The Design of a Pluggable Protocol architecture for a CORBA Audio/Video Streaming Service

Nagarajan Surendran and Douglas C. Schmidt

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

Abstract

Distributed multimedia application development with stringent Quality of Service (QoS) requirements is gaining importance with the emergence of new technologies such as IP Telephony. The end-to-end QoS guarantees for such applications can be provided by different QoS protocols such as RSVP and Differentiated Services for IP application traffic or by the ATM layer. Such difference in requirements for diverse multimedia applications calls for the availability of different flow and transport protocol implementations. This implies the need to provide a highly customizable communication protocol infrastructure that allows the application developer to build different protocol stacks as per the application delivery requirements. This differs significantly from the traditional protocol architectures that assume a pre-installed protocol stack that cannot be customized.

This paper makes two contributions to the study of the CORBA-based distributed multimedia streaming framework. First, it describes the design and performance of an implementation of a Pluggable Protocols Framework in TAO's Audio/Video (A/V) Streaming Service. This framework provides an Object Oriented (OO) protocol architecture that allows for the dynamic building of flow and transport protocol stacks. Second, it discusses our experience in adapting an existing Videoconferencing tool to use this framework.

Keywords: CORBA-based Multimedia Streaming, QoS-enabled OO Middleware, Pluggable Protocols

1 Introduction

There is an emergence of distributed multimedia applications exhibiting significantly stringent quality of service requirements than conventional data-oriented applications. This calls for new transport protocols with different characteristics to co-exist and be integrated with single applications. The different delivery requirements posed by these diverse multimedia applications often imply the need for highly customized protocol implementations. Hence, application developers are faced with the threat of code obsolescence caused by the development of even newer delivery techniques. [1]. Also, with recent advances in distributed object computing middleware and high speed networking, multimedia streaming is gaining significance. This is evident from the popularity of streaming products like RealVideo and QuickTime and their increasing use over the internet. OMG has defined a reference, control and management model for Audio/Video Streams based on CORBA. Our prior work in implementing the CORBA A/V Service [2] has shown that it is a very flexible model for developing distributed multimedia applications. In this paper we explore our further research on the design and implementation of a Pluggable Protocols Framework for a CORBA based Audio/Video Streaming Model, specified by the OMG, in TAO, a real-time ORB.

The remainder of the paper is organized as follows: Section 2 outlines the motivation for a pluggable protocols framework; Section 3 gives an overview of the existing implementation of the CORBA A/V Streaming Service in TAO, a real-time CORBA ORB; Section 4 provides details of the design of the Pluggable Protocol Architecture; Section 5 describes the Flow Specification; Section 6 gives details of adapting Vic, an existing videoconferencing tool, to use A/V Service objects; Section 7 analyzes the performance of different flow protocols of TAO's A/V Streaming Service; Section 8 discusses the future work on enabling QoS in the A/V Streaming Service to provide QoS guarantees between stream suppliers and consumers; Section 9 outlines related work in the area of data streaming; and Section 10 provides a concluding summary.

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

2 Motivation for Pluggable Protocols

There were two major reasons for the design and implementation of the Pluggable Protocols Framework. First, to provide an extensible infrastructure to plug in different flow and transport protocols and configure them dynamically and statically. For example, a wireless link based application may want to use the wireless link protocol to stream the data. RTP is gaining wide significance as the transport protocol for streaming audio and video data over the Internet. Our framework provides an implementation of RTP/RTCP for the audio/video profile. RTP is internet-centric and doesn't provide facilities to transport IDL-defined flows. SFP is a Simple Flow Protocol specified by the OMG for A/V Streaming and it can be mapped to different transports like UDP, RTP and ATM AAL5. Our framework provides an implementation of SFP that runs over UDP and multicast UDP. Second, it provides a framework that can be leveraged for our future work in providing end-to-end QoS guarantees with different QoS protocols on different network layers such as IP and ATM.

3 The CORBA Audio/Video Streaming Architecture

The CORBA A/V Streaming Service defines standard interfaces for the control and management components. These components make use of the Object Property Service [2] to set/get dynamic attributes called Properties and their values.

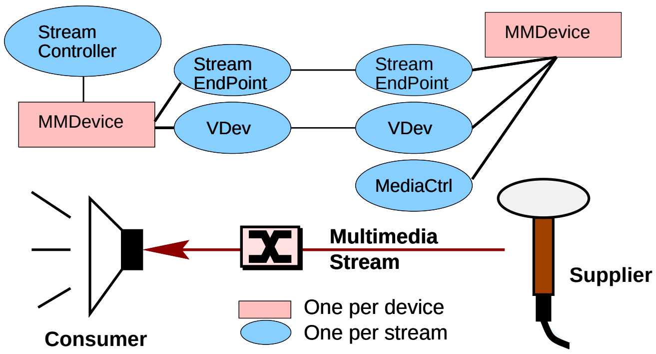

Figure 1: **A/V Streaming Service Components**

Stream interfaces: The following are stream interfaces defined in the CORBA A/V Streaming Service.

- **Flow:** It is the simplex data streaming channel that is set up between two network endpoints that are identified by the 5-tuple.

- **Stream:** A stream is a collection of the individual flows streaming data in both directions.

- **MMDevice:**

- **MMDevice:** This represents the Multimedia device. A Multimedia device could represent a physical or a logical device. Examples of MMDevices are VideoPhone,VoD (Video on Demand) Server, VideoConferencing System.

- **StreamCtrl:** This interface represents the Stream and has operations to start/stop the stream. A Stream can contain many flows and all operations on the stream will be applied to all the specified flows.The *StreamCtrl* interface is used to bind two *MMDevices* together with a specified QoS in a point-to-point binding. It also supports a point-to-multipoint binding.

- **VDev:** This interface represents the device specific aspects of the stream. It has operations to set and get the format and device parameters for each flow in the stream. *VDev* objects are created per stream by the *MMDevice*.

- **StreamEndPoint :** This interface represents the network specific aspects of a stream. It logically contains and controls the flow endpoints. The StreamEndPoint has 2 types: *StreamEndPoint_A* and *StreamEndPoint_B* both of which can contain producer and consumer flow endpoints except in the modified binding. In the case of multipoint binding the *StreamEndPoint_A* can contain only Producer flow endpoints and *StreamEndPoint_B* can contain only Consumer flow endpoints. StreamEndPoint objects are created per stream by the *MMDevice*.

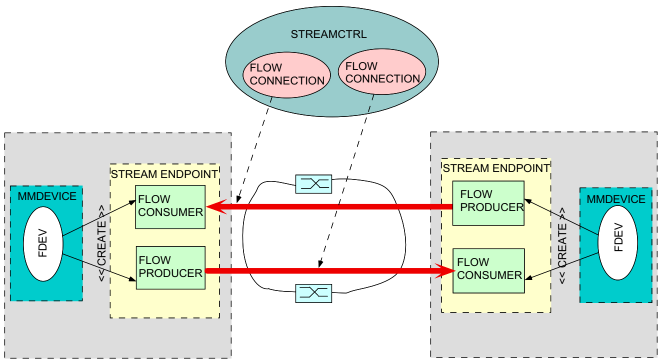

Figure 2: **A/V Streaming Service Components**

Flow interfaces: The CORBA A/V Streaming service specifies two different profiles for implementing the specification. The Light Profile is optimized for low memory footprint applications and the flow interfaces are not exposed ,that is, the individual flows cannot be controlled and only the above mentioned interfaces are used. The Full Profile implementation will make use of the following flow interfaces for accessing/controlling each flow within a stream. This profile provides applications with more flexibility. TAO's implementation of the A/V Service provides a full profile implementation. The following are the flow interfaces:

- **FlowConnection:** This interface represents a flow. It has operations to connect two flow endpoints and also to start and stop the flow. It also has operations to add producers and consumers in the case of a multipoint binding.

- **FlowEndPoint:** This interface represents all the aspects of a flow i.e network and flow device specific aspects. A FlowEndPoint is either a *FlowProducer* or a *FlowConsumer* and is created for each flow.

- **FDev :** This interface represents the Flow Device. A Flow Device is similar to the *MMDevice* for a stream, except that it creates only a *FlowProducer* or *FlowConsumer* object.

4 Pluggable Protocols Architecture

The OMG specification defines the flow specification syntax that is to be used for the bind_devs calls for connection establishment. It defines the protocol names and also the syntax for specifying the transport/flow protocol information but it doesn't define any interfaces for the protocol implementation as such. Our framework bridges this gap by defining a uniform API for the different flow protocols like RTP and SFP while taking care of the variations using a Policy Interface as described in section 4.2.5.We solve this with our Pluggable Protocol Framework using design patterns like *Layer*, *Acceptor*, *Connector*, *Facade* and *Abstract Factory*;

This section describes the various components of the Pluggable Protocol Architecture. It talks about the different design issues that were faced in providing the pluggable protocol functionality and how they were resolved.

4.1 Pluggable Protocol Framework Components

The pluggable protocols implementation makes use of the following components as shown in figure 3

TAO_AV_Core: This component is a singleton in that there'll be only one instance per process. It is a container for the flow protocol factories and also for the transport protocol factories. An application using TAO's A/V implementation has to initialize this before using any of the A/V classes like *TAO_StreamCtrl* and *TAO_MMDevice*. During the initialization, the *AV_Core* class loads all the flow protocol factories,

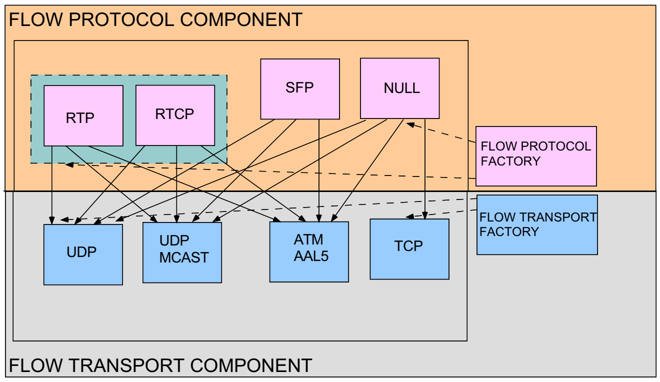

Figure 3: **Pluggable Protocol components**

control protocol factories and transport factories from the *Service_Configurator* and creates default instances for the known protocols.

Connector and Acceptor Registry: The *TAO_AV_Connector_Registry* and *TAO_AV_Acceptor_Registry* classes provide the Facade to access the abstract flow and transport factories.

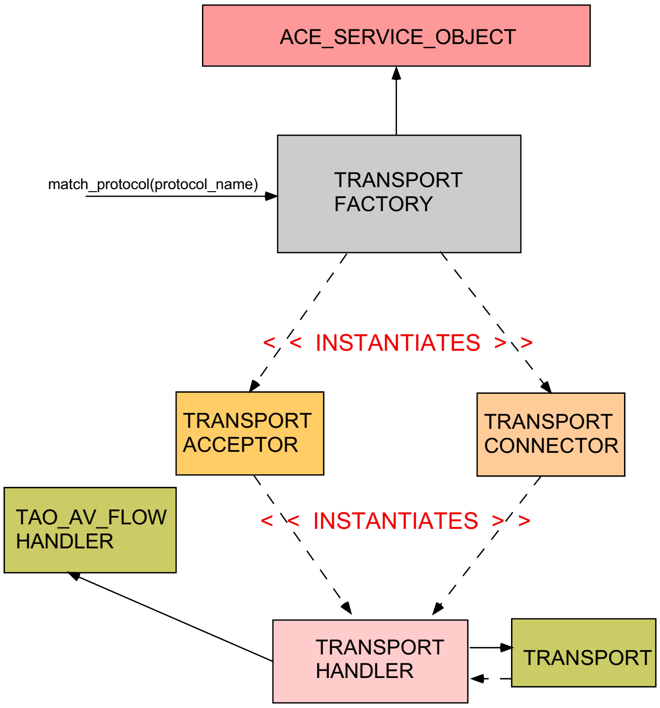

Figure 4: **Pluggable Transport Components**

Transport Components: The following are the components required for each transport protocol as shown in figure 4

- *Acceptor and Connector* These are implementations of the Acceptor and Connector [3] design patterns. In the case of connectionless protocols like UDP, calling accept and connect will result in the creation of handlers immediately.
- *Flow_Handler* All the transport handlers derive from the Flow_Handler which has methods to start and stop and provide flow specific functionality for timeout upcalls to the *Callbacks*.
- *Transport_Factory* This is an Abstract_Factory providing interfaces for creating Acceptors and Connectors.

Flow Protocol Components: The following are the components required for each flow protocol as shown in figure 5

- *Flow_Protocol_Factory* This is an implementation of the *Abstract_Factory* to create flow protocol objects.
- *Protocol_Object* This is a concrete implementation providing the flow protocol functionality. Applications use this interface to send frames and the *Protocol_Object* uses application specified *Callback* objects to deliver frames.

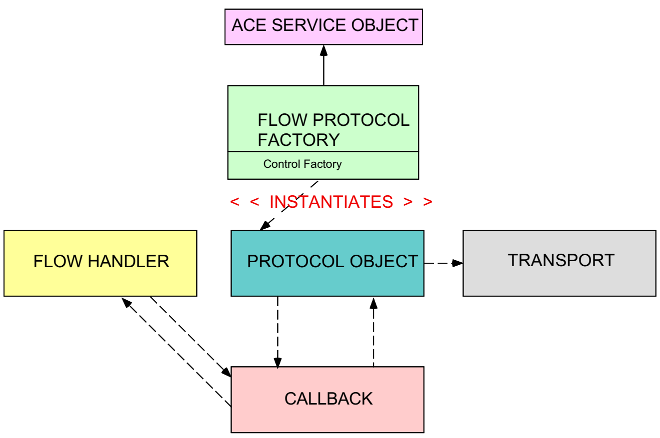

Figure 5: **Pluggable Flow Protocol Components**

4.2 Issues faced and resolved in the Design of Pluggable Protocol Architecture

In this subsection we address the design issues in providing a pluggable protocol functionality and discuss how we resolved these challenges using design patterns [4].

4.2.1 Adding New Transports Transparently

Context: Multimedia applications have different QoS requirements. A video application over an intranet may want to take advantage of the native ATM transport to reserve bandwidth. An audio application in a Video Conferencing application may want to use a reliable transport like TCP since loss of audio is more visible to users than video and the bit-rate of audio flows are low (in the order of 8 kbps using compressions like GSM), whereas a video application might not want the retransmission and slow starts offered by a TCP, instead it can use the facilities of a transport like UDP since losing a couple of frames doesn't bother it much.

Problem: It should be possible to add new transports to the pluggable protocols framework without making changes to the rest of the A/V framework. Thus the framework must be open for extensions but closed to modifications, *i.e*, the Open-Closed principle [5]. Ideally creating the new protocol and configuring it into the A/V framework should be all that is required.

Solution: Use a registry to maintain a collection of *Abstract Factories* [4]. In the Abstract Factory pattern, a single class defines an interface for creating families of related objects, without specifying their concrete types. Subclasses of the Abstract Factory are responsible for creating concrete classes that collaborate amongst themselves. In the context of pluggable protocols, each Abstract Factory can create the `Connector`, `Acceptor` classes for a particular protocol.

Applying the solution in AV: In TAO's A/V framework , the `Connector_Registry` plays the role of the protocol registry. This registry is created by the *TAO_AV_Core* class. Figure 6 depicts the connector registry and its relation to the abstract factories.

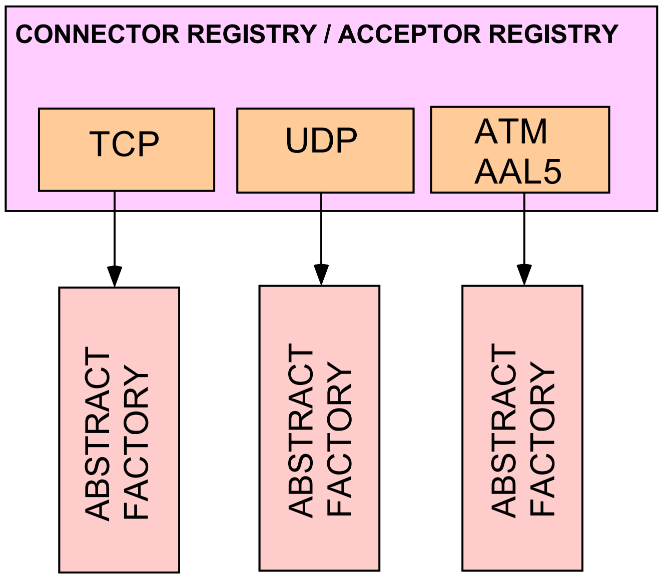

Figure 6: **Connector Registry**

These factories are accessed via the *Facade* [4] pattern in order to hide the complexity of manipulating multiple factories behind a simpler interface. The Registry described above plays the Facade role.

4.2.2 Adding New Flow Protocols Transparently

Context: Multimedia flows typically use 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 might need a timestamp so that the receiver can play the frame at the right time. Also sequence numbers will be needed if a connectionless protocol like UDP is received so that the applications can do resequencing. Also information like source identification number will be needed in the case of multicast flows to separate the flows from different sources. RTP, the Real-time transport protocol provides such facilities to transport in-band data. But RTP is internet-centric and cannot be used to carry IDL-typed flows. For example RTP specifies that all header fields should be in network-byte order whereas SFP uses the CDR encoding and carries the byte-order in each header. SFP is a simple flow protocol defined by the OMG A/V specification and provides transporting such in-band data.

Problem: Provide an abstraction for the flow protocols . Also the flow protocols should be able to run over different transports.

Solution: To solve the problem of a flow protocol running over different transports the *Layers* pattern is used. TAO's A/V framework provides a uniform transport layer for UDP, UDP MCast and TCP. In cases where the mapping of the flow protocol differs over different transports they can provide a different mapping. The *AV_Core* class maintains a registry of flow protocol factories. TAO's AV framework provides a flow protocol layer using a *Protocol_Object* interface.

4.2.3 Adding New Protocols Dynamically

Context: When developing new pluggable protocols, it is inconvenient to recompile the AV framework and applications just to validate a new protocol implementation. Moreover, it is often useful to experiment with different protocols, *example*, systematically compare their performance, footprint size, and QoS guarantees. Also, in telecom systems which are 24×7 systems with high availability requirements, it is important to configure flow and transport protocols dynamically, even while the system is running. This level of flexibility helps simplify upgrades and protocol enhancements.

Problem: How to populate the registry *dynamically* with a set of flow and transport factories.

Solution: Use the Service Configurator [6] pattern, which decouples the implementation of a service from its configuration into the application.

The Service Configurator dynamically loads the set of entries in the Registry. For instance, a registry can simply parse a configuration script and dynamically link the services listed in it. This is the most flexible approach, but it requires more code, *example*, to parse the configuration script, load the objects dynamically, etc.

Applying the solution in AV: The *TAO_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[1] Service Configurator implementation [7]. Figure 7 depicts the connector registry and its relation to the Service Configurator.

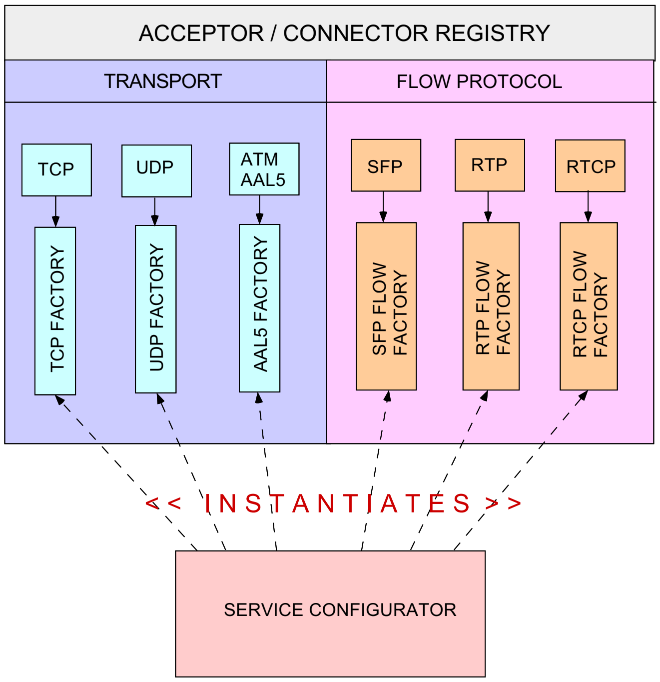

Figure 7: Connector, Acceptor Registry and the Service Configurator

4.2.4 Control Protocols

Context: RTP has a control Protocol RTCP associated with it. Every RTP participant has to send RTCP frames since it

[1] ACE provides a rich set of reusable and efficient components for high-performance, real-time communication, and forms the portability layer of the AV framework. (http://www.cs.wustl.edu/ schmidt/ACE.html)

provides the control information eg. the name of the participant and the tool being used. Also RTCP sends reception reports for each source from which it received.

Problem: Protocols like SFP make use of the A/V interfaces to exchange control interfaces. The use of RTP for a flow necessitates it to transmit RTCP information. RTCP extracts such control information from RTP packets. We need to provide an extensible interface for such control protocols and also provide means for interaction between the data and control protocols.

Solution: The solution is to make the control protocol information as part of the flow protocol. For example, RTP knows that RTCP is its control protocol. To reuse the pluggablity features make the control protocol use the same interfaces as its data components.

Applying the solution in AV: During stream establishment *Registries* will first check for the flow factory for the flow protocol mentioned. Once the listen or connect is done for the data flow, the *Registry* will check if the flow factory has a control factory. If yes, then it will do the same processing for the control factory except the endpoint will be one higher than the data endpoint. Since the A/V Specification doesn't mention a portable way of specifying Control flow endpoint information, we followed this approach as a temporary solution until the OMG comes up with a solution.

The RTCP implementation in AV uses the same interfaces as RTP like *Flow_Protocol_Factory* and *Protocol_Object*. RTP will call the *handle_control_input* method on the RTCP *Protocol_Object* when a RTP frame is received. This will let the RTCP Object extract the control information like sequence number of the last frame.

4.2.5 Interface for Variations in Flow Protocols:

Context: In the previous paragraph we explained how different multimedia flow protocols have been abstracted out and we provide a uniform interface to all. But, there are inherent variations in such protocols. For example RTP needs to transmit the payload type i.e format of the flow in each frame, whereas SFP makes use of the control and management interface in AV to set and get the format values for a flow. Similarly, RTP has a Control Protocol RTCP, which periodically sends out sender information like Name and E-mail, while SFP doesnt transmit such information. But such information are not changed with every frame. For eg. the Name and E-mail information of a participant in a conference will not change for a session. Also the properties of the transport needs to be controlled by the application sometimes. For example a conferencing application may not want to have Multicast Loopback.

Problem: The A/V framework should allow the user to set such variations and also provide a single interface for the different flow protocols. Also the interface should be open to changes with the addition of new flow protocols and transports.

Solution: The solution to the above problem is the *Policy* mechanism as defined in CORBA. 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.

5 Flow Specification

An important specification for interoperability of flows between two A/V Streaming implementations is the flow specification. This is passed to the bind_devs call on the StreamCtrl object. The syntax for the flow specification is specified in the following BNF grammar.

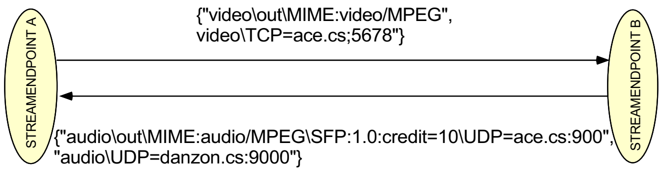

Figure 8: **Flow Specification**

The flow specification is passed between the two streamendpoints to convey per-flow information. It represents all the aspects of a flow such as its name, format, flow protocol being used and the transport name and address. The flow specification string is analogous to the IOR in the CORBA model. Standardization of this ensures that two different Streamendpoints from two different implementations can interoperate. There are two different flow specifications depending on the direction in which the flowspec is travelling. If it is from the A StreamEndPoint to the B StreamEndPoint then it is a forward flowspec and the reverse direction is the reverse flowspec.

TAO's A/V Service provides helper classes *TAO_Forward_FlowSpec_Entry* and *TAO_Reverse_FlowSpec_Entry* for the application to construct these flow specification string from their components without worrying about the syntactic details. For example the entry takes the address as both an ACE_Addr and a string. It provides convenient parsing utilities for the strings. On the other hand one could give it a flow_spec string and get the individual components like flowname, format and transport information.

6 Vic - VideoConferencing Application

Vic [8]is a videoconferencing application developed at the University of California, Berkeley. We have adapted Vic to use TAO's A/V Service objects and its pluggable protocols framework. Vic currently uses the RTP/RTCP implementation in TAO. The aim of adapting this application to the A/V Streaming Service was to demonstrate the ease with which an application using a new flow protocol RTP can be easily built using the pluggable protocol framework infrastructure provided by the A/V Streaming Service.

6.1 Architecture of Vic:

Vic provides a video conferencing application. The audio conferencing is done with another tool, Vat. The Vic family of tools do synchronization between media using a conference bus mechanism, which is implemented as multicast datagram sockets bound to the loopback interface.

The Architecture of Vic is driven largely by the *TclObject* interface. *TclObject* provides operations so that operations on the object can be invoked from a Tcl script. By using a Tcl based approach Vic provides rapid prototyping and reconfiguration of the encode/decode paths.

One of the design challenges that we faced while adapting Vic to use the A/V framework was to integrate both the GUI and ORB event loops. This was solved using the Reactor Pattern [9]. We can write a wrapper reactor that uses the GUI's event loop and use that for the ORB.

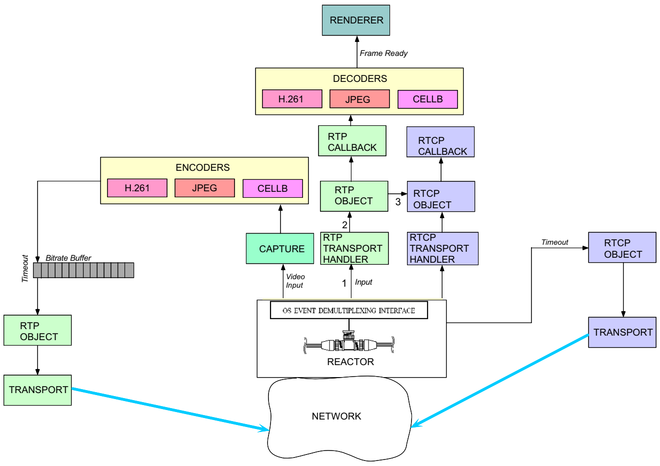

Figure 9: **Architecture of Vic using AV framework**

6.2 Vic using A/V Objects

In this subsection we discuss the A/V framework Objects in Vic. Vic uses the MultiPoint-to-MultiPoint binding in the A/V framework. The first step was the proper abstraction for the conference device. Since a Videoconferencing application is both a source and sink, we need a source and sink MMDevice. Also to be extensible for future integration with vat and other tools, Vic uses the flow interfaces so that Video is considered as a flow within the conference stream. Since Vat runs in a separate address space the flow interfaces have to be exposed and hence we used the Full Profile Flow interfaces i.e FDev, FlowProducer and FlowConsumer.

Vic was highly coupled with RTP and the 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 the *Protocol_Object* interface . The modified Vic still preserves the Application Level Framing concept of RTP. Vic's RTCP functionality has been abstracted into the A/V framework and the framework automatically defines a RTCP flow for a RTP flow. The modifed Vic is independent from the network specific details of opening connections and io handling since it uses the *Pluggable Protocols* framework in A/V.

The next step was to define the *Callback* objects for the source and sink FlowEndPoints. The *Source_Callback* uses the timer functionality to schedule timer events to send the flows. The 9 explains the sequence of events that lead to the sending of frames. When the input becomes ready on the VideoCard, the grabber reads it and gives it to the *transmitter* which 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 get the RTCP functionality Vic implements a RTCP_Callback which derives from the *TAO_RTCP_Callback* to provide Vic-specific Source objects.

Vic can function both as a participant in a centralized conference as well as a loosely-coupled conference. This 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 which registers itself with the Naming Service, all the Vics will become part of a centralized conference, which could be controlled from the control process. On the other hand, when no such process is run, Vic reverts to the loosely controlled model by creating its own streamctrl and transmitting on the multicast address.

7 Benchmarking

7.1 Performance Comparisons of different Flow Protocols with and without the pluggable rotocol framework (Blackbox and White box analysis

8 Future Work

8.1 Enabling QoS in A/V Streaming Service

Our next step would be to enable end-to-end QoS for the flows protocols. The Pluggable Protocol Framework enables us to provide QoS guarantees using either IP end-to-end QoS protocols like RSVP and Differentiated Services or the ATM. This is ensured by the flexibility that this framework provides for building different protocol stacks as per the application requirements. So if we want to use the ATM QoS guarantees then we choose the ATM AAL5 transport protocol. If we want to use RSVP or Differentiated Service QoS provisions then we choose the TCP or UDP transport protocols. Also this framework helps us dynamically choose the flow and transport protocols that we want to use according to the data that we are streaming. For example we would like to use reliable TCP for audio transmission and unreliable UDP for video transmission.

The AV Streaming Service specification has provisions for the application developer to specify the QoS requirements when setting up a connection. We can specify these QoS parameters for each flow through a sequence of properties which are name, value pairs. But these are application level QoS parameters and the specification leaves it to the AV Streaming Service implementation to translate the application level QoS parameters like video frame rate to network level QoS parameters like bandwidth. We would like to build a QoS framework that we can use to ensure end-to-end QoS for the flows. We have identified three main components for this framework namely the QoS Mapping, QoS Monitoring and Adaptation and QoS-Based Transport API.

The QoS mapping component translates QoS specifications between different levels such as application and network levels. This is required to reserve network sufficient resources at connection establishment. Good mapping rules are required to prevent reservation of too much (or too little) resources. The QoS Monitoring and Adaptation component measures end-to-end QoS parameters over a finite time period and takes actions based on the measured QoS and the application requirements. It facilitates renegotiation of the QoS parameters between the sender and receiver. The QoS-Based Transport API component provides calls for provisioning control (renegotiation and violation notification) and media transfer enforcing end-to-end network QoS. The ACE framework provides QoS API's that provide us these functionalities. The ACE QoS API's use the G-QoS and RAPI API's to enforce end-to-end network QoS.

9 Related Work

RTSP, the Realtime Streaming Protocol is an IEFT draft standard and is aimed to control streams similar to the A/V framework. It defines all the messages that are to be exchanged between the client and the server and also provides ways for the client to request particular resources example using a URI to get a specific file. It has been implemented in the RealPlayer. It is similar to HTTP/1.1 and takes advantage of HTTP/1.1 features like caching, tunellinng and proxies. The A/V framework differs from RTSP in that it aims to provide a higher-level framework for multimedia application developer and can be transparently adapted to interoperate with different control protocols like RTSP and H.323 using bridges.

Java Media Framework is an API for incorporating time-based media into Java applications and applets. It provides a plugging API for the decoders and encoders and provides implementation for common formats like AVI, MPEG and WAV files. It also defines common user interface and rendering components. The goal of the A/V framework is to provide a framework for configuring and finding network streaming services leaving the end components like decoders and encoders to implementors. JMF is a complimentary framework and can be nicely integrated with the A/V framework.

10 Concluding Remarks

Real-Time multimedia streaming and conferencing are gaining importance with the increase in network bandwidth and the spread of the internet. 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 protocols and transports for multimedia. In such situations a flexible framework to adapt to the new developments is needed. CORBA Audio/Video Streaming Service provides flexibility by using an open standard and using standard interfaces.

Furthermore, distributed multimedia application developers need a framework which makes use of the A/V interfaces and also abstracts the network/ protocol details. In this paper we have contributed a pluggbable protocols framework which provides the capability to rapidly adapt to new transports and flow protocols. TAO's implementation of the CORBA Audio/Video Streaming service provides a complete implementation of the specification i.e flow interfaces, point-to-multipoint binding. In addition it also provides a multipoint-to-multipoint binding for conferencing applications.

The paper also talks about how we envision to leverage the Pluggable Protocols Framework in our on-going work on providing end-to-end QoS for the individual multimedia flows.

References

[1] J.-F. Huard and A. Lazar, "A Programmable Transport Architecture with QOS Guarantees," *IEEE Communications Magazine*, vol. 36, pp. 54–62, October 1998.

[2] S. Mungee, N. Surendran, and D. C. Schmidt, "The Design and Performance of a CORBA Audio/Video Streaming Service," in *Proceedings of the Hawaiian International Conference on System Sciences*, Jan. 1999.

[3] D. C. Schmidt, "Acceptor and Connector: Design Patterns for Actively and Passively Initializing Network Services," in *Workshop on Pattern Languages of Object-Oriented Programs at ECOOP '95*, (Aarhus, Denmark), August 1995.

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

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

[6] P. Jain and D. C. Schmidt, "Service Configurator: A Pattern for Dynamic Configuration of Services," in *Proceedings of the 3^{rd} Conference on Object-Oriented Technologies and Systems*, USENIX, June 1997.

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

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

[9] D. C. Schmidt, "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Dispatching," in *Proceedings of the 1^{st} Annual Conference on the Pattern Languages of Programs*, (Monticello, Illinois), pp. 1–10, August 1994.