The Design and Performance of a CORBA Audio/Video Streaming Service

Sumedh Mungee, Nagarajan Surendran, Douglas C. Schmidt

{sumedh,naga,schmidt}@cs.wustl.edu
Department of Computer Science, Washington University
St. Louis, MO 63130, USA*

Abstract

Recent advances in network bandwidth and processing power of CPUs has led to the emergence of multimedia streaming frameworks, such as NetShow, Realvideo and Vxtreme. These frameworks typically rely on proprietary stream establishment and control mechanisms to access multimedia context. To facilitate the development of standards-based distributed multimedia streaming applications, the OMG has defined a CORBA-based specification that stipulates the key interfaces and semantics needed to control and manage audio/video streams.

This paper makes two contributions to the study of CORBA-based distributed multimedia streaming frameworks. First, it describes the design and performance of an implementation of the OMG audio/video (A/V) streaming model based on TAO, which is a real-time CORBA ORB. Second, it describes the design and performance of a distributed application that uses TAO's A/V streaming framework to establish and control MPEG video streams. Our experience with TAO's A/V streaming framework indicates that CORBA defines a flexible and efficient model for developing standards-based multimedia streaming applications.

Keywords: CORBA-based Multimedia Streaming, QoS-enabled OO Middleware, Performance Measurements

1 Introduction

1.1 Motivation

Traditional distributed object computing (DOC) middleware such as CORBA, DCOM, and Java RMI support request/response semantics for distributed applications. However, an increasingly important class of applications require transfer of continuous media data streams. For instance, popular Internet-based streaming mechanisms, such as Realvideo [1] and Vxtreme [2], allow suppliers to transmit continuous streams of audio and video packets to consumers. Likewise, non-continuous media applications, such as medical imaging servers [3] and network management agents [4], employ streaming to transfer bulk data efficiently from suppliers to consumers.

Stringent performance requirements for streaming data often preclude DOC middleware from being used as the transport mechanism for multimedia applications [5]. For instance, inefficient CORBA Internet Inter-ORB Protocol (IIOP) [6] implementations often perform excessive data-copying and memory allocation *per-request*, which increases packet latency [7]. Likewise, inefficient marshaling/demarshaling in DOC middleware decreases streaming data throughput [8].

If the design and performance of DOC middleware can be improved, however, the stream establishment and control components of distributed multimedia applications can benefit greatly from the portability and flexibility provided by middleware. To address these issues, the Object Management Group (OMG) has defined a specification for the control and management of A/V streams [9], based on the CORBA reference model [10].

The CORBA A/V streaming specification defines a model for implementing an open distributed multimedia streaming framework. This model integrates (1) well-defined modules, interfaces, and semantics for stream establishment and control with (2) efficient transport-level mechanisms for data transmission. In addition to defining standard stream establishment and control mechanisms, the OMG specification allows distributed multimedia applications to leverage the portability and flexibility provided by DOC middleware.

Our prior research on CORBA middleware has explored several dimensions of real-time ORB endsystem design including static [11] and dynamic [12] real-time scheduling, real-time request demultiplexing [13], real-time event pro-

*This work was supported in part by Boeing, GDIS/CDI, DARPA contract 9701516, Lucent, Motorola, NSF grant NCR-9628218, Siemens, and Sprint.

cessing [14], real-time I/O subsystem integration [15], and the real-time performance of various commercial and research ORBs [16] over ATM networks. This paper focuses on a previously unexamined point in the real-time ORB endsystem design space: *the design and performance of the CORBA A/V streaming service specification*.

1.2 Design Challenges

We have developed the first freely available implementation of the OMG A/V streaming model using TAO [11], which is a real-time CORBA ORB that has been ported to most OS platforms. In addition to implementing the components defined by the OMG specification, TAO's A/V streaming service uses patterns [17] to resolve the following key design challenges that arise when developing distributed multimedia streaming frameworks:

Flexibility in stream endpoint creation strategies: The OMG specification defines the interfaces and roles of stream components. Many performance-sensitive multimedia applications require fine-grained control over the strategies governing the creation of stream components. For instance, our past studies of Web server performance [18, 3] motivate the need to support *adaptive* concurrency strategies to develop efficient and scalable streaming applications.

In the context of our A/V streaming service, we determined that the supplier-side of our MPEG application described in Section 3 required a process-based concurrency strategy to maximize stream throughput by allowing parallel processing of separate streams. Other types of applications required different implementations, however. For example, the consumer-side of our MPEG application benefited from the creation of reactive [19] suppliers that contain all related endpoints within a single process.

To achieve a high degree of flexibility, therefore, our A/V streaming service design decouples the *behavior* of stream components from the strategies governing their *creation*. We achieved this decoupling via the *Factory Method* and *Abstract Factory* patterns [17], as described in Section 2.2.1.

Flexibility in transport protocol: A streaming service may need to select from a variety of transport protocols. For instance, an Internet-based streaming application like Realvideo [1] may use the UDP protocol, whereas a local intranet video-conferencing tool [20] might prefer the QoS features offered by native high-speed ATM protocols. Thus, it is essential that a streaming service support a range of transport protocols.

The OMG streaming service makes no assumptions about the transport protocol used for data streaming. Consequently, the stream establishment components in our A/V streaming service framework provide flexible mechanisms that allow applications to define and use multiple transport endpoints, such as sockets and TLI, and multiple protocols, such as TCP, UDP, or ATM.

Another design challenge, therefore, is to define stream establishment components that can work with a variety of transport endpoints. To resolve this challenge, we applied the *Strategy* pattern [17], as explained in Section 2.2.5.

Flexibility in stream control interfaces: An A/V streaming framework must provide flexible mechanisms that allow developers to define and use different operations for different streams. For instance, a video application typically supports a variety of *operations*, such as `play`, `stop` and `rewind`. Conversely, a stream in a stock quote application might support operations like `start` and `stop`. Because the operations provided by the stream are application-defined, it is useful for the control logic component in a streaming service to be very flexible.

Therefore, another design challenge facing designers of streaming services is to allow applications the flexibility to define their own stream control interfaces.

Flexibility in managing states of stream supplier and consumers: The transport component of a streaming application often needs to change behavior depending on the current *state* of the system. For instance, invoking the `play` operation on the stream control interface of a video supplier may cause it to enter into a PLAYING state. Likewise, subsequently sending it the `stop` operation may cause it to transition to the STOPPED state. More complex state machines can result due to additional operations, such as `rewind` and `fast_forward` operations.

Thus, an important design challenge for developers is designing flexible applications whose states can be extended. In addition, the behavior of supplier/consumer applications, and the A/V streaming framework itself, in each state must be well-defined, To address this issue we applied the *State Pattern* [17], as described in Section 3.1.

1.3 Paper Organization

The remainder of this paper is organized as follows: Section 2 describes our implementation of the OMG A/V streaming service specification using TAO [11], which is a real-time CORBA ORB; Section 3 outlines the design of an MPEG streaming application that uses TAO's A/V streaming service; Section 4 analyzes the performance results of TAO's A/V streaming service over a high-speed ATM network; Section 5 summarizes related work; and Section 6 presents concluding remarks. For completeness, Appendix A outlines the OMG CORBA reference model and Appendix B presents a brief overview of the CORBA Property Service, which is used to transfer QoS information between consumers and suppliers in the A/V streaming service.

2 The Design of TAO's Audio/Video Streaming Service

This section presents an overview of the key architectural components in the OMG A/V streaming model. It also describes the design challenges facing developers of A/V streaming frameworks and explains how TAO's A/V streaming service resolves these challenges.

2.1 Overview of the OMG Audio/Video Streaming Specification

The OMG A/V streams specification [9] presents an architectural model and OMG IDL interfaces for building distributed multimedia streaming frameworks. The goals of the OMG A/V streaming model are the following:

Standardized stream establishment and control mechanisms: Using these mechanisms, consumers and suppliers can be developed independently, while still being able to establish streams with one another.

Support multiple transport protocols: To avoid unnecessary overhead, the A/V streaming model separates control signaling from the data transfer protocol, such as TCP, UDP, or ATM.

Support various types of sources and sinks: Common stream sources include a video-on-demand server, a video camera attached to the network, or a stock quote server. Common sinks include a video-on-demand client, a display device attached to a network, or a stock quote client.

Figure 1 shows a *multimedia stream*, which is represented as a flow between two *flow data endpoints*. One endpoint acts

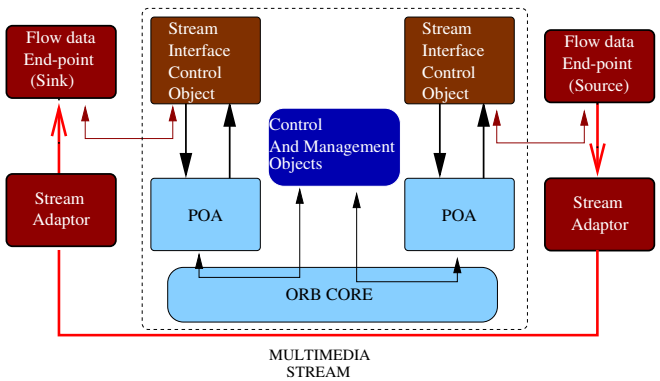

Figure 1: **OMG Streams Architecture**

as a source of the data and the other endpoint acts as a sink. Note that the control and signaling operations pass through the GIOP/IIOP-path of the ORB, demarcated by the dashed box. In contrast, the data stream uses *out-of-band* stream(s), which can be implemented using protocols that are more suitable for multimedia streaming than IIOP. Maintaining this separation of concerns is crucial to achieve high performance.

Each stream endpoint consists of three logical entities: (1) a *stream interface control object* that exports an IDL interface, (2) a *data source or sink*, and (3) a *stream adaptor* that is responsible for sending and receiving frames over a network.

Control and Management objects are responsible for the establishment and control of streams. The OMG A/V specification defines the interfaces and interactions of the *Stream Interface Control Objects* and the Control and Management objects. Section 2.2 describes the various components in Figure 1 in detail.

2.2 OMG A/V Streaming Service Components

The OMG A/V streaming specification defines a set of standard IDL interfaces that can be implemented to provide a distributed multimedia streaming framework. Figure 2 illustrates the key components of the CORBA streaming framework. This subsection describes the TAO's implementation of

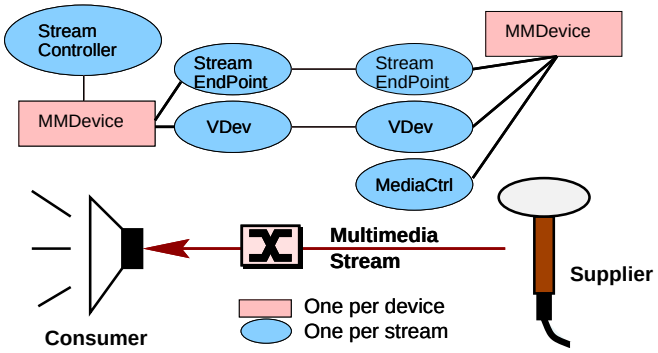

Figure 2: **A/V Streaming Service Components**

the A/V streaming service framework components shown in Figure 2. The corresponding OMG interface name for each role is provided in brackets. In addition, we discuss how TAO provides solutions to the design challenges outlined in Section 1.2. Readers who are already familiar with the OMG A/V streaming specification may want to skip to Section 3, which describes how we developed an MPEG player application using TAO's implementation of this service.

2.2.1 Multimedia Device Factory (`MMDevice`)

The `MMDevice` component abstracts the behavior of a multimedia device. The actual device can be *physical*, such as a video microphone or speaker. Likewise, a device can be *logical*, such as a program that reads video clips from a file or a

database that contains information about stock prices. There is typically one `MMDevice` per physical or logical device.

The `MMDevice` encapsulates the device-specific parameters of a multimedia device, as shown in Figure 3. For instance, a particular device might support `MPEG-1` [21] compression or `ULAW` audio [22]. Such parameters are termed "properties" of the `MMDevice`. Properties can be associated with the `MMDevice` using the CORBA Property Service [23], which is described in Appendix B.

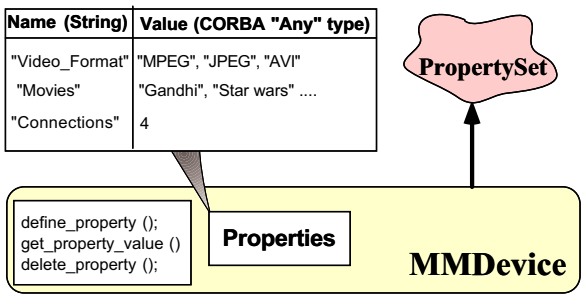

Figure 3: **Multimedia Device Factory**

The `MMDevice` is an endpoint factory, *i.e.*, it is responsible for creating new endpoints for new stream connections. Each endpoint consists of a pair of objects: (1) a virtual device (`VDev`), which encapsulates the device-specific parameters of the connection and (2) the `StreamEndpoint`, which encapsulates the transport-specific parameters of the connection. The roles of `VDev` and `StreamEndpoint` are described in Section 2.2.2 and Section 2.2.5, respectively.

The `MMDevice` component also encapsulates the implementation of *strategies* that govern the creation of the `VDev` and `StreamEndpoint` objects. For instance, the implementation of `MMDevice` in TAO's A/V streaming service framework provides the following two concurrency strategies:

Process-based strategy: The process-based concurrency strategy creates new virtual devices and stream endpoints in a new process, as shown in Figure 4. This strategy is useful for applications that create a separate process to handle each new endpoint. For instance, the supplier in our MPEG player application described in Section 3.1 creates separate processes to stream the audio and video data to the consumer concurrently.

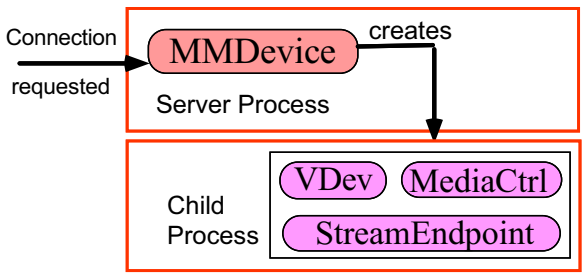

Figure 4: `MMDevice` **Process-based Concurrency Strategy**

Reactive strategy: In this strategy, endpoint objects for each new stream are created in the same process as the factory, as shown in Figure 5. This means that a single process handles all the simultaneous connections *reactively* [19]. This strategy is useful for applications that dedicate one process to control multiple streams. For instance, the consumer of the MPEG A/V player application described in Section 3.2 creates the video and audio endpoints in the same process using this strategy to minimize synchronization overhead.

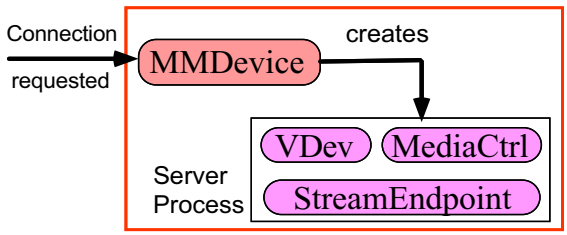

Figure 5: `MMDevice` **Reactive Concurrency Strategy**

We are enhancing TAO's A/V streaming framework to support other `MMDevice` concurrency strategies, such as a thread-based strategy that creates new stream endpoints to run in separate threads within the same process.

In TAO's A/V streaming service, the `MMDevice` uses the *Abstract Factory* pattern [17] to decouple (1) the creation strategy of the stream endpoint and virtual device from (2) the concrete classes that define it. Thus, applications that use the `MMDevice` can subclass both the strategies described above, as well as the `StreamEndpoint` and the `VDev` that are created.

Subclassing allows applications to customize the concurrency strategies to suit their needs. For instance, by default, the reactive strategy creates new stream endpoints using dynamic allocation, *e.g.*, via the `new` operator in C++. Applications can override this behavior via subclassing so they can allocate stream endpoints using other allocation techniques, such as thread-specific storage [24] or special framebuffers.

2.2.2 Virtual Device (`VDev`)

The virtual device (`VDev`) component is created by the `MMDevice` factory in response to a request for a new stream connection. There is one `VDev` per stream. The `VDev` is used by the application to define its response to `configure` requests. For instance, if a consumer of a stream wants to use the MPEG video format, it can invoke the `configure` operation on the supplier `VDev`, as shown in Figure 6.

Stream establishment is a mechanism defined by the OMG A/V streaming specification to permit the negotiation of QoS

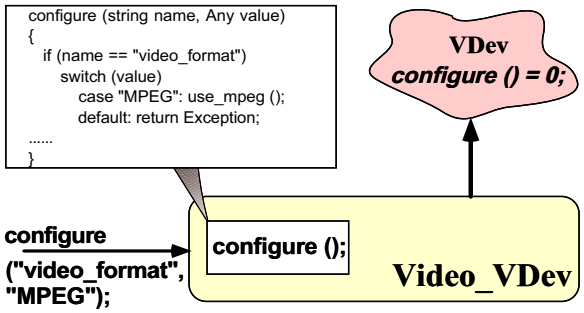

Figure 6: **Virtual Device**

parameters via *properties*. Properties are *name-value* pairs, *i.e.*, they have a `string` name and a corresponding value. The properties used by the A/V streaming framework are implemented using the CORBA Property Service, described in Appendix B.

The OMG A/V streaming specification specifies the names of the common properties used by the `VDev` objects. For instance, the property `currformat` is a string that contains the current encoding format *e.g.*, "MPEG." During the stream establishment, each `VDev` can use the `get_property_value` operation on its peer `VDev` to ensure that the peer uses the same encoding format.

When a new pair of `VDev` objects are created, each `VDev` uses the `configure` operation on its peer to set the stream configuration parameters. If the negotiation fails the stream can be torn down and resources released immediately.

Section 2.3.1 describes the OMG A/V streaming service stream establishment mechanism in detail.

2.2.3 Media Controller (`MediaCtrl`)

The Media Controller (`MediaCtrl`) is an IDL interface that defines operations for controlling a stream. The `MediaCtrl` interface is *not* defined by the OMG A/V streaming service specification. Instead, it is defined by application developers to support operations for a specific stream, such as the following OMG IDL for a video service:

```
interface video_media_control
{
  void select_video (string name_of_movie);
  void play ();
  void rewind (short num_frames);
  void pause ();
  void stop ();
};
```

The OMG A/V streaming service provides developers with the flexibility to associate an application-defined `MediaCtrl` interface with a stream. Thus, the A/V streaming service can be used with an infinitely extensible variety of streams, such as audio and video, as well as non-multimedia streams, such as a stream of stock quotes.

The `VDev` object represented device-specific parameters, such as compression format or frame rate. Likewise, the `MediaCtrl` interface is device-specific since different devices support different control interfaces. Therefore, the `MediaCtrl` is associated with the `VDev` object using the Property Service [23].

There is typically one `MediaCtrl` per stream. In some cases, however, application developers may choose to control multiple streams using the same `MediaCtrl`. For instance, the video and audio streams for a movie might have a common `MediaCtrl` to enable a single CORBA operation, such as `play`, to start both audio and video playback simultaneously.

2.2.4 Stream Controller (`StreamCtrl`)

The Stream Controller (`StreamCtrl`) interface abstracts a continuous media transfer between virtual devices (`VDevs`). It supports operations to bind two `MMDevice` objects together using a stream. Thus, the `StreamCtrl` component binds the supplier and consumer of a stream, *e.g.*, a video-camera and a display. It is the key participant in the *Stream Establishment* protocol described in Section 2.3.1.

The `StreamCtrl` object is generally instantiated by an application developer. There is one `StreamCtrl` per stream, *i.e.*, per consumer/supplier pair.

2.2.5 Stream Endpoint (`StreamEndpoint`)

The `StreamEndpoint` object is created by the `MMDevice` in response to a request for a new stream. There is one `StreamEndpoint` per stream. A `StreamEndpoint` encapsulates the transport-specific parameters of a stream. For instance, a stream that uses UDP as the transport protocol will use a host name and a port number to identify its `StreamEndpoint`.

TAO's A/V streaming service implementation of the `StreamEndpoint` uses patterns, such as Double Dispatching and Template Method [17], to allow applications to define and exchange transport-level parameters flexibly. This interaction is shown in Figure 7 and occurs as follows:

Step 1: An A/V streaming application can inherit from the `StreamEndpoint` class and override the operation `handle_connection_requested` in the new subclass `TCP_StreamEndpoint`.

Step 2: While binding two `MMDevices`, the `StreamCtrl` invokes `connect` on one `StreamEndpoint` with the peer `TCP_StreamEndpoint` as a parameter.

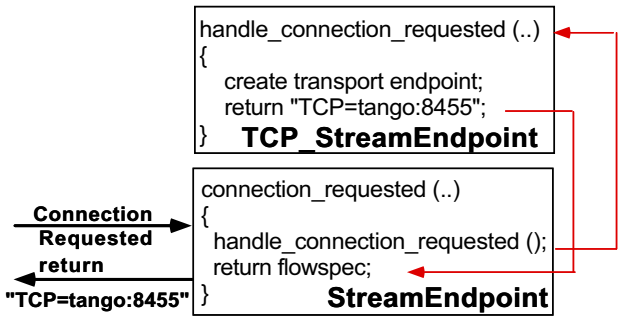

Figure 7: **Interaction Between `StreamEndpoint` and the Application**

Step 3: The `StreamEndpoint` then requests the `TCP_StreamEndpoint` to establish the connection for this stream using the transport addresses it is listening on.

Step 4: The virtual `handle_connection_requested` operation of the `TCP_StreamEndpoint` is invoked and connects with the listening transport address on the peer side.

Thus, the `StreamEndpoint` design uses patterns that allow each application to configure its own transport protocol, while reusing the generic stream establishment control logic in TAO's A/V streaming service framework.

2.3 Interaction Between Components in the OMG Audio/Video Streams Model

Section 2.2 described the structure of components that constitute the OMG A/V streaming model. The remainder of this section describes how these components *interact* to provide two key A/V streaming service features: stream establishment and flexible stream controls.

2.3.1 Stream Establishment

An important feature provided by the OMG A/V streaming specification is a standard mechanism to establish a binding between streams. Stream establishment is the process of binding two peers who need to communicate via a *stream*. Standardizing this binding mechanism is important because it allows suppliers and consumers to be developed independently, yet still be able to establish streams with one another via a common protocol.

Several components are involved in the stream establishment. A key motivation for providing an elaborate stream establishment protocol is to allow components to be configured independently of one another. This allows the stream establishment mechanism to remain standard, and yet provide sufficient hooks for framework developers to customize this process for a specific set of requirements. For instance, the `MMDevice` can be configured to use one of several concurrency strategies to create stream endpoints. Thus, at each stage of the stream establishment process, individual components can be configured to implement the desired policies.

The OMG A/V specification identifies the two peers in stream establishment as the `A` party and the `B` party. These terms define complimentary relationships, *i.e.*, a stream always has an `A` party at one end and a `B` party at the other. The `A` party may be the *sink*, *i.e.*, the consumer, of a video stream, whereas the `B` party may be the *source*, *i.e.*, the supplier, of a video stream and vice versa.

Note that the OMG A/V streaming specification defines two *distinct* IDL interfaces for the `A` and `B` type endpoint. Hence, for a given stream, there will be two distinct types for the supplier and the consumer. Thus, the OMG A/V streaming specification ensures that the complimentary relationship between suppliers and consumers is typesafe. An exception will be raised if a supplier accidentally tries to establish a stream with another supplier.

Stream establishment in TAO's A/V streaming service occurs in several steps, as illustrated in Figure 8. This figure shows a stream controller (`aStreamCtrl`) binding the `A` party together with the `B` party of a stream. The stream controller need not be collocated with either end of a stream. To simplify the example, however, we assume that the controller is collocated with the `A` party, and is called the `aStreamCtrl`. Each step shown in Figure 8 is explained below:

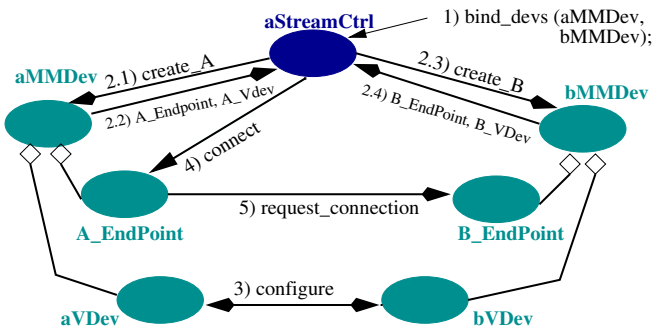

Figure 8: **Stream Establishment Steps in the A/V Streaming Service**

1. The aStreamCtrl binds two Multimedia Device (`MMDevice`) objects together: Application developers invoke the `bind_devs` operation on `aStreamCtrl`. They provide the controller with the object references of two `MMDevice` objects. These objects are factories that create the two `StreamEndpoints` of the new stream.

2. Stream Endpoint creation: In this step, `aStreamCtrl` requests the `MMDevice` objects, *i.e.*, `aMMDevice` and

bMMDevice, to create the StreamEndpoints and VDev objects. The aStreamCtrl invokes create_A and create_B operations on the two MMDevice objects. These operations request them to create A_Endpoint and B_Endpoint endpoints, respectively.

3. VDev configuration: After the two peer VDev objects have been created, they can use the configure operation to exchange device-level configuration parameters. For instance, these parameters can be used to designate the video format and compression technique used for subsequent stream transfers.

4. Stream setup: In this step, aStreamCtrl invokes the connect operation on the A_Endpoint. This operation instructs the A_Endpoint to initiate a connection with its peer. The A_Endpoint initializes its transport endpoints in response to this operation. In TAO's A/V streaming framework, applications can customize this behavior using the *Double Dispatch* pattern described in Section 2.2.5.

5. Stream Establishment: In this step, the A_Endpoint invokes the request_connection operation on its peer endpoint. The A_Endpoint passes its transport endpoint parameters, *e.g.*, hostname and port number, as parameters to this operation. When the B_Endpoint receives the request_connection operation, it initializes its end of the transport layer connection. It subsequently connects to the transport endpoint passed to it by the A_Endpoint.

After these five steps are complete, a transport-level stream has been established between the two endpoints of the stream. Section 2.3.2 describes how the *Media Controller* (MediaCtrl) can control an established stream, *e.g.*, by starting or stopping the stream.

2.3.2 Stream Control

Each MMDevice endpoint factory can be configured with an application-defined MediaCtrl interface, as described in Section 2.2.3. Each stream has one MediaCtrl and every MediaCtrl controls one stream. Thus, if a particular movie has two streams, one for audio and the other for video, it will have two MediaCtrls.

After a stream has been established by the stream controller, applications can obtain object references to their MediaCtrls from their VDev. These object references control the flow of data through the stream. For instance, a video stream might support operations like play, rewind, and stop and be used as shown below:

```
// The Audio/Video streaming service invokes this
// application-defined operation to give the
// application a reference to the media controller
// for the stream.
Video_Client_VDev::set_media_ctrl
  (CORBA::Object_ptr media_ctrl,
   CORBA::Environment &env)
{
  // "Narrow" the CORBA::Object pointer into
  // a media controller for the video stream.
  this->video_control_ =
    Video_Control::_narrow (media_ctrl);
}
```

The video control interface can be used to control the stream, as follows:

```
// Select the video to watch.
this->video_control_->select_video ("gandhi");

// Start playing the video stream.
this->video_control_->play ();

// Pause the video.
this->video_control_->stop ();

// Rewind the video 100 frames.
this->video_control_->rewind (100);
```

3 Design and Implementation of an Audio/Video Streaming Application

We have developed a CORBA-based A/V streaming application that uses the components and interfaces described in Section 2.2. This application is an enhanced version of a non-CORBA MPEG player developed at the Oregon Graduate Institute [25]. Our application plays movies using the MPEG-1 video format [21] and the Sun ULAW audio format [22]. Figure 9 shows the architecture of our A/V streaming application.

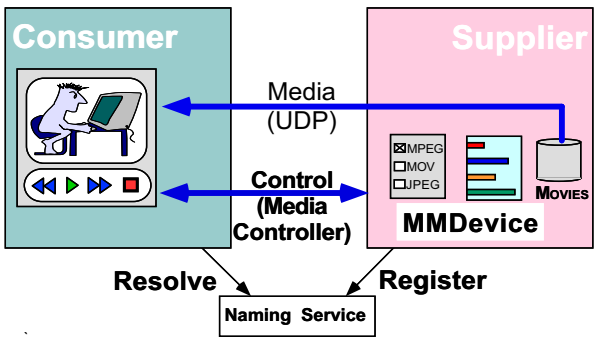

Figure 9: **Architecture of the A/V Streaming Application**

The MPEG player application uses a supplier/consumer design implemented using TAO. The consumer locates the supplier using the CORBA Naming Service [26]. Future versions of our MPEG application will use the Trading Service [26] to find suppliers that match the consumer's requirements. For instance, a consumer might want to locate a supplier that has a

particular movie or a supplier with the least number of consumers currently connected to it.

Once the consumer obtains the supplier's `MMDevice` object reference it requests the supplier to establish two streams, *i.e.*, a video stream and an audio stream, for a particular movie. The streams are established as described in Section 2.3.1. The consumer then uses the `MediaCtrl` to control the stream, as described in Section 2.2.3.

The supplier is responsible for sending A/V packets via UDP to the consumer. For each consumer, the supplier sends two streams, one each for the MPEG video packets and the Sun ULAW audio packets. The consumer decodes these streams and plays these packets in a viewer, as shown in Figure 10.

Figure 10: **The TAO Audio/Video player**

This section describes the various components of the consumer and supplier in detail. The following table illustrates the number of lines of C++ source required to develop this system and application.

Component	Lines of code
TAO CORBA ORB	61,524
TAO Audio/Video (A/V) streaming service	3,208
TAO MPEG video application	47,782

Using the ORB and the A/V streaming service greatly reduced the amount of software that otherwise would have been written from scratch.

3.1 Supplier Architecture

The supplier in the A/V streaming application is responsible for streaming `MPEG-1` video frames and `ULAW` audio samples to the consumer. The files can be stored in a filesystem accessible to the supplier process. Alternately, the video frames and the audio packets can be sent by live source, such as a video camera. Our experience with the supplier indicates that it can support ∼10 concurrent consumers simultaneously on a Sun Ultrasparc-II with 256MB of RAM over a 155 mbps ATM network.

The role of the supplier is to read audio and video frames from a file, encode them, and transmit them to the consumer across the network. Figure 11 depicts the key components in the supplier architecture.

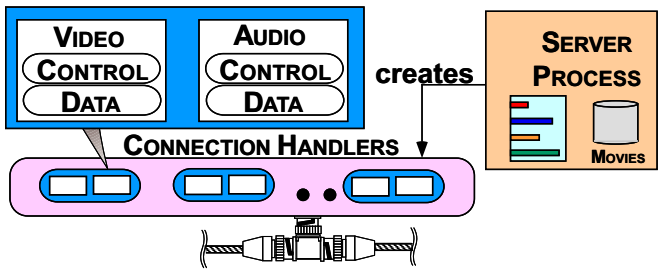

Figure 11: **TAO Audio/Video Supplier Architecture**

The main supplier process contains the `MMDevice` endpoint factory described in Section 2.2.1. The `MMDevice` creates connection handlers in response to consumer connections, using *process-based concurrency strategy*. Each connection triggers the creation of one audio process and one video process. These processes respond to multiple events. For instance, the video supplier process responds to CORBA operations, such as `play` and `rewind`, and sends video frames periodically in response to timer events.

Each component in the supplier architecture is described below.

3.1.1 The Media Controller Component

This component in the supplier process is a servant that implements the Media Controller interface (`MediaCtrl`) described in Section 2.2.3. The Media Controller responds to CORBA operations from the consumer. The interface exported by the `MediaCtrl` component represents the various operations supported by the supplier, such as `play`, `rewind`, and `stop`.

At any point in time, the supplier can be in several states, such as PLAYING, REWINDING, or STOPPED. Depending on the supplier's state, its behavior may change in response to consumer operations. For instance, the supplier ignores a consumer's `play` operation when the supplier is already in the PLAYING state. Conversely, when the supplier is in the STOPPED state, a consumer `rewind` operation transitions the supplier to the REWINDING state.

The key design forces that must be resolved while implementing `MediaCtrls` for A/V streaming are (1) allowing the same object to respond differently, based on its current state, (2) providing hooks to add new states, and (3) providing extensible operations to change the current state.

To provide a flexible design that meet these requirements, the control component is implemented using the *State* pattern [17]. This implementation is shown in Figure 12. The

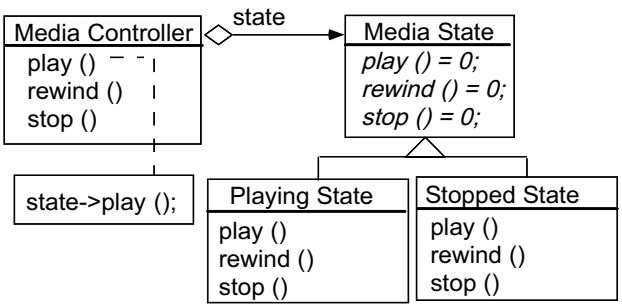

Figure 12: **State pattern implementation of the Media Controller**

`MediaCtrl` has a `state` object pointer. The object being pointed to by the Media Controller's `state` pointer represents the current state. For simplicity, the figure shows the `Playing State` and the `Stopped State`, which are subclasses of the `Media State` abstract base class. Additional states, such as the `Rewinding State`, can be added by subclassing from `Media State`.

The diagram lists three operations: `play`, `rewind` and `stop`. When the consumer invokes an operation on the `Media Controller`, this class delegates the operation to the *state object*. A state object implements the response to each operation in a particular state. For instance, the `rewind` operation in the `Playing State` contains the response of the media controller to the `rewind` operation when it is in the PLAYING state. State transitions can be made by changing the object being pointed to by the `state` pointer of the `Media Controller`.

In response to consumer operations, the current `state` object instructs the data transfer component discussed in Section 3.1.2 to modify the stream flow. For instance, when the consumer invokes the `rewind` operation on the `Media Controller` while in the STOPPED state, the `rewind` operation in the `Stopped State` object instructs the data component to play frames in reverse chronological order.

3.1.2 The Data Transfer Component

The data component is responsible for transferring data to the consumer. Our MPEG supplier application reads video frames from a `MPEG-1` file and audio frames from a Sun `ULAW` audio file. It sends these frames to the consumer, fragmenting long frames if necessary. The current implementation of the data component uses the UDP protocol to send A/V frames.

A key design challenge related to data transfer is to have the application respond to CORBA operations for the stream control objects, *e.g*, the `MediaCtrl`, as well as the data transfer events, *e.g.*, video frame timer events. An effective way to do this is to use the *Reactor* pattern, as shown in Figure 13.

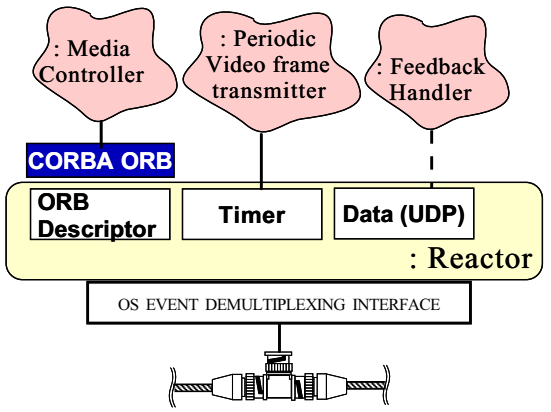

Figure 13: **Reactive Architecture of the Video Supplier**

The video supplier registers two event handlers with TAO's ORB `Reactor`. One is a signal handler for the video frame timer events. The other is a UDP socket event handler for feedback events coming from the consumer. The frames sent by the data component correspond to the current state of the `MediaCtrl` object, as outlined above. Thus, in the PLAYING state, the data component plays the audio and video frames in chronological order.

Future implementations of the data transfer component in our MPEG player application will support multiple encoding protocols via the simple flow protocol (SFP) [9]. SFP encoding encapsulates frames of various protocols within an SFP frame. It provides standard framing and sequence numbering mechanisms. SFP uses the CORBA CDR encoding mechanism to encode frame headers and uses a simple *credit-based* flow control mechanism described in [9].

3.2 Consumer Architecture

The role of the consumer is to read audio and video frames off the network, decode them, and play them synchronously. The audio and video servers stream the frames separately. A/V frame synchronization is performed on consumer. Figure 14 depicts the key components in the consumer architecture:

The original non-CORBA MPEG consumer [25] used a process-based concurrency architecture. Our CORBA-based consumer maintain this architecture to minimize changes to the code. Separate processes are used to do the buffering, decoding, and playback, as explained below:

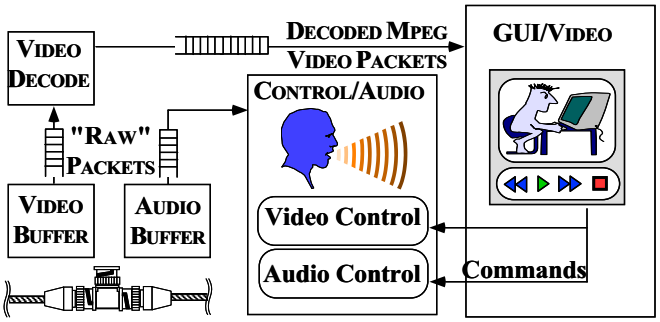

Figure 14: **TAO Audio/Video Consumer Architecture**

1. Video Buffer: The video buffering process is responsible for reading UDP packets from the network and enqueueing them in shared memory. The Video Decoder process dequeues these packets and performs MPEG decoding operations on them.

2. Audio Buffer: Similarly, the audio buffering process is responsible for reading UDP packets of the network and enqueueing them in shared memory. The Control/Audio Playback process dequeues these packets and sends them to `/dev/audio`.

3. Video Decoder: The video decoding process reads the raw packets sent to it by the Video Buffer process and decodes them according to the MPEG-1 video specification. These decoded packets are sent to the GUI/Video process, which displays them.

4. GUI/Video process: The GUI/Video process is responsible for the following two tasks:

- *GUI* – It provides a GUI to the user, where the user can select operations like `play`, `stop`, and `rewind`. These operations are sent to the Control/Audio process via a UNIX domain socket [27].
- *Video* – This component is responsible for displaying video frames to the user. The decoded video frames are stored in a shared memory queue.

5. Control/Audio Playback process: The Control/Audio process is responsible for the following tasks:

- *Control* – This component receives control messages from the GUI process and sends the appropriate CORBA operation to the `MediaCtrl` servant in the supplier process.
- *Audio playback* – The audio playback component is responsible for dequeueing audio packets from the Audio Buffer process and playing them back using the multimedia sound hardware. Decoding is unnecessary because the supplier uses the ULAW format. Therefore, the data received can be directly written to the sound port, which is `/dev/audio` on Solaris.

4 Performance Results

This section describes the design and results of three performance experiments we conducted using TAO's A/V streaming service.

4.1 CORBA/ATM Testbed

The experiments in this section were conducted using a FORE systems ASX-1000 ATM switch connected to two dual-processor UltraSPARC-2s running Solaris 2.5.1. The ASX-1000 is a 96 Port, OC12 622 Mbs/port switch. Each UltraSPARC-2 contains a 300 MHz Super SPARC CPUs with a 1 Megabyte cache per-CPU. The Solaris 2.5.1 TCP/IP protocol stack is implemented using the STREAMS communication framework [28].

Each UltraSPARC-2 has 256 Mbytes of RAM and an ENI-155s-MF ATM adaptor card, which supports 155 Megabits per-sec (Mbps) SONET multimode fiber. The Maximum Transmission Unit (MTU) on the ENI ATM adaptor is 9,180 bytes. Each ENI card has 512 Kbytes of on-board memory. A maximum of 32 Kbytes is allotted per ATM virtual circuit connection for receiving and transmitting frames (for a total of 64 Kb). This allows up to eight switched virtual connections per card. The CORBA/ATM hardware platform is shown in Figure 15.

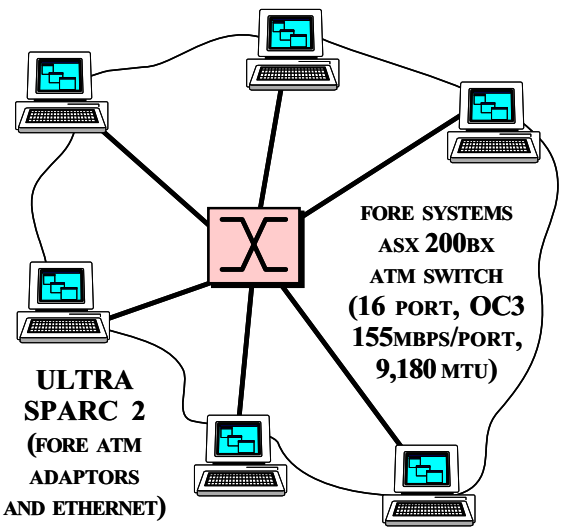

Figure 15: Hardware for the CORBA/ATM Testbed

4.2 CPU Usage of the MPEG decoder

The aim of this experiment is to determine the CPU overhead associated with decoding and playing MPEG-1 frames in software. To measure this, we used the MPEG/ULAW A/V player application described in Section 3.

We used the application to view two movies, one of size 128x96 pixels and the other of size 352x240 pixels. We measured the percentage CPU usage for different *frame rates*. The frame rate is the number of video frames displayed by the viewer per second.

The results are shown in Figure 16. These results indicate

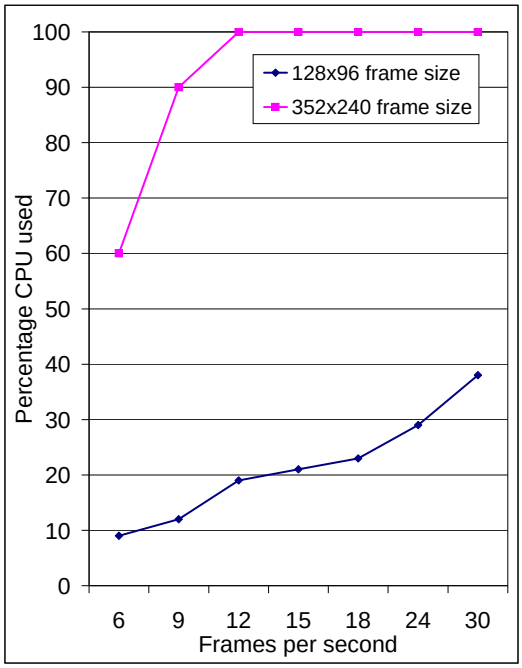

Figure 16: **CPU Usage of the MPEG Decoder**

that for large frame sizes (352x240), MPEG decoding in software becomes expensive, and the CPU usage becomes 100% while playing 12 frames per second, or higher. However, for smaller frame sizes (128x96), MPEG decoding in software does not cause heavy CPU utilization. At 30 frames per second, CPU utilization is ∼38%.

4.3 A/V Stream Throughput

The aim of this experiment is to illustrate that TAO's A/V streaming service does not introduce appreciable overhead in transporting data. To demonstrate this, we wrote a TCP-based data streaming component and integrated it with TAO's A/V service. The producer in this application establishes a stream with the consumer, using the stream establishment mechanism discussed in Section 2.3.1. Once the stream is established, it streams data via TCP to the consumer.

We measured the throughput, *i.e.*, the number of bytes per second sent by the supplier to the consumer, obtained by this streaming application. We then compared this throughput with the following two configurations:

- *TCP transfer – i.e.*, by a pair of application processes that do not use the OMG stream establishment mechanism. In this case, sockets and TCP were the transport mechanism. This is the "ideal" case since there is no additional ORB-related or presentation layer overhead.
- *ORB transfer – i.e.*, the throughput obtained by a stream that used an *octet stream* passed through the TAO [11] CORBA ORB. In this case, the IIOP data path was the transport mechanism.

We measured the throughput obtained by varying the buffer size of the sender, *i.e.*, the number of bytes written by the supplier in one `write` system call. In each stream, the supplier sent 64 megabytes of data to the consumer.

The results shown in Figure 17 indicate that, as expected, the A/V streaming service does not introduce any appreciable overhead to streaming the data. In the case of using the IIOP path through the ORB as the transport layer can incur more performance overhead. This overhead could arise from the dynamic memory allocation, data-copying, and marshaling/demarshaling performed by the ORB's IIOP protocol engine [8].But TAO could achieve almost the socket performance at higher buffer sizes due to its optimizations, in particular for octet data [6]

The largest disparity occurred for smaller buffer sizes, where the performance of the ORB was approximately half that of the TCP and A/V streaming implementations. As the buffer size increases, however, the ORB performance improves considerably and attains nearly the same throughput as TCP and A/V streaming. Clearly, there is a fixed amount of overhead in the ORB that is amortized and minimized as the size of the data payload increases.

4.4 Stream Establishment Latency

This experiment measures the time required to establish a stream using TAO's implementation of the OMG CORBA A/V stream establishment protocol described in Section 2.3.1. We measured the stream establishment latency for the two concurrency strategies, process-based strategy and reactive strategy, described in Section 2.2.1.

The timer starts when the consumer gets the object reference for the supplier's `MMDevice` servant from the Naming

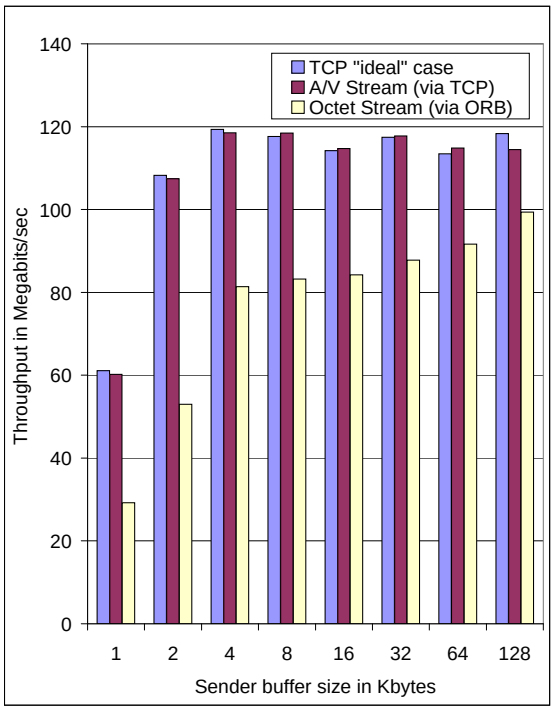

Figure 17: **Throughput Results**

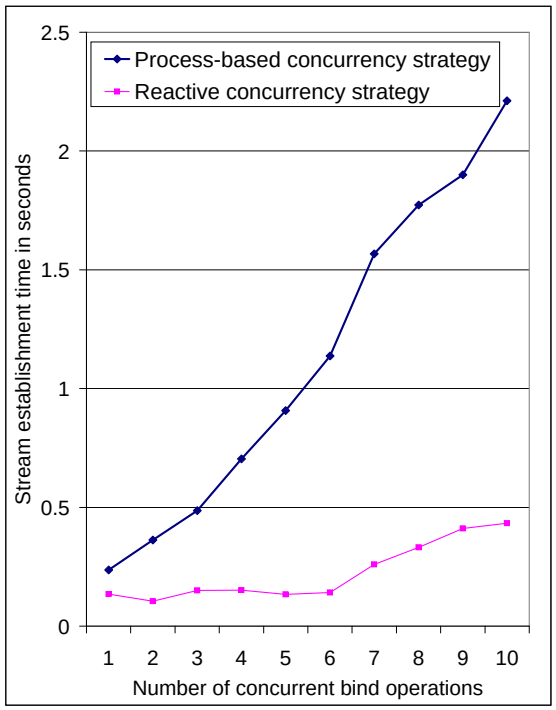

Figure 18: **Stream Establishment Latency Results**

Service. The timer stops when the stream has been established, *i.e.*, when a transport-layer TCP connection has been established between the consumer and the supplier.

We measured the stream establishment time as the number of concurrent consumers establishs connections with the supplier increased from 1 to 10. The results are shown in Figure 18. When the supplier's `MMDevice` is configured to use the process-based concurrency strategy (described in Section 2.2.1), the time taken to establish the stream is higher, due to the overhead of process creation. For instance, when 10 concurrent consumers establish a stream with the producer simultaneously, the average latency observed is about 2.25 seconds with the process-based concurrency strategy. With the reactive concurrency strategy, the latency is only about 0.4 seconds.

The process-based strategy is well-suited for supplier devices that have multiple streams, *e.g.*, a video camera that broadcasts a live feed to many clients. In contrast, the reactive concurrency strategy is well-suited for consumer devices that have few streams, *e.g.*, a display device that has only one or two streams.

5 Related Work

Distributed multimedia streaming frameworks have received increasing focus in the R&D community. A popular Internet-based streaming mechanism is Realvideo [1], from Real Networks. Like the MPEG application described in Section 3, the Realvideo system uses the UDP protocol to send A/V packets from the supplier to the consumer. However, the Realvideo application uses proprietary stream establishment and control protocols, as well as a proprietary audio and video format. Microsoft's Vxtreme [2] is another popular streaming mechanism that is similar to Realvideo.

IONA Inc. has developed *Orbix MX* [29], which is an implementation of the CORBA A/V streaming specification. The key features of Orbix MX are similar to TAO's implementation of the A/V Streaming service, *i.e.*, support for multiple transport protocols, flexible stream controls, and support for multiple concurrency strategies while creating stream endpoints.

The NEC C&C Laboratories have implemented a preliminary prototype of the A/V streaming specification [30]. Their prototype has been implemented with Orbix2.2 and OrbixWeb2.0.1. The `flowAdapters` in their implementation are similar to the `StreamEndpoint` of the A/V specifica-

tion, *i.e.*, they deal with the network specific aspects of a `flow` within a stream. `Flows` are a forthcoming extension to TAO's A/V implementation.

The Distributed Multimedia Research Group at the University of Lancaster is working on standardization of Open Distributed Systems using CORBA middleware. Towards this goal, they propose the *explicit open bindings* concept [31], which is a mechanism using which application developers can explicitly set up an additional transport connection between two CORBA objects. This connection can then be used for streaming data.

The H.323 standards specified by `ITU` ensures interoperability between heterogeneous multimedia devices over heterogeneous networks. The H.323 document defines standards for video/audio coding/decoding, signalling and control and also provides facilities for network and bandwidth management. The A/V streaming service can interoperate with H.323 clients/servers using an `H.323-Adapter`. The `H.323-Adapter` is a CORBA object that converts the H.323 control messages into appropriate Audio/Video CORBA control messages.

6 Concluding Remarks

The demand for high quality multimedia streaming is growing, both over the Internet and for intranets. Distributed object computing is also maturing at a rapid rate due to middleware technologies like CORBA. The flexibility and adaptability offered by CORBA makes it very attractive for use in streaming technologies, as long as the requirements of performance-sensitive multimedia applications can be met.

This paper illustrates an approach to building standards-based, flexible, adaptive, multimedia streaming applications using CORBA. While designing and implementing the CORBA A/V streaming service, we learned a number of lessons. First, we found that CORBA simplifies a number of common network programming tasks, such as parsing untyped data and performing byte-order conversions. Second, we found that using CORBA to define the operations supported by a supplier in an IDL interface made it much easier to express the capabilities of the application, as described in Section 2.2.3.

However, our measurements described in Section 4 revealed that while CORBA provides solutions to many recurring problems in network programming, using CORBA for data transfer in bandwidth-intensive applications is not as efficient as using lower-level protocols like TCP, UDP, or ATM directly. Thus, an important benefit of the TAO A/V Streaming service is to provide applications the advantages of using CORBA IIOP in their stream establishment and control modules, while allowing the use of more efficient transport-layer protocols for data streaming.

Enhancing an existing A/V streaming application to use CORBA was a key design challenge. By applying patterns, such as the *State*, *Strategy*, [17] and *Reactor* [19], we found it was much easier to address these design issues. Thus, the use of patterns helped us rework the architecture of an existing MPEG A/V player and make it more amenable to a distributed technology such as CORBA.

Building the CORBA A/V streaming service also helped us improve TAO, the CORBA ORB used to implement the service. An important feature added to TAO was support for *nested upcalls*. This feature allows a CORBA-enabled application to respond to incoming CORBA operations, while it is making a CORBA operation on a remote object. During the development of the A/V streaming service, we also applied many optimization to TAO and its IDL compiler, particularly for sequences of `octets` and the `CORBA::Any` type.

All the C++ source code, documentation, and benchmarks for TAO and its A/V streaming service is available at `www.cs.wustl.edu/~schmidt/TAO.html`.

Acknowledgments

We would like to thank Alexander Arulanthu for implementing the CORBA Property Service using TAO. Also, we would like to thank Marina Spivak and Sergio Flores for implementing the CORBA Naming service. Finally, we would like to thank Dr. Aniruddha Gokhale and Irfan Pyarali for their extensive constructive comments on this paper.

References

[1] RealNetworks, "Realvideo player." www.real.com, 1998.

[2] Vxtreme, "Vxtreme player." www.microsoft.com/netshow/vxtreme/, 1998.

[3] J. Hu, S. Mungee, and D. C. Schmidt, "Principles for Developing and Measuring High-performance Web Servers over ATM," in *Proceeedings of INFOCOM '98*, March/April 1998.

[4] D. C. Schmidt and T. Suda, "An Object-Oriented Framework for Dynamically Configuring Extensible Distributed Communication Systems," *IEE/BCS Distributed Systems Engineering Journal (Special Issue on Configurable Distributed Systems)*, vol. 2, pp. 280–293, December 1994.

[5] I. Pyarali, T. H. Harrison, and D. C. Schmidt, "Design and Performance of an Object-Oriented Framework for High-Performance Electronic Medical Imaging," *USENIX Computing Systems*, vol. 9, November/December 1996.

[6] A. Gokhale and D. C. Schmidt, "Optimizing a CORBA IIOP Protocol Engine for Minimal Footprint Multimedia Systems," *Journal on Selected Areas in Communications special issue on Service Enabling Platforms for Networked Multimedia Systems*, vol. 17, Sept. 1999.

[7] A. Gokhale and D. C. Schmidt, "Measuring and Optimizing CORBA Latency and Scalability Over High-speed Networks," *Transactions on Computing*, vol. 47, no. 4, 1998.

[8] A. Gokhale and D. C. Schmidt, "Measuring the Performance of Communication Middleware on High-Speed Networks," in *Proceedings of SIGCOMM '96*, (Stanford, CA), pp. 306–317, ACM, August 1996.

[9] Object Management Group, *Control and Management of A/V Streams specification*, OMG Document telecom/97-05-07 ed., October 1997.

[10] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.2 ed., Feb. 1998.

[11] D. C. Schmidt, D. L. Levine, and S. Mungee, "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications*, vol. 21, pp. 294–324, Apr. 1998.

[12] C. D. Gill, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-Time CORBA Scheduling Service," *The International Journal of Time-Critical Computing Systems, special issue on Real-Time Middleware*, 2000, to appear.

[13] A. Gokhale and D. C. Schmidt, "Evaluating the Performance of Demultiplexing Strategies for Real-time CORBA," in *Proceedings of GLOBECOM '97*, (Phoenix, AZ), IEEE, November 1997.

[14] T. H. Harrison, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-time CORBA Event Service," in *Proceedings of OOPSLA '97*, (Atlanta, GA), ACM, October 1997.

[15] F. Kuhns, D. C. Schmidt, and D. L. Levine, "The Design and Performance of a Real-time I/O Subsystem," in *Proceedings of the 5^{th} IEEE Real-Time Technology and Applications Symposium*, (Vancouver, British Columbia, Canada), pp. 154–163, IEEE, June 1999.

[16] D. C. Schmidt, S. Mungee, S. Flores-Gaitan, and A. Gokhale, "Alleviating Priority Inversion and Non-determinism in Real-time CORBA ORB Core Architectures," in *Proceedings of the 4^{th} IEEE Real-Time Technology and Applications Symposium*, (Denver, CO), IEEE, June 1998.

[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[18] J. Hu, I. Pyarali, and D. C. Schmidt, "Measuring the Impact of Event Dispatching and Concurrency Models on Web Server Performance Over High-speed Networks," in *Proceedings of the 2^{nd} Global Internet Conference*, IEEE, November 1997.

[19] D. C. Schmidt, "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching," in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), pp. 529–545, Reading, MA: Addison-Wesley, 1995.

[20] D. D. et al., "Vaudeville: A High Performance, Voice Activated Teleconferencing Application," Department of Computer Science, Technical Report WUCS-96-18, Washington University, St. Louis, June 1996.

[21] International Organisation for Standardisation, *Coding Of Moving Pictures And Audio For Digital Storage Media At Up To About 1.5 Mbit/s*, 1993.

[22] Sun Microsystems, Inc., *Sun Audio File Format*, 1992.

[23] Object Management Group, *Property Service Specification*, 1.0 ed., July 1996.

[24] T. Harrison and D. C. Schmidt, "Thread-Specific Storage: A Pattern for Reducing Locking Overhead in Concurrent Programs," in *OOPSLA Workshop on Design Patterns for Concurrent, Parallel, and Distributed Systems*, ACM, October 1995.

[25] S. Chen, C. Pu, R. Staehli, C. Cowan, and J. Walpole, "A Distributed Real-Time MPEG Video Audio Player," in *Fifth International Workshop on Network and Operating System Support of Digital Audio and Video*, Apr. 1995.

[26] Object Management Group, *CORBAServices: Common Object Services Specification, Revised Edition*, 97-12-02 ed., Nov. 1997.

[27] W. R. Stevens, *UNIX Network Programming, Second Edition*. Englewood Cliffs, NJ: Prentice Hall, 1997.

[28] D. Ritchie, "A Stream Input–Output System," *AT&T Bell Labs Technical Journal*, vol. 63, pp. 311–324, Oct. 1984.

[29] IONA, "IONA Orbix MX." www.iona.com, 1998.

[30] J.-P. Redlich, "A Distributed Object Architecture for QoS-sensitive Networking," in *OpenArch98*, April 1998.

[31] T. Fitzpatrick, G. Blair, G. Coulson, N. Davies, and P. Robin, "Supporting Adaptive Multimedia Applications through Open Bindings," in *International Conference on Configurable Distributed Systems (ICCDS '98)*, May 1998.

[32] E. Eide, K. Frei, B. Ford, J. Lepreau, and G. Lindstrom, "Flick: A Flexible, Optimizing IDL Compiler," in *Proceedings of ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI)*, (Las Vegas, NV), ACM, June 1997.

A Overview of the CORBA Reference Model

CORBA Object Request Brokers (ORBs) allow clients to invoke operations on distributed objects without concern for object location, programming language, OS platform, communication protocols and interconnects, and hardware. Figure 19 illustrates the key components in the CORBA reference model that collaborate to provide this degree of portability, interoperability, and transparency.[1] Each component in the CORBA

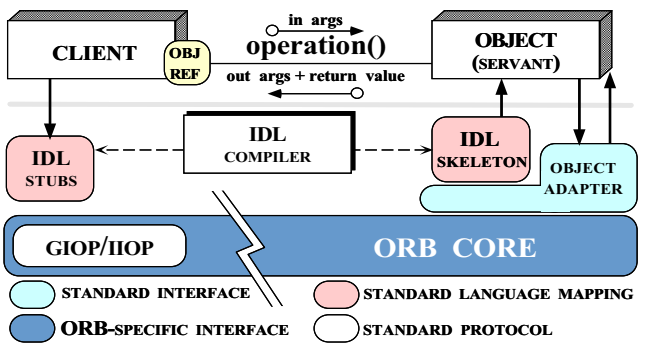

Figure 19: Key Components in the CORBA 2.x Reference Model

reference model is outlined below:

Client: A client is a role that obtains references to objects and invokes operations on them to perform application tasks. Objects can be remote or collocated relative to the client. Ideally, a client can access a remote object just like a local object, *i.e.*, `object→operation(args)`. Figure 19 shows how the underlying ORB components described below transmit remote operation requests transparently from client to object.

[1] This overview only focuses on the CORBA components relevant to this paper. For a complete synopsis of CORBA's components see [10].

Object: In CORBA, an object is an instance of an OMG Interface Definition Language (IDL) interface. Each object is identified by an *object reference*, which associates one or more paths through which a client can access an object on a server. An *Object Id* associates an object with its implementation, called a servant, and is unique within the scope of an Object Adapter. Over its lifetime, an object has one or more servants associated with it that implement its interface.

Servant: This component implements the operations defined by an OMG IDL interface. In languages like C++ and Java that support object-oriented (OO) programming, servants are implemented using one or more class instances. In non-OO languages, like C, servants are typically implemented using functions and `structs`. A client never interacts with servants directly, but always through objects identified by object references.

ORB Core: When a client invokes an operation on an object, the ORB Core is responsible for delivering the request to the object and returning a response, if any, to the client. An ORB Core is implemented as a run-time library linked into client and server applications. For objects executing remotely, a CORBA-compliant ORB Core communicates via a version of the General Inter-ORB Protocol (GIOP), most commonly the Internet Inter-ORB Protocol (IIOP) that runs atop the TCP transport protocol. In addition, custom Environment-Specific Inter-ORB protocols (ESIOPs) can also be defined.

OMG IDL Stubs and Skeletons: IDL stubs and skeletons serve as a "glue" between the client and servants, respectively, and the ORB. Stubs implement the *Proxy* pattern [17] and provide a strongly-typed, *static invocation interface* (SII) that marshals application parameters into a common data-level representation. Conversely, skeletons implement the *Adapter* pattern [17] and demarshal the data-level representation back into typed parameters that are meaningful to an application.

IDL Compiler: An IDL compiler transforms OMG IDL definitions into stubs and skeletons that are generated automatically in an application programming language like C++ or Java. In addition to providing programming language transparency, IDL compilers eliminate common sources of network programming errors and provide opportunities for automated compiler optimizations [32].

Object Adapter: An Object Adapter associates servants with objects, creates object references, demultiplexes incoming requests to servants, and collaborates with the IDL skeleton to dispatch the appropriate operation upcall on a servant. CORBA 2.2 portability enhancements [10] define the Portable Object Adapter (POA), which supports multiple nested POAs per ORB. Object Adapters enable ORBs to support various types of servants that possess similar requirements. This design results in a smaller and simpler ORB that can support a wide range of object granularities, lifetimes, policies, implementation styles, and other properties.

B Overview of the CORBA Property Service

B.1 Motivation

A CORBA object consists of (1) an identify, *i.e.*, an object reference, (2) an interface, *i.e.*, defined in IDL and consisting of operations and attributes, and (3) an implementation of the interface, *i.e.*, one or more servants. The operations and attributes in an IDL interface are *static*, *i.e.*, they are defined *a priori*. In general, statically-typed IDL interfaces enhance application robustness by preventing accidental violations of the typesystem.

When building frameworks like the A/V streaming service described in this paper, however, certain attributes cannot be defined statically because the names, types, and values of these attributes will vary depending on how the application uses the framework. For example, when a video output device is represented as an `MMDevice`, the typical attributes of `MMDevice` might be *video encoding format* and *frame rate*. In contrast, if it is an audio output device, the `MMDevice` attributes might be *audio format* and *sample rate*, as shown in Figure 20.

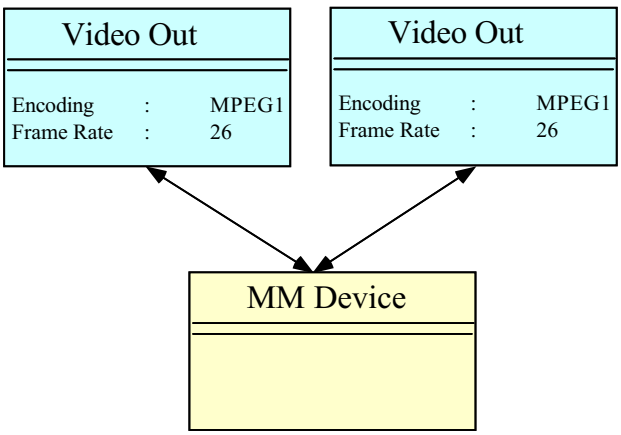

Figure 20: **Properties for AV Streams**

To maximize flexibility, therefore, the A/V streaming framework requires attributes that contain *dynamic* types and values. The CORBA Property Service provides this flexibility via the following features:

Dynamic property association: The Property Service provides the ability to dynamically associate named values with objects more flexibly than the statically defined IDL-type system. Thus, they allow applications to associate *dynamic attributes* with object. By using the Property Service, applica-

tions can create and delete new properties, change the values of properties, and associate properties with modes, such as `readonly` mode.

Dynamically typed values: The Property Service defines operations to create and manipulate sets of *name-value* and *name-value-mode* tuples. Names are OMG IDL strings and values are OMG IDL `anys`. The use of `anys` allows a Property Service implementation to handle any value that can be represented in the OMG IDL-type system.

Figure 3 shows how the `MMDevice` interface uses the Property Service to store properties related to the multimedia device that it represents.

B.2 Design Overview

The UML diagram in Figure 21 shows the components in the Property Service. These components are described below.

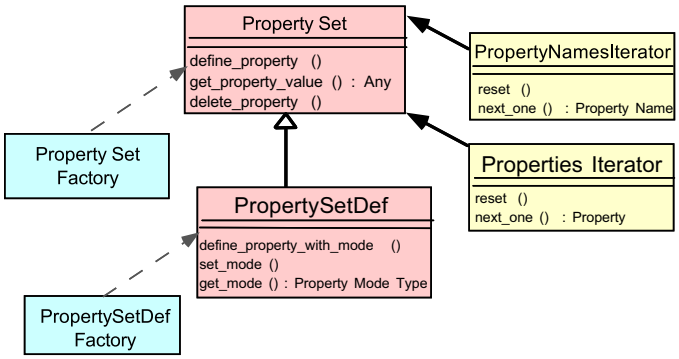

Figure 21: **UML for the Property Service**

PropertySet: This interface supports a set of properties. A `property` is a tuple consisting of `<property_name, property_value>`. The `property_name` is a `string` that names the property. The `property_value` is a type `any` that contains the value assigned to the property.

PropertySetDef: This interface is a subclass of the `PropertySet` interface that exposes characteristics of each property, *e.g.*, readonly or read/write access. There are two factory interfaces: one for the `PropertySet` interface and the other for the `PropertySetDef` interface. Iterators are defined to iterate over the property names and properties.

B.3 Associating Properties with CORBA Objects

Properties can be associated with a CORBA object in either of the following ways:

Inheritance: The application IDL interface can inherit directly from the `PropertySet` or `PropertySetDef` interfaces, as shown in Figure 22. In this approach,

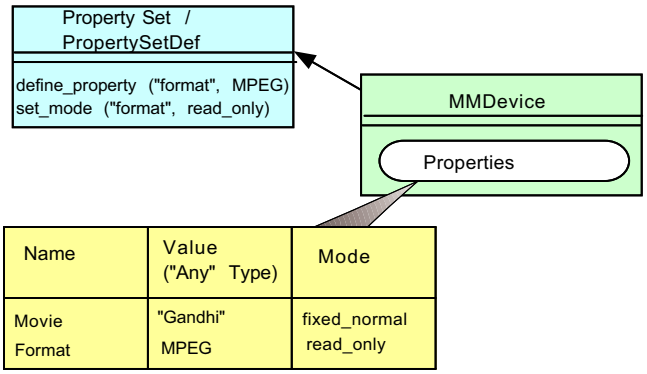

Figure 22: **Using the Property Service Via Inheritance**

interface `MMDevice` inherits from `PropertySet` or `PropertySetDef` interface. If it is a public inheritance, clients of `MMDevice` will also have access to the Property Service operations. For example, a client may define a new property and associate that with a servant that implements `MMDevice`.

Factory interfaces: As an alternative to inheritance, *factory methods* [17] can be used to create `PropertySets` or `PropertySetDefs`. This approach is shown in Figure 23. In this approach, the object `AV_Server` obtains one or more

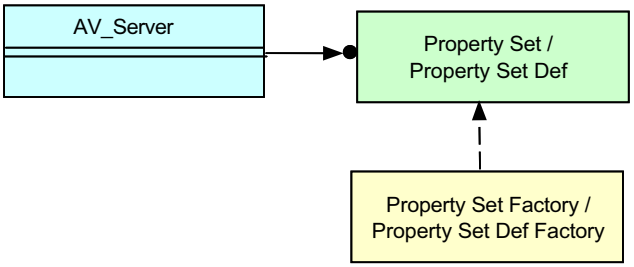

Figure 23: **Using The Property Service Via Factory Methods**

`PropertySet` or `PropertySetDef` objects through the factory methods. Objects can keep properties under different `PropertySets` depending on how they are related.

Objects should use the inheritance approach, if they want to allow the clients to access the properties with the servants. For example, `MMDevice` interface of A/V streams inherits from the `PropertySet` interface and hence the clients can invoke property service operations on the servants. Factory approach of the property service should be used when the objects want to keep track of some properties internally. For example, as

shown in Figure 23, an AV_Server object can have a sequence of PropertySets or PropertySetDefs to keep track of the various properties of all its clients.

B.4 Advanced Features of the Property Service

As with CORBA attributes, clients can read and write property values. In addition, clients can use the Property Service to dynamically create and delete properties associated with a remote object. Clients can manipulate properties individually or in *batched mode* using a sequence of the Property data type called *Properties*. For example, to define new properties, the define_properties operation can be called with a sequence of Properties, which are a dynamically-sized array of name-value pairs.

If objects support the PropertySetDef interface, clients can create and manipulate properties and their characteristics, such as the property mode *e.g.*, readonly and fixed_readonly. The PropertySetDef interface also provides operations for clients to retrieve constraint information about a PropertySet, such as the list of all the property types that are allowed in this PropertySet or the list of all the property names that are allowed in this PropertySet. This constraint information can be specified using the factory creation operations when the PropertySet is created.