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

Sumedh Mungee, Nagarajan Surendran
Yamuna Krishnamurthy
{sumedh,naga,yamuna}@cs.wustl.edu
Department of Computer Science, Washington University
St. Louis, MO 63130

Douglas C. Schmidt
{schmidt}@uci.edu
Electrical & Computer Engineering Dept.
University of California, Irvine, CA 92697*

This paper has been submitted as a chapter in the book *Design and Management of Multimedia Information Systems: Opportunities and Challenges*, edited by Mahbubur Syed and to be published by Idea Group Publishing, Hershey, USA, in 2000

1 Introduction

Motivation: Advances in network bandwidth and CPU processing power have enabled the emergence of multimedia applications, such as tele-conferencing or streaming video, that exhibit significantly more diverse and stringent quality-of-service (QoS) requirements than traditional data-oriented applications, such as file transfer or email. For instance, popular Internet-based streaming mechanisms, such as Realvideo [RealNetworks, 1998] and Vxtreme [Vxtreme, 1998], allow suppliers to transmit continuous streams of audio and video packets to consumers. Likewise, non-continuous media applications, such as medical imaging servers [Hu et al., 1998] and network management agents [Schmidt and Suda, 1994], employ streaming to transfer bulk data efficiently from suppliers to consumers.

However, many distributed multimedia applications rely on custom and/or proprietary low-level stream establishment and signaling mechanisms to manage and control the presentation of multimedia content. These types of applications run the risk of becoming obsolete as new protocols and services are developed [Huard and Lazar, 1998]. Fortunately, there is a general trend to move from programming custom applications manually to integrating applications using reusable components based on open distributed object computing (DOC) middleware, such as CORBA [Object Management Group, 1999], DCOM [Box, 1997], and Java RMI [Wollrath et al., 1996].

Although DOC middleware is well-suited to handle request/response interactions among client/server applications, the stringent QoS requirements of multimedia applications have historically precluded DOC middleware from being used as their data transfer mechanism [Pyarali et al., 1996]. For instance, inefficient CORBA Internet Inter-ORB Protocol (IIOP) [Gokhale and Schmidt, 1999] implementations perform excessive data-copying and memory allocation *per-request*, which increases packet latency [Gokhale and Schmidt, 1998]. Likewise, inefficient marshaling/demarshaling in DOC middleware decreases streaming data throughput [Gokhale and Schmidt, 1996].

As the performance of DOC middleware steadily improves, however, the stream establishment and control components of distributed multimedia applications can benefit greatly from the portability and flexibility provided by DOC middleware. Therefore, to facilitate the development of standards-based distributed multimedia applications, the Object Management Group (OMG) has defined the CORBA Audio/Video (A/V) Streaming Service specification [OMG, 1997a], which defines common interfaces and semantics necessary to control and manage A/V streams.

The CORBA A/V Streaming Service specification defines an architecture for implementing open distributed multimedia streaming applications. This architecture integrates (1) well-defined modules, interfaces, and semantics for stream establishment and control with (2) efficient data transfer protocols for multimedia data transmission. In addition to defining standard stream establishment and control mechanisms, the CORBA A/V Streaming Service specification allows distributed multimedia applications to leverage the inherent portability and flexibility benefits provided by standards-based DOC middleware.

Our prior research on CORBA middleware has explored the efficiency, predictability, and scalability aspects of ORB endsystem design, including static [Schmidt et al., 1998a] and dynamic [Gill et al., 2000] scheduling, I/O subsystem [Kuhns et al., 1999] and pluggable ORB transport protocol [O'Ryan et al., 2000] in-

*This work was supported in part by Boeing, NSF grant NCR-9628218, DARPA contract 9701516, and Sprint.

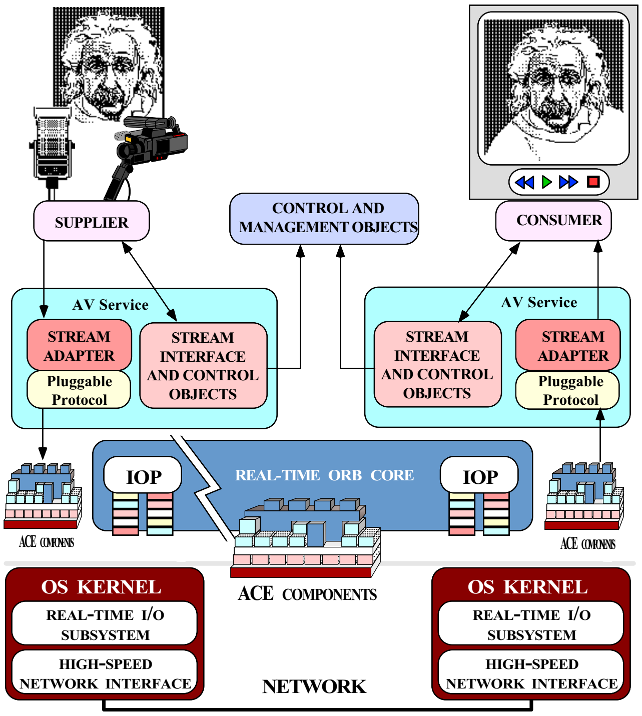

Figure 1: Layering of TAO's A/V Streaming Service Atop the TAO ORB Endsystem

tegration, synchronous [Schmidt et al., 2000a] and asynchronous [Arulanthu et al., 2000] ORB Core architectures, event processing [Harrison et al., 1997], optimization principle patterns for ORB performance [Pyarali et al., 1999], and the performance of various commercial and research ORBs [Gokhale and Schmidt, 1996, Schmidt et al., 1998b] over high-speed ATM networks. This chapter focuses on another important topic in ORB endsystem research: *the design and performance of the CORBA A/V Streaming Service specification*.

The vehicle for our research on the CORBA A/V Streaming Service is TAO [Schmidt et al., 1998a]. TAO is a high-performance, real-time Object Request Broker (ORB) endsystem targeted for applications with deterministic and statistical QoS requirements, as well as best effort requirements. The TAO ORB endsystem contains the network interface, OS I/O subsystem, communication protocol, and CORBA-compliant middleware components and services shown in Figure 1.

Figure 1 also illustrates how TAO's A/V Streaming Service is built over the TAO ORB subsystem. TAO's real-time I/O (RIO) [Kuhns et al., 2000] subsystem runs in the OS kernel and sends/receives requests to/from clients across high-speed, QoS-enabled networks, such as ATM or IP Integrated [IETF, 2000b] and Differentiated [IETF, 2000a] Services. TAO's ORB components, such as its ORB Core, Object Adapter, stubs/skeletons, and servants, run in user-space and handle connection management, data transfer, endpoint and request demultiplexing, concurrency, (de)marshaling, and application operation processing. TAO's A/V Streaming Service is implemented atop its user-space ORB components. At the heart of TAO's A/V Streaming Service is its *pluggable A/V protocol framework*. This framework provides the "glue" that integrates TAO's A/V Streaming Service with the underlying I/O subsystem protocols and network interfaces.

Chapter organization: The remainder of this chapter is organized as follows: Section 2 illustrates how we applied patterns to develop and optimize the CORBA A/V Streaming Service to support the standard OMG interfaces; Section 3 describes two case studies that illustrate how to develop distributed multimedia applications using TAO's A/V Streaming Service and its pluggable A/V protocol framework; Section 4 presents the results of empirical benchmarks we conducted to illustrate the performance of TAO's A/V Streaming Service; Section 5 presents concluding results. For completeness, Appendix A outlines the intents of all the patterns applied in TAO's A/V Streaming Service; Appendix B summarizes the CORBA reference model and Appendix C illustrates the various point-to-point and point-to-multipoint stream and flow endpoint bindings implemented in TAO's A/V Streaming Service.

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

This section first presents an overview of the key architectural components in the CORBA A/V Streaming Service. We then summarize the key design challenges faced when developing TAO's CORBA A/V Streaming Service and outline how we applied patterns [Gamma et al., 1995, Buschmann et al., 1996, Schmidt et al., 2000b] to resolve these challenges. Finally, we describe the design and performance of the pluggable A/V protocol framework integrated into TAO's A/V Streaming Service.

2.1 Overview of the CORBA Audio/Video Streaming Service Specification

The CORBA Audio/Video (A/V) Streaming Service specification [OMG, 1997a] defines an architectural model and standard OMG IDL interfaces that can be used to build interoperable distributed multimedia streaming applications. Below, we outline the architectural components and goals of the CORBA A/V Streaming Service specification.

2.1.1 Synopsis of Components in the CORBA A/V Streaming Service

The CORBA A/V Streaming Service specification defines *flows* as a continuous transfer of media between two multimedia devices. Each of these flows is terminated by a *flow endpoint*. A set of flows, such as audio flow, video flow and data flow, constitute a *stream*, which is terminated by a *stream endpoint*. A stream endpoint can have multiple flow endpoints.

Figure 2 shows a *multimedia stream*, which is represented as a flow between two *flow endpoints*. One flow endpoint acts as

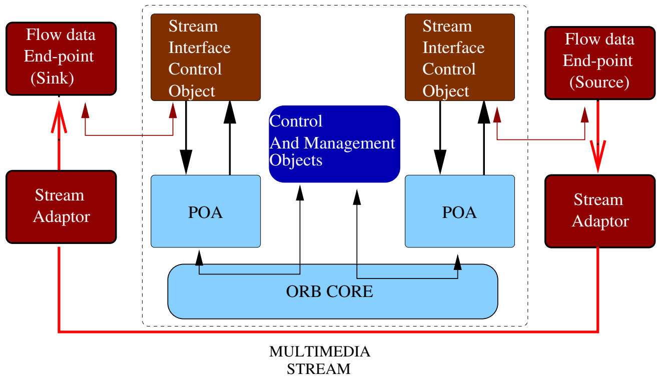

Figure 2: CORBA A/V Streaming Service Architecture

a source of the data and the other flow 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 communication protocols that are more suitable for multimedia streaming than IIOP. Maintaining this separation of concerns is crucial to meeting end-to-end QoS requirements.

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 CORBA A/V Streaming Service specification defines the interfaces and interactions of the *Stream Interface Control Objects* and the Control and Management objects. Section 2.3 describes the various components in Figure 2 in detail.

2.1.2 Synopsis of Goals for the CORBA A/V Streaming Service

The goals of the CORBA A/V Streaming Service include the following:

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

Support for multiple data transfer protocols: The CORBA A/V Streaming Service architecture separates its stream establishment and control protocols from its data transfer protocols, such as TCP, UDP, RTP, or ATM, thereby allowing applications to select the most suitable data transfer protocols for a particular network environment or set of application requirements.

Provide interoperability of flows: A *flow specification* is passed between two stream endpoints to convey per-flow information, such as format, network host name and address, and flow protocol, required to bind or communication between two multimedia devices.

Support many types of sources and sinks: Common stream sources include video-on-demand servers, video cameras attached to a network, or stock quote servers. Common sinks include video-on-demand clients, display devices attached to a network, or stock quote clients.

2.2 Overview of Design Challenges and Resolutions

Below, we present an overview of the key challenges faced when we developed TAO's CORBA A/V Streaming Service and outline how we applied patterns [Gamma et al., 1995, Schmidt et al., 2000b] to resolve these challenges. Sections 2.3 and 2.4 then examine these design and optimization pattern techniques in more depth. Appendix A outlines the intents of all the patterns applied in TAO's A/V Streaming Service.

Flexibility in stream endpoint creation strategies: The CORBA A/V Streaming Service 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 their stream components. For instance, our past studies of Web server performance [Hu et al., 1997, Hu et al., 1998] 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 case-study application (described in Section 3.1.1) 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 (described in Section 3.1.2) benefited from the creation of reactive

[Schmidt, 1995] consumers that contain all related endpoints within a single process.

To achieve a high degree of flexibility, therefore, TAO's 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 [Gamma et al., 1995], as described in Section 2.3.1.

Flexibility in data transfer protocol: A CORBA A/V Streaming Service implementation may need to select from a variety of transfer protocols. For instance, an Internet-based streaming application, such as Realvideo [RealNetworks, 1998], may use the UDP protocol, whereas a local intranet video-conferencing tool [et al., 1996] might prefer the QoS features offered by native high-speed ATM protocols. Likewise, RTP [Schulzrinne et al., 1994] is gaining acceptance as a transfer protocol for streaming audio and video data over the Internet. Thus, it is essential that a A/V Streaming Service support a range of data transfer protocols *dynamically*.

The CORBA A/V Streaming Service defines a simple specialized protocol *Simple Flow Protocol* (SFP), which makes no assumptions about the communication protocols used for data streaming and provides an architecture independent flow content transfer. Consequently, the stream establishment components in TAO's A/V Streaming Service provide flexible mechanisms that allow applications to define and use multiple network programming APIs, such as sockets and TLI, and multiple communication protocols, such as TCP, UDP, RTP, or ATM.

Therefore, another design challenge we faced was to define stream establishment components that can work with a variety of data transfer protocols. To resolve this challenge, we applied the *Strategy* pattern [Gamma et al., 1995], as explained in Section 2.3.5.

Providing a uniform API for different flow protocols: The CORBA A/V Streaming Service specification defines the flow specification syntax that can be used for connection establishment. It defines the protocol names and syntax for specifying the flow and data transfer protocol information, but it does not define any interfaces for protocol implementations. We resolved this omission with our *pluggable A/V protocol framework* (described in Section 2.4) using design patterns, described in Appendix A, such as *Layer* [Buschmann et al., 1996] , *Acceptor-Connector* [Schmidt et al., 2000b], *Facade* and *Abstract Factory* [Gamma et al., 1995]. Moreover, TAO's A/V Streaming Service defines a uniform API for the different flow protocols, such as RTP and SFP, that can handle variations using the policy interface described in Section 2.4.2.

Flexibility in stream control interfaces: A/V streaming middleware should 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 may support other operations, such as `start` and `stop`. Since the operations provided by the stream are application-defined, it is useful for the control logic component in streaming middleware to be flexible and adaptive.

Therefore, another design challenge facing designers of CORBA A/V Streaming Services is to allow applications the flexibility to define their own stream control interfaces and access these interfaces in an extensible, type-safe manner. In TAO's A/V Streaming Service implementation, we used the *Extension Interface* [Schmidt et al., 2000b] pattern to resolve this challenge.

Flexibility in managing states of stream supplier and consumers: The data transfer component of a streaming application often must 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 a PLAYING state. Likewise, 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. Moreover, in each state, the behavior of supplier/consumer applications, and the A/V Streaming Service itself, must be well-defined. To address this issue we applied the *State Pattern* [Gamma et al., 1995], as described in Section 3.1.1. The `State` pattern is described in Appendix A.

Providing a uniform interface for full and light profiles: To allow developers and applications to control and manage flows and streams, the CORBA A/V Streaming Service specification exposes certain of their IDL interfaces. There are two levels of exposure defined by the CORBA A/V Service: (1) the *light profile*, where only the stream and stream endpoint interfaces are exposed and the flow interfaces are not exposed and (2) the *full profile*, where flow interfaces are also exposed. This two-level design provides more flexibility and granularity of control to applications and developers since flow interfaces are CORBA interfaces and are not locality constrained.

Therefore, the design challenge was to define a uniform interface for both the light and full profiles to make use of TAO's pluggable A/V protocol framework. We resolved this challenge by deriving the full and light profile endpoints from a base interface and by generating the flow specification using the `Forward_FlowSpec_Entry` and

Reverse_FlowSpec_Entry classes, as mentioned in section 2.3.6.

Providing multipoint-to-multipoint bindings: Different multimedia applications require different stream endpoint bindings. For example, video-on-demand applications require point-to-point bindings between consumer and supplier endpoints whereas video-conferencing applications require a multipoint-to-multipoint bindings. The CORBA A/V specification defines a point-to-multipoint binding, but not a multipoint-to-multipoint binding, which is left as a responsibility of implementors.

Thus, we faced the design challenge of providing multipoint-to-multipoint bindings for applications that use multicast protocols provided by the underlying network. We have provided a solution based on IP multicast and used to Adapter pattern [Gamma et al., 1995] to adapt it to ATM's multicast model. The Adapter pattern is used to allow multiple components to work together, even if they were not originally designed to work together. This adaptation was done by having TAO's A/V Streaming Service set source ids for the flow producers so that the flow consumers can distinguish the sources. We added support in both SFP and RTP to allow them to be adapted for such bindings. Our implementation of Vic, described in Section 3.2, uses TAO's A/V Streaming Service multipoint-to-multipoint binding and its RTP adapter.

2.3 CORBA A/V Streaming Service Components

The CORBA A/V Streaming Service specification defines a set of standard IDL interfaces that can be implemented to provide a reusable framework for distributed multimedia streaming applications. Figure 3 illustrates the key components of the CORBA A/V Streaming Service. This subsection describes the design of TAO's A/V Streaming Service components shown in Figure 3. The corresponding IDL interface name for each role is provided in brackets. In addition, we illustrate how TAO provides solutions to the design challenges outlined in Section 2.2.

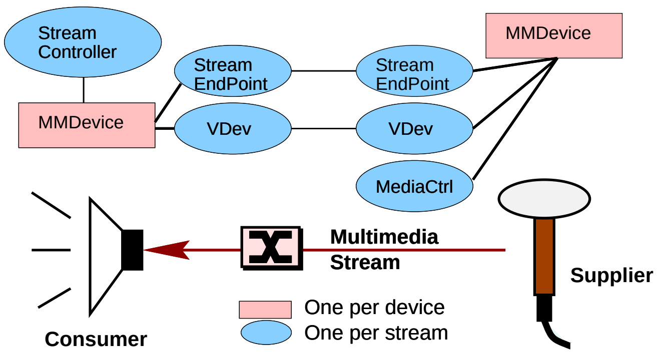

Figure 3: A/V Streaming Service Components

2.3.1 Multimedia Device Factory (MMDevice)

An MMDevice abstracts the behavior of a multimedia device. The actual device can be *physical*, such as a video microphone or speaker, or 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.

For instance, a particular device might support MPEG-1 [ISO, 1993] compression or ULAW audio [SUN Microsystems, 1992]. Such parameters are termed "properties" of the MMDevice. Properties can be associated with the MMDevice using the CORBA Property Service [OMG, 1996], as shown in Figure 4.

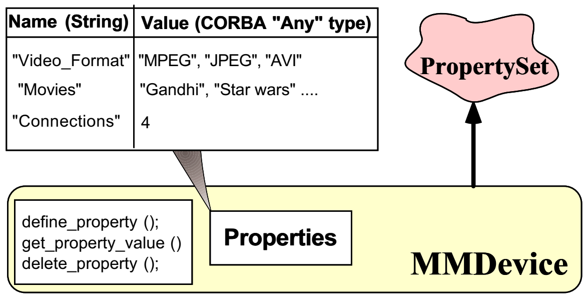

Figure 4: Multimedia Device Factory

An MMDevice is also an endpoint factory that creates 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 data transfer-specific parameters of the connection. The roles of VDev and StreamEndpoint are described in Section 2.3.2 and Section 2.3.5, 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 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 5. 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 3.1.1 creates separate processes to stream the audio and video data to the consumer concurrently.

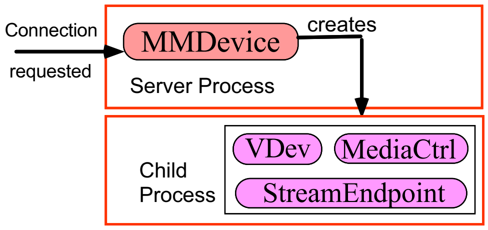

Figure 5: `MMDevice` Process-based Concurrency Strategy

Reactive strategy: In this strategy, endpoint objects for each new stream are created in the same process as the factory, as shown in Figure 6. Thus, a single process handles all the simultaneous connections *reactively* [Schmidt, 1995]. This strategy is useful for applications that dedicate one process to control multiple streams. For instance, to minimize synchronization overhead, the consumer of the MPEG A/V player application described in Section 3.1.2 uses this strategy to create the audio and video endpoints in the same process.

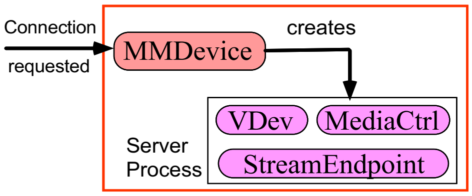

Figure 6: `MMDevice` Reactive Concurrency Strategy

In TAO's A/V Streaming Service, the `MMDevice` uses the *Abstract Factory* pattern [Gamma et al., 1995] 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.

The Abstract Factory pattern 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 [Schmidt et al., 2000b] or special framebuffers.

2.3.2 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 an 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 7.

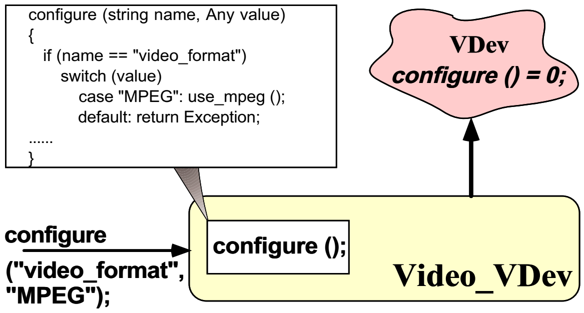

Figure 7: Virtual Device

Stream establishment is a mechanism defined by the CORBA A/V Streaming Service specification to permit the negotiation of QoS 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 Service are implemented using the CORBA Property Service [OMG, 1996].

The CORBA A/V Streaming Service 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 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 its resources released immediately.

Section 2.3.6 describes the CORBA A/V Streaming Service stream establishment protocol in detail.

2.3.3 Media Controller (`MediaCtrl`)

The Media Controller (`MediaCtrl`) is an IDL interface that defines operations for controlling a stream. A `MediaCtrl` interface is *not* defined by the CORBA A/V Streaming Service specification. Instead, it is defined by multimedia application developers to support operations for a specific stream, such as the following IDL interface 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 CORBA 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 [OMG, 1996].

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.

2.3.4 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 2.3.6.

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

2.3.5 Stream Endpoint (`StreamEndpoint`)

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

In TAO's A/V Streaming Service, the `StreamEndpoint` implementation uses patterns, such as *Double Dispatching* and *Template Method* [Gamma et al., 1995], described in Appendix A, to allow applications to define and exchange data transfer-level parameters flexibly. This interaction is shown in Figure 8 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`.

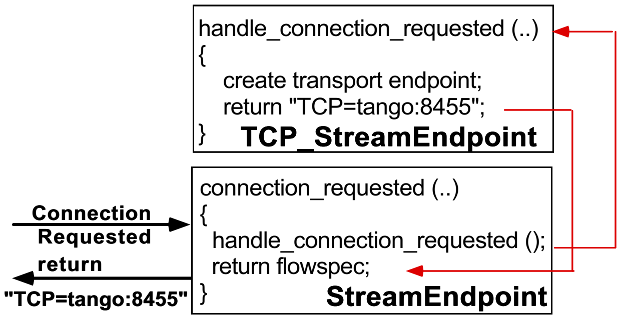

Figure 8: Interaction Between `StreamEndpoint` and a Multimedia Application

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

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

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

Thus, by applying patterns, the `StreamEndpoint` design allows each application to configure its own data transfer protocol, while reusing the generic stream establishment control logic in TAO's A/V Streaming Service.

2.3.6 Interaction Between Components in the CORBA Audio/Video Streaming Service Model

The preceding discussion in Section 2.3 described the structure of components that constitute the CORBA A/V Streaming Service. Below, we describe how these components *interact* to provide two key A/V Streaming Service features: *stream establishment* and *flexible stream control*.

Stream establishment: Stream establishment is the process of binding two peers who need to communicate via a *stream*. The CORBA A/V Streaming Service specification defines a standard protocol for establishing a binding between streams. Several A/V Streaming Service components are involved in stream establishment. A key motivation for providing an elaborate stream establishment protocol is to allow components to be configured independently. This allows the stream establishment protocol to remain standard, and yet provide sufficient hooks for multimedia application developers to customize this process for a specific set of requirements. For instance, an `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 desired policies.

The CORBA A/V Streaming Service specification identifies two peers in stream establishment, which are known 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 CORBA A/V Streaming Service specification defines two *distinct* IDL interfaces for the A and B party endpoints. Hence, for a given stream, there will be two distinct types for the supplier and the consumer. Thus, the CORBA A/V Streaming Service specification ensures that the complimentary relationship between suppliers and consumers is type-safe. An exception will be raised if a supplier tries to establish a stream with another supplier accidentally.

Stream establishment in TAO's A/V Streaming Service occurs in several steps, as illustrated in Figure 9. This

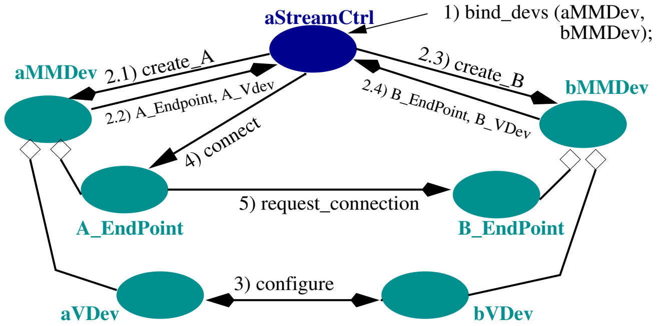

Figure 9: Stream Establishment Protocol in the A/V Streaming Service

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 9 is explained below:

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 data transfer endpoints in response to this operation. In TAO's A/V Streaming Service, applications can customize this behavior using the *Double Dispatch* [Gamma et al., 1995] pattern described in Section 2.3.5.

5. Stream Establishment: In this step, the A_Endpoint invokes the request_connection operation on its peer endpoint. The A_Endpoint passes its network 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 data transfer connection. It subsequently connects to the data transfer endpoint passed to it by the A_Endpoint.

After completing these five stream establishment protocol steps, a data transfer-level stream is established between the two endpoints of the stream. Section 2.3.6 describes how the *Media Controller* (MediaCtrl) can control an established stream, *e.g.*, by starting or stopping the stream.

Stream control: Each MMDevice endpoint factory can be configured with an application-defined MediaCtrl interface, as described in Section 2.3.3. 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. The MediaCtrl is an Extension Interface described in Appendix A.

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 certain operations, such as 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);
```

Flow specification: When binding two multimedia devices, a flow specification is passed between the two `StreamEndpoints` to convey per-flow information. A flow specification represents key aspects of a flow, such as its name, format, flow protocol being used, and the network name and address. A flow specification string is analogous to an *interoperable object reference* (IOR) in the CORBA object model. The syntax for the interoperable flow specifications is shown in Figure 10. Standardizing the flow specifications ensures that

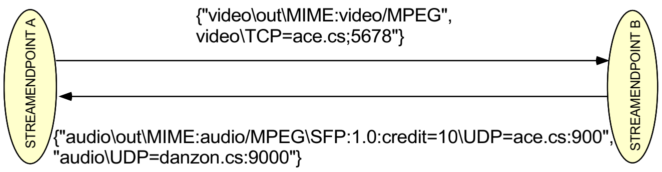

Figure 10: Flow Specification

two different `StreamEndpoints` from two different implementations can interoperate. There are two different flow specifications, depending on the direction in which the flowspec is traveling. If it is from the `A` party's `StreamEndpoint` to the `B` party's *StreamEndpoint* then it is a "forward flowspec;" the opposite direction is the "reverse flowspec."

TAO's CORBA A/V Streaming Service implementation defines two classes, `Forward_FlowSpec_Entry` and `Reverse_FlowSpec_Entry`, that allow multimedia applications to construct the *flow specification string* from their components without worrying about the syntactic details. For example, the entry takes the address as both an `INET _Addr` and a `string` and provides convenient parsing utilities for strings.

2.4 The Design of a Pluggable A/V Protocol Framework for TAO's A/V Streaming Service

At the heart of TAO's A/V Streaming Service is its *pluggable A/V protocol framework*, which defines a common interface for various flow protocols, such as TCP, UDP, RTP, or ATM. This framework provides the "glue" that integrates its ORB components with the underlying I/O subsystem protocols and network interfaces. In this section, we describe the design of the pluggable A/V protocol framework provided in TAO's A/V Streaming Service and describe how we resolved key design challenges that arose when developing this framework.

2.4.1 Overview of TAO's Pluggable A/V Protocol Framework

The pluggable A/V protocol framework in TAO's A/V Streaming Service consists of the components shown in Figure 11. Each of these components is described below.

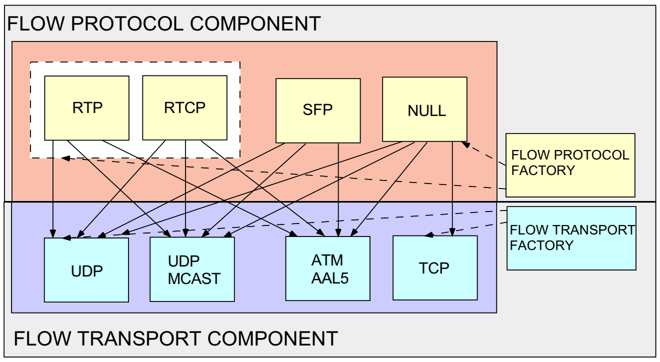

Figure 11: Pluggable A/V Protocol Components in TAO's A/V Streaming Service

AV_Core: This singleton [Gamma et al., 1995] component is a container for flow and data transfer protocol factories. An application using TAO's A/V implementation must initialize this singleton before using any of its A/V classes, such as `StreamCtrl` and `MMDevice`. During initialization, the `AV_Core` class loads all the flow protocol factories, control protocol factories, and data transfer factories dynamically using the `Service Configurator` pattern [Schmidt et al., 2000b] and creates default instances for each known protocol.

Data Transfer components: The components illustrated in Figure 12 and described below are required for each data transfer protocol:

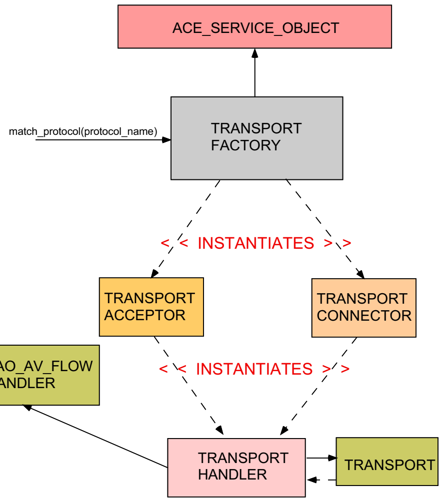

Figure 12: TAO's A/V Streaming Service Pluggable Data Transfer Components

- **Acceptor and Connector:** These classes are implementations of the `Acceptor-Connector` pattern [Schmidt et al., 2000b], which are used to accept connections passively and establish connections actively, respectively.

- **Transport_Factory:** This class is an abstract factory [Gamma et al., 1995] that provides interfaces to create `Acceptors` and `Connectors` in accordance to the appropriate type of data transfer protocol.

- **Flow_Handler:** All data transfer handlers derive from the `Flow_Handler` class, whose methods can start, stop, and provide flow-specific functionality for timeout upcalls to the `Callback` objects, which are described in the following paragraph.

Callback interface: TAO's A/V Streaming Service uses this callback interface to deliver frames and to notify `FlowEndPoints` of start and stop events. Multimedia application developers subclass this interface for each flow endpoint, *i.e.*, there are producer and consumer callbacks. TAO's A/V Streaming Service dispatches timeout events automatically so that applications need not write event handling mechanisms. For example, all flow producers are automatically registered for a timer events with a Reactor. The value for the timeout is obtained through the `get_timeout` hook method on the `Callback` interface. This hook method is called whenever a timeout occurs since multimedia applications typically have adaptive timeout values.

Flow protocol components: Flow protocols carry in-band information for each flow that a receiver can use to reproduce the source stream. The following components are required for each flow protocol supported by TAO's A/V Streaming Service:

- **Flow_Protocol_Factory:** This class is an abstract factory that creates flow protocol objects.

- **Protocol_Object:** This class defines flow protocol functionality. Applications use this class to send frames and the `Protocol_Object` uses application-specified `Callback` objects to deliver frames.

Figure 13 illustrates the relationships among the flow protocol components in TAO's pluggable A/V protocol framework.

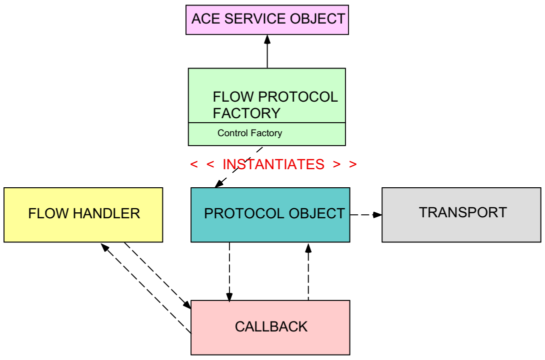

Figure 13: TAO's A/V Streaming Service Pluggable A/V Protocol Components

AV_Connector and AV_Acceptor Registry: As mentioned above, different data transfer protocols require the creation of corresponding data transfer factories, acceptors, and connectors. The `AV_Core` class creates the `AV_Connector` and `AV_Acceptor` registry classes to provide a facade that maintains and accesses the abstract flow and data transfer factories both for *light* and *full* profile objects. This design gives users a single interface that hides the complexity of creating and manipulating different data transfer factories.

2.4.2 Applying Patterns to Resolve Design Challenges for Pluggable A/V Protocols Frameworks

Below, we outline the key design challenges faced when developing TAO's pluggable A/V protocol framework and discuss how we resolved these challenges by applying various patterns [Gamma et al., 1995, Buschmann et al., 1996, Schmidt et al., 2000b].

Adding new data transfer protocols transparently:

- **Context:** Different multimedia applications often have different QoS requirements. For example, a video application over an intranet may want to take advantage of native

ATM protocols to reserve bandwidth. An audio application in a video-conferencing application may want to use a reliable data transfer protocol, such as TCP, since loss of audio is more visible to users than video and the bit-rate of audio flows are low (~8 kbps using GSM compression). In contrast, a video application might not want the overhead of retransmission and slow-start congestion protocol incurred by a TCP [Stevens, 1993]. Thus, it may want to use the facilities of an unreliable data transfer protocol, such as UDP, since losing a small number of frames may not affect perceived QoS.

- **Problem:** It should be possible to add new data transfer protocols to TAO's pluggable A/V protocol framework without modifying the rest of TAO's A/V Streaming Service. Thus, the framework must be open for extensions but closed to modifications, *i.e.*, the Open/Closed principle [Meyer, 1989]. Ideally, creating a new protocol and configuring it into TAO's pluggable A/V protocol framework should be all that is required.

- **Solution:** Use a registry to maintain a collection of *abstract factories* based on the Abstract Factory pattern [Gamma et al., 1995]. In this pattern, a single class defines an interface for creating families of related objects, without specifying their concrete types. Subclasses of abstract factories are responsible for creating concrete classes that collaborate amongst themselves. In the context of pluggable A/V protocols, each abstract factory can create concrete `Connector` and `Acceptor` classes for a particular protocol.

- **Applying this solution in TAO's A/V Streaming Service:** In TAO's A/V Streaming Service, the `Connector_Registry` plays the role of the protocol registry. This registry is created by the `AV_Core` class. Figure 14 depicts the `Connector_Registry` and its relation to the abstract factories. These factories are accessed via a facade defined according to the *Facade* pattern [Gamma et al., 1995]. This design hides the complexity of manipulating multiple factories behind a simpler interface. The `Connector_Registry` described above plays the facade role.

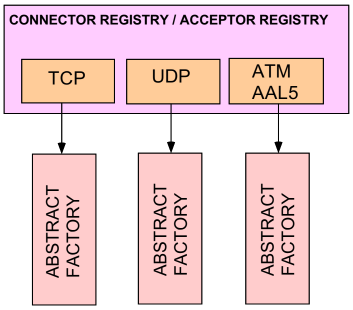

Figure 14: Connector Registry

Adding new A/V protocols transparently:

- **Context:** Multimedia flows often require a flow protocol since most multimedia flows need to carry in-band information for the receiver to reproduce the source stream. For example, every frame may need a timestamp so that the receiver can play the frame at the right time. Moreover, sequence numbers will be needed if a connectionless protocol, such as UDP, is used so that the applications can do resequencing. In addition, multicast flows may require information, such as a source identification number, to demultiplex flows from different sources.

  SFP is a simple flow protocol defined by the CORBA A/V Streaming Service specification to transport in-band data. Likewise, the Real-time Transport Protocol (RTP) [Schulzrinne et al., 1994] defines facilities to transport in-band data. RTP is Internet-centric, however, and cannot carry CORBA IDL-typed flows directly. For example, RTP specifies that all header fields should be in network-byte order, whereas the SFP uses CORBA's CDR encoding and carries the byte-order in each header.

- **Problem:** Flow protocols should be able to run over different data transfer protocols. This configuration of a flow protocol over different data transfer protocol should be done easily and transparently to the application developers and users.

- **Solution:** To solve the problem of a flow protocol running over different data transfer protocols, we applied the *Layers* pattern [Buschmann et al., 1996] described in Appendix A. We have structured the flow protocols and data transfer protocols as two different layers. The flow protocol layer creates the frames with the in-band flow information. The data transfer layer performs the connection establishment and sends the frames that are sent down from the flow protocol layer onto the network. The layered approach makes the flow and data transfer protocols independent of each other and hence it is easy to tie different flow protocols with different data transfer protocols transparently.

- **Applying this solution in TAO's A/V Streaming Service:** TAO's A/V Streaming Service provides a uniform data transfer layer for a variety of flow protocols, including UDP unicast, UDP multicast, and TCP. TAO's A/V Streaming Service provides a flow protocol layer using a `Protocol_Object` interface. Likewise, its `AV_Core` class maintains a registry of A/V protocol factories.

Adding new protocols dynamically:

• **Context:** When developing new pluggable A/V protocols, it is inconvenient to recompile TAO's A/V Streaming Service and applications just to validate a new protocol implementation. Moreover, it is often useful to experiment with different protocols to compare their performance, footprint size, and QoS guarantees systematically. Moreover, in telecom systems with 24×7 availability requirements, it is important to configure protocols dynamically, even while the system is running. This level of flexibility helps simplify upgrades and protocol enhancements.

• **Problem:** The user would like to populate the registry *dynamically* with a set of factories during run-time and avoid the inconvenience of recompiling the AV Service and the applications when different protocols are plugged in. The solution explains how we can achieve this.

• **Solution:** We can solve the above stated problem using the *Service Configurator* pattern [Schmidt et al., 2000b], which decouples the implementation of a component from the point in time when it is configured into the application. By using this pattern, a pluggable A/V protocol framework can dynamically load the set of entries in a registry. For instance, a registry can simply parse a configuration script and dynamically link the services listed in it.

• **Applying this solution in TAO's A/V Streaming Service:** The `AV_Core` class maintains all parameters specified in a configuration script. Adding a new parameter to represent the list of protocols is straightforward, *i.e.*, the default registry simply examines this list and links the services into the address-space of the application, using the ACE Service Configurator implementation [Schmidt and Suda, 1994]. ACE provides a rich set of reusable and efficient components for high-performance, real-time communication, and forms the portability layer of TAO's A/V Streaming Service. Figure 15 depicts the connector registry and its relation to the ACE Service Configurator framework, which is an implementation of the Component Configurator pattern [Schmidt et al., 2000b].

Control protocols:

• **Context:** RTP has a control protocol – RTCP – associated with it. Every RTP participant must transmit RTCP frames that provide control information, such as the name of the participant and the tool being used. Moreover, RTCP sends reception reports for each of its sources.

• **Problem:** Certain flow protocols, such as SFP, use A/V interfaces to exchange control interfaces. The use of RTP for a flow necessitates it to transmit RTCP information. RTCP extracts this control information from RTP packets. Therefore, TAO's A/V Streaming Service must provide an extensible interface for these control protocols, as well as provide a means for interacting between the data and control protocols.

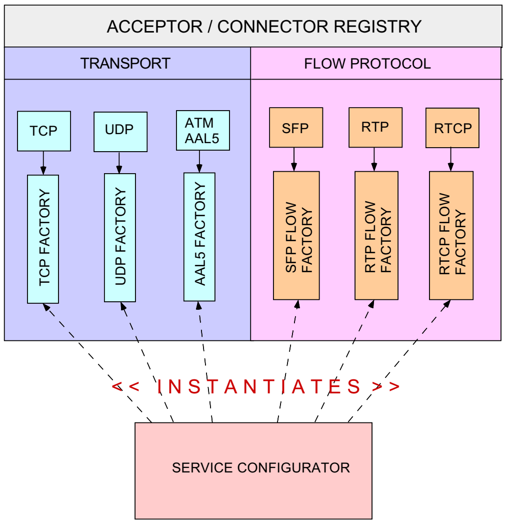

Figure 15: Acceptor-Connector Registry and Service Configurator

• **Solution:** The solution is to make the control protocol information part of the flow protocol. For example, RTP knows that RTCP is its control protocol. Therefore, to reuse pluggability features, it may be necessary to make the control protocol use the same interfaces as its data components.

• **Applying this solution in TAO's A/V Streaming Service:** During stream establishment, `Registry` objects will first check the flow factory for the configured flow protocol. After the `listen` or `connect` operation has been performed for a particular data flow, the `Registry` will check if the flow factory has a control factory. If so, it will perform the same processing for the control factory, except the network endpoint port will be one value higher than the data endpoint. Since the CORBA A/V Streaming Service specification does not define a portable way to specify control flow endpoint information, we followed this approach as a temporary solution until the OMG comes up with a portable solution.

The RTCP implementation in TAO's A/V Streaming Service uses the same interfaces that RTP does, including the `Flow_Protocol_Factory` and `Protocol_Object` classes. Thus, RTP will call the `handle_control_input` method on the RTCP `Protocol_Object` when a RTP frame is received. This method enables the RTCP object to extract the necessary control information, such as the sequence number of the last frame.

Interface for variations in flow protocols:

• **Context:** Above, we explained how TAO's pluggable A/V protocol framework factors out different flow protocols and provides a uniform flow protocol interface. In certain cases, however, there are inherent variations in such protocols.

For example, RTP must transmit the payload type, *i.e.*, the format of the flow in each frame, whereas SFP uses the control and management interface in TAO's A/V Streaming Service to set and get the format values for a flow.

Similarly, the RTP control protocol, RTCP, periodically transmits participant information, such as the senders name and email address, whereas SFP does not transmit such information. Such information does not changed with every frame, however. For example, the name and email address of a participant in a conference will not change for a session. In addition, the properties of the transfer may need to be controlled by applications. For instance, a conferencing application may not want to have multicast loopback.

• **Problem:** An A/V Streaming Service should allow end-users to set protocol-specific variations, while still providing a single interface for different flow protocols. Moreover, this interface should be open to changes with the addition of new flow protocol and data transfer protocols.

• **Solution:** The solution to the above problem is to apply the *CORBA Policy* framework defined in the CORBA specification [Object Management Group, 1999]. The CORBA Policy framework allows the protocol component developer to define policy objects that control the behavior of the protocol component. The policy object is derived from the CORBA Policy interface [Object Management Group, 1999] which stores the Policy Type [Object Management Group, 1999] and the associated values.

• **Applying this solution in TAO's A/V Streaming Service:** By defining a policy framework, which is extensible and follows the CORBA Policy model, the users will have shorter learning curve to the API and be able to add new flow protocols flexibly. We have defined different policy types used by different flow protocols that can be accessed by the specific transport and flow protocol components during frame creation and dispatching. For example we have defined the TAO_AV_PAYLOAD_TYPE_POLICY which allows the RTP protocol to specify the payload type.

3 Case Study: Developing Multimedia Applications using TAO's A/V Streaming Service

To evaluate the capabilities of the CORBA-based A/V Streaming Service, we have developed several multimedia applications that use the components and interfaces described in Section 2. Thus, this section describes the design of two distributed multimedia applications that use TAO's A/V Streaming Service and pluggable A/V protocol framework to establish and control MPEG and interactive audio/video streams.

3.1 Case Study 1: an MPEG A/V Streaming Application

This application is an enhanced version of a non-CORBA MPEG player developed at the Oregon Graduate Institute [Chen et al., 1995]. Our application plays movies using the `MPEG-1` video format [ISO, 1993] and the Sun `ULAW` audio format [SUN Microsystems, 1992]. Figure 16 shows the architecture of our A/V streaming application.

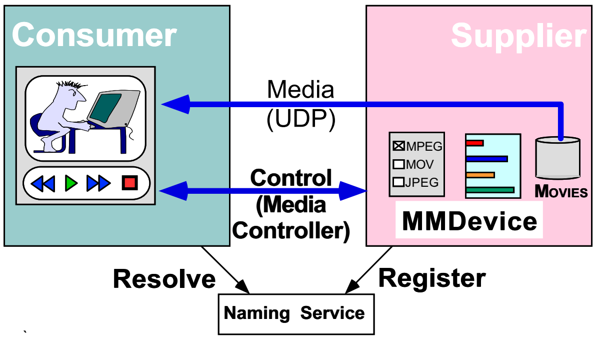

Figure 16: Architecture of the MPEG A/V Streaming Application

The MPEG player application uses a supplier/consumer design implemented using TAO. The consumer locates the supplier using the CORBA Naming Service [OMG, 1997b] or the Trading Service [OMG, 1997b] 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 a 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. These streams are established using the protocol described in Section 2.3.6. The consumer then uses the `MediaCtrl` to control the stream, as described in Section 2.3.3.

The supplier is responsible for sending A/V packets via UDP to the consumer. For each consumer, the supplier transmits two streams, one for the MPEG video packets and one for the Sun ULAW audio packets. The consumer decodes these streams and plays these packets in a viewer, as shown in Figure 17.

This section describes the various components of the consumer and supplier. 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

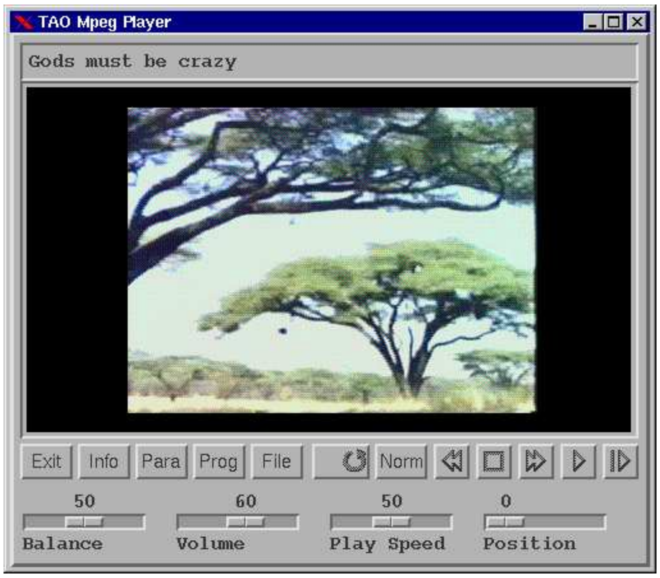

Figure 17: A TAO-enabled Audio/Video player

Using the ORB and the A/V Streaming Service greatly reduced the amount of software that otherwise would have been written manually.

3.1.1 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 dual-CPU 187 Mhz Sun Ultrasparc-II with 256 MB 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 18 depicts the key components in the supplier architecture.

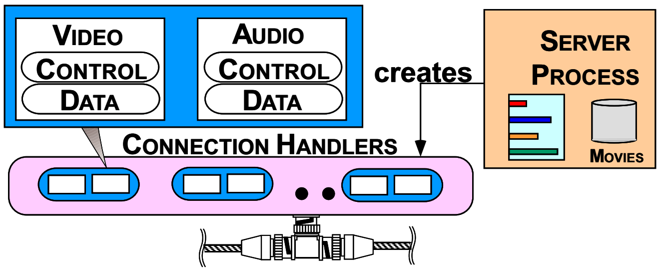

Figure 18: TAO Audio/Video Supplier Architecture

The main supplier process contains an `MMDevice` endpoint factory described in Section 2.3.1. This `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:

The Media controller component: This component in the supplier process is a servant that implements the Media Controller interface (`MediaCtrl`) described in Section 2.3.3. 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 of our MPEG player application is implemented using the *State* pattern [Gamma et al., 1995]. This implementation is shown in

Figure 19. The `MediaCtrl` has a `state` object pointer.

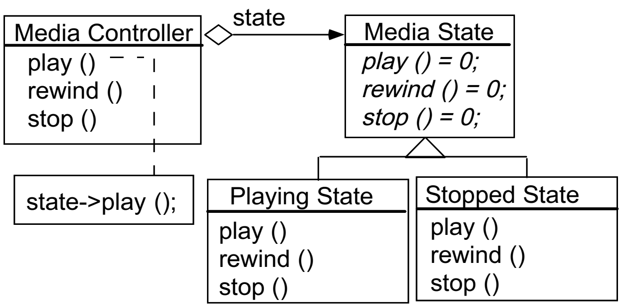

Figure 19: State Pattern Implementation of the Media Controller

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 3.1.1 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.

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 [Schmidt et al., 2000b], as shown in Figure 20. The Reactor pattern is described in Appendix A.

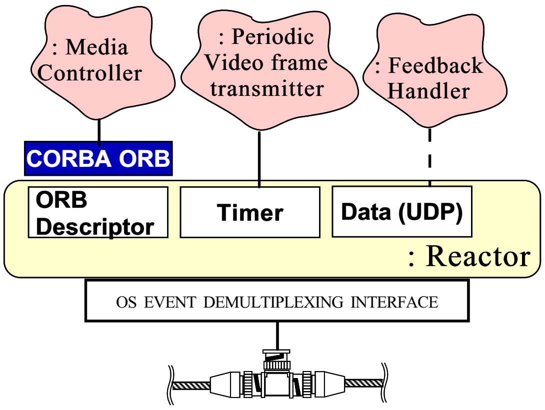

Figure 20: Reactive Architecture of the Video Supplier

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) [OMG, 1997a]. 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 [OMG, 1997a].

3.1.2 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 21 depicts the key components in the consumer architecture:

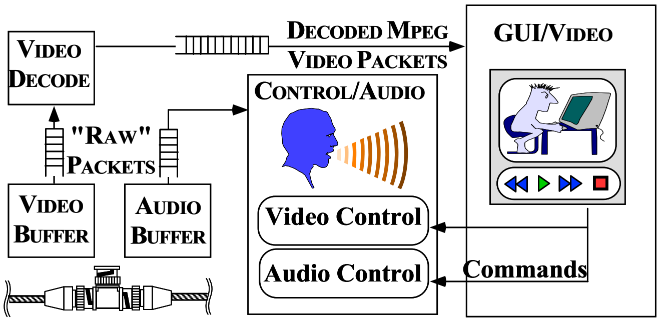

Figure 21: TAO Audio/Video Consumer Architecture

The original non-CORBA MPEG consumer [Chen et al., 1995] 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:

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 [Stevens, 1997].

- **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.

3.2 Case Study 2: The Vic Video-Conferencing Application

Vic [McCanne and Jacobson, 1995] is a video-conferencing application developed at the University of California, Berkeley. We have adapted Vic to use TAO's A/V Streaming Service components and its pluggable A/V protocol framework described in Section 2. The Vic implementation in TAO uses RTP/RTCP as its flow and data transfer protocols.

3.2.1 Architecture of Vic

Vic provides a video-conferencing application. Audio conferencing is done with another tool, Vat [NRG, LBNL, 1995]. The Vic family of tools synchronize media streams using a conference bus mechanism, which is the "localhost" synchronization mechanisms used via loopback sockets.

The Architecture of Vic is driven largely by the `TclObject` interface [McCanne and Jacobson, 1995]. `TclObject` provides operations so that operations on the object can be invoked from a Tcl script. By using Tcl, Vic allows rapid prototyping and reconfiguration of its encode/decode paths.

One design challenge we faced while adapting Vic to use TAO's A/V Streaming Service was to integrate both the GUI and ORB event loops. This was solved using the Reactor pattern [Schmidt et al., 2000b]. In particular, we developed a Reactor wrapper facade [Schmidt et al., 2000b] that unified the GUI and ORB into a single event loop.

3.2.2 Implementing Vic using TAO's A/V Streaming Service

Below, we discuss the steps we followed to adapt Vic to use TAO's A/V Streaming Service.

1. Structuring of conferencing protocols: In this step, we decomposed the flow, control and data transfer protocols using TAO's pluggable A/V protocol framework. The original Vic application was highly coupled with RTP. For instance, its encoders and decoders were aware of the RTP headers. We decoupled the dependencies of the encoders/decoders from RTP-specific details by using the `frame_info` structure and using TAO's A/V Streaming Service `Protocol_Object` interface. The modified Vic still preserves the application-level framing (ALF) [Clark and Tennenhouse, 1990] model embodied in RTP. Moreover, Vic's RTCP functionality was abstracted into the TAO's pluggable A/V protocol framework, so the framework automatically defines a RTCP flow for a RTP flow. The modified Vic is independent from the network specific details of opening connections and I/O handling since it uses the pluggable A/V protocol framework provided by TAO's A/V Streaming Service.

Vic uses the multipoint-to-multipoint binding provided by TAO's A/V Streaming Service, which is described in Appendix C. Thus, our first step when integrating into TAO was to determine the proper abstraction for the conference device. A video-conferencing application like Vic serves as both a source and sink; thus, we needed a source and sink `MMDevice`. Moreover, to be extensible for future integration with Vat and other multimedia tools, Vic uses flow interfaces, *i.e.*, video is considered as a flow within the conference stream. Since Vat runs in a separate address space, its flow interfaces must be exposed using TAO's full profile flow interfaces, *i.e.*, `FDev`, `FlowProducer`, and `FlowConsumer`.

2. Define callback objects: In this step, we defined `Callback` objects for all the source and sink `FlowEndPoints`. The `Source_Callback` uses the timer functionality to schedule timer events to send the frames. Figure 22 illustrates the sequence of events that trigger the sending of frames. When the input becomes ready on the video card, the grabber reads it and gives it to the `transmitter`.

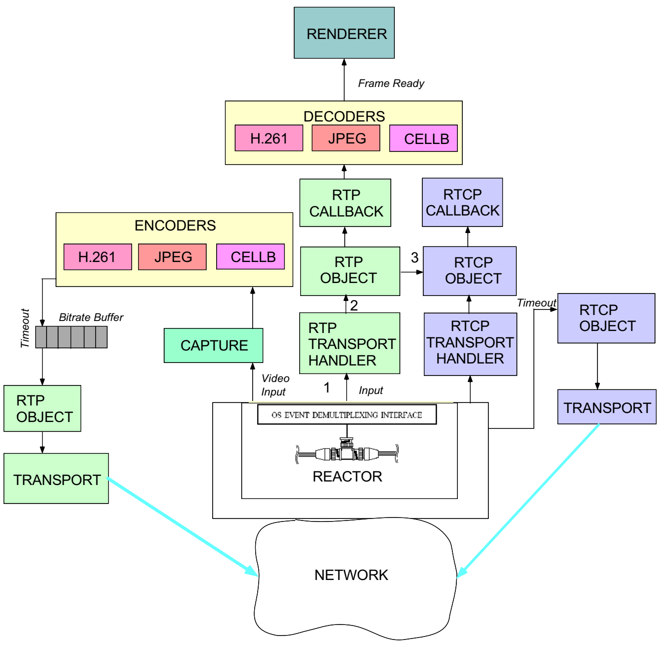

Figure 22: Architecture of Vic using TAO's A/V Streaming Service

The transmitter then uses the Source_Callback object to schedule a timer to send the frames at the requested bit rate using a bitrate buffer.

On the sink-side, when a packet arrives on the network the receive_frame upcall is done on the Sink_Callback object which using the frame_info structure gives it to the right Source object , which then passes it to the right decoder. To implement RTCP functionality, Vic implements a RTCP_Callback to provide Vic-specific source objects.

3. Select a centralized or distributed conference configuration: In this step, we have ensured that Vic can function both as a participant in a centralized conference, as well as a loosely-coupled distributed conference. This flexibility is achieved by checking for a StreamCtrl object in the Naming Service and creating new StreamCtrl if one is not found in the Naming Service. Thus, by running a StreamCtrl control process that registers itself with the Naming Service, all Vic participants will become part of a centralized conference, which can be controlled from the control process. Conversely, when no such process is run, Vic reverts to the loosely controlled model by creating its own Streamctrl and transmitting on the multicast address.

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 [Ritchie, 1984].

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 23.

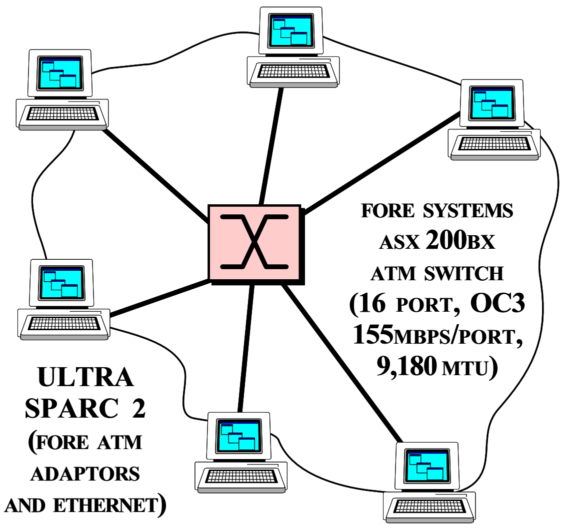

Figure 23: Hardware for the CORBA/ATM Testbed

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 24. These results indicate 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%.

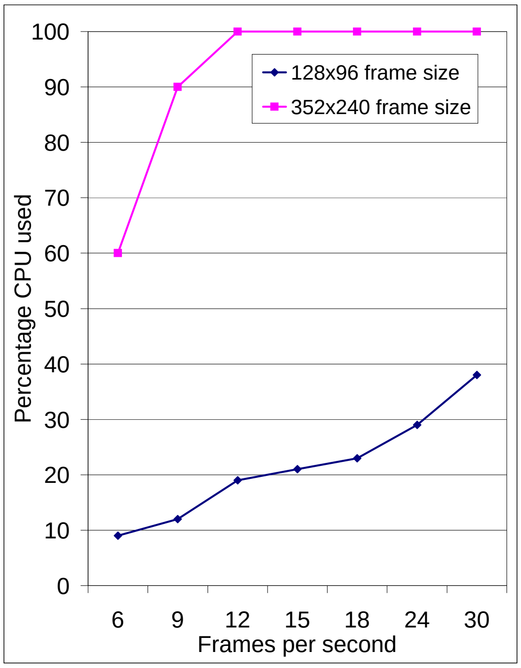

Figure 24: CPU Usage of the MPEG Decoder

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 2.3.6. 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 CORBA A/V Streaming Service stream establishment protocol. In this case, Sockets and TCP were the network programming API and data transfer protocol, respectively. 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 [Schmidt et al., 1998a] CORBA ORB. In this case, the IIOP data path was the data transfer 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 25 indicate that, as expected, the A/V Streaming Service does not introduce any appreciable overhead to streaming the data. In the case of us-

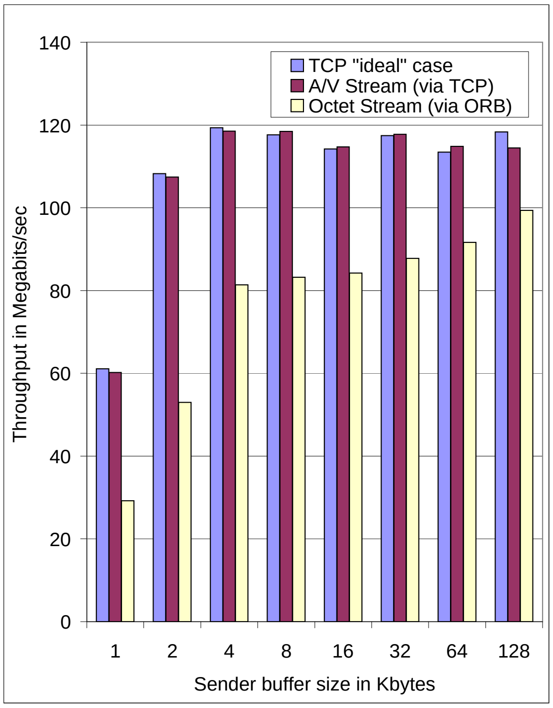

Figure 25: Throughput Results

ing IIOP as the data transfer layer, the benchmark incurs additional performance overhead. This overhead arises from the dynamic memory allocation, data-copying, and marshaling/demarshaling performed by the ORB's IIOP protocol engine [Gokhale and Schmidt, 1996]. In general, however, a well-designed ORB can achieve performance equivalent to sockets for higher buffer sizes due to various optimizations, such as eliding (de)marshaling overhead for octet data [Gokhale and Schmidt, 1999]

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 CORBA A/V stream establishment protocol described in Section 2.3.6. We measured the stream establishment latency for the two concurrency strategies, process-based strategy and reactive strategy, described in Section 2.3.1.

The timer starts when the consumer gets the object reference for the supplier's `MMDevice` servant from the Naming Service. The timer stops when the stream has been established, *i.e.*, when a 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 26. When the supplier's `MMDevice` is configured to use the process-based concurrency strategy (described in Section 2.3.1), 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.

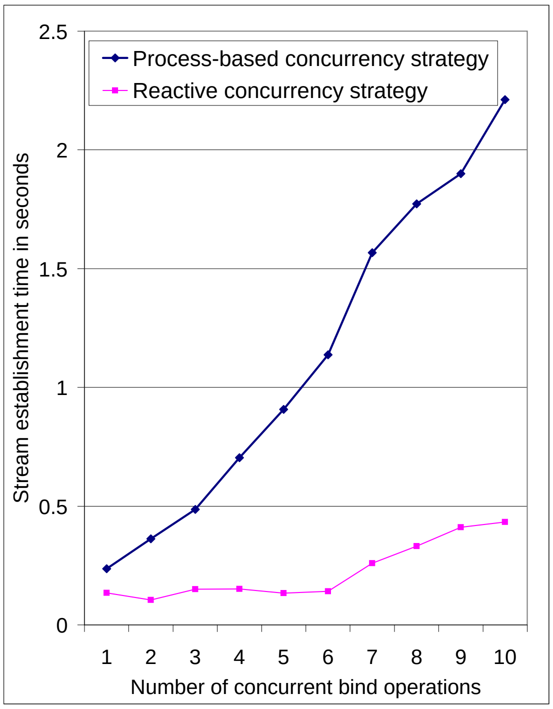

Figure 26: Stream Establishment Latency Results

5 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 chapter illustrates an approach to building standards-based, flexible, adaptive, multimedia streaming applications using CORBA.

Furthermore, there is a lot of activity in the codec community in designing new formats for audio and video transmission. Active research is also being done in designing new flow and data transfer protocols for multimedia. In such situations, a flexible framework which makes use of the A/V interfaces and also abstracts the network/protocol details is needed to adapt to the new developments. In this chapter we have presented a pluggable A/V protocol framework which provides the capability to rapidly adapt to new flow and data transfer protocols.

With growing demand for real-time multimedia streaming and conferencing with increase in network bandwidth and the spread of the Internet, TAO provides the first freely-available, open-source implementation of the CORBA Audio/Video Streaming Service specification *i.e.*, flow interfaces, point-to-multipoint binding and multipoint-to-multipoint binding for conferencing applications. Our experience with TAO's A/V implementation indicates that the standard CORBA specification defines a flexible and efficient model for developing flexible and high-performance multimedia streaming applications.

While designing and implementing the CORBA A/V Streaming Service, we learned a number of lessons:

1: We found that CORBA simplifies a number of common network programming tasks, such as parsing untyped data and performing byte-order conversions.

2: 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 2.3.3.

3: Our measurements presented 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 data transfer protocols for multimedia streaming.

4: Enhancing an existing A/V streaming application to use CORBA was a key design challenge. By applying patterns, such as the *State*, *Strategy*, [Gamma et al., 1995] and *Reactor* [Schmidt et al., 2000b], 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 distributed object computing middleware, such as CORBA.

5: 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 optimizations 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`.

References

[Arulanthu et al., 2000] Arulanthu, A. B., O'Ryan, C., Schmidt, D. C., Kircher, M., and Parsons, J. (2000). The Design and Performance of a Scalable ORB Architecture for CORBA Asynchronous Messaging. In *Proceedings of the Middleware 2000 Conference*. ACM/IFIP.

[Box, 1997] Box, D. (1997). *Essential COM*. Addison-Wesley, Reading, MA.

[Buschmann et al., 1996] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996). *Pattern-Oriented Software Architecture - A System of Patterns*. Wiley and Sons.

[Chen et al., 1995] Chen, S., Pu, C., Staehli, R., Cowan, C., and Walpole, J. (1995). A Distributed Real-Time MPEG Video Audio Player. In *Fifth International Workshop on Network and Operating System Support of Digital Audio and Video*.

[Clark and Tennenhouse, 1990] Clark, D. D. and Tennenhouse, D. L. (1990). Architectural Considerations for a New Generation of Protocols. In *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, pages 200–208, Philadelphia, PA. ACM.

[Deering and Cheriton, 1990] Deering, S. E. and Cheriton, D. R. (May 1990). Multicast routing in datagram internetworks and extended LANs. *ACM Transactions on Computer Systems*, 8(2):85–110.

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

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

[Fan et al., 1998] Fan, L., Cao, P., Almeida, J., and Broder, A. (1998). Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. In *SIGCOMM 98*, pages 254–265. SIGS.

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

[Gill et al., 2000] Gill, C. D., Levine, D. L., and Schmidt, D. C. (2000). 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*.

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

[Gokhale and Schmidt, 1998] Gokhale, A. and Schmidt, D. C. (1998). Measuring and Optimizing CORBA Latency and Scalability Over High-speed Networks. *Transactions on Computing*, 47(4).

[Gokhale and Schmidt, 1999] Gokhale, A. and Schmidt, D. C. (1999). 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*, 17(9).

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

[Henning and Vinoski, 1999] Henning, M. and Vinoski, S. (1999). *Advanced CORBA Programming With C++*. Addison-Wesley Longman.

[Hu et al., 1998] Hu, J., Mungee, S., and Schmidt, D. C. (1998). Principles for Developing and Measuring High-performance Web Servers over ATM. In *Proceeedings of INFOCOM '98*.

[Hu et al., 1997] Hu, J., Pyarali, I., and Schmidt, D. C. (1997). 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.

[Huard and Lazar, 1998] Huard, J.-F. and Lazar, A. (1998). A Programmable Transport Architecture with QOS Guarantees. *IEEE Communications Magazine*, 36(10):54–62.

[IETF, 2000a] IETF (2000a). Differentiated services (diffserv). http://www.ietf.org/html.charters/diffserv-charter.html.

[IETF, 2000b] IETF (2000b). Integrated services (intserv). http://www.ietf.org/html.charters/intserv-charter.html.

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

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

[Kuhns et al., 2000] Kuhns, F., Schmidt, D. C., O'Ryan, C., and Levine, D. (2000). Supporting High-performance I/O in QoS-enabled ORB Middleware. *Cluster Computing: the Journal on Networks, Software, and Applications*.

[McCanne and Jacobson, 1995] McCanne, S. and Jacobson, V. (1995). Vic: A Flexible Framework for Packet Video. In *ACM Multimedia 95*, pages 511–522, New York. ACM Press.

[Meyer, 1989] Meyer, B. (1989). *Object Oriented Software Construction*. Prentice Hall, Englewood Cliffs, NJ.

[Mungee et al., 1999] Mungee, S., Surendran, N., and Schmidt, D. C. (1999). The Design and Performance of a CORBA Audio/Video Streaming Service. In *Proceedings of the Hawaiian International Conference on System Sciences*.

[NRG, LBNL, 1995] NRG, LBNL (1995). LBNL Audio Conferencing Tool (vat). ftp://ftp.ee.lbl.gov/conferencing/vat/.

[Object Management Group, 1999] Object Management Group (1999). *The Common Object Request Broker: Architecture and Specification*. Object Management Group, 2.3 edition.

[OMG, 1996] OMG (1996). *Property Service Specification*. Object Management Group, 1.0 edition.

[OMG, 1997a] OMG (1997a). *Control and Management of A/V Streams specification*. Object Management Group, OMG Document telecom/97-05-07 edition.

[OMG, 1997b] OMG (1997b). *CORBAServices: Common Object Services Specification, Revised Edition*. Object Management Group, 97-12-02 edition.

[O'Ryan et al., 2000] O'Ryan, C., Kuhns, F., Schmidt, D. C., Othman, O., and Parsons, J. (2000). The Design and Performance of a Pluggable Protocols Framework for Real-time Distributed Object Computing Middleware. In *Proceedings of the Middleware 2000 Conference*. ACM/IFIP.

[Pyarali et al., 1996] Pyarali, I., Harrison, T. H., and Schmidt, D. C. (1996). Design and Performance of an Object-Oriented Framework for High-Performance Electronic Medical Imaging. *USENIX Computing Systems*, 9(4).

[Pyarali et al., 1999] Pyarali, I., O'Ryan, C., Schmidt, D. C., Wang, N., Kachroo, V., and Gokhale, A. (1999). Applying Optimization Patterns to the Design of Real-time ORBs. In *Proceedings of the 5^{th} Conference on Object-Oriented Technologies and Systems*, San Diego, CA. USENIX.

[RealNetworks, 1998] RealNetworks (1998). Realvideo player. www.real.com.

[Ritchie, 1984] Ritchie, D. (1984). A Stream Input–Output System. *AT&T Bell Labs Technical Journal*, 63(8):311–324.

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

[Schmidt et al., 1998a] Schmidt, D. C., Levine, D. L., and Mungee, S. (1998a). The Design and Performance of Real-Time Object Request Brokers. *Computer Communications*, 21(4):294–324.

[Schmidt et al., 1998b] Schmidt, D. C., Mungee, S., Flores-Gaitan, S., and Gokhale, A. (1998b). 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.

[Schmidt et al., 2000a] Schmidt, D. C., Mungee, S., Flores-Gaitan, S., and Gokhale, A. (2000a). Software Architectures for Reducing Priority Inversion and Non-determinism in Real-time Object Request Brokers. *Journal of Real-time Systems, special issue on Real-time Computing in the Age of the Web and the Internet*.

[Schmidt et al., 2000b] Schmidt, D. C., Stal, M., Rohnert, H., and Buschmann, F. (2000b). *Pattern-Oriented Software Architecture: Patterns for Concurrency and Distributed Objects, Volume 2*. Wiley & Sons, New York, NY.

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

[Schulzrinne et al., 1994] Schulzrinne, H., Casner, S., Frederick, R., and Jacobson, V. (1994). RTP: A Transport Protocol for Real-Time Applications. *Internet-Draft*.

[Stevens, 1993] Stevens, W. R. (1993). *TCP/IP Illustrated, Volume 1*. Addison Wesley, Reading, Massachusetts.

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

[SUN Microsystems, 1992] SUN Microsystems, I. (1992). *Sun Audio File Format*. Sun Microsystems, Inc.

[Vxtreme, 1998] Vxtreme (1998). Vxtreme player. www.microsoft.com/netshow/vxtreme/.

[Wollrath et al., 1996] Wollrath, A., Riggs, R., and Waldo, J. (1996). A Distributed Object Model for the Java System. *USENIX Computing Systems*, 9(4).

A Design Patterns used in the TAO A/V Streaming Service

This section outlines the intents of all the patterns used in TAO's A/V Streaming Service and its pluggable A/V protocol framework. The references explore each pattern in greater depth.

Abstract Factory pattern [Gamma et al., 1995]: This pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes.

Acceptor-Connector pattern [Schmidt et al., 2000b]: This pattern decouples the connection and initialization of cooperating peer services in a distributed system from the processing performed by these peer services once they are connected and initialized.

Adapter [Gamma et al., 1995]: This pattern allows two classes to collaborate that were not designed originally to work together.

Component Configurator [Schmidt et al., 2000b]: This pattern decouples the implementation of services from the time when they are configured.

Double Dispatching [Gamma et al., 1995]: In this pattern, when a call is dispatched to a method on a target object from a parent object, the target object in turn makes method calls on the parent object to access certain attributes in the parent object.

Extension Interface [Schmidt et al., 2000b]: This pattern prevents bloating of interfaces and breaking of client code when developers add or modify functionality to existing components. Multiple extensions can be attached to the same component, each defining a contract between the component and its clients.

Facade pattern [Gamma et al., 1995] : This pattern provides a unified higher-level interface to a set of interfaces in a subsystem that makes the subsystem easier to use.

Factory Method pattern [Gamma et al., 1995]: This defines an interface for creating objects, but lets subclasses decide which class to instantiate.

Leader/Follower pattern [Schmidt et al., 2000b]: This pattern provides a concurrency model where multiple threads efficiently demultiplex events received on I/O handles shared by the threads and dispatch event handlers that process the events.

Layer pattern [Buschmann et al., 1996]: This pattern helps to structure applications that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction.

Reactor pattern [Schmidt et al., 2000b]: This pattern demultiplexes and dispatches requests that are delivered concurrently to an application by one or more clients.

State pattern [Gamma et al., 1995]: This pattern allows an object to alter its behavior when its internal state changes. The object will appear to change its class.

Strategy pattern [Gamma et al., 1995]: This pattern defines and encapsulates a family of algorithms and makes them interchangeable.

Template Method [Gamma et al., 1995]: This pattern defines the skeleton of an algorithm in an operation, deferring certain steps to subclasses.

B 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 [Henning and Vinoski, 1999]. Figure 27 illustrates the key components in the CORBA reference model [Object Management Group, 1999] that collaborate to provide this degree of portability, interoperability, and transparency.[1] Each component in the CORBA reference model is outlined below:

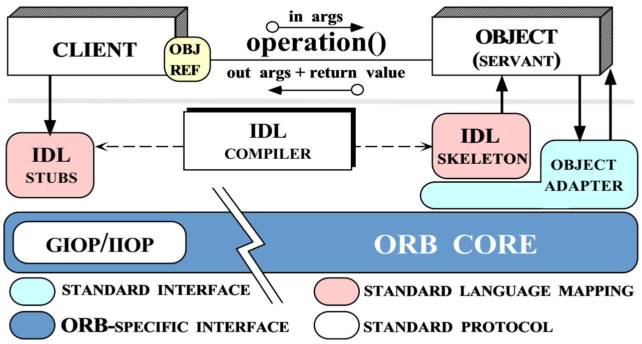

Figure 27: Key Components in the CORBA 2.x Reference Model

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 27 shows how the underlying ORB components described below transmit remote operation requests transparently from client to object.

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 object-oriented (OO) languages, such as C++ and Java, servants are implemented using one or more class instances. In non-OO languages, such

[1] This overview only focuses on the CORBA components relevant to this paper. For a complete synopsis of CORBA's components see [Object Management Group, 1999].

as 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), such as 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 [Gamma et al., 1995] and provide a strongly-typed, *static invocation interface* (SII) that marshals application parameters into a common message-level representation. Conversely, skeletons implement the *Adapter* pattern [Gamma et al., 1995] and demarshal the message-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, such as 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 [Eide et al., 1997].

Object Adapter: An Object Adapter is a composite component that 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. 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.

C Supporting Multiple Endpoint Binding Semantics in TAO's A/V Streaming Service

The CORBA A/V Streaming Service can construct different topologies for establishing streams between stream endpoints. For instance, one-to-one, one-to-many, many-to-one, and many-to-many sources and sinks may need to be configured in the same stream binding. The need for certain stream endpoint bindings is dictated by the multimedia applications. For example, a video-on-demand application may require a point-to-point binding when sources and sinks are pre-selected. However, a video-conferencing application may require a multipoint-to-multipoint binding to receive from and transmit to various sources and sinks simultaneously.

This section illustrates the various stream and flow endpoint bindings that have been implemented in TAO's A/V Streaming Service and shows how stream endpoints are created and the connections are established. In TAO's A/V Streaming Service, we have implemented the standard point-to-point and point-to-multipoint bindings of the stream endpoints. In addition, we have used these configurations as building blocks for multipoint-to-multipoint bindings.

C.1 Point-to-Point Binding

Below, we describe the sequence of steps during a point-to-point stream establishment, as defined by the CORBA A/V specification and implemented in TAO's A/V Streaming Service. In our example, we consider the stream establishment in a video-on-demand (VoD) application that is similar to the MPEG player application described in Section 3.1. As shown in Figure 28, the VoD server and VoD client device with two audio and video flows. The audio flow is carried

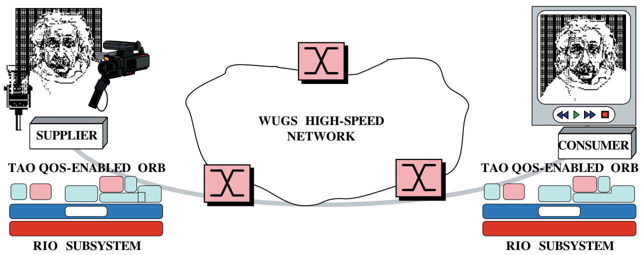

Figure 28: Video-on-Demand Consumer and Supplier

over TCP and video over UDP. The client must first locate the Server `MMDevice` reference and then pass its `MMDevice` as the `A` party and the Server `MMDevice` as the `B` party to the `StreamCtrl`.

Endpoint creation: At this point, the `VDev` and `StreamEndpoint` are created for this stream from the `MMDevices`. The client and server applications can choose either `Process_Strategy`, where the endpoints are created in a separate process, or a `Reactive_Strategy`, where the endpoints are created in the same process. The pluggable A/V protocol framework in TAO's A/V Streaming Service provides flexible *Concurrency Strategies* [Mungee et al., 1999] to create the endpoints, as described in Section 2.3.1.

Creation of flowendpoints: To create a full profile, an MMDevice can act as a container for FDevs. In this case, the MMDevice will create a FlowProducer or FlowConsumer from the FDev, depending on the direction of the flow specified in the flow specification parameter. The flow direction is always with respect to the A side. Thus, the direction "out" means that the flow originates from the A side to the B side, whereas "in" means that the flow originates from the B side to the A side.

In the above case, the server is streaming the data to the client. Therefore, the direction of the flow for both audio and video will be "in" and the MMDevice will create a Flowproducer from the audio and video FDevs on the server and a FlowConsumer from the audio and video FDevs on the client. These FlowProducers and FlowConsumers are then added to the StreamEndpoint using the add_fep call.

The advantage of using the flow interfaces is that the FDevs can be shared across different applications. In our VoD server example, the audio and video processes could be running as two different processes and contain only the flow objects and a control process could add the FDevs from these two processes to the stream. Both flows can now be controlled through the same StreamCtrl interface. This configuration is a much more scalable and extensible approach than the implementation of a MPEG player described in Section 3.1, where the audio and video were treated as two separate streams.

VDev configuration: The StreamCtrl then calls set_peer on each of the VDevs with the other VDevs. For light profiles, multimedia application developers are responsible for implementing the set_peer call to check if all flows are compatible. For full profiles, the VDev interface is not used because the FlowEndPoint contain these configuration operations.

Stream setup: During this step the actual connections for the different flows are established. For light profiles, the flows do not have any interfaces and the flow specification should contain the transfer information for each flow. For example, the following flow specs are typically passed to the bind_devs call from the VoD client:

"audio\in\MIME:audio/mpeg\\TCP=ace.cs.wustl.edu;10000"
and
"video\in\MIME:video/mpeg\\UDP=ace.cs.wustl.edu;8080"

In these flow specs, the client is offering to listen for a TCP connection and the server will connect to the client. This configuration might be useful if the server is behind a firewall. The StreamCtrl calls connect on one of the StreamEndpoints passing the other StreamEndpoint, QoS, and the flow spec.

Stream QoS negotiation: The StreamEndpoint will first check if the the other StreamEndpoint has a negotiator property defined. If it does, StreamEndpoint calls negotiate on the negotiator and the client and server can negotiate the QoS. TAO's A/V Streaming Service provides a default implementation that can be overridden by the applications. The StreamEndpoint then queries the "AvailableProtocols" property on the other StreamEndpoint. If there is no common protocol the Stream setup will fail and the exception StreamOpDenied will be thrown.

Light profile connection establishment: The A party StreamEndpoint will then try to setup the stream for all its flows. For light profiles, the following steps are done for each flow:

1: The StreamEndpoint will extract the flow protocol and data transfer protocol information from the flow spec entry for this flow. If a network address is not specified then a default stream endpoint is picked.

2: The StreamEndpoint then does the following actions.

3: It goes through the list of flow protocol factories in the AV_Core instance to find if there is any matching flow protocol. If no flow protocol is specified, it passes the protocol as the flow protocol string. TAO's A/V Streaming Service provides "no-op" implementations for all data transfer protocols so that the layering of the architecture is preserved and a uniform API is presented to the application. These no-op flow protocols do not process the frames – they simply pass them to the underlying data transfer protocol.

4: If a flow protocol factory matches the specified flow protocol/transfer protocol, the StreamEndpoint then checks for the data transfer protocol factory that matches the protocol specified for this flow.

5: After finding a matching data transfer protocol factory, it creates a one-shot acceptor for this flow passing the *FlowProtocolFactory* to the acceptor.

6: If the flow protocol factory has an associated control protocol factory, the StreamEndpoint then tries to match the data transfer factory for that, as well.

Figure 29 illustrates the sequence of steps outlined above. In each step, the StreamEndpoint uses base interfaces, such as Protocol_Factory, Transport_Factory, and AV_Acceptor. Therefore, it can be extended easily to support new flow and data transfer protocols. In addition, the address information is opaque to the StreamEndpoint and is passed down to the Acceptor that knows how to interpret it. Moreover, since the flow and data transfer protocols can be linked dynamically via the ACE Service Configurator mechanisms, applications can take advantage of these protocols by simply changing the name of the protocols in the flow spec.

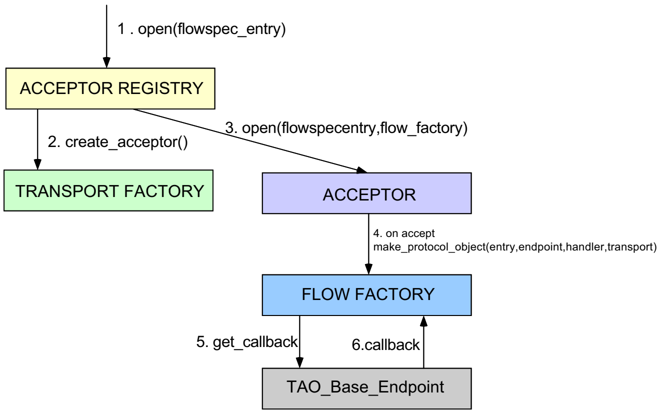

Figure 29: Acceptor Registry

After completing the preceding steps, the `StreamEndpoint` then calls the `request_connection` operation on the `B StreamEndpoint` with the flowspec. The `StreamEndpoint_B` performs the following steps for each of the flow:

1: It extracts the flow and data transfer protocol information from the flow spec entry for this flow. If a network address is not specified then a default stream endpoint is picked.

2: The `StreamEndpoint` then performs the following actions.

3: Finds a flow protocol factory matching the flow protocol specified for this flow and in the absence of a flow protocol tries to match a null flow protocol for the specified data transfer protocol.

4: Finds a matching data transfer protocol factory and creates a connector for it. Then it calls `connect` on the connector, passing it the flow protocol factory.

5: Upon establishing a data transfer connection, the connector creates a protocol object for this flow.

6: The flow protocol factory typically creates the application-level callback object and sets the protocol object on the `Base_EndPoint` interface passed to it.

7: If an address was not specified for this flow then the `StreamEndpoint` does the similar steps for listening for those flows and extracts the network endpoints and inserts it into the flowspec to be sent back to the `A StreamEndpoint`.

The `A StreamEndpoint` after receiving the reverse flowspec does the `connect` for all the flows for which `B StreamEndpoint` is listening and also sets the peer address for connectionless protocols, such as UDP.

Full profile connection establishment: In the full profile, the flow specification does not contain the data transfer information for each flow since the flows are represented by flow interfaces and they need not be collocated in the same process. A `StreamCtrl` can be used to control different flows, each of which could reside on a different machine. In this case, each `FlowEndPoint` will need to know the network address information. In the full profile stream setup, `bind` is called on the `StreamCtrl`, passing the two `StreamEndpoints`.

Figure 30 illustrates the sequence of steps performed for a full profile point-to-point stream setup.

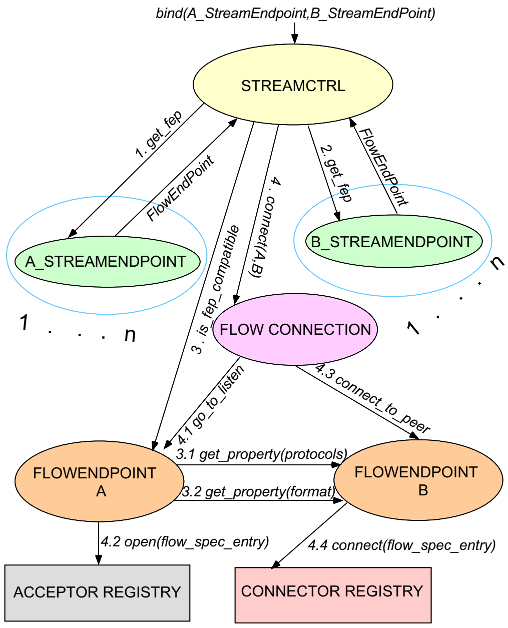

Figure 30: Full Profile Point to Point Stream Establishment

1: Flow endpoint matching: The `StreamCtrl` obtains the flow names in each `StreamEndpoint` by querying the "flows" property. For each flow name, it then obtains the `FlowEndPoint` using the `get_fep` method on the `StreamEndpoint`. If the flowspec is empty all the flows are considered. Otherwise, only the specified flows are considered for stream establishment.

It then goes through the list of `FlowEndPoints` trying to find a match between the `FlowEndPoints` on the `A` and `B` side. Two `FlowEndPoints` are said to match if `is_fep_compatible` returns true. This call checks to make sure that the format and the protocols of the two `FlowEndPoints` match. Applications can override this behavior to do more complex checks, such as checking for semantic nuances of device parameters. For example, the `FlowEndPoint` may want only a French audio stream, whereas the other `FlowEndPoint` may support only English. These requested semantics can be checked by querying the property "devParams" and by checking the value for "language."

The StreamEndpoint then tries to obtain a FlowConnection from the StreamCtrl. The application developer can set the FlowConnection object for each flow using the StreamCtrl. All operations on a stream are applied to the contained FlowConnections and by setting specialized FlowConnections the user can customize the behavior of the stream operations. If the stream does not have a FlowConnection then a default FlowConnection is created and set for that flow. The StreamEndpoint then calls connect on the FlowConnection with the producer and consumer endpoints with the flow QoS.

2: Flow configuration: The FlowConnection calls set_peer on each of the FlowEndPoints during the connect operation and this will let the FlowEndPoints to check and set the peer FlowEndpoint's configuration. For example, a video consumer can check the ColourModel, ColourDepth, and VideoResolution and allocate a window for the specified resolution and also other display resources, *i.e.*, colormap, etc. In the case of audio, the quantization property can be used by the consumer to allocate appropriate decoder resources.

3: Flow connection establishment: In this step, the FlowConnection calls go_to_listen on one of the FlowEndPoints with the is_mcast parameter set to false and also passes the flow protocol that was set on the FlowConnection using the use_flow_protocol operation. The FlowEndPoint can raise an exception failedToListen in which case the FlowConnection calls go_to_listen on the other FlowEndPoint.

In TAO's implementation the go_to_listen does the sequence of operations shown in Figure 29 to accept on the selected flow protocol and data transfer protocol and also if needed the control protocol for the flow. Since the FlowEndPoint also derives from Base_EndPoint the Callback and Protocol_Objects will be set on the endpoint. In the case of the FlowProducer the get_timeout will be called on the Callback object to register for timeout events.

The FlowConnection then calls connect_to_peer on the other FlowEndPoint with the address returned by the listening FlowEndPoint and also the flowname. In the case of connectionless protocols, such as UDP, the listening FlowEndPoint may need to know the reverse channel to send the data in which case it can call the get_rev_channel operation to get it.

When FlowEndPoint calls connect_to_peer, sequence of steps shown in Figure 31 will occur to connect to the listening endpoint. With the above sequence of steps a stream will be established in a point-to-point binding between two multimedia devices.

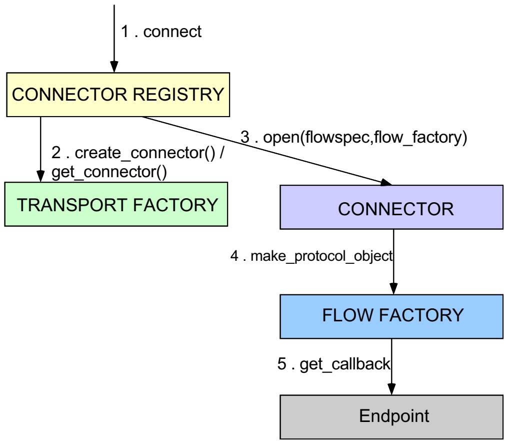

Figure 31: Connector Registry

C.2 Point-to-Multipoint Binding

TAO's point-to-multipoint binding support is essential to handle broadcast/multicast streaming servers. With new technologies, such as Web Caching [Fan et al., 1998], multicast updates of web pages and streaming media files is becoming common place. In addition, it has become common on websites to broadcast live events using technologies like RealPlayer. For example, during the World Cup Cricket 99, millions of people listened to the live commentaries of the matches from the BBC website.

In such cases, it would be ideal for the servers to use multicast technologies like IP multicast to reduce server connections and load. TAO's point-to-multipoint binding essentially provides such an interface for a source to multicast the flows to multiple sinks as shown in figure 32. TAO's A/V Streaming Service implementation provides a binding based on IP multicast [Deering and Cheriton, 1990]. In this section we explain the sequence of steps that lead to a point-to-multipoint stream establishment both in the light and full profiles.

Adding a multipoint source: A multipoint source MMDevice must be added before any sinks can be added to the Stream. For example, the multicast server could add itself to the StreamCtrl and expose the StreamCtrl interface through a standard CORBA object location service, such as Naming or Trading. If the B party MMDevice parameter to bind_devs is nil, the source is assumed to be a multicast source. As with a point-to-point stream, the endpoints for the source device are created, *i.e.*, the StreamEndpoint and VDev for light profiles, and the StreamEndpoint containing FlowProducers for the full profile. Unlike the point-to-point stream, however, there can only be

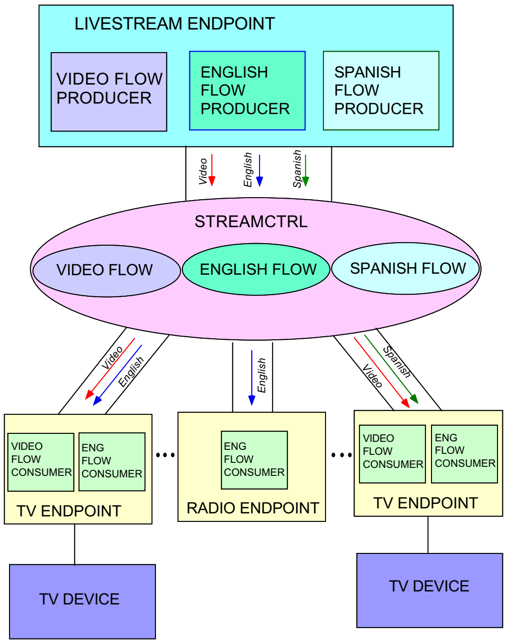

Figure 32: Point-to-Multipoint Binding

`FlowProducers` in the `MMDevice`. Figure 33 shows the

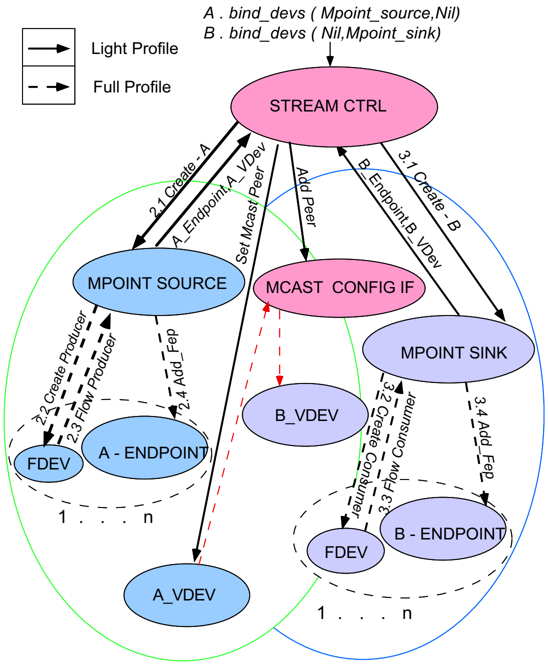

Figure 33: Creation of Endpoints in the Point-to-Multipoint Binding

creation of endpoints in point-to-multipoint bindings.

Multicast configuration interface: In the case of a multipoint binding there can be numerous sinks. Therefore, the CORBA A/V Streaming Service specification provides an `MCastConfigIf` interface, which is used instead of using point-to-point `VDev` configurations. Upon addition of a multipoint source, the `StreamCtrl` creates a new `MCastConfigIf` interface and sets it as the multicast peer of the source `VDev`. This design allows the stream binding to use multicasting technologies to distribute the stream configuration information instead of using numerous point-to-point configurations.

The `MCastConfigIf` interface provides operations to set the initial configuration of the stream example via the `set_initial_configuration` operation. This operation can be called by the source `VDev` during the `set_MCast_peer` call. This information is conveyed to the multicast sink `VDev` during the `set_peer` call on the `MCastConfigIf` when a multicast sink is added. The `MCastConfigIf` performs the configuration operation using point-to-point invocation on all sink `VDevs`.

Adding multicast sinks: When a sink wants to join a stream as a multicast sink, it can call `bind_devs` with a nil `A` party `MMDevice`. This call will create the endpoints for the multicast sink, *i.e.* the `StreamEndpoint` and the `VDev`. For full profiles, the `StreamEndpoint` will contain `FlowConsumers`. For light profiles, the `VDev` is added to the `MCastConfigIf`.

Multicast connection establishment: The `StreamCtrl` then calls `connect_leaf` on the multicast source endpoint for the multicast sink endpoint. In TAO, the `connect_leaf` operation will throw the `notSupported` exception. The `StreamCtrl` will then try the IP multicast model using the `multiconnect` call on the source `StreamEndpoint`. The following steps occur when `multiconnect` is called on a `StreamEndpoint_A` for each flow in the full profile:

1: The `StreamEndpoint` makes sure that the endpoint is indeed a `FlowProducer`.

2: It then checks to see if a `FlowConnection` interface exists for this flow in the `StreamCtrl`, which is obtained through the `Related_StreamCtrl` property.

3: In the absence of a `FlowConnection`, the `StreamEndpoint_A` will create a `FlowConnection` and set the multicast address to be used for this flow on the `FlowConnection`. An application configure this address by passing it to the `StreamEndpoint` during its initialization. The A/V specification does not define how multicast addresses are allocated to flows. Thus, TAO's `StreamEndpoint` uses a base multicast address and assigns different ports for the flows and sets the `FlowConnection` on the `StreamCtrl`. We ultimately plan to strategize this allocation so applications can decide on the multicast addresses to use for each flow.

4: The `StreamEndpoint` then calls `add_producer` on `FlowConnection`.

5: The call to `add_producer` will result in a `connect_mcast` on the `FlowProducer`, passing the multicast address with which to connect. The `FlowProducer` then returns the address to which it will multicast the flow. If the return address is complete with network address, then IP Multicast is used. In contrast, if the return address specifies only the protocol name an ATM-style multicast is used.

6: In addition, the `FlowConnection` creates a `MCastConfigIf` if it has not been created and sets it as the multicast peer on the `FlowProducer`. Since the same `MCastConfigIf` is used for both `FlowEndPoint` and `VDev`, the parameters to `MCastConfigIf` are passed as CORBA objects. It is the responsibility of `MCastConfigIf` to check whether the peer is a `VDev` or a `FlowEndpoint`.

7: The `connect_mcast` does the actual connection to the multicast address and results in the sequence of steps for multicast accept using the pluggable A/V protocols.

Figure 34 illustrates these steps graphically. The steps described above occur for each multipoint sink that is added to the stream. TAO's pluggable A/V protocol framework is configured with both full profile and light profile objects. It is also configured in the point-to-point and point-to-multipoint bindings. Thus, the control and management implementation objects can be closed for modifications, yet new flow and data transfer protocols can be added flexibly to the framework without modification to these interface implementations. A similar set of steps happens when `multiconnect` is called on the `StreamEndpoint_B`.

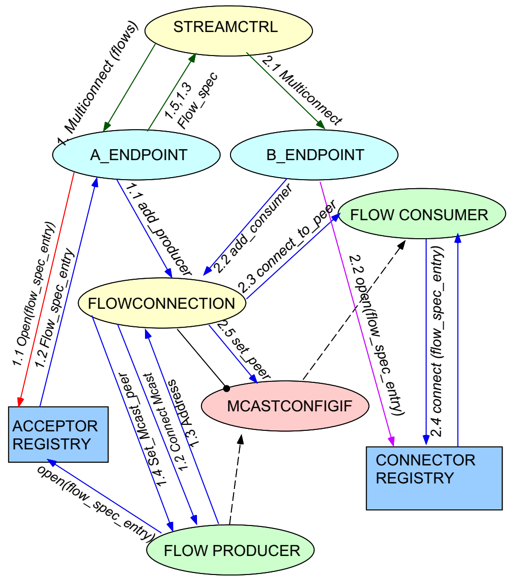

Figure 34: Connection Establishment in the Point-to-Multipoint Binding

C.3 Multipoint-to-Multipoint Binding

The multipoint-to-multipoint binding is important for applications, such as video-conferencing, where there are multiple source and sink participants. The CORBA A/V Streaming Service specification does not mandate any particular protocol for multipoint-to-multipoint binding, leaving it to implementors to provide this feature. In TAO, we provide a multipoint-to-multipoint binding by extending the point-to-multipoint binding based on IP multicast. We assume a Leader/Follower pattern [Schmidt et al., 2000b] for the sources, where the first source that is added to the stream will become the *leader* for the multipoint-to-multipoint binding, and every other source become a *follower*. This design implies that all stream properties, such as format and codec, will be selected by the leader.