



The ATON Project

Center for Research in Electronic Art Technology
University of California, Santa Barbara
Computer Vision and Robotics Research Laboratory
University of California, San Diego
CalTrans Test-Bed Center For Interoperability



ATON Report 2000.1: The Design of a Protocol for High-Performance Distributed Multimedia Data Processing

Stephen Travis Pope (stp@create.ucsb.edu), Frode Holm, Alex Kouznetsov, and SunWoo Kim
CREATE, UCSB, June, 2000

Contents

- Preface2
- Introduction2
- Requirements and Available Solutions9
- ATON High-Performance Distributed Multimedia11
- A Simple Example12
- Design Issues14
- Conclusions15
- References15
- Appendix 1: Extended Example of Extended IDL16
- Appendix 2: CORBA A/V Streaming Description18
- Appendix 3: Examples of Point-to-Point CORBA A/V Streams22
- Appendix 4: CORBA A/V Device and Stream Parameters25

Preface

Distributed real-time object-oriented software is a topic of much current interest; for example, the June, 2000 issue of the *IEEE Computer* magazine is a special issue on the subject. This document describes phases 1 and 2 of the High-Performance Distributed Multimedia (HPDM) task that is being carried out in the CREATE Lab. at UCSB in partnership with teams at the CalTrans TCFI and UCSD CVRRL. This is Research Thrust 5 of the DiMI ATON Project. Our goal is to provide a flexible and scalable distributed computing environment for the other ATON software development tasks, which involve multi-user virtual reality, distributed robots, complex real-time signal processing, large-scale simulation, distributed real-time databases, and wide-area (and wireless) networking.

The introduction discusses general requirements for writing and deploying large-scale distributed software systems, such as is needed by the ATON DRIVE (Distributed Real-time Interactive Virtual Environment) and robotics efforts. After presenting the basic technology of the CORBA (Common Object Request Broker Architecture) system, we develop a proposal for an extension to the CORBA interface definition language (IDL) that adds Quality-of-Service (QoS) attributes to CORBA interfaces and provides information that can be used by an advanced CORBA run-time, naming and trading services. Taking these extensions, we investigate the CORBA audio/video (A/V) streaming standard with respect to our requirements and our infrastructure. The appendices present an in-depth example of CREATE's ExIDL (Extended IDL) language, as well as CORBA-related examples and property lists.

Introduction

A great deal of effort has been invested over the last decade in scalable distributed computing on multiprocessor systems. Recently, the CORBA (Common Object Request Broker Architecture) standard has gained wide acceptance as a framework for interprocess communication and coordination over (potentially) wide-area heterogeneous networks. Several software suppliers now have CORBA-compliant software packages (Object Request Brokers or ORBs) that allow programs written in a variety of object-oriented languages (e.g., C++, Java, or Smalltalk) running on a wide range of hardware platforms to interoperate. There is also a competing single-vendor system called DCOM from Microsoft.

Given these developments, it is now possible to perform wide-area distributed processing with little of the programming overhead associated with traditional multi-tier applications. This advance in technology has led to a number of new application architectures, and to the widespread use of CORBA in many application domains, even areas where real-time interaction and controlled latency are required.

Background

In order to develop software that runs complex tasks distributed over several computers, one needs a program that is well-partitioned, and an infrastructure for starting, managing, and stopping subtasks over a local-area (or wide-area) network.

There are a number of reasons for building distributed software systems, the most obvious of which are to achieve performance scaling (e.g., for large databases, digital signal process-

ing, rendering, or multi-user virtual environments), or to achieve fault-tolerance (in high-availability systems). A third class of applications involve physically (geographically) distributed programs (as needed for network management)

The architecture of the ATON software infrastructure presents three levels. The lowest level provides the communications infrastructure for mobile networking and flexible user I/O. This will likely be based on one of the new standards for medium-range (on the order of tens of meters) high-speed communication. The two candidates here are the 2 Mbits/sec. “Blue-Tooth” consortium technology, and the recently standardized IEEE-802.11b technology (with CCK modulation), which offers up to 11 Mbits/sec.

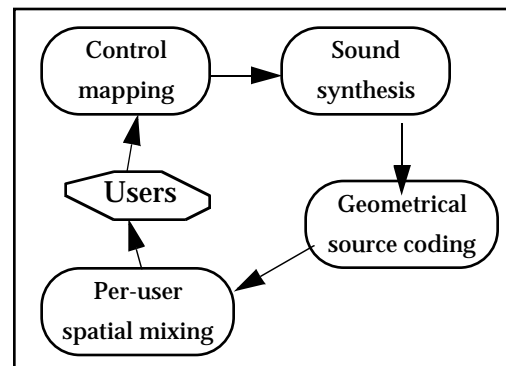
The middle-level software provides a uniform LAN-oriented distributed computing platform based on multi-vendor standards such as CORBA and Java RMI. In the HPDM task we are developing a high-level API for distributed multimedia signal processing that is independent of the external data representation, remote function call discipline, and transport details.

At the applications-level, we are developing integrated virtual reality simulation applications for visualization and interaction with both virtual and robotic worlds.

Examples of Distributed Systems

To motivate our discussion, we present several examples of distributed systems that pose different requirements on the network infrastructure and software libraries and tools.

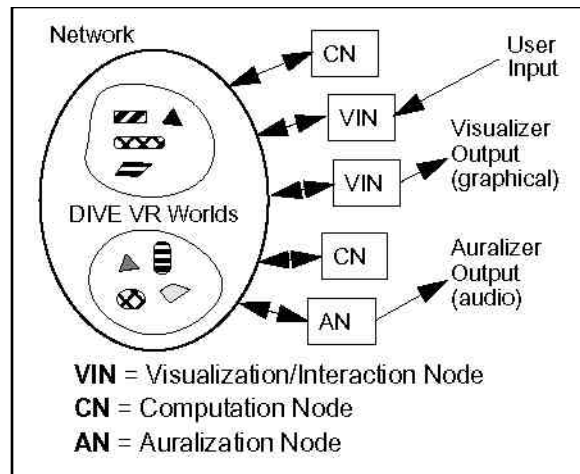
The first example is a distributed real-time digital signal processing for sound synthesis and spatialization for use in an immersive user interface (i.e., surround sound for a virtual reality system). Here we see one or more users controlling objects in the virtual world, these input gestures being mapped to high-level control parameters that drive a synthesis, sampling, or streaming audio system. The virtual source geometries are managed by another component, which updates the database that can be read by the geometry-based spatial sound mixers for one or more users.



A distributed real-time event collection and signal processing application such as this requires a language able to express the data usage characteristics of individual function calls, the run-time behavior and performance scaling of algorithms, and the hardware devices used in certain application object interfaces. The run-time infrastructure for this application should support task migration among servers, sophisticated stream buffer management, negotiable packet-dropping policies, and fault-tolerant process re-start.

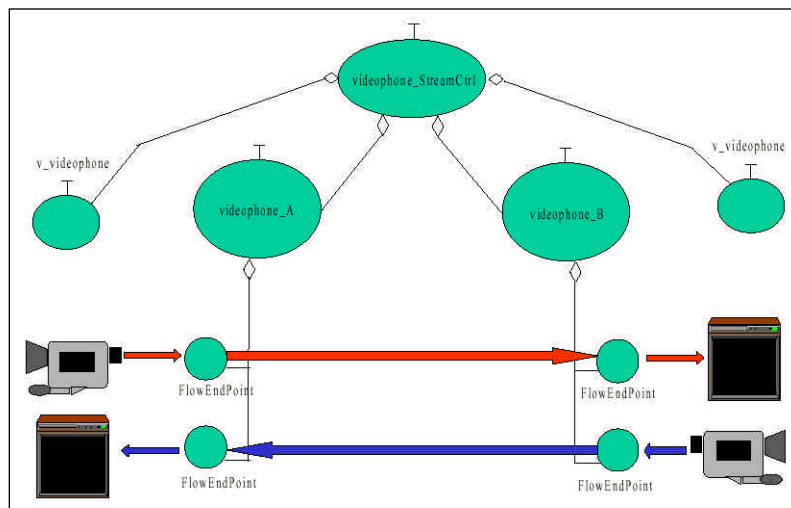
The second example is the DRIVE (Distributed Real-time Interactive Virtual Environment) distributed virtual reality system. DRIVE is a multi-user virtual reality system that is implemented within a coarse-grained heterogeneous multiprocessing environment (i.e., a network of different UNIX workstations). The system is a tool kit for building distributed interactive applications based on the simulation of virtual worlds.

Independent DRIVE applications (called “AIs” after W. Gibson), can run on various nodes distributed within a local- or wide-area network; AIs can update a shared (node-replicated) object database. This architecture makes possible the distribution of the computational load of object animation and interaction. User input and output processes are loosely coupled with the rest of the system, allowing for flexible user input device handling, and several modes of graphical (and audio) rendering.



For DRIVE, we need (a) efficient 1-to-many event broadcasting, (b) good portability to heterogeneous systems (i.e., different platforms, development languages, and network connections); and (c) support for special-purpose (and host-specific) I/O devices such as user-tracking input or stereo-optic visual output.

The final example is a video-conferencing application where two users set up a bidirectional video stream that is managed by software components running on each user’s desktop machine. The stream management software communicates over a different connection (network and protocol) than that used by the real-time video data stream and can dynamically negotiate the Quality-of-Service (QoS) for the video and audio streams in response to network bandwidth availability.



Multi-user real-time data streaming applications for video, audio, screen updates, and other multimedia data require system support for managing out-of-band (OOB) data streams (i.e., those that use different networks), and routing media data connections. The application must be able to specify its needs (in terms of QoS negotiations) to the run-time streaming manager. Some of these issues are addressed in the CORBA Audio/Video streaming specification (CORBA A/V).

Each of these three example systems have different distribution requirements, and place different demands on the underlying network, the software development aids, and the multi-host run-time system. In the following sections, we will discuss the basic language and pro-

programming tools, the run-time managers and infrastructure components, and the currently available solutions for building large-scale, real-time distributed systems.

Components of Distributed Software Systems

To execute a parallel or distributed program, one needs a run-time task scheduler and data manager, some manner of task monitoring, and before-the-fact tools for task partitioning for use on distributed architectures. The programming language of choice should have some support for the description of formal published external interfaces, and libraries are needed to make remote “proxy” objects as transparent (i.e., indistinguishable from an true local object) as possible.

These partially conflicting requirements drive many interesting design issues in building an infrastructure for powerful real-time multi-user distributed programs.

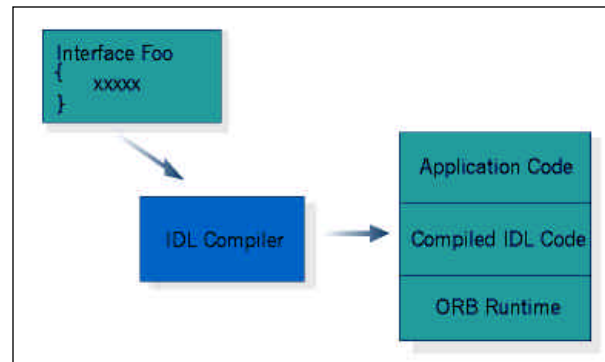
Writing Software for Distributed Processing

Any distributed programming library must include at least two components: (1) a standardized external data representation (**XDR**, which describes how data is to be passed over the network in a machine-independent fashion), and (2) a standardized format for remote procedure calls (**RPC**) or (in the case of object-oriented programming languages) remote method invocations (**RMI**). Distributed programming packages often provide a format for describing XDR/RMI interfaces in an implementation-language-independent way, for example, using a special interface definition language (**IDL**).

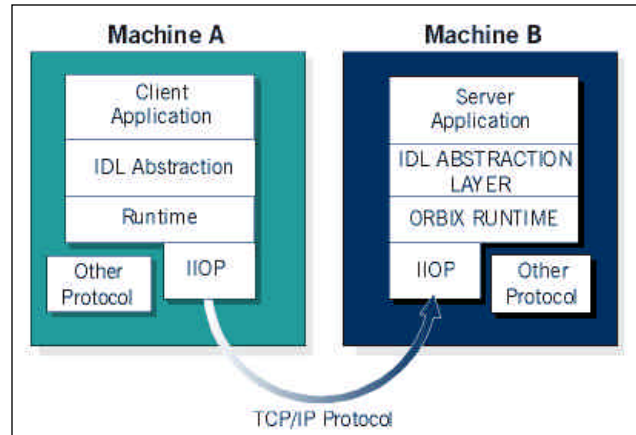
Two other desirable features of distributed programming libraries for multimedia applications are: (3) a mechanism for asynchronous 1-to-many communication among components (also called event distribution, notification, publish/subscribe services, the dependency mechanism, or signals), and (4) a method of specifying an application's quality-of-service (**QoS**) requirements. in terms of network delays (latency) and “burstiness” (latency jitter).

Typical distributed programming frameworks (e.g., CORBA, COM, or JavaRMI) consist of a user program (client) that can send messages to another (server) using the server's published interface (which is written in some interface definition language, e.g., CORBA IDL). To do this, the support libraries must include XDR converters that take the arguments from the client and turn them into a linear, machine-independent format, and then convert them back for delivery to the server function. The RPC/RMI interface manages how the remote procedure call is passed over the network, and how the client maps from the network format of the call to an actual program function address.

Traditional socket-based distributed programming tools provide the first two features (XDR and RPC/RMI), but generally lack the latter two (events and QoS). The CORBA specification standardizes XDR, RMI, and “event channels” but lacks any notion of QoS.



CORBA also introduces overhead that may make it too inefficient for many real-time, high bandwidth, low latency, bounded latency jitter applications. The CORBA RMI and event mechanisms also have very different performance characteristics and scalability considerations. In applications where deterministic response time (bounded latency) is a requirement, it may be difficult to characterize the suitability of CORBA RMI or event channels.



Even the emerging real-time CORBA standard focuses on fixed priority scheduling algorithms, which are unsuitable for applications with non-deterministic and non-static QoS requirements (such as we are addressing). ATM networks provide a fine-grained model of QoS, but few higher-level libraries provide application-level “hooks” that allow programmers to make simple use of these facilities.

Naming and Trading services

To run any non-trivial distributed system, some form of **naming service** for locating remote objects is required. Basic naming services assume a well-known semi-global namespace known to both servers and clients (often passed among programs as absolute inter-object references (or IORs). This is the equivalent of being able to locate an object by name string and host machine (and getting the IOR of the object). Some refer to a naming service as the “white pages” model — if you know the exact name and location...

A **trading service** can locate remote objects based on their interfaces — “find me an object that exports this interface method”. This is more like the “yellow pages” model — search by description. Advanced traders can take run-time criteria into account in giving out IORs in response to client requests. The criteria might include machine-load, interface bandwidth requirements, co-location with databases, or access to specialized I/O devices.

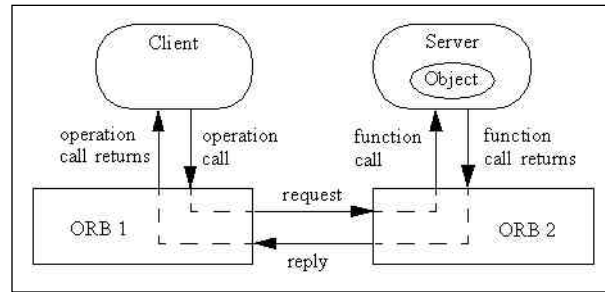
CORBA defines naming and trading services as add-on specifications, outside of the functionality of the basic common ORB. Support for the advanced CORBA services, and the performance of the servers themselves, varies greatly among ORB vendors.

Much of our effort in HPDM year 2 will focus on developing advanced trader implementations that support an extended IDL language.

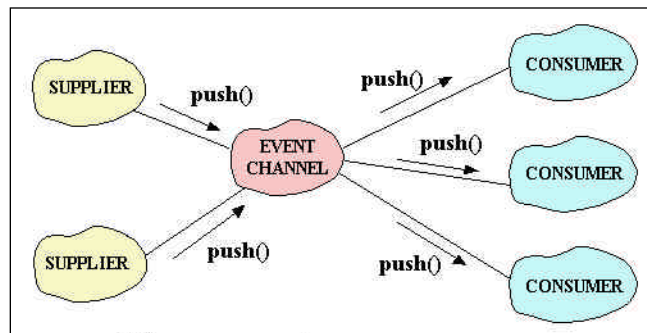
Directory services allow object location by some traversal of a site- or application-defined hierarchy or heterarchy.

Remote Method Invocation and Publish/Subscribe Event Services

The normal model of object interactions assumes a single “client” and a single well-known “server” object for each remote procedure call or remote method invocation, as shown in the Figure to the right. In this case, the client sends a message to (or calls a function in) the receiver, and the client’s thread is blocked until the receiver responds.

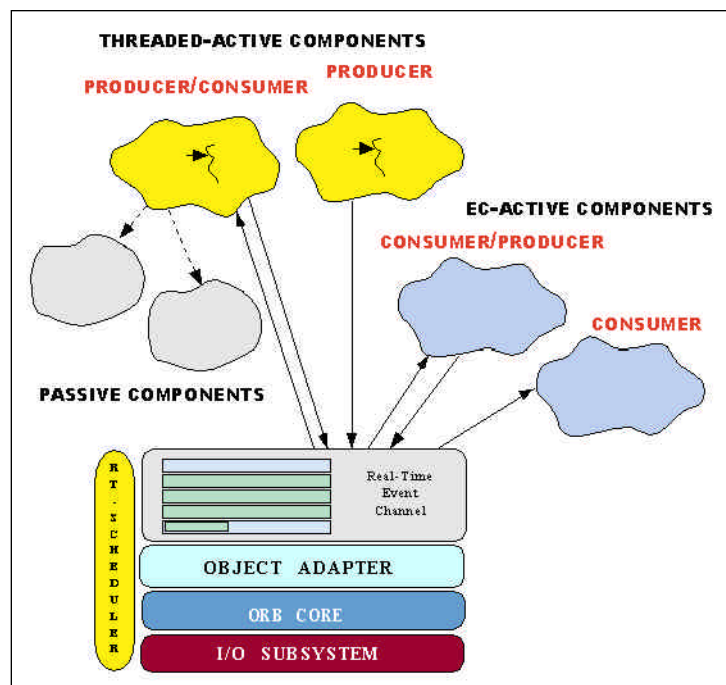


While this model is sufficient for a large class of applications, there are also cases where a 1-to-many non-blocking asynchronous RMI is needed as in the publish-subscribe mechanisms offered by other software systems (e.g., using semaphores, message queues, event distribution, notification, publish/subscribe services, the dependency mechanism, or signals). In CORBA, one



can use the **Event Channel** service, whereby an event channel may have multiple event producers and/or consumers, each of whom does not have to know about the others. Event channels can apply filters, so that events can be routed only to consumers interested in them.

A CORBA ORB can be extended with a efficient (or even “real-time”) event service, and producers and consumers can connect to it as shown in the Figure on the right. The ORB sits in the lower-left, with an event service layer that provides the I/O for components that are actively producing events, and others waiting for a notification from the event service.

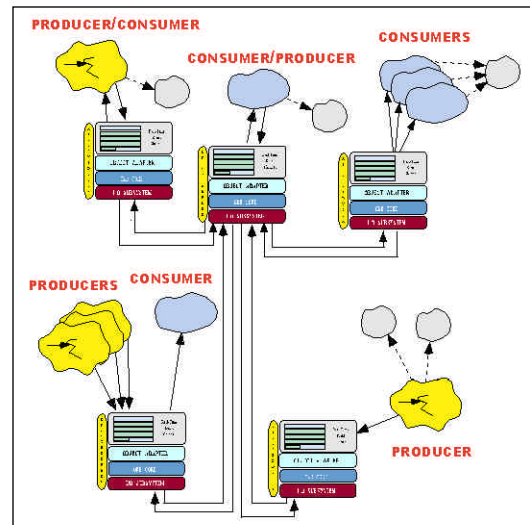


In this case, the ORB is a central point of coordination of both synchronous RMI and asynchronous event distribution. The scheduler that synchronizes RMI and event activity is shown on the left of both the ORB and the real-time event channel object.

Some of the components involved are in regularly scheduled threads (called the “Thread-Active” components in the Figure), which some objects live in threads that are activated by in-coming event activity (the “EC-Active” objects).

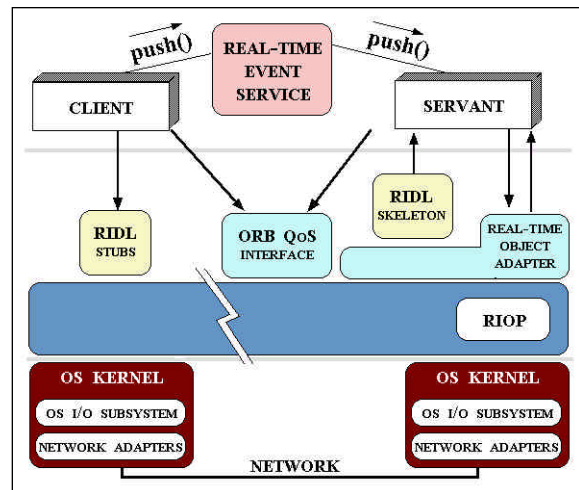
In a multi-computer (possibly geographically distributed) networked application, multiple event services need to communicate and coordinate, just like the ORBs themselves do for distributed RMI. This architecture then looks like that shown in the Figure on the right.

For this architecture, we need a distributed notification service with “smart” filter migration that can minimize the network traffic and uses a priority model for event delivery.



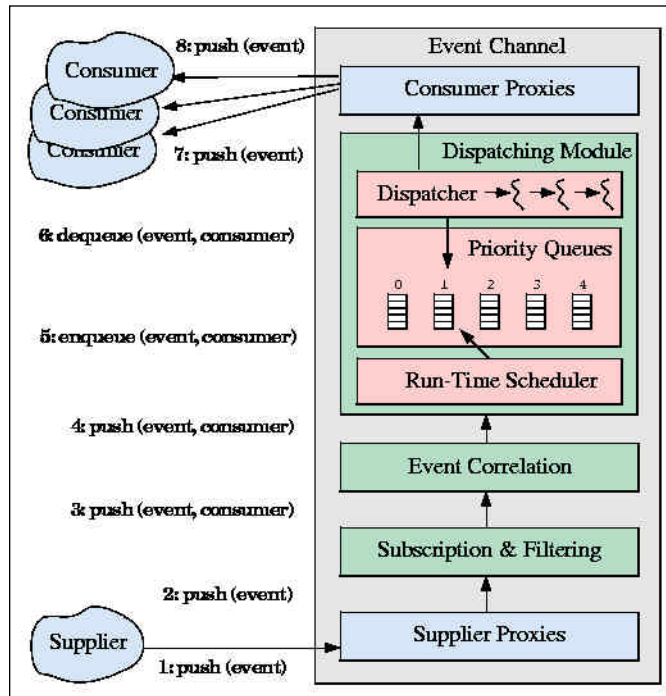
To make asynchronous publish/subscribe systems such as event channels work efficiently, it is often necessary to provide a special event server or “notification manager” that can handle event filtering at both the producer and consumer (and possibly intermediate) processes, and to have message queues for various processor or even individual objects.

This Figure shows the event service as separate from the ORB, and shows the ORB’s interface to the lower-level operating system’s networking facilities. Real-time ORBs (such as those described by the Real-Time CORBA specification, CORBA R/T) also use the host platforms priority scheduling support. This real-time example also shows the use of an extended protocol and a QoS interface between the client and servant.

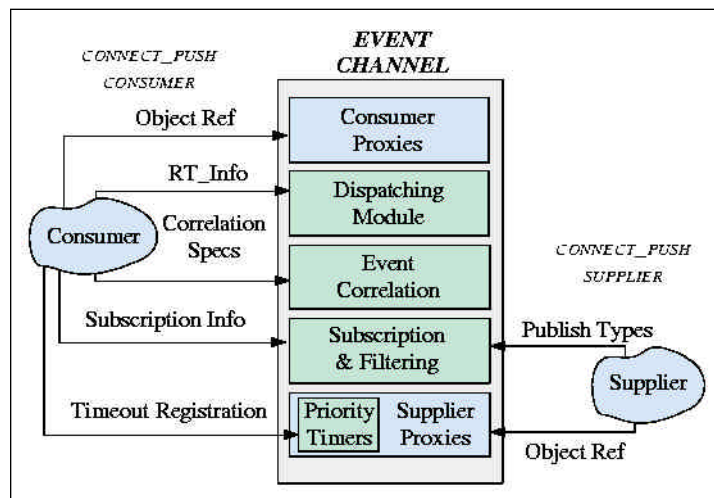


The internal structure of an efficiency, low-latency event distribution service might look like the example in the Figure on the right (taken from Douglas Schmidt's TAO system). This shows the stages of event creation (at the bottom), filtering and allocation, correlation, queueing, and scheduling/delivery.

The event channel object manages (potentially distributed) subscription and filtering, and allocates several priority queues for event delivery.



Another view of an advanced event channel implementation is shown in this Figure; it breaks the software up to show the internals of the actual event channel object and the software interfaces (the labelled arrows) used by the supplier and the consumer objects.



Requirements and Available Solutions

Based on the the examples given above we can create of a list of basic requirements for distributed systems of interest to us. For each of a number of criteria, we present brief summaries of the way they are addressed by basic CORBA (with optional CORBA services), by the extended CORBA A/V and R/T specifications, and by the soon-to-be proposed (see below) ExIDL language and HPDM libraries and run-time system.

Table 1: Requirements and Solutions for HPDM

Requirement	CORBA Core + Services	CORBA A/V or R/T	Java RMI	ExIDL + HPDM Libs. + Run-Time
<u>Language and Object Models</u>				
IDL/XDR	CORBA IDL	CORBA IDL	Java	ExIDL
RPC/RMI	CORBA RMI	CORBA RMI	RMI	ORBs or stand-alone
1-many Events	Optional event service	Optional event service	Java Exceptions, JMS	Event Channels
QoS Model	none	CORBA Messaging Service QoS	none	ExIDL QoS model
I/O Device Models	3rd-party	A/V Obj. Models	3rd-party	HPDM Libraries
OOB Streaming Data	none	A/V Obj. Models	none	HPDM Libraries
<u>Development Libraries & Tools</u>				
IDL Compiler	Included w/ ORB	Included w/ ORB	Java Tools	Several ExIDL bindings and compilers
Configuration management via IDL	none	none	none	HPDM Tools
Streaming libraries	none	A/V stream model	3rd-party	HPDM Libraries
Models of I/O devices	none	A/V device model	3rd-party	HPDM Libraries
<u>Run-time Infrastructure</u>				
Naming Service	Optional service, semi-portable	Optional service	JNDI	HPDM Naming
Trading Service	Optional service, semi-portable	Optional service	none	HPDM Trading
Process monitoring	none, 3rd-party	none	none	HPDM Tools
Notification	Optional, 3rd-party extended notification managers	Optional service	Java Exceptions, JMS	HPDM Notification Manager
Load-balancing	none	none	none	HPDM Tools
QoS negotiation	none	Possible to implement	none	HPDM Libraries

Table 1: Requirements and Solutions for HPDM

Requirement	CORBA Core + Services	CORBA A/V or R/T	Java RMI	ExIDL + HPDM Libs. + Run-Time
Management of OOB streams	none	CORBA A/V	possible in Java	Supported in ExIDL
Object/process migration	none, 3rd-party	none	none, 3rd-party	Supported by HPDM Run-Time
Performance				
Data efficiency (object size ratio over network)	400% typical	400-150%	200% typical	<150% desired (compact formats)
Minimum latency for RMI	2-5 msec typical	2-5 msec typical	1-5 msec typical	< 1msec desired
Scalability to very large systems	good	unknown	poor	hopefully very good scalability
Software Development Complexity	medium	high	medium	low desired complexity

ATON High-Performance Distributed Multimedia

In the HPDM task, we are developing a basic application programmer's interface (API) for use in real-time multimedia software (such as the DRIVE system) that incorporates all of the features discussed above. We are implementing versions of this API on top of low-level UDP/IP socket libraries, one or more CORBA ORBs, and possibly other infrastructures.

We aim to provide a flexible, portable, multi-paradigm development API for high-level programming of real-time multimedia applications. Given this “first-generation” HPDM API and library, we will attempt to address the last three issues of streaming, caching, run-time management, and fault tolerance.

The ATON HPDM API programming libraries will give application programmers a platform- and network-independent tool kit for developing distributed multimedia software. At the simplest level, they will be able to use the XDR and RMI features without having to know the details of the network transport or low-level libraries. The higher-level features such as QoS and automatic replication may not be supported on all platforms and network types, but the provision of a high-level API will at least allow their specification in a flexible and portable manner.

Given the ATON HPDM API and at least two implementations of it, we can carry out detailed performance tests, and determine which language/hardware/network combinations suit themselves for which kind of multimedia applications (e.g., to answer the question “can we run DRIVE over a WAN using CORBA?”).

HPDM Extended IDL

The basic CORBA IDL object model supports inheritance, abstraction, full implementation-hiding, and both strong and weak typing. Modules can contain classes related by composition and inheritance. Class interfaces can have data and function members (AKA instance variables and methods); function members support precise annotation of arguments and return values. CORBA method descriptions can specify the exceptions that can be raised within a method, and this can be used for distributed exception handlers.

It is important to remind the reader that CORBA applications that export IDL interfaces can be reused by clients in different languages and using different ORBs (assuming they share the IIOP protocol). The IDL can be compiled by a client developer into another programming language, using another vendor's ORB and hardware/OS platform.

To support formal interface descriptions that incorporate QoS specifications and run-time context data, we extend the CORBA-style IDL, giving HPDM Extended IDL: **ExIDL**.

The HPDM Scheduler/Trader/Run-time

To run ExIDL programs, we need:

- A basic remote object infrastructure such as a set of CORBA ORBs (one per machine, not necessarily identical), or smaller stand-alone HPDM managers;
- A naming or trading service for locating object references, possibly based on intimate knowledge of the run-time environment and the interface used;
- An event publish/subscribe system that allows many-to-many asynchronous message-passing with efficient event filtering and routing; and
- A run-time task manager for start/stop/monitor/tune operations.

Each of these components will be the subject of further investigation, design, and prototyping as part of the HPDM task.

ATON HPDM Platforms

To allow ExIDL support libraries for non-CORBA platforms, we cannot tie ourselves too strongly to commercial IDL compilers, ORB run-time support, or standard CORBA services such as naming, trading, or event channels. This implies that we specify a minimal protocol and set of "managers" that can run light-weight ExIDL applications without any ORB or significant run-time support. The other extreme would be an all-CORBA system that uses the information in the HPDM extensions to IDL to tune the run-time managers and schedulers.

A Simple Example

To motivate the discussion below, we introduce a simple example of an application that requires an advanced API for distributed programming and flexible Q-S model. For the application domain of realistic surround sound for a multi-user VR system, we require scalable spatial sound mixer/reverberators. These consist of programs running on networked workstations performing the coarse-grained tasks of (a) sound synthesis (or sample playback), (b) source/room/listener geometry mapping, (c) spatial filtering, (d) physical-model-based reverberation, and (e) mixing and output buffering. These tasks may all be running on a sin-

gle computer (for simple worlds and limited processing), or they may each be delegated to a separate processor on a local- or even a wide-area-network.

For the purpose of this document, we'll describe a basic API for task (b) above -- mapping a sound source and its position and the user's virtual position into a structure that encodes the psychoacoustical cues we need to synthesize in the surround-sound spatialization. We will start describing this in extended IDL. The full example is given in Appendix 1. We use a simple "keyword: value" syntax, but this could be replaced by #macro pre-processor directives, "<special-token>" syntax, full XML, or any of several notations.

```
// Sound source placement call -- take (a) sound source (streaming object),
// (b, c) the source and user position/orientation, and (c) the room model and fill-in
// the placement object with spatial processing information.

void place_source(                                // function name and return value
    in sound_generator a,                        // in parameters (can be passed by value):
    in point_6D source_position,                // the streaming sound object
    in point_6D listener_position,             // the source's position and orientation
    in room_geometry room,                     // the user's position and orientation
                                                // the model of the room's reflecting surfaces
    inout sound_source placement,              // inout parameter (passed by reference or copied out):
                                                // the spatialized sound object

                                                // (We may describe the exceptions raised by the call.)

                                                // function description and QoS
    called: EVENT_LIKE,                         // called irregularly
    frequency: USER_IO,                       // called on user (or source) movement
    processing_demands: ON2(room),              // proc. time scales as the square of room complexity
    max_buffer: 40,                            // can be 40 msec or so ahead of real-time
    max_jitter: 10,                            // needs less than 10 msec latency jitter
    late_policy: DROP_UPDATE,                  // drop the call if it's too late
);
```

The expressions in ALL-CAPS are system-defined constants for the range of QoS and run-time policy options, for example, they might be defined as follows.

```
// Call frequency ranges
enum call_frequency {
    PER_SAMPLE,                                // called on every sample
    PER_BUFFER,                                // on every output buffer or frame
    EXCEPTIONAL,                               // only on certain exceptions
    USER_MVMNT,                               // called irregularly on user our source movement
    USER_IO                                    // called on user input or action
};
```

```

// Processing demand scale
enum processing_demands {
    CONSTANT,          // constant processing demand
    ON(x),             // linear in # of arguments
    ON2(x),            // squared in argument complexity
    OExp()             // exponential complexity relative to the argument
};

```

The example begins with a standard function prototype, giving the arguments as pre-defined object types. The added annotation keywords describe the run-time execution context and scaling behavior of the function.

Design Issues

We are extending CORBA IDL for partitioning and QoS management. The added directives in the function prototype above illustrate the domain of knowledge that we want to capture in ExIDL interfaces. The precise syntax is irrelevant at present, and will have to be mapped to C++, Java, Smalltalk, and C over time, probably starting with an open-source IDL compiler such as the MICO system's example.

XDR

The HPDM Project's requirements on the external data representation comprise the normal CORBA IDL features, plus support for event filters, out-of-band streams, and I/O device models.

RMI

The characteristics and requirements for remote method invocation are similar to those provided by a typical CORBA ORB, with the addition of the ability to schedule RMI delivery according to a priority scheme under the control of a manager.

Events

The HPDM Notification Manager publish/subscribe event distribution framework must support very efficient event delivery, distributed event filtering, and event streams.

QoS

The diverse properties of quality-of-service for distributed multimedia applications require a multi-faceted QoS model, support for application extensions and dynamic negotiation, and effective trader integration.

Run-Time System

The ATON/HPDM run-time system and scheduler must allow the start/stop/monitoring of HPDM tasks, dynamic stream rerouting, object and process migration, and fault tolerance.

Traders and Object Services

To run a CORBA A/V application, one needs development libraries and run-time management tools. System management tools need to be integrated with an A/V-aware HPDM trad-

ing service for sophisticated traffic management (including dynamic QoS negotiation), adaptive bandwidth allocation, replication, and load-balancing.

Conclusions

The DiMI ATON Project has many facets that integrate into a single, functioning high-performance distributed sensing and monitoring application. At UCSB, we are carrying out several tasks under ATON Research Thrust 5, the most important of which is to provide a simple, flexible, and powerful development environment for large-scale real-time multimedia software to our colleagues at UCSD.

The HPDM interface description language ExIDL, and the libraries, trader, and run-time tools necessary to support it, will evolve over the rest of 2000 in close coordination with our "users" at UCSD and TCFI.

References

The final reports from the CREATE's two previous (1997) projects in the ATON domain are available on the Web from the following references.

IDOT (Impact of Distributed Object Technology Using CORBA):

<http://www.create.ucsb.edu/idot/report.html>

DRIVE (Distributed Real-Time Interactive Virtual Environments):

<http://www.create.ucsb.edu/drive/phase2/report.html>

CORBA

"OMG CORBA Event Services Specification." OMG document 97-12-11. Available from www.omg.org.

"OMG CORBA Time Service Specification." OMG document 97-12-21. Available from www.omg.org.

"OMG CORBA Telecommunications Domain (Audio/Video Streams) Specification." OMG document 98-07-13. Available from www.omg.org.

"OMG CORBA Wireless Access and Terminal Mobility RFP." OMG document telecom/99-05-05. Available from www.omg.org.

"OMG CORBA Mgmt. of Event Networks RFP." OMG document telecom/98-09-05. Available from www.omg.org.

"OMG CORBA Draft Adopted Real-time CORBA 1.0 Specification." OMG document ptc/99-05-03. Available from www.omg.org.

"An Overview of the Real-time CORBA Specification." By Douglas Schmidt and Fred Kuhns, IEEE Computer, 6/00, also available from <http://www.cs.wustl.edu/~schmidt/corba-research-realtime.html>.

Distributed VE

Frécon, Emmanuel and Märten Stenius. "DIVE: A scaleable network architecture for distributed virtual environments." Available at <http://www.sics.se/~emmanuel/publications/dsej/>. See also <http://www.sics.se/publications/dive.html>

Hagsand, Olof. "Interactive MultiUser VEs in the DIVE System." IEEE Multimedia, 3(1), 1996.

Appendix 1: Extended Example of Extended IDL

This is the complete ExIDL necessary to read the `place_source()` example from the report.

```
module ATON_HPDM {
interface geometry_engine { // The API of the spatial geometry process

    // Processing QoS flags
    enum call_format {
        STREAMING, // continuous streaming API calls
        EVENT_LIKE // event-like API calls
        BACKGROUND // background processing
    };

    // Call frequency ranges
    enum call_frequency {
        PER_SAMPLE, // called on every sample
        PER_BUFFER, // on every output buffer or frame
        EXCEPTIONAL, // only on certain exceptions
        USER_MVMNT, // called irregularly on user our source movement
        USER_IO // called on user input or action
    };

    // Processing demand scale
    enum processing_demands {
        CONSTANT, // constant processing demand
        ON(x), // linear in # of arguments
        ON2(x), // squared in argument complexity
        OExp() // exponential complexity relative to the argument
        ...
    };

    // 6-D point
    struct point_6D { float y, y, z, alpha, delta, theta }

    // Room description
    struct room_geometry {
        sequence walls... // sequence of wall coordinates
    }

    // Delay line taps
    struct delay_line {
        sequence <float> taps; // sequence of tap delays
    };
};
```

```

// 2-speaker spatialization parameters
struct stereo_placement {
    float IATD;                // inter-aural time delay
    float IAAD;                // inter-aural amplitude difference
    delay_line left_taps;      // delay lines for headphone listening
    delay_line right_taps;
    filter left_filter;        // spatial (HRTF-like) filters
    filter right_filter;
}

// Ambisonic spatial model
struct ambisonic_placement {
    float W, X, Y, Z,          // 1st-order coefficients
    float R, S, T, U, V;      // 2nd-order coefficients
}

// Sound_source struct for mapped source cues
struct sound_source {
    sound_generator source;    // the source object
    point_6D relative_location; // the position (6-D) relative to the listener
    floatscaled_amplitude;    // the distance-scaled amplitude
    stereo_placement phones;   // parameters for stereo playback
    ambisonic_placement ambi; // full 2nd-order ambisonic model
}

// Exceptions ...

// Sound source placement call -- this is the interesting part

void place_source(
    in sound_generator a,      // in parameters
    in point_6D source_position,
    in point_6D listener_position,
    in room_geometry room,
    inout sound_source placement, // inout parameter

    // function description and QoS
    called: event_like,       // called irregularly
    frequency: user_movement, // called on user (or source) movement
    processing_demands: ON_room, // proc. demands linear in room complexity
    max_buffer: 40,           // can be 40 msec or so ahead of real-time
    max_jitter: 10,           // needs less than 10 msec latency jitter
);

}; // end of interface
}; // end of module

```

Appendix 2: CORBA A/V Streaming Description

The Object Management Group (OMG) is the software industry standards body that manages the CORBA specification. They have a number of Special Interest Groups (SIGs), one of which tackled the task of describing an object model for streaming audio-visual data. The CORBA A/V model includes a complex set of objects, interfaces, and run-time relationships, and has been deemed “drastic overkill” for the tasks at hand in ATON. Nevertheless, we have studied it in detail, and hope to learn from both its successes and its mistakes.

We present an excerpt from the CORBA A/V streaming specification below for reader reference in the context of HPDM and ExIDL.

CORBA A/V overview

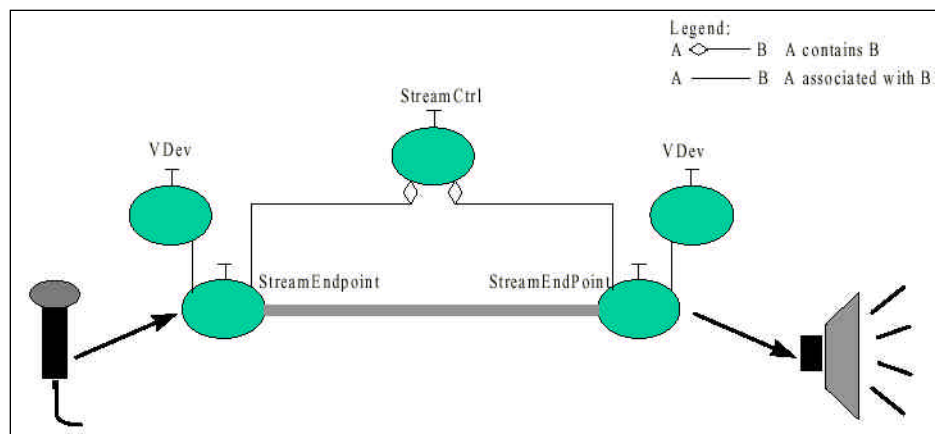
This specification proposes a set of interfaces which implement a distributed media streaming framework. The principal components of the framework are:

- Virtual Multimedia Devices and Multimedia device - represented by the VDev and MMDevice interfaces respectively
- Streams - represented by the StreamCtrl interface
- Stream endpoints - represented by the StreamEndPoint interfaces
- Flows and flow endpoints - represented by FlowConnection and FlowEndPoint interfaces respectively
- Flow Devices - represented by the FDev interface

A stream represents continuous media transfer, usually between two or more virtual multimedia devices. A stream endpoint terminates a stream.

A simple stream between a microphone device (audio source or producer) and speaker device (audio sink or consumer) is shown in the Figure below.

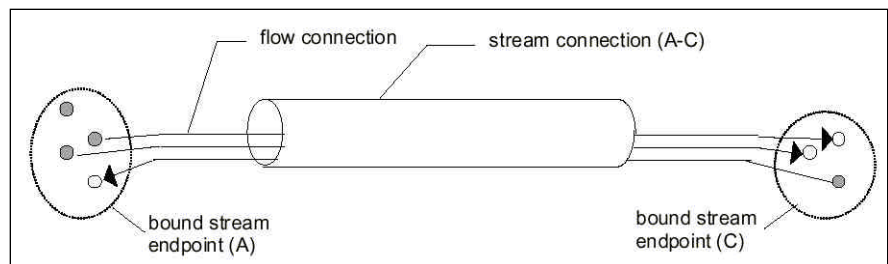
A stream may contain multiple flows. Each flow carries data in one direction so a flow endpoint may be either a source (producer) or a sink (consumer). An operation on a stream (for example, stop or start) may be applied to all flows within the stream simultaneously or just a subset of them.



A stream endpoint may contain multiple flow endpoints. Both flow producer endpoints and flow consumer endpoints may be contained in the same stream endpoint. There may be a

CORBA object representing each flow endpoint and flow connection (i.e., the flow itself), but not all systems are required to expose IDL interfaces to these flow objects. Figure 2-3 illustrates a stream which consists of several different flow connections. Note that not all flow endpoints are involved in the stream (i.e., there may be dangling flow endpoints). Note also that flows can travel in both directions within the same stream. When two stream endpoints which support separate flow endpoints are bound, a compatibility rule can be used to determine which flow endpoints connect to each other.

A multimedia device abstracts one or more items of multimedia hardware and acts as a factory for virtual multimedia devices. A multimedia device can support more than one



stream simultaneously. For example, a microphone device streaming audio to two speaker devices using separate non-multicast connections. For each stream connection requested, the multimedia device creates a stream endpoint and a virtual multimedia device.

The StreamEndPoint interface type has two specializations: 1) StreamEndPoint_A and 2) StreamEndPoint_B. This does not imply that the flows always start at the A party and flow to the B party, indeed both A and B parties may have a mixture of flow producers and flow consumers. Stream endpoints are distinguished in this way in order to help the implementation determine the directionality of their contained flow endpoints. For example, a videophone stream may contain four flows labeled video1, video2, audio1, audio2. When a videophone A party endpoint is created it can automatically set video1 to be a consumer, video2 to be a producer, and so on. Similarly, when the videophone B party endpoint is created it can set video1 to be a producer, video2 to be a consumer, and so on. In this way the videophone A and B parties can act like a 'plug and socket' with all pins and ports facing in the right direction, relieving the application programmer from the chore of manually ensuring that this is the case.

CORBA A/V QoS

The application programmer can use either network level or application-level QoS parameters to set up a connection. The application level QoS will be translated to network level QoS internally. The QoS definition is essentially a named list of properties and their values. IDL for the QoS structures is shown below:

```
// From Property Service
typedef string PropertyName;
struct Property{
    PropertyName property_name;
    any property_value;
};
typedef sequence<Property> Properties;
```

```

struct QoS{
    string QoSType;
    Properties QoSParams;
};
typedef sequence<QoS> streamQoS;

```

The QoSType is a convenience label used to group QoS parameters and identify which flow they pertain to. Most operations for modifying or binding streams take a parameter of type streamQoS. This allows the application programmer to specify QoS on a flow-by-flow basis. For example, consider a stream which has two flows, one called "video" and one called "audio." When establishing a stream of this type the application programmer will typically call bind_devs() with a streamQoS parameter which has two elements (one to specify the QoS for the video and one for the audio). For example:

```

<
{"video_QoS"
  <
    {"video_framerate" 25}
    {"video_colorDepth" 8}
  >
}
{"audio_QoS"
  <
    {"audio_sampleRate" 8000}
    {"audio_numChannels" 2}
  >
}
>

```

There are a couple of things to note about this. First, the QoS for a particular flow is indicated using the name of the flow followed by "_QoS." Second, the QoS parameters shown in this example are application level QoS parameters (i.e., they relate to the performance of the application as opposed to the network QoS needed to transport the flow). A number of these QoS parameters are standardized in this document (see "The A/V Streams Registration Space" on page 2-63). It is up to the A/V Streams implementation to translate the application-level QoS parameters such as "video_framerate" to suitable network level QoS parameters such as "Bandwidth." If the application programmer prefers, s/he may directly specify the StreamQoS above using Network-level QoS parameters instead of application-level QoS parameters. In this case, the above example might become something like the following:

```

<
{"video_QoS"
  <
    {"ServiceType" 0} // Best effort
    {"Bandwidth" 1500000}
    {"Bandwidth_Min" 1250000}
  >
}
>

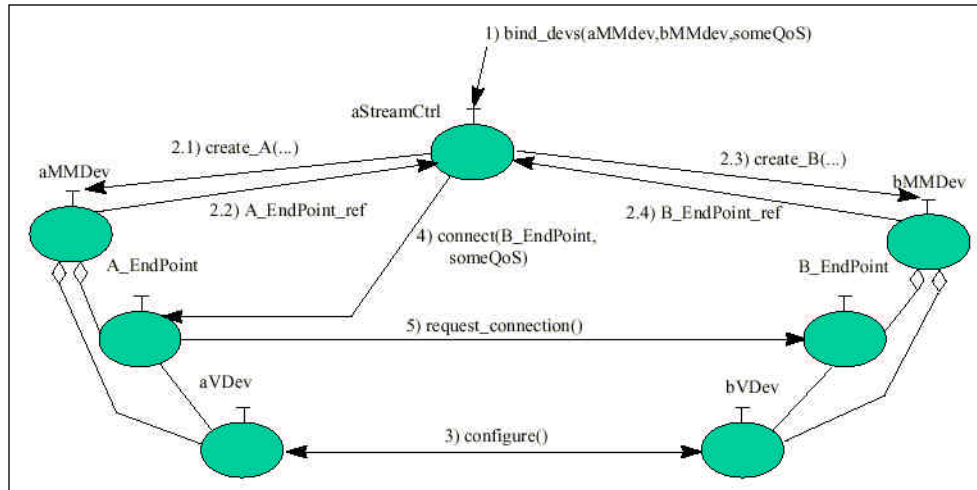
```

```
    {"Delay" 100}
    ...
  >
}
{"audio_QoS"
  <
  {"ServiceType" 1} // Guaranteed
  {"Bandwidth" 8000}
  {"Delay" 100}
  ...
  >
}
>
```

A set of common network-level QoS parameters are also specified in this document. There is a reserved QoSType value "Network_QoS," this is used to indicate a QoS structure which contains only network-level QoS parameters.

Appendix 3: Examples of Point-to-Point CORBA A/V Streams

Setting up a CORBA A/V distributed application involves creating and “wiring together” an array of multimedia objects such as devices, stream and flow end points, and various control objects.



The code fragment below illustrates a simple stream binding in C++.

```
// Declare the local and remote videophone multimedia
// devices
videophone_ptr myPhone = ...;
videophone_ptr johnsPhone = ...;
//Declare the stream controller

videophone_StreamCtrl_ptr myStream;

// Some code here to initialize the local MMDevice (myPhone)
...
// Bind johnsPhone
...
myStream = myPhone->bind(johnsPhone,
    someQoS,
    &wasQoSmet, // Was requested QoS honored
    nilFlowSpec); // Bind all flows
myStream->start(nilFlowSpec);

cout << "Hit any key to hang up..." << endl;
cin >> buf;
myStream->stop(nilFlowSpec)
myStream->destroy(nilFlowSpec);
```

The code fragment below illustrates another way of achieving the same effect.

```
// Declare the local and remote videophone multimedia
// devices
videophone_ptr myPhone = ...;
videophone_ptr johnsPhone = ...;

//Declare the stream controller
videophone_StreamCtrl myStream;

// Some code here to initialize the local MMDevice (myPhone)
...
// Bind johnsPhone
...
// Bind the two devices using a stream with a specified QoS
wasQoSmet = myStream.bind_videophone_devs(
    myPhone,johnsPhone,QoSspec,nilFlowSpec);
myStream.start();

cout << "Hit any key to hang up..." << endl;
cin >> buf;

myStream.stop();
myStream.destroy();
```

One point of interest in the above fragment is that no explicit code for reading from or writing to the stream is shown. This is because each flow has a thread associated with it which loops around reading from the network and writing to the multimedia hardware or vice versa. The application programmer, however, is not compelled to use separately threaded `FlowEndPoint`. You can loop around calling `read()` or `write()` style operations on the `FlowEndPoint`. These operations can be untyped for 'octetstream' flows or typed for non-octetstream flows. Another point to note is the use of the specialized typesafe `bind_videophone_devs()` call instead of `bind_devs()`. The following example shows how `StreamEndpoints` can be used independently of `MMDevices`. On the client side:

```
// Declare local and remote phones
videophone_B_ptr remote_phone;
videophone_A_ptr local_phone = new videophone_A(...);
videophone_StreamCtrl my_stream_controller;

// Bind the remote_phone
...
my_stream_controller.bind(local_phone,
    remote_phone,QoSspec,nilFlowSpec);
```

```

my_stream_controller.start(nilFlowSpec);

cout << "Hit return to hang up! " << endl;
cin >> buf;

my_stream_controller.stop(nilFlowSpec);
my_stream_controller->destroy(nilFlowspec);

```

Using the StreamEndPoint interface directly, a stream can exist independently of a multi-media device. The bind family of calls on the StreamCtrl all work in one of two ways:

1. In the full version of the Media Streaming Framework, the stream compatibility rules are used to determine a viable stream setup. For each matching pair of source/sinks, the StreamCtrl calls `go_to_listen()` on the sink FlowEndPoint and `connect_to_peer()` on the source FlowEndPoint.

2. In the light version of the Media Streaming Framework the stream is set up by calling `A_end-point->connect(B_Adapter,QoS,flowSpec)`. The `connect()` operation basically works by setting up a number of communications channels to the peer StreamEndPoint (remote_phone). This involves:

- Choosing protocols which are supported by the StreamEndPoint_B
- Performing a QoS translation from application level parameters to protocol specific parameters.
- Optionally create 'sockets' and start listening on any flows which terminate in the StreamEndPoint_A.
- Requesting connection of all the flows to StreamEndPoint_B, pass transport addresses of any sockets that are being listened on.
- The StreamEndPoint_B sets up 'sockets' which listen on the appropriate transport addresses and returns these addresses. At this point, the StreamEndPoint_B implementation may choose to connect to any listening sockets on StreamEndPoint_A.
- The StreamEndPoint_A may then choose to connect its remaining unconnected flows to addresses of listening 'sockets' in StreamEndPoint_B and returns.

Appendix 4: CORBA A/V Device and Stream Parameters

Stream establishment and management is subject to a large number of potential parameters for Quality of Service and other attributes. In order to ensure compatibility between different implementations, a standard set of parameters and their values need to be defined. For generic network-level QoS the following parameters are currently registered (see CORBA RFC. "The A/V Streams Registration Space" page 2-63):

Network QoS, parameter set 1

- ServiceType - Best Effort, Guaranteed, Predicted
- ErrorFree - True or False
- Delay - long value
- Delay_Max - long value
- Bandwidth - long value
- Bandwidth_Min - long value
- PeakBandwidth - long value
- PeakBandwidth_Min - long value
- TokenRate - long value
- TokenRate_Min - long value
- TokenBucketSize - long value
- TokenBucketSize_Min - long value
- Jitter - float value
- Jitter_Max - float value
- Cost - float value
- Cost_Max - float value
- Protection - short value, 0= default, no encryption, 1= encryption level 1 This parameter set makes no assumptions about the semantics of policing and shaping policies. This is beyond the scope of this specification.

Network QoS, parameter set 2

The following optional Network QoS parameter set is defined as an alternative to parameter set 1:

- Duplication - enum dup {IGNORE, DELETE}
- Damage - enum dam {DAM_IGNORE, DAM_NOTIFY, DAM_DELETE, DAM_CORRECT}
- Damage_method - Type to be specified
- Reorder - enum reord {REORDER_CORRECT, REORDER_IGNORE}

- Loss - long, {-1 = LOSS_IGNORE, -2 = LOSS_NOTIFY}, positive integer denotes number of retry attempts before the receiver is presumed dead
- Size_Min - long, min bytes in a data unit
- Size_Max - long, max bytes in a data unit
- Size_avg - long, average number of bytes in a data unit
- Size_avg_span - long, number of subsequent data units sent with an interval (ival_const)
- Ival_Const - long, {-2 = IVAL_MAX, -1 = IVAL_ANY}, positive integer denotes constant time interval between transport requests
- Ival_Max - long, the maximum acceptable value if Ival_Const cannot be met
- Delay - long, {0 = DELAY_VOID, -1 = DELAY_ANY, -2 = DELAY_MIN}, DELAY_MIN denotes best effort with minimal delay, DELAY_ANY = best effort, low cost. Positive integer denotes delay required.
 - Delay_Max - long, indicates maximum acceptable delay
 - Delay_Cum - long, indicates acceptable cumulative delay for concatenated stream
 - Jitter - long
 - Jitter_Max - long
 - ErrDamRatio - float, Ratio of damaged data units {0.0 = RATIO_DAM_VOID, -1.0 = RATIO_DAM_ANY, -2.0 = RATIO_DAM_MIN}, where RATIO_DAM_MIN is a request for minimal error ratio, RATIO_DAM_ANY is request for less costly ratio. A number between 0-1.0 indicates the desired ratio.
 - ErrDamRatio_Max - float, maximum acceptable error ratio for damaged data units
 - ErrLossRatio - float, ratio of lost data units { 0.0 = RATIO_LOSS_VOID, -1.0 = RATIO_LOSS_ANY, -2.0 = RATIO_LOSS_MIN}, where RATIO_LOSS_MIN is a request for minimal error ratio, RATIO_LOSS_ANY is request for less costly ratio. A number between 0-1.0 indicates the desired ratio.
 - ErrLossRatio_Max - float, maximum acceptable ratio of lost data units
 - Workahead_Mode - enum {AHEAD_BLOCKING, AHEAD_NONBLOCKING}
 - Workahead_Max - long, the maximum number of data units the producer may be ahead of the consumer
 - Playback_Mode - Type to be specified, indicates playback strategy
 - Playback_Max - long, The maximum delay introduced by the producer to counter jitter effects

It is mandatory for an implementation of streams to support Network QoS parameter set 1. It is optional to support Network QoS parameter set 2. Not all of the Network QoS parameters need to be used to describe network level QoS. For example, in a toll-free environment where only best-effort limits are used the "Network QoS" QoS structure could use only the properties: ServiceType, ErrorFree, Delay, Bandwidth, and Jitter.

Devices and streams themselves have a number of properties associated with them. These properties can be read by using the Object Property Service interfaces from which StreamCtrl, MMDevice, StreamEndPoint, FlowEndPoint, FlowConnection, and VDev are derived.

StreamCtrl

The suggested properties for a StreamCtrl are:

- Type - string (empty string implies generic stream type)
- Status - seqflowStatus
- QoS - streamQoS
- Network_QoS - streamQoS
- Flows - sequence of strings, current flows (named from A side)
- A_parties - sequence of StreamEndPoint_A
- B_parties - sequence of StreamEndPoint_B
- flowConnections - sequence of FlowConnection

MMDevice

Some suggested properties for MMDevices are:

- Flows - sequence of flow names supported
 - FlowNameX_dir - string, directionality indicators
 - FlowNameX_availableFormats - sequence of <format_name> strings
 - FlowNameX_SFPStatus - sfp status structure
 - FlowNameX_PublicKey - sequence of octets
 - MaxStreams - long, maximum of streams supported
 - CurrentLoad - float (a percentage) indicates load on physical device
- The flowNameX_availableFormats property lists all the possible coders/decoders supported by the device which is associated with that flow. This is designated using a

<format_name> which takes the following form:

<format_name> ::= <format_category> [":" <fname>]

<format_category> ::= "MIME" | "IDL" | "UNS"

The MIME format category is managed by the IETF [7] and is known there as Media Type. It is referred to here by its older, more familiar name of MIME Content-Type. Valid values for <fname> when the <format_category> is MIME are listed in the registration section (see “The A/V Streams Registration Space” on page 2-63). The <format_category> IDL is used to describe IDL-typed flows. The <fname> will be the full IDL of the flow element. The <format_category> "UNS" indicates unspecified and is not followed by <fname> information. Further <format_category> values can be registered with the OMG (see “The A/V

Streams Registration Space” on page 2-63). The property "FlowNameX_dir" states the possible directions for this flow supported by the device (i.e., in, out, or inout).

FDev

The properties for the FDev interface are very similar to those for MMDevice:

- Flow - string, name of flow supported
- Dir - string, indicates directionality
- AvailableFormats - sequence <format_name> strings
- SFPStatus - SFP status structure
- PublicKey - sequence of octets
- MaxFlows - long, Maximum number of flows supported
- CurrentLoad - float (a percentage) indicates load on physical device Properties are especially important on the VDev interface. This is because during the configuration phase the devices may need to query each other's current settings.

VDev

The properties supported by VDev are:

- Related_StreamEndPoint
- Related_MMDevice
- Flows - sequence of string
- FlowNameX_dir - string, directionality indicators
- FlowNameX_availableFormats - sequence of <format> strings
- FlowNameX_currFormat - <format> string
- FlowNameX_devParams - Properties
- FlowNameX_SFPStatus - sfp status structure (only where SFP is in use)
- FlowNameX_status - flow status structures
- FlowNameX_related_mediaCtrl - Object
- FlowNameX_PublicKey - sequence of octets

The flowNameX_availableFormats property states which formats this VDev can support (e.g., MPEG, MJPEG, etc.). The flowNameX_currFormat property states which of these is currently in use. The flowNameX_related_mediaCtrl property holds a reference to a media controller for a flow. These media controllers can be used to implement functionality like rewind and fast forward and can support any interface at all. “

The flowNameX_devParams property states the settings associated with the current codec or device. This document describes the following common device parameters for audio, video and other devices.

- language - string, from the set {...,"English(UK)","English(US)",...,"Irish",...}
- audio_sampleSize - short, number of bits per sample
- audio_sampleRate - long, Hertz
- audio_numChannels - short
- audio_quantization - short, 0 = linear, 1 = u-law, 2 = A-law, 3 = GSM
- video_framerate - long
- video_colorDepth - short (e.g., 2, 4, 8, 12, 16, 24, 32)
- video_colorModel - short 0 = RGB, 1 = CMY, 2 = HSV, 3 = YIQ, 4 = HLS
- video_resolution - struct resolution

StreamEndPoint

The properties exposed by a StreamEndPoint are:

- Related_VDev
- Related_StreamCtrl
- Negotiator - A negotiator object ref.
- Flows - sequence of flow names supported
- FlowNameX_dir - string, directionality indicators
- FlowNameX_currFormat - string, <format> string
- FlowNameX_address - string indicates protocol and address
- FlowNameX_status - stopped, started, destroyed
- FlowNameX_flowProtocol - string (<flowProtocol>)
- FlowNameX_PublicKey - sequence octet
- AvailableProtocols - sequence of string (protocol names)
- ProtocolRestriction - sequence of string (protocol names)
- PeerAdapter - StreamEndPoint reference

The AvailableProtocols property states what protocols are available to this StreamEndPoint. The ProtocolRestriction property lists the restriction currently placed on what protocols may be used for the purposes of connecting to another StreamEndPoint.

The flowNameX_address property is formatted according to <transport_address> syntax described later in this document. For example, it could typically have the format "TCP=cod.fish.net:2222". The directionality is expressed from the A-side, so if the direction of a flow is "in" on a B end-point that means that the flow is originating on the B side and terminating on the A side.

FlowEndPoint

The properties exposed by a flow endpoint are:

- FlowName - string
- Format - sequence of <format> string
- CurrFormat - <format> string
- DevParams - property list, describes
- Status - Stopped/started
- FlowProtocol - string (<flow protocol>)
- Active - Boolean
- Dir - enum sink/source (State actual type name)
- flowProtocol - string, flow protocol name in <flowProtocol> syntax (e.g., "SFP1.0")
- SFPStatus - SFP status structure
- Related_mediaCtrl - Object, the related media controller
- Address - Formatted string
- AvailableProtocols - protocol spec
- CurrProtocol - <protocolname>