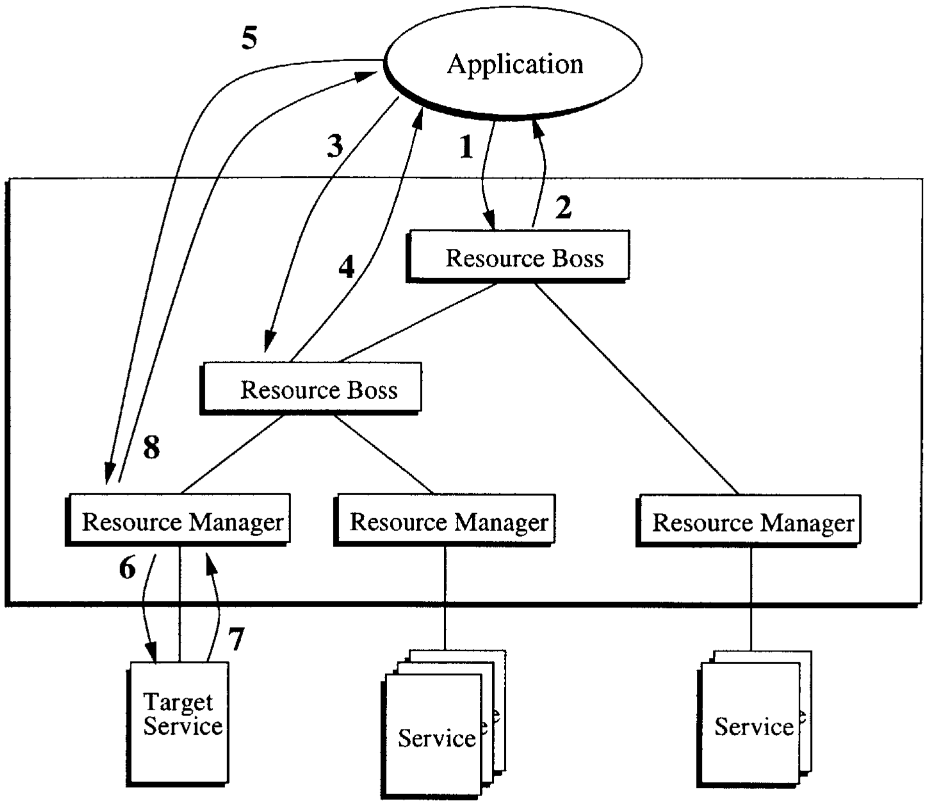


Figure 3-4: Service Invocation Process. Steps required for invocation are numbered in order of operation.

Figure 3-4 shows the steps that an application must take to submit a request for a service. First, an application asks the local resource boss for a resource manager that can handle its request. If the resource boss refers the application to another resource boss, the application poses the same query to that resource boss. The application continues to query resource bosses until a suitable resource manager is returned. When a resource manager is finally returned, the request is submitted to the resource manager, and the application waits until the request is complete.

The steps that an application must take to submit a request can be summarized into a couple of simple rules:

1. Ask resources bosses to find a resource manager that can handle the request until either an error or a resource manager is returned.

2. When a resource manager that can handle the request is found, submit the

request to the resource manager and wait until it is completes.

3.5 Infrastructure

Thus far we have identified the primary differences between the components and layers of the system. Despite their differences, many shared mechanisms are needed between components of the same layer and components between layers. In the integration framework, we implemented a set of common mechanisms that we call infrastructure.

Examples of infrastructure that are used in the integration framework are a universally unique identifier generator, a distributed file system, an event channel, and a data transfer service. We discuss each of these infrastructure components in greater detail in appendix B.

Chapter 4

Services

Services are shared computational units that perform a domain specific operation on an asset. In terms of the integration framework, they compose the lowest layer.

In this chapter, we discuss services in greater detail. In particular, we discuss how services are implemented, how they operate, and how they can be integrated into an asset management system. This section deals almost exclusively with services and ignores the other components of the framework. For an overview for how services interact with the rest of the integration framework, see chapter 3.

4.1 Basic Description

Comprising the lowest layer of the integration framework, services perform domain specific, computational tasks for asset management. As a contrast, infrastructure includes more general, basic services that are often needed across the entire integration framework. Services typically make use of some the infrastructure components such as the event channel, distributed file system, and socket copy service.[1]

Services are implemented as objects which perform a specific function based on their type. Services with the same type perform the same function whereas services of different types perform different functions. For each type of service, multiple instances of a service may exist. Hence, the type actually names a group of instances

[1]A more detailed description of the infrastructure components that we use is in appendix B.

of a service. For example, an application that requests a service `MPEGEnc` is actually asking for an instance of a service with a type `MPEGEnc`.

Because services perform computational tasks, they exhibit demands on a host machine for resources such as CPU time, memory space, hard disk space, and network bandwidth. Services are thus limited resources. Sometimes a service requires specialized hardware devices, such as a video capture card. In this case, a service can only run on machines that have the required devices. Such a service is further limited by the number of the specialized devices available on a particular machine.

Services can be connected and removed from the system dynamically as the rest of the system runs. An instance of a service connects to the system by registering with the resource management layer. Once a service registers with the resource management layer, it announces its availability to the system. To remove a service, one needs only to stop it. In the current implementation, we rely on the resource management layer to detect when a service stops. We determine that a service has stopped when it fails to handle requests.

The two basic types of service operations are invocation and registration. A service invocation is initiated by the resource management layer on behalf of an application. Registration is used by services to announce their availability as a shared resource to the asset management system.

As in RPC, services are invoked synchronously. An invoked service does not return until it finishes performing a task. The thread in the application that submitted the request can wait until the invocation is complete. However, we do support non-blocking semantics which allow an application thread to continue processing while a request is being processed.

Registration of services is accomplish by sending messages through an event channel. The event channel, which is an infrastructure component described in appendix B, acts as a shared bus and is basically a server which accepts messages from a set of clients called *event producers*. These messages are routed to another set of clients called *event consumers*. Barring any server crashes, every message sent by an event producer is guaranteed to arrive at most once to an event consumer. The event channel is capable of supporting an arbitrary number of event producers and consumers at one time.

4.2 Components of a Service

Services can be divided into two major parts: a software component to be integrated and a *wrapper*. We refer to the component as a *black box* for metaphorical reasons. The wrapper is used to integrate the black box into the complete system.

Recall the "purchase and integrate" approach for constructing software systems that was introduced in chapter 1. We metaphorically name a purchased component as a black box because we interact with the component only through external interface. The implementation details are generally hidden from our view by the vendor. Each of the black boxes that we purchase likely has differing interfaces. Example interfaces for a black box include a programming library, an executable, or even a graphical user interface. The number of integration possibilities increases when we consider that each black box might be developed for different platforms.

We contain this heterogeneity at the service level by forcing all services to be implemented as distributed objects with a single `invoke` method. When the `invoke` method of an object is called, the service is invoked. Some software is needed to augment a black box to actually convert it into a service. This software is called the wrapper.

A wrapper is composed of three smaller objects. These objects are called the `ServiceInterface`, the `EventAdapter`, and the `WrapperAdapter`. The `Service-Interface` and `EventAdapter` objects provide a general interface and set of mechanisms needed for each service. Hence, the same `ServiceInterface` and `Event-Adapter` is used for every service. The `WrapperAdapter` contains code that translates the invocation on the distributed object into operations on the native interface of the black box. The `WrapperAdapter` is thus different for each type of service. Figure 4-1 shows the components that make up a service.

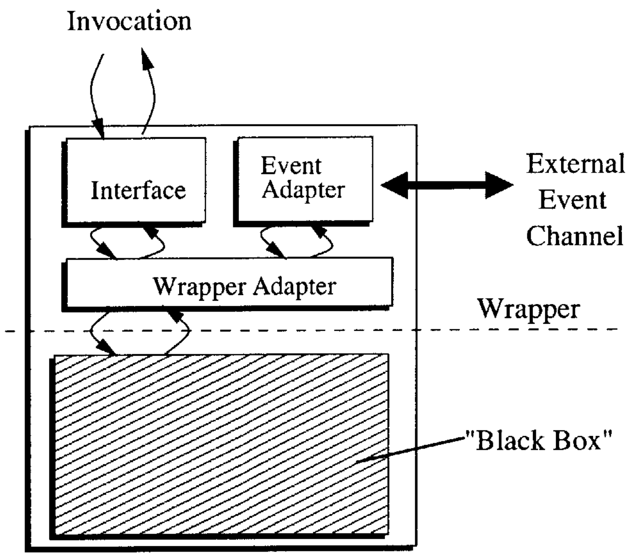


Figure 4-1: Components of a Service

4.2.1 Service Interface

The `ServiceInterface` is an object that exports the `invoke` method. Resource managers use the `invoke` method to invoke a service. When the `ServiceInterface` is invoked, it forwards the invocation to the `WrapperAdapter`.

In each `ServiceInterface`, a status variable is maintained to keep track of a service's current state. A service is in either a *busy*, *available*, or *error* state.

4.2.2 Event Adapter

The `EventAdapter` provides a way to register a service with the resource management layer. The `EventAdapter` uses the event channel described in appendix B to send and receive messages to a particular resource manager in the resource management layer.

The messages sent to a resource manager generally register the service. Registration often includes some interaction with the `WrapperAdapter`, which often contains relevant information such as the type of the service. Sometimes, a service receives a message instructing it to send a message to its resource manager. To allow such events to be handled even as a service is processing a request, the `EventAdapter` runs a separate thread.

The `EventAdapter` passes messages to the `WrapperAdapter` if the message is not related to registration. Thus, new types of messages can be defined and handled by services. Such messages might be used for communicating between services or canceling requests that are being processed by a service.

4.2.3 Wrapper Adapter

The `WrapperAdapter` is software that is specific to each type of service. When an invocation occurs, the `WrapperAdapter` translates the request from the `Service-Interface` and uses the arguments in the request to perform the appropriate operations on the black box. The `WrapperAdapter` is also responsible for propagating return values or error messages from the black box back to the `ServiceInterface`. Since the black boxes vary widely in their interface, so too must the software that interacts with it. Thus, the `WrapperAdapter` must be customized for each black box.

As mentioned in the previous section, the `WrapperAdapter` is also used by the `EventAdapter`. During registration, the `WrapperAdapter` is called upon by the `EventAdapter` to supply registration information that is specific to the service.

Thus, the `WrapperAdapter` is an intermediate between the black box and the `ServiceInterface` and `EventAdapter` components. All the code that is used to integrate the black box into the system is contained in the `WrapperAdapter`.

A number of black box interfaces are possible. One such interface might be a library of functions for the C language. For this interface, a `WrapperAdapter` would contain code that translates Java calls into C calls. The `ServiceInterface` would call the `WrapperAdapter` in Java and the `WrapperAdapter` would call the black box using its native C interface. The arguments would be converted by the `WrapperAdapter` from Java types into C types. The return values would be converted from C types into Java types and returned to the `ServiceInterface`. Such a mechanism already exists as the Java Native Interface.

Another example of a black box interface could be an executable binary that is compiled to run on a particular platform. In this case, the `WrapperAdapter` would invoke the binary through the system shell. On most platforms, a system call can be used to run a program within a shell.

A final example of a black box interface could be a graphical user interface. In this case, we would have to write a program that would interact with the graphical user interface. We could simulate events such as a mouse click or a key press using the programming interface of the graphical user interface. Such an approach is feasible although inelegant.

In all these examples, only the `WrapperAdapter` needed to be changed to integrate a component into the system. Neither the `ServiceInterface` nor the `EventAdapter` need to be modified.

4.3 Service Operations

In this section, we discuss in greater detail the operations of service invocation and service registration.

4.3.1 Service Invocation

Figure 4-1 shows the sequence of method calls involved in a service invocation. A service is invoked when a resource manager calls the `invoke` method of a `Service-Interface`. The `invoke` method of the `ServiceInterface` calls an `invoke` method on the `WrapperAdapter`. The `invoke` method of the `WrapperAdapter` then performs the appropriate operations on the black box. When the operations are complete, the `invoke` method of the `WrapperAdapter` returns back to the `ServiceInterface`.

For an invocation, a service accepts a set of parameters and returns a set of values. Each type of service accepts a different set of parameters and returns a different set of return values. As a result, the parameter passing mechanism needs to be flexible enough to accommodate all types of services.

Another requirement for our parameter passing mechanism is the ability to create data structures. Being composed from existing software components, services are relatively high level software components. High level invocations require poten-

```
<Filename>file:///news.wav</Filename>

<AudioQuality>
    <SamplingRate>44kHz</SamplingRate>
    <Mode>Stereo</Mode>
</AudioQuality>
```

Figure 4-2: Two examples of XML elements

tially larger numbers of parameters. Such parameters are usually organized in some structured format.

To build such a flexible, structured set of data, we use XML, also known as the extensible markup language [5]. XML is a hypertext language which allows data to be structured hierarchically in a human readable form. The basic data object in XML is an *element*. An element contains a *tag* and a *value*. A value can be either a string, a set of elements, or a combination of both. A tag is a name that refers to the values. For our purposes, we restrict a value to be either a string or a set of elements. Figure 4-2 shows two XML elements which correspond to each of these types of values.

We add a further restriction to the XML document model. For every value that is a set of elements, none of the tags for these elements may be same. Hence, the tag for each element is unique with respect to the other elements. This uniqueness restriction does not apply to the children of these elements. Figure 4-3 shows an XML element that violates this uniqueness restriction and an element that meets the restriction. With these two restrictions, we use a subset of the data structures supported in XML to package arguments and return values for services.

For passing parameters to and from services, we have defined a convention which makes use of the hierarchy allowed by XML. When passing arguments to a service of a particular type, the name of the root element of the XML document is that type. Thus for a service of type `ServiceA`, the name of the root will also be `ServiceA`. The next level consists of three elements named `IN`, `OUT`, and `Exception`. The `IN` element contains a set of the elements that are the arguments to be passed to the service. The `OUT` element contains a set of elements that are the return values from the service.

```
<Invalid_Element>
    <Filename>file:///news1.wav</Filename>
    <Filename>file:///news2.wav</Filename>
</Invalid_Element>

<Valid_Element>
    <Filename>file:///news1.wav</Filename>
    <Nested_File>
        <Filename>file:///news2.wav</Filename>
    </Nested_File>
</Valid_Element>
```

Figure 4-3: Elements demonstrating the uniqueness restriction of XML elements

The `Exception` element contains a message describing an error that occurred in the service.

Figure 4-4 shows an example of a request made to an audio playback service. The service requires arguments describing the name of the file to be played and the quality of the audio file. When complete, the service returns the duration of the audio and a return code.

Retrieving the values of parameters in the XML document is a simple parsing task given the proper series of tag names. We have simplified the parsing task by creating a set of functions which automatically extract the value when given a name that is constructed from the hierarchy. For example given the document in figure 4-4 and the name `AudioPlaybackService.IN.AudioQuality.Mode`, the value that is resolved will be `Stereo`. These hierarchical names are also used to set values much in the same way that they are used to retrieve values.

4.3.2 Service Registration

A service registers itself by sending a message to a resource manager. Each service registers itself with exactly one resource manager. The resource manager to which a service registers itself is determined ahead of time by an administrator. A service uses its `EventAdapter` to send an event of type `RegisterResource`. The event contains

```
<AudioPlaybackService>
    <IN>
        <Filename>file:///news.wav</Filename>
        <AudioQuality>
            <SamplingRate>44kHz</SamplingRate>
            <Mode>Stereo</Mode>
        </AudioQuality>
    </IN>
    <OUT>
        <TimePlayed>01:30:04</TimePlayed>
        <ReturnCode>0</ReturnCode>
    </OUT>
</AudioPlaybackService>
```

Figure 4-4: Sample Service Return Value (includes Invocation Parameters)

the following information:

1. the type of the service.
2. the status of the service.
3. the name of the service's resource manager.
4. a Java RMI remote reference to the service.

When the named resource manager receives the registration event, it adds a new resource to its list of available resources. The information from the registration event is included in the in this list of resources.

During the course of operation, a resource manager might be started after a service sends its registration event. In this case, the service needs to send the registration event after the resource manager is started. To make sure that the service is registered, a resource manager sends out a `CallForResource` event soon after starting. The event contains the name of the resource manager. Any service which matches the name of its resource manager with the name in the event, responds by sending a registration event.

Hence, we have two simple rules of registration for a service:

1. After a service starts, it sends a `RegisterResource` event.

2. When a service receives a `CallForResource` event that originated from its resource manager, send a `RegisterResource` event.

Given the registration rules, a resource manager may receive a duplicate registration event if the `RegisterResource` and `CallForResource` events are sent at approximately the same time. To suppress duplicate registration events, the resource manager compares the remote reference included in an event to the those in its list of available resources. When a match is found, the registration event is a duplicate and can be ignored.

Chapter 5

Resource Management

The resource management layer mediates between applications and services. Services register themselves with the resource management layer whereas applications submit requests. This layer provides indirection which is important to the flexibility of systems created with the integration framework.

This chapter describes in detail the different components and processes of the the resource management layer of the integration framework.

5.1 Basic Description

Shared resources are limited and must be managed. In the case of the integration framework, each instance of a service is a resource. Because services are the computational units of the asset management system, they are the most resource intensive operations in the system and thus should be managed.

Since the system has shared resources, multiple applications can invoke services simultaneously. When a request is accepted by the resource management layer, a resource manager must arbitrate between the different application requests in deciding how and when services are invoked.

The resource management layer is composed of primarily two types of components: *resource managers* and *resource bosses.* Each resource manager is responsible for a set of services. Each resource boss is responsible for a set of resource bosses and resource managers. Together, the resource managers and bosses handle application requests and manage the use of services.

5.2 Handling Application Requests

A request for a service may contain more than one *job*. We define a *job* to be a request for a single service. Hence a request may actually invoke multiple services. Using multiple jobs in a request allows the set of jobs to be scheduled at the same time. In such cases, tasks might have to occur simultaneously as in the case of real-time applications. Generally however, requests contain just one job.

When an application wishes to invoke a service, it submits a request to the resource management layer. This submission process is called *request acceptance.* After the request is accepted, the application has the option of waiting until the service completes. Should the application choose not to wait, it could continue processing and check for the result of the service invocation at a later time.

After a request is accepted, it is placed in a queue. The set of jobs associated with the request determine which services are needed to fulfill the request. When the requested services are available, the jobs are scheduled to run according to some scheduling policy.

5.2.1 Request Acceptance

When an application wishes to submit a request, it first locates a resource manager that can handle its request by contacting a resource boss. On behalf of the application, the resource boss looks for a resource manager that has a service that can handle the request. The resource boss is aware of resource managers in its resource domain since resource managers register themselves with resource bosses.

The contacted resource boss finds a suitable resource manager by querying each registered resource manager to see if it can handle the request. Since requests are typed objects, a resource boss sends queries to resource managers asking if they can process a request of that particular type. The entire request need not be sent. After

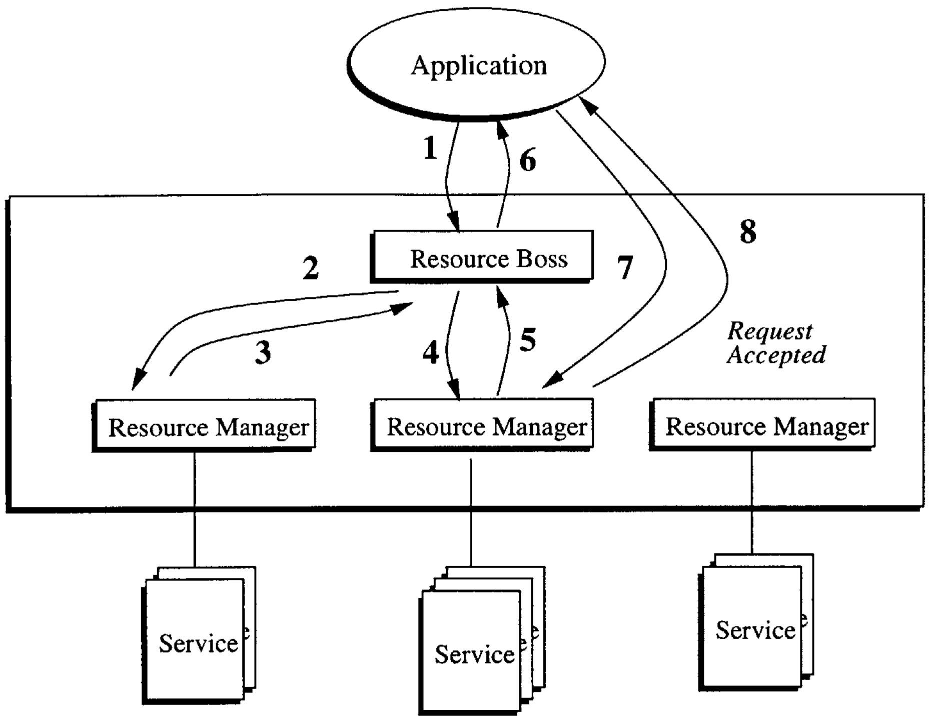


Figure 5-1: Sample Request Acceptance

a resource manager replies in the affirmative, the resource boss returns a reference to the resource manager to the application. The application can then connect to the resource manager using this reference and submit its request.

Figure 5-1 shows this process of request acceptance for a system with one resource boss and three resource managers. The second resource manager can process the application's request. Hence, the resource boss ceases querying resource managers after the second resource manager replies affirmatively. A reference to the resource manager is returned to the application in the sixth transition. The request is then accepted by the resource manager.

Once a request is submitted, the application can wait until the completion of the request. The resource manager communicates to the application the completion of the request through a `JobHandle` object. The `JobHandle` object is created and returned to the application after the application submits the request. The application can then wait on the `JobHandle` object or continue processing. If the application waits, the resource manager will notify the waiting application when the request completes.

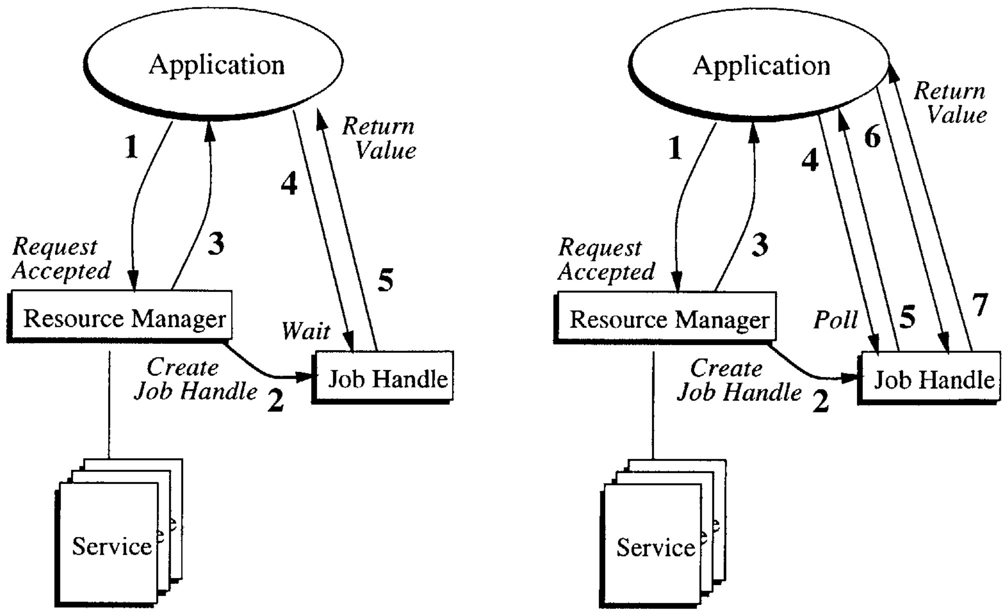


Figure 5-2: Waiting for a Request On A Job Handle

If the application does not wait, it could check the `JobHandle` later to see if the request completes. When the request completes, the result can be acquired from the `JobHandle` object.

Figure 5-2 shows two diagrams describing the sequence of events after a request is accepted. In both diagrams, the application submits a request and the resource manager creates a `JobHandle` for the request. In the diagram to the left, the application then waits on the `JobHandle` object until the request is completed. The results of the request are returned to the application after it finishes waiting.

In the diagram to the right, the application periodically polls the the `JobHandle` object until the request is complete. The results of the request are returned to the application after the request is completed.

5.2.2 Job Scheduling

Requests that are submitted by an application to a resource manager are placed in a queue to be scheduled with other requests. To schedule these requests, a resource manager must also have a list of resources and some associated data such as the type

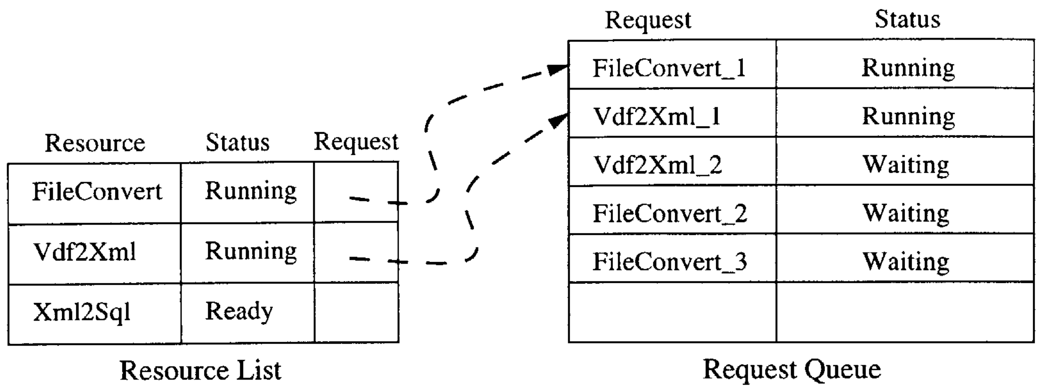


Figure 5-3: Resource Manager State for a FIFO Scheduling Policy

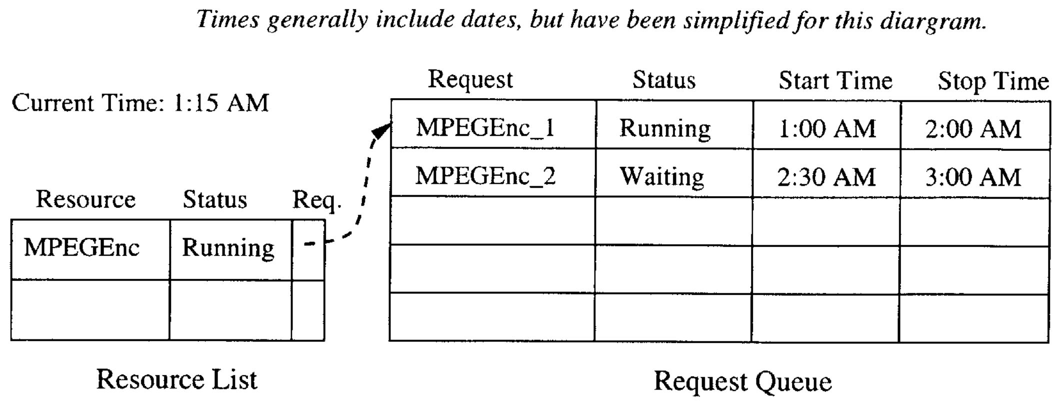
of the resource, the current state of the resource, and a reference to the resource. Recall that a resource is really an instance of a service in the framework.

In a system accommodating many different types of services, no single scheduling algorithm is optimal or even applicable to the different types of services. To accommodate this heterogeneity, different scheduling policies can implemented for each resource manager. Services which fit a particular scheduling policy should be registered with a resource manager embodying that policy. In our integration framework, two different scheduling policies are currently implemented.

The simplest scheduling policy is *first in, first out*, or FIFO for short. In this policy, a request is processed once the services needed for the request are available. When more than one request needs the same type of service, the first request to be submitted to the resource manager is the first to be processed by the service.

Figure 5-3 shows the system state for a resource manager that is using the FIFO scheduling policy. The resource manager is managing three services and has five requests in the queue. Two of the services have requests for them in the queue and are busy processing some of those requests. The requests are being processed one at a time in the order that they were submitted. The third service is idle because no requests for it are in the queue.

The second scheduling policy that is currently supported by the framework is a


Times generally include dates, but have been simplified for this diargram.
Current Time: 1:15 AM
Resource List
Request Queue

Resource	Status	Req.
MPEGEnc	Running	-

Request	Status	Start Time	Stop Time
MPEGEnc_1	Running	1:00 AM	2:00 AM
MPEGEnc_2	Waiting	2:30 AM	3:00 AM

Figure 5-4: Resource Manager State for an Actual Time Scheduling Policy

policy called *actual time*.[1] In this policy, each request carries with it timing constraints that specify when the request must be processed. The requests have actual start and stop times which describe when services for a request should be invoked and when the request expects that it will be done. In the event that a scheduled request uses a service past its reserved time, a request scheduled after it still must wait for the request to finish. If the waiting request fails to run during its requested time, an error is returned to the application waiting on the request.

Figure 5-4 shows the system state for a resource manager that is using the actual time scheduling policy. The request queue must have two additional parameters specifying start and stop times for a service invocation. A request can be added only if the time interval in which it is supposed to run does not intersect with the time intervals of requests already in the queue. Thus a new request starting at 1:30 AM and ending at 2:30 AM would be rejected by the resource manager.[2]

Additional scheduling policies could be implemented in the framework to optimize some criterion or to accommodate new types of services. However for our purposes, these two scheduling policies were adequate.

[1]We call this scheduling policy *actual time* as opposed to *real time* to avoid confusion. Real time schedulers typically have mechanisms that allow processes to specify very strict timing requirements. While our actual time scheduling also supports timing requirements, our mechanisms do not enforce these timing requirements strictly enough to be called real-time.

[2]In the current implementation of the actual time scheduling policy, we cannot accommodate for multiple instances of services.

5.3 Resource Manager Registration

Resource managers are registered to resource bosses in a way similar to how services are registered with resource managers. The registration of services with a resource manager was discussed in chapter 4.

Like services, a resource manager has an `EventAdapter` with which it sends and receives events. Upon being started, a resource manager sends a `RegisterManager` event to a resource boss. A `RegisterManager` event simply contains a reference to the resource manager being registered and the name of the resource boss to whom the event is directed. Like services, resource managers register themselves to exactly one resource boss.

As in service to resource manager registration, resource bosses send a `CallForManager` event to all resource managers when they start. When a resource manager receives a `CallForManager` event from its particular resource boss, it responds by sending a `RegisterManager` event.

A resource boss is registered to another resource boss in a similar fashion. The only difference between resource boss and resource manager registration is that `RegisterBoss` and a `CallForBoss` events are used in place of `RegisterManager` and `CallForManager` events.

Chapter 6

Video Capture System

Using the integration framework, we developed a video capture system. We integrated a heterogeneous set of software and hardware into a system that was able to automatically capture video streams. Some of the services in the system included metadata extraction, metadata storing, and format conversion services.

We present this video capture system as a case study for how the integration framework is used to create an asset management system. We first describe how the system works before discussing each service in brief detail. We conclude this chapter by describing how applications were constructed for end users to use the services that were integrated.

6.1 System Model

Before any asset management system can be constructed, some notion of how the system is supposed to work must be specified. The main purpose of an asset management system is to manage data. Hence, we begin by discussing the data model for our video capture system. We also describe a task model which specifies how a set of services can be integrated to support the capture of video streams.

Asset ID
1035635

MBR

Asset ID	Start Time	Stop Time	Video File
1035635	1:00 PM	1:58 PM	/file1.mpg

Broadcast

Asset ID	Video Quality	Audio Quality
1035635	Standard	CD

MPEGEnc

Asset ID	RM File	Codec
1035635	/file2.rm	28.8K
1035635	/file3.rm	1.5 M

FileConvert

Figure 6-1: Simplified Version of Video Capture System Data Model

6.1.1 Data Model

The purpose of an asset management system is to manage assets. Assets are persistent information objects that are composed of a set of smaller information objects. From the point of view of a user, an asset is a unit of information.

In our video capture system, video streams are captured from an analog signal and stored as an asset in a repository. Our video assets are composed of many parts such as a video file, still images, an audio file, author names, and titles. Since video files tend to be very large, we also have some performance limitations which force us to store versions of the video which are smaller and of lower quality but can be quickly transferred through a network.

The information composing an asset can be more finely identified as either data or metadata. In the case of our system, the data is the set of images and audio samples that together form the video stream. Metadata includes information such as the title of the video and the author of the video. Our system also maintains low quality versions of the video as different formats of the data. To maintain the association between these different formats and the data from which they were derived, we use an asset identifier.

The asset identifier is a UUID[1] and is used throughout our system to associate data and metadata with an asset. This asset identifier marks every piece of metadata and data that is stored in the system. The asset identifier also marks every service invocation so that the system can maintain a history of each asset.

Much of the metadata and data is stored in a relational database although larger units of data are stored in a distributed file system. Collectively, we call these persistent storage facilities the *repository*.

Figure 6-1 shows a simplified version of the tables that we used in our video capture system. The tables labeled as `MPEGEnc` and `FileConvert` correspond to services which performed video encoding and format conversion. The Broadcast table contains basic information about a completed video capture. The MBR, short for Master Business Record, simply contains a list of all the asset identifiers that we have stored in our system.

The figure shows the state of our video capture system with just one asset. The asset is named by the asset identifier `1035635`. This asset identifier is sprinkled through each table identifying the records which belong to the asset for which it is named.

It is worth mentioning that file names need not be related to the asset identifier in any way. The metadata needed to associate a file with an asset is contained within the tables. Thus, the file can be given any name so long as the name appears in the correct location in the table. The only constraint that we place on new file names is that they be unique.

Although it is not shown in figure 6-1, we also store bibliographical information in a database. We are using a schema based on the Dublin Core standard [7]. As with the other metadata records in our tables, we again use an asset identifier to associate these bibliographical records with an asset.

[1]UUID is short for Universally Unique Identifier. See appendix B for more information about UUID's.

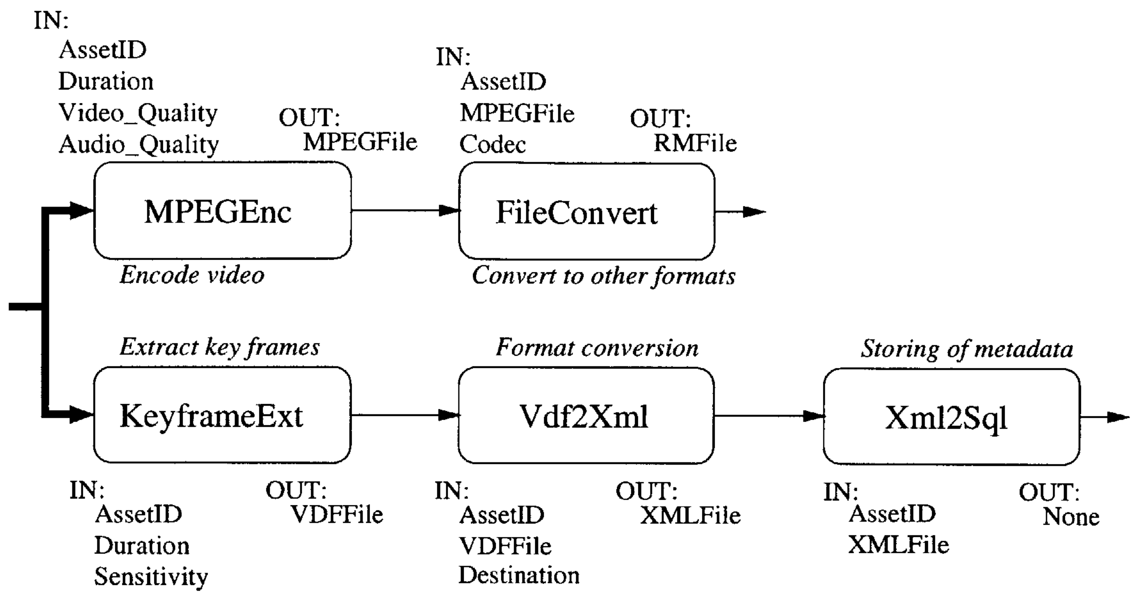


Figure 6-2: Task Model of Video Capture System

6.1.2 Task Model

We next specify a task model which divides our services into a set of basic functions. We define a *task* to be the operations performed by a single service. This task model describes the order in which services can occur by considering precedence constraints between services. Given a set of content services, we can be very specific about how each service relates to the others.

Figure 6-2 shows the task model for our video capture system. The figure shows the dependencies between services and a simplified interface to each service. Because of the dependencies, certain sequences of tasks must run serially although some sequences may run in parallel. In particular, the `FileConvert` service can process an asset only after the `MPEGEnc` service has finished with the asset.[2] However, `FileConvert` can process the asset at the same time as the `Vdf2Xml` service if the `MPEGEnc` and `KeyframeExt` have already finished with the asset.

Indicated by the dark line between the `MPEGEnc` and `KeyframeExt` services, the two services run in parallel but are not dependent on other services. However, these

[2]We are excluding the possibility that the `FileConvert` service can operated on the MPEG file as it is being encoded.

services must also obey one additional constraint. Both services operate on the same analog video signal and hence must operate at the same time in order to generate consistent data and metadata.[3] This constraint is taken into account before a request is scheduled for either service. If an application request demands both services, the resource manager ensures that an instance of both services is available before it accepts the request.

6.2 Content Services

We now launch into a discussion about each of the content services in the video capture system. Specification for each service can be found in appendix A.

We have five automated services in our system. Of the automated services, two operate on a video stream while it is being encoded while the other three operate as post-processing tasks. We shall first discuss the two actual-time services in this section followed by the three post-processing services.

6.2.1 MPEG Encoding

The MPEG encoding service (named as `MPEGEnc`) uses software and hardware to encode an analog video signal to a file into MPEG format. When the service is invoked, it encodes a video stream for some duration. In addition to the duration, a variety of parameters are used to set the quality of the video and audio encoding. In the case of video, the quality is determined by the frame rate and image resolution. For audio, quality is determined by the bit rate. With the MPEG format, a user may also specify different degrees of lossiness for the compression of a file.

Rather than allowing all the parameter options, we reduced both the video and audio quality parameters to three preset options. For video the options are qualitatively described as *High Quality*, *Standard Quality*, and *Low Quality*. For the audio,

[3]Hence, we need a distributed synchronization mechanism. Unfortunately, we do not provide any general mechanism for the framework. Currently, we synchronize the services by transmitting messages between the services using the event channel.

the available options are *CD Quality (44 kHz)*, *Radio Quality (22 kHz)*, and *Telephone Quality (11 kHz)*.

During the encoding process, the MPEGEnc service uses the local file system as a disk cache. If the service attempted to use a network file system, much data would be lost because the amount of data bandwidth would be insufficient. After the encoding completes, the captured video file is moved to the repository. The socket copy service described in appendix B is used to quickly move a file from the machine where the MPEGEnc is located to the repository machine.

6.2.2 Key Frame Extraction

Like the MPEGEnc service, the key frame extraction service (named as KeyframeExt) operates on an analog video signal.[4] The purpose of the KeyframeExt service is to extract a set of images that summarize a video stream. The service uses a sensitivity threshold between 1 and 100 to help it decide which frames to extract. These frames are called *key frames*.

We do not know the actual algorithm used to find key frames because the extraction engine was purchased from an outside vendor. To us, the extraction engine is merely a "black box." We only know that the engine picks fewer key frames at lower sensitivities than at higher sensitivities. However, we do not need to know exactly how the extraction engine works since we are only interested in the key frames.

For each key frame, the service also notes its exact time of position in the video stream. This time is relative to the start of the video stream and identifies the exact frame from which the image was extracted. These times can be used to create a summary or an index and are hence a useful piece of metadata that we store with the key frames.

[4]The KeyframeExt service operates on an analog video signal because the component from which it is integrated operates on an analog video signal.

6.2.3 File Conversion Service

The file conversion service (named as `FileConvert`) is a post-processing service that converts an MPEG file into RealMedia format. Video and audio streams stored in RealMedia format are much smaller than those stored in MPEG format. This reduction allows acceptable performance when transferring the video or audio stream across a network. The RealMedia format has the further benefit of being a streaming format. Thus, video and audio files can be viewed at a receiving machine as it is arrives across the network. This improved delivery is achieved at a cost of image and audio quality, but is absolutely necessary for clients with slower network connections.

The RealMedia format is capable of accommodating a variety of performance and quality tradeoffs. By using different encodings and compression settings, the RealMedia format can encoding a video or audio stream in a setting that is optimized for specific performance specifications. RealMedia accommodates both these different settings by using a *codec*, which is short for coder-decoder. A codec describes various encoding parameters and compression settings for a video file.

The `FileConvert` service allows a variety of codecs to be used for encoding a video or audio stream. A user can decide which codec to use by passing a proper codec name to the service. Example codec names are `Video 28.8, High Action` and `Audio 56K ISDN, Music - Stereo`.

6.2.4 VDF-to-XML Conversion Service

The VDF-to-XML conversion service (named as `Vdf2Xml`) is a post-processing service that runs after the `KeyframeExt` service. The `Vdf2Xml` service is needed because the output of the `KeyframeExt` service is in `VDF` format, a non-standard format for describing video streams. The `VDF` file contains the key frames and their times of occurrence in the video stream. The purpose of the `Vdf2Xml` service is hence to convert a `VDF` file into a set of key frames and their associated times.[5] The key frames

[5]While it might be argued that the `Vdf2Xml` should be integrated with the `KeyframeExt` service, we found that this service demonstrated a useful precedence constraint.

are converted into a set of JPEG files while their associated times are put into an XML document.[6]

6.2.5 XML-to-SQL Conversion Service

The XML-to-SQL conversion service (named as `Xml2Sql`) is a post-processing operation that runs after the `Vdf2Xml` service. The `Xml2Sql` service reads the metadata in the XML file returned by the `Vdf2Xml` service and executes SQL[7] statements which inserts the metadata into a relational database.

As a performance optimization, this service creates a sequence number for each key frame. The sequence number indicates the relative order of each key frame. Thus, when a query returns a set of key frames for a video file, the order can be quickly reconstructed by ordering the key frames by their sequence numbers.[8]

6.3 Service Integration

Once the content services and the overall system models are specified, the services can be integrated with a set of resource management components to make the services available to applications. Figure 6-3 is a picture of our video capture system at the three levels of the integration framework.

At the lowest level are the five services that we described in the previous section. The actual-time processing services, `MPEGEnc` and `KeyframeExt`, are both under the management of a single resource manager embodying the actual-time scheduling policy. The three post-processing services on the other hand are managed by a resource manager under the FIFO scheduling policy. Both the actual-time and FIFO scheduling policies were discussed in chapter 5.

At the resource management layer, the two resource managers were named the

[6]Bill Graham, a researcher at the IBM Almaden Research Center, implemented the Vdf2Xml service.

[7]SQL stands for Standard Query Language.

[8]Bill Graham, a researcher at the IBM Almaden Research Center, implemented the Xml2Sql service.

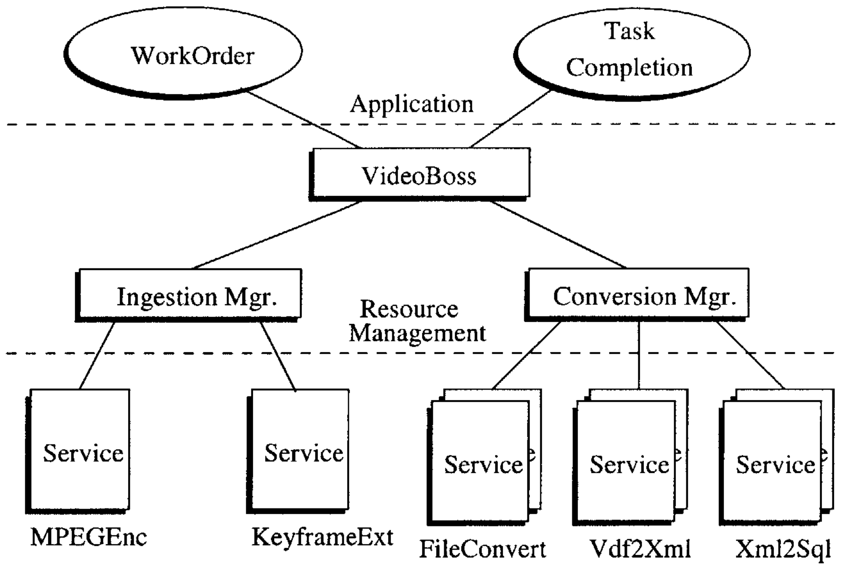


Figure 6-3: System Diagram of the Video Capture System

IngestionManager and the ConversionManager. The ConversionManager follows the simple FIFO scheduling policy while the IngestionManager follows the actual-time scheduling policy. One resource boss, named the VideoBoss serves the entire system. Both resource managers are registered to this resource boss. This resource boss is consulted by applications wishing to use any one of the services in the video capture system.

6.4 Applications

We have two applications in our video capture system. The applications can perform concurrently and can generate a number of simultaneous requests. The two applications that we have in the system are called the WorkOrder and the TaskCompletion applications.

6.4.1 Work Order

The WorkOrder application is used by a user to initiate a set of tasks for capturing video streams. The application presents a form which allows the user to choose which services to invoke for the capture. The user can also specify some of the capture parameters such as the quality of the video and audio encoding. If the user decides to extract key frames, he would also supply a sensitivity parameter between 1 and 100 which would be used to perform key frame extraction. Similarly, he would specify a RealMedia codec if he wished to perform a file conversion.

When the user has finished setting up his capture, he submits the form, and the `WorkOrder` application creates a new asset identifier. The new asset identifier is entered into the master business record, thus committing the creation of a new asset. The `WorkOrder` application then sends requests for services to the appropriate resource managers in an order that satisfies the precedence constraints specified in the task model. Requests are sent until all are complete or a failure occurs.

Multiple instances of the `WorkOrder` application can be run at the same time. Because the resource management layer handles the resource allocation issues, we do not have to worry about inconsistencies that arise from concurrent requests for services.

6.4.2 Task Completion

The `TaskCompletion` application allows a user to perform tasks on existing assets. The application is particularly useful for performing tasks which were not performed during the initial capture. For example, if a user of our video capture system wished to create a smaller version of a video file, he could use the `TaskCompletion` application to invoke the `FileConvert` service so long as the `MPEGEnc` service had successfully completed for the asset.

The `TaskCompletion` application is thus an application which complements the `WorkOrder` application. Whereas the `WorkOrder` is used to create new video assets, the `TaskCompletion` allows a user to invoke services on an existing asset.

6.5 State of the System

The system is currently running on a small group of computers at the IBM Almaden Research Center. The group consists of six IBM PC based computers and four AIX workstations all connected to the regular IBM internal network.

With the system, we were able to successfully capture video broadcasts using our applications. All the content services performed as reliably as the components upon which they were constructed. Thanks to the distributed nature of the system, we were able to run the system remotely, thereby allowing its services to be shared amongst a greater number of users.

After deploying the system, we replicated some of the post-processing services simply by installing the software for the services on other machines. Because the post-processing services were implemented entirely in software, duplicating these services was quite trivial. When we started those services, they registered themselves with the appropriate resource manager and became instantly available to applications. The ease with which we were able to add replicas of existing services led us to characterize the process as "plug-and-play."

We also attempted to integrate a new type of service after deploying the system. Our integration efforts involved creating a service from a new "black box" component and adding the service to the system. We were successful in adding the new type of service with relative ease although it was not as easy as adding a replica service. Some implementation details still need to be simplified before new services can be added with as little effort as replicas.

Chapter 7

Conclusion

In this thesis, we have described an extensible integration framework for the creation of asset management systems. Divided into three layers, the integration framework provides a systematic way to integrate heterogeneous services into a complete asset management system.

Each layer of the framework handles separate issues that are needed to perform any set of tasks. The layers we define in ascending order are the service layer, the resource management layer, and the application layer. Each of the three layers handles subsequently higher level issues. The application layer decides what tasks need to be performed. The resource management layer decides how the tasks are to be performed. The service layer actually performs the tasks. This separation allows the systematic implementation of tasks supporting an asset management system.

The division of functions amongst the three separate layers has the added benefit that applications and services are decoupled from each one another. Services can thus be added and removed dynamically without affect applications. Similarly, applications can be started and stopped without affecting services.

In other integration technologies that we have explored, none explicitly offers the same set of resource management mechanisms that we provide. Owing to the fact to no resource management scheme could possibly be optimal for all cases, we can extend the resource management mechanisms to handle other cases.

With this integration framework, we have successfully constructed a video capture system. When we added replicas of existing services to the running system, we found the process to be quite trivial. Similarly, new types of services were also added although the process involved a bit more effort.

7.1 Future Work

We have described the virtues and the successes of the integration framework. However, more work could be done to expand the framework to deal with more of the issues of asset management systems.

The integration framework was developed from a very narrow point of view for a very specific type of system. Hence, many assumptions were made about how services and applications interact. A more general and complex model of services and applications should be considered and possibly supported by the integration framework. As the service and application models are modified, so too must the manner in which resources are managed. The end result is that the entire framework might have to be redesigned.

From the asset management point of view, we have only dealt with the issue of integrating services for an asset management system. The services that we have used to demonstrate our framework are but a minute subset of the functions needed to build a real asset management system. More work needs be done to develop some of the domain specific services and infrastructure needed to build an asset management system for digital objects. As these services are developed, they will likely reveal more complicated interactions which have not been considered in the integration framework.

Appendix A

Service Specifications of the Video Capture System

Type	MPEGEnc
Hardware	Broadway MPEG Encoding Card
Software	Broadway
Vendor	Data Translation, Inc.
Platform	Windows 95/NT
Interface	GUI
Arguments	**AssetID**: asset identifier **VideoQuality**: High Quality or Standard Quality or Low Quality **AudioQuality**: CD Quality (44 kHz) or Radio Quality (22 kHz) or Telephone Quality (11 kHz) **Duration**: Amount of time to capture in seconds
Return Value	**MPEGFile**: Name of resulting MPEG file

Table A.1: MPEG Capture Service Specification

Type	FileConvert
Hardware	None
Software	MediaPalette version 1.2
Vendor	Cinax, Inc.
Platform	Windows 95/NT
Interace	GUI and command line
Arguments	**AssetID**: asset identifier **MPEGFile**: Name of MPEG file to be converted **Codec**: Must be one of following codecs: Audio 14.4, Voice Audio 28.8, Music - Mono Audio 28.8, Voice Audio 28.8, Music - Stereo Video 28.8, High Action Audio 56K Dial-Up Modem, Music - Mono Audio 56K Dial-Up Modem, Music - Stereo Video 56K Dial-Up Modem, Music Video 56K Dial-Up Modem, Voice Audio 56K ISDN, Music - Mono Audio 56K ISDN, Music - Stereo Video 56K ISDN, Music Video 56K ISDN, Voice Audio 112K, Dual ISDN, Music - Mono Audio 112K, Dual ISDN, Music - Stereo Video 112K Dual ISDN, Music Video High-Bite Rate 200K, Music Video High-Bite Rate 200K, Voice Video High-Bite Rate 300K, Music Video High-Bite Rate 300K, Voice
Return Value	**RMFile**: Name of converted file in RealMedia format

Table A.2: File Conversion Service Specification

Type	KeyframeExt
Hardware	Flashpoint Video Card
Software	Virage VideoLogger
Vendor	Virage, Inc.
Platform	Windows 95/NT
Interace	GUI
Arguments	**AssetID**: asset identifier **Sensitivity**: key frame extraction sensitivity. Range in [1, 100] **Duration**: Amount of time to capture in seconds
Return Value	**VDFFile**: Name of VDF file containing key frames

Table A.3: Key Frame Extraction Service Specification

Type	Vdf2Xml
Hardware	None
Software	Virage C++ VDF API
Vendor	Virage, Inc.
Platform	Windows 95/NT
Interace	command line binary created from C++ VDF API
Arguments	**AssetID**: asset identifier **VDFFile**: Name of VDF file containing key frames **Destination**: Location where key frame files should be written
Return Value	**XMLFile**: XML file containing listing of times of occurrence for each key frame

Table A.4: VDF-to-XML Conversion Service Specification

Type	Xml2Sql
Hardware	None
Software	IBM Alphaworks XML For Java, IBM DB2, JDBC
Vendor	IBM, Sun Microsystems
Platform	any
Interace	Java classes
Arguments	**AssetID**: asset identifier **XMLFile**: XML file containing listing of times of occurrence for each key frame
Return Value	None

Table A.5: XML-to-SQL Conversion Service Specification

Appendix B

System Infrastructure

Infrastructure consists of the common mechanisms that are needed across the three layers of the integration framework. The infrastructure components that were needed in the framework and are discussed in this chapter include a universally unique identifier generator, a distributed file system, an event channel, and a data transfer service.

B.1 Universal Unique Identifier Generators

The universal unique identifier (known also as a UUID) is an important infrastructure component. In a distributed system, we often need to create a unique object. The best way to ensure that all references to the object are also unique is to give it a unique name. For example, if we have multiple services running in parallel and sharing a file system, we will need to make sure the services use unique file names when writing new files. If two services use the same file name to write data, some data will be lost.

The purpose of a UUID generator is to generate unique names for objects. The scheme that we use to generate UUIDs is specified in [8]. This scheme uses a 64 bit timestamp, a 16 bit sequence number, and a 48 bit identifier base. The timestamp is the current time in milliseconds. The sequence number is used in case more than one UUID is created in the same millisecond.

The identifier base is a number that is supposed to be unique to the generator. When the 48 bit MAC address of an Ethernet card is available to be used as the

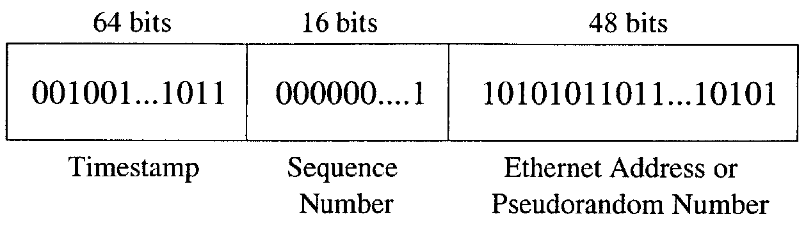


Figure B-1: Sample 128 Bit Universal Unique Identifier

identifier base, the UUID is guaranteed to be unique. However, in cases when the MAC address is unavailable, a random number is used instead. In this case, the UUID is not guaranteed to produce unique identifiers, although the probability of a name collision occurring is extremely low. The identifier base, sequence number, and timestamp are concatenated to form a 128 bit unique identifier. Figure B-1 shows a sample UUID.[1]

An additional attribute of our UUID generator is that multiple generators can be used in any number of locations. Generating unique identifiers in a decentralized manner ensures that UUID generation will not be a performance bottleneck while avoiding a single point of failure in our system.

B.2 Distributed File System

A distributed file system is used to share files and directories across multiple machines. To provide for a uniform naming system for files across multiple platforms, we created an platform independent abstraction for file names. The syntax of the abstraction is similar to that of a URL.[2]

[1]Bill Graham, a researcher at the IBM Almaden Research Center, implemented the UUID generator used in the integration framework.

[2]URL stands for Uniform Resource Locator.

B.2.1 Construction

The distributed file system was cobbled together using the NFS and SAMBA[9] protocols.[3] With these protocols, UNIX and Window 95/NT machine export directories which are then mounted on other machines.[4] Hence, the distributed file system is implemented as a set of network file systems.

Given the amount of overhead for the NFS and SAMBA protocols, the file system performs poorly for large data transfers. Currently, the file system serves as an interim solution until a more robust and optimized distributed file system becomes available for multiple platforms.

B.2.2 Uniform Naming

Different platforms have different ways of naming files. In the standard UNIX file system, a single root directory exists from which all other files and directories may be reached. In the DOS based FAT file system, the file system can have as many as 26 root directories, each named by a letter in the alphabet. Within each of these directories, the file system supports subdirectories. Both file systems thus support hierarchical directory trees.

One minor difference between the FAT and UNIX file systems occurs in how they separate names in a path name. In UNIX, the separator character is a slash, whereas in DOS, the character is a backslash.

To hide this platform dependent detail, we use an abstraction called the `VDFSFile`. The syntax used for the `VDFSFile` abstraction is similar to that of a URL. We use the word "vdfsfile" as the protocol followed by three slashes. The rest of the URL is the path name of a file.

As a convention, we resolve the `VDFSFile` root to the `/vdfs` directory in UNIX

[3]The NFS protocol is used to mount UNIX directories on UNIX workstations. SAMBA is used to mount Windows 95/NT directories on a UNIX workstation and UNIX directories onto Windows 95/NT machines.

[4]Aaron Van Devender constructed the distributed file system at the IBM Almaden Research Center. In addition, he implemented the first version of the VDFS file abstraction to provide for platform independent naming of file.

Naming System	File Name
UNIX	/vdfs/realvideo/videofile.rm
DOS	V:\realvideo\videofile.rm
VDFS	vdfsfile:///realvideo/videofile.rm

Table B.1: Comparison of Names per Naming System

file systems. On FAT file systems, the `V:\` directory is the root. Table B.1 compares these different naming systems for files.

With this `VDFSFile` abstraction, we can pass references to files across multiple platforms. When a `VDFSFile` reference is passed to a service, the service determines if it is operating on system with a UNIX or an FAT file system. The service can then translate the `VDFSFile` to its native file name format depending on the platform.

B.3 Event Channel

System components send and receive events by creating clients that connect to the event channel. If these clients produce events, they are *event producers*. If these clients receive events, they are called *event consumers*. Clients that send and receive events are called *event adapters*.

The event channel serves as an intermediary through which messages may be sent from event producers to event consumers. When the event channel receives a message from an event producer, it sends the message to all connected event consumers. We call this message an *event* since it usually demands some sort of action of an event consumer. Events are guaranteed to be received exactly once unless a failure occurs.

Using the event channel as an intermediate simplifies the task of producing and consuming events. With an event channel, a program can broadcast an event to all the event consumers without needing to know the number or location of the consumers. The event channel thus provides a layer of indirection for communicating between clients.

The event channel was implemented using Java RMI according to the CORBASer-

Host Name	VDFS File URL	Native File
vilnius	vdfsfile:///vilnius/temp/	F:\temp\
ngcs	vdfsfile:///ngcs/temp/	/usr/local/src/
tv1000	vdfsfile:///tv1000/	E:\
xfiles	vdfsfile:///xfiles/	E:\
viagra	vdfsfile:///viagra/	E:\
ice	vdfsfile:///ice/	/image1/
realvideo	vdfsfile:///realvideo/	/video/

Table B.2: Sample VDFSFileMap file

vices specification for an event channel [10].[5]

B.4 Socket Copy Service

The socket copy service is an mechanism better suited for transferring large amounts of data across a network [11]. The socket copy service was created because the distributed file system was poorly suited for large data transfers.

The socket copy service works by opening a direct TCP connection between two machines which are transferring a file. One of the machines is a server while the other is a client. A server is typically placed on repository machines where large amounts of data are transferred. Clients initiate connections with a server from some machine in the system.

A client initiates a connection for a socket copy service by sending two `VDFSFile` names to a server. One of the names corresponds to a source URL while the other is a destination URL. With the source and destination URLs, the server and client can determine if the file is to be transferred *to* the server or *from* the server. If the source is located at the client and the destination at the server, the file is copied *from* the client *to* the server. If the source is located at the server and the destination at the client, the file is copied *to* the client *from* the server.

In order for the socket copy service to determine where a source or destination

[5]Bill Graham, researcher at the IBM Almaden Research Center, implemented a Java version of the event channel for the integration framework.

file is actually located, it needs to be able to resolve a `VDFSFile` name to a native file name. This information is maintained in a `VDFSFileMap` file. The `VDFSFileMap` is a table that maps a `VDFSFile` name to a directory name and a host name. The directory name is the native directory name referenced by the `VDFSFile` URL. The host name is the name of the machine that exports the directory.

Table B.2 shows an example `VDFSFileMap` file. From the table, we can see that the `VDFSFile` directory `vdfsfile:///tv1000/` is located on the host `tv1000` and is the `E:\` directory on that host.

Using this `VDFSFileMap`, the socket copy service can determine whether the file to be copied and its destination is on the client or the server. If the socket copy service cannot determine one of the valid cases, it automatically resorts to using the distributed file system instead of the socket copy service.

Rough measurements showed that the socket copy service transferred files more than an order of magnitude faster than our distributed file system.

Bibliography

[1] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59+, February 1984.

[2] Jay Ongg. An architectural comparison of distributed object technologies. Master's thesis, Massachusetts Institute of Technology, 1997.

[3] Object Management Group, Framingham, Massachusetts. *Common Object Request Broker: Architecture and Specification*, 1997.

[4] Java remote method invocation specifications. Web page available at `http://www.javasoft.com/products/jdk/1.1/docs/guide/rmi/`.

[5] World Wide Web Consortion, Cambridge, Massachusetts. *Extensible Markup Language (XML) 1.0*, February 1998.

[6] Robert Kahn and Robert Wilensky. A framework for distributed digital object services. ARPA research project, Corporation for National Research Initiatives, Reston, Virginia, May 1995. Soft copy located at `http://www.cnri.reston.va.us/home/cstr/arch/k-w.html`.

[7] Diane Hillman. A user guide for simple dublin core. Web page located at `http://purl.org/dc`, July 1988.

[8] Paul J. Leach and Rich Salz. UUIDs and GUIDs. Internet Draft on Standards Track, February 8, 1998.

[9] John D. Blair. *SAMBA: Integrating UNIX and Windows.* Specialized Systems Inc., Seattle, Washington, 1998.

[10] Object Management Group, Framingham, Massachusetts. *CORBAServices: Common Object Services Specification*, 1997.

[11] Rajat Mukherjee, October 1998. Personal communication.

[12] Butler W. Lampson, M. Paul, and H.J. Siegert (Editors). *Distributed Systems – Architecture and Implementation: An Advanced Course.* Lecture Notes in Computer Science. Sprinter-Verlag, Berlin, Germany; New York, New York, 1981.

[13] Sape Mullender (Editor). *Distributed Systems.* ACM Press, New York, New York, 1993.

[14] Chris J. Date. *An Introduction to Database Systems.* The Systems Programming Series. Addison-Wesley, Reading, Massachusetts, sixth edition, 1995.

[15] Wilson WindowWare, Inc., Seattle, Washington. *Windows Interface Language Reference Manual*, 1997.

[16] David Curtis. Java, RMI and CORBA. Soft copy available at `http://www.omg.org/library/wpjava.html`, 1997.

7433-57