Audio/Video Stream Specification

New Edition: January 2000
(no text changes since June 1998)

Copyright 2000, IONA Technologies, Plc
Copyright 2000, Lucent Technologies, Inc.
Copyright 2000, Siemens-Nixdorf, AG

The companies listed above have granted to the Object Management Group, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version. Each of the copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

PATENT

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

NOTICE

The information contained in this document is subject to change without notice. The material in this document details an Object Management Group specification in accordance with the license and notices set forth on this page. This document does not represent a commitment to implement any portion of this specification in any company's products.

WHILE THE INFORMATION IN THIS PUBLICATION IS BELIEVED TO BE ACCURATE, THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR PARTICULAR PURPOSE OR USE. In no event shall The Object Management Group or any of the companies listed above be liable for errors contained herein or for indirect, incidental, special, consequential, reliance or cover damages, including loss of profits, revenue, data or use, incurred by any user or any third party. The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

RESTRICTED RIGHTS LEGEND. Use, duplication, or disclosure by government is subject to restrictions as set forth in subdivision (c) (1) (ii) of the Right in Technical Data and Computer Software Clause at DFARS 252.227.7013 OMG®and Object Management are registered trademarks of the Object Management Group, Inc. Object Request Broker, OMG IDL, ORB, CORBA, CORBAfacilities, CORBAservices, and COSS are trademarks of the Object Management Group, Inc. X/Open is a trademark of X/Open Company Ltd.

ISSUE REPORTING

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the issue reporting form at *http://www.omg.org/library/issuerpt.htm.*

Contents

1. Overview 1-1

1.1 About the Object Management Group 1-1

1.1.1 What is CORBA? 1-1

1.2 Associated Documents 1-2

1.3 Acknowledgments 1-2

2. Control and Management of Audio/Video Streams 2-1

2.1 Introduction 2-1

2.2 Architecture Overview 2-3

2.2.1 Principal Components 2-3

2.2.2 (Virtual) Multimedia Device Interface 2-5

2.2.3 StreamCtrl Interface 2-6

2.2.4 StreamEndPoint Interface 2-7

2.2.5 Flow Endpoints, Flow Connections, and FDevs 2-9

2.2.6 Properties of Streams 2-12

2.2.7 Quality of Service 2-12

2.2.8 Stream Specification 2-14

2.2.9 Flow Protocol 2-16

2.2.10 Examples for point-to-point streams 2-20

2.2.11 Issues in Modifying QoS 2-23

2.2.12 Issues in Multipoint Streams 2-23

2.2.13 Extending Stream Management Functionality . 2-25

2.2.14 Device and Stream Parameters 2-25

2.3 IDL Interfaces 2-31

2.3.1 The Basic_StreamCtrl 2-31

2.3.2 The StreamCtrl 2-33

Contents

2.3.3 The StreamEndpoint 2-36
2.3.4 The StreamEndPoint_A and StreamEndPoint_B 2-43
2.3.5 The MMDevice 2-44
2.3.6 The VDev 2-47
2.3.7 The FlowEndPoint 2-48
2.3.8 The FlowConnection 2-52
2.3.9 FDev 2-54

2.4 Conformance Criteria 2-55
2.4.1 Light vs Full Profile 2-55
2.4.2 Flow Protocol 2-56
2.4.3 Network QoS Parameters 2-56

Appendix A - Complete OMG IDL A-1

Appendix B - Requirements for Control and Management of A/V Streams B-1

Appendix C - Relationship to DAVIC Work C-1

Appendix D - References D-1

Overview 1

1.1 About the Object Management Group

The Object Management Group, Inc. (OMG) is an international organization supported by over 800 members, including information system vendors, software developers and users. Founded in 1989, the OMG promotes the theory and practice of object-oriented technology in software development. The organization's charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. Primary goals are the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. Conformance to these specifications will make it possible to develop a heterogeneous applications environment across all major hardware platforms and operating systems.

OMG's objectives are to foster the growth of object technology and influence its direction by establishing the Object Management Architecture (OMA). The OMA provides the conceptual infrastructure upon which all OMG specifications are based.

1.1.1 What is CORBA?

The Common Object Request Broker Architecture (CORBA), is the Object Management Group's answer to the need for interoperability among the rapidly proliferating number of hardware and software products available today. Simply stated, CORBA allows applications to communicate with one another no matter where they are located or who has designed them. CORBA 1.1 was introduced in 1991 by Object Management Group (OMG) and defined the Interface Definition Language (IDL) and the Application Programming Interfaces (API) that enable client/server object interaction within a specific implementation of an Object Request Broker (ORB). CORBA 2.0, adopted in December of 1994, defines true interoperability by specifying how ORBs from different vendors can interoperate.

1.2 Associated Documents

The CORBA documentation set includes the following books:

- *Object Management Architecture Guide* defines the OMG's technical objectives and terminology and describes the conceptual models upon which OMG standards are based. It defines the umbrella architecture for the OMG standards. It also provides information about the policies and procedures of OMG, such as how standards are proposed, evaluated, and accepted.

- *CORBA: Common Object Request Broker Architecture and Specification* contains the architecture and specifications for the Object Request Broker.

- *CORBAservices: Common Object Services Specification* contains specifications for OMG's Object Services.

- *CORBAfacilities: Common Facilities Architecture and Specification* describes an architecture for Common Facilities. Additionally, it includes specifications based on this architecture that have been adopted and published by the OMG.

The OMG collects information for each book in the documentation set by issuing Requests for Information, Requests for Proposals, and Requests for Comment and, with its membership, evaluating the responses. Specifications are adopted as standards only when representatives of the OMG membership accept them as such by vote. (The policies and procedures of the OMG are described in detail in the *Object Management Architecture Guide*.)

OMG formal documents are available from our web site in PostScript and PDF format. To obtain print-on-demand books in the documentation set or other OMG publications, contact the Object Management Group, Inc. at:

OMG Headquarters
492 Old Connecticut Path
Framingham, MA 01701
USA
Tel: +1-508-820 4300
Fax: +1-508-820 4303
pubs@omg.org
http://www.omg.org

1.3 Acknowledgments

This specification represents the hard work and contribution of many individuals and companies. We would like to acknowledge the following for their contributions, both large and small:

- Fore Systems, Inc.
- IONA Technologies, Plc.

- Lucent Technologies, Inc.
- Siemens-Nixdorf, AG

Control and Management of Audio/Video Streams 2

The OMG document used to create this formal document was telecom/97-05-07 (and its errata).

Contents

This chapter contains the following sections.

Section Title	Page
"Introduction"	2-1
"Architecture Overview"	2-3
"IDL Interfaces"	2-31
"Conformance Criteria"	2-55

2.1 Introduction

A stream is a set of flows of data between objects, where a flow is a continuous sequence of frames in a clearly identified direction. A stream interface is an aggregation of one or more source and sink flow endpoints associated with an object. Although any type of data could flow between objects, this specification focuses on applications dealing with audio and video exchange with Quality of Service (QoS) constraints.

The Control and Management of Audio/Video Streams specification addresses the following issues: Topologies for streams, multiple flows, stream description and typing, stream interface identification and reference, stream set-up and release, stream modification, stream termination, multiple protocols, Quality of Service (QoS), flow synchronization, interoperability, and security. This specification addresses the first 11 issues and provides hooks for solutions to the last issue (security).

Note – The term "Stream Control" is used throughout this specification as an abbreviation for the name of the specification.

The specification uses an architecture based upon Figure 2-1.

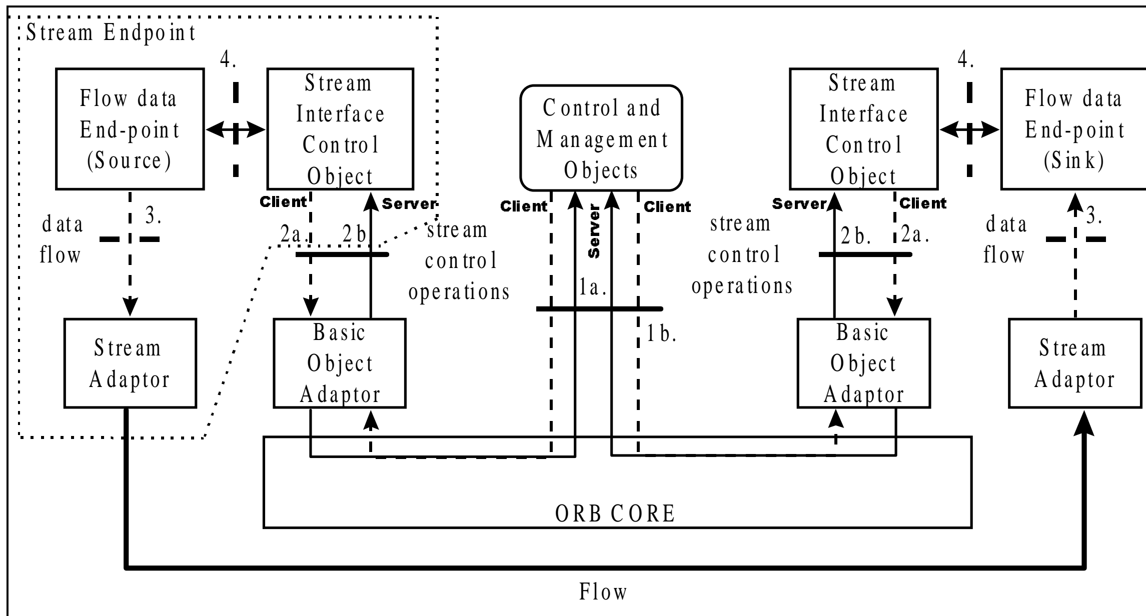

Figure 2-1 Example stream architecture

Figure 2-1 shows a stream with a single flow between two stream end-points, one acting as the source of the data and the other the sink. Each stream end-point, shown as a dotted encapsulation in Figure 2-1, consists of three logical entities:

1. A stream interface control object that provides IDL defined interfaces (as server, 2b) for controlling and managing the stream (as well as potentially, outside the scope of this specification, invoking operations as client, 2a, on other server objects).
2. A flow data source or sink object (at least one per stream endpoint) that is the final destination of the data flow (3).
3. A stream adaptor that transmits and receives frames over a network.

The Stream interface control object is shown in Figure 2-1 as using a basic object adaptor that transmits and receives control messages in a CORBA-compliant way.

Note – This specification does not require any changes to the BOA/POA or IDL language bindings to accommodate the control of A/V streams.

When a stream is terminated in hardware, the source/sink object and the stream adaptor may not be visible as distinct entities. Please note that how the stream interface control object communicates with the source/sink object and perhaps indirectly with the stream adaptor (interface 4 in Figure 2-1) and how the source/sink object communicates with the stream adaptor (interface 3 in Figure 2-1) are outside the scope of this specification.

This specification provides definitions of the components that make up a stream and for interface definitions onto Stream control and management objects (interface number 1a.), and for interface definitions onto stream interface control objects (interface number 2b.) associated with individual stream endpoints.

In particular, CORBA interface references for stream interface control objects are used to refer to all stream endpoints in parameters to stream control operations defined in this specification. Thus, this specification does not require a new IDL data type for stream interface reference.

2.2 Architecture Overview

2.2.1 Principal Components

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 Figure 2-2 on page 2-4.

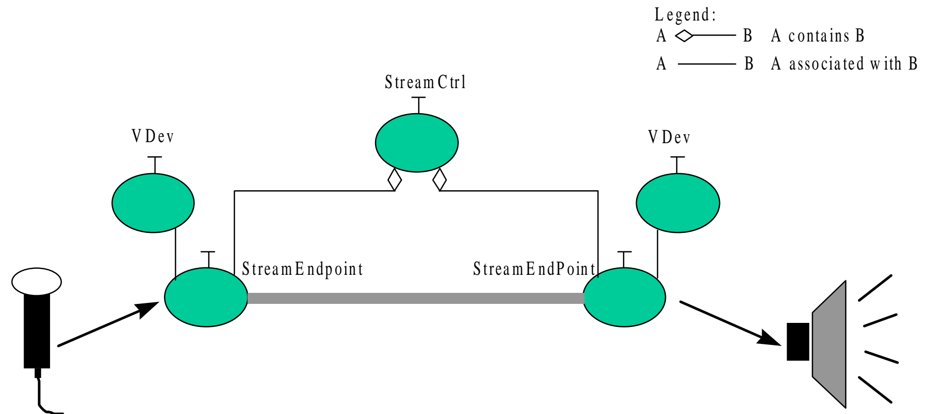

Figure 2-2 A basic stream configuration

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.

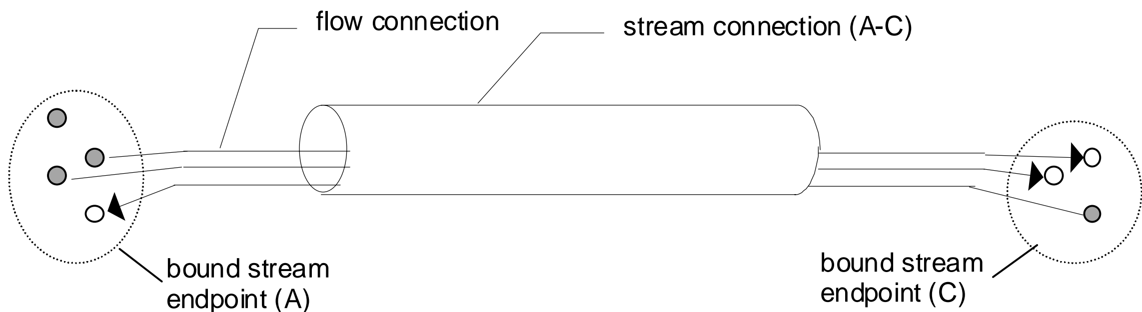

Figure 2-3 Stream connection compatibility rules can allow unconnected flow endpoints

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.

Note – Throughout this document the verb 'binding' when applied to a stream, flow, or set of multimedia devices is used as meaning stream or flow establishment.

When 'binding' is used as a noun, it refers to a stream or flow which is established (i.e., has active transport connections).

The sections below discuss each of the main IDL interfaces in more detail. There are two basic profiles for the streaming service:

- The full profile in which flows endpoints and flow connections have accessible IDL interfaces. This profile is optimized for flexibility.
- The light profile in which flows endpoints and flow connections do not expose IDL interfaces. The light profile is a subset of the full profile. It is optimized for systems which need to minimize memory footprint and the number of CORBA invocations.

2.2.2 (Virtual) Multimedia Device Interface

A multimedia device abstracts one or more items of multimedia hardware and is described by the IDL interface **MMDevice**. A multimedia device can be connected or 'bound' to one or more compatible multimedia devices using a stream. A multimedia device potentially can support any number of streams to other multimedia devices. Each stream connection will be supported by creation of a virtual multimedia device (**VDev**) and a stream endpoint (**StreamEndPoint**) representing the device specific and network specific aspects of a stream endpoint respectively. Virtual devices will have configuration parameters associated with them. For example, a microphone device might be capable of encoding audio data using either μ-law or A-law. The sampling frequency might also vary. When two virtual devices are connected using a stream they

must ensure that they are both appropriately configured. For example, if the microphone device is using A-law encoding and a sampling frequency of 8-kHz, then it must ensure that the speaker device it is connected to is configured similarly. It does this by calling configuration operations such as:

- **set_format("audio1","MIME:audio/basic")**, and
- **set_dev_params("audio1",myaudioProperties)**

on the **VDev** interface for the speaker device. This procedure occurs when a **StreamCtrl** is establishing a stream between two virtual devices. Changes in configuration can also occur while a stream is flowing between two virtual devices. There are some important points to note about this procedure. The **VDev** negotiation procedure is initiated in a point-to-point scenario by the streamCtrl first calling **set_peer()** on the 'A-party' **VDev** with the 'B-party' **VDev** as a parameter and then calling **set_peer()** on the 'B-party' **VDev** with the 'A-party' **VDev** as a parameter. It will generally be the case that the programmer who implements the **VDev** (either an application programmer or a multimedia device vendor) will be responsible for writing the behavior of the **set_peer()** call. This behavior should normally consist of checking that all flows which originate in this **VDev** can be understood by the corresponding flow consumers in the peer **VDev**. The operations **set_format()** and **set_dev_params()** have been expressly included for this purpose. A variation on this procedure is also used for multicast streams and is explained in "The StreamCtrl" on page 2-33.

An **MMDevice** supports operations to set up a stream between two or more peer MMDevices on the network. For example, to bind two multimedia devices using a stream the application programmer could call the following operation on the **MMDevice** interface:

```
StreamCtrl bind(in MMDevice peer_device, inout streamQoS the_qos,
    out boolean is_met, flowSpec the_spec);
```

The parameter peer_device specifies the device which should be bound to this device using the stream. The the_spec parameter is a list which specifies the flows within the stream which should be initially connected. If the list is empty, then all flows are connected. This operation will return a reference to the **StreamCtrl** which the application programmer can then use to stop, start, or otherwise manipulate the stream. This **StreamCtrl** will be co-located with the **MMDevice** on which the bind call is made; therefore, it can be considered a form of first-party stream establishment.

2.2.3 StreamCtrl Interface

The **StreamCtrl** interface abstracts a continuous media transfer between virtual devices. It supports operations to bind multimedia devices using a stream. There are also operations to start and stop a stream. If the application programmer requires more complex functionality (for example, rewind) s/he can extend the basic stream interface to support this.

When the application programmer requests a stream between two or more multimedia devices s/he can explicitly specify the Quality of Service (QoS) of the stream. QoS can mean different things at different levels. For example, a multimedia device which supports video will be concerned with QoS parameters like frame-rate and color depth. This type of QoS will be referred to as application-level QoS. A stream, however, will be supported by underlying network protocols. The QoS for a network protocol may include parameters like minimum bandwidth and jitter. This type of QoS will be referred to as network-level QoS. An application programmer can establish a stream between two devices by calling the operation **bind_devs()** on the **StreamCtrl** interface:

```
boolean bind_devs(in MMDevice a_party, in MMDevice b_party,
    inout streamQoS the_qos, flowSpec the_spec);
```

The **StreamCtrl** object is generally instantiated by the application programmer (see “Examples for point-to-point streams” on page 2-20). The **bind()** operation on the **MMDevice** interface is simply a shorthand for creating a **StreamCtrl** and calling **bind_devs()** on it. The argument **the_qos** will generally be an application-level QoS specification. The appropriate protocols will be chosen by the **Media Streaming Framework** and a translation will occur between application level QoS and network level QoS for these protocols. If the **MMDevices** decide they can accept the connection, they will each create a **StreamEndPoint** and a **VDev** to support the stream between them.

The **Media Streaming Framework** supports point-to-multipoint streams. These allow a single ‘source’ device to be connected to many sink devices via a multicast stream. For example, the application programmer could do something like the following to set up a video broadcast:

```
MyStream->bind_devs(cameraDev,nilObject,someQoS,nilFlowSpec)
```

Using a reference to a nil Object as the b_party parameter has the effect of adding a camera device as a multicast A party. The application programmer can then add an arbitrary number of TVs to the broadcast by calling:

```
MyStream->bind_devs(nilObject,TVDev,someQoS,nilFlowSpec)
```

Note that in the point-to-point case, an A party may contain flow sinks and the B party the corresponding flow sources, but in the multicast case only flow sources may be present in an A-party and only flow sinks in a B-party. The A and B stream endpoints are therefore symmetrical except for the multicast case.

2.2.4 StreamEndPoint Interface

A stream endpoint logically contains and controls the flow endpoints for each of the individual flows in a stream. There are two flavors of stream endpoint: 1) an ‘A-party’ and 2) a ‘B-party’ (represented by the interfaces **StreamEndPoint_A** and **StreamEndPoint_B**). An A party can contain producer flow endpoints as well as consumer flow endpoints, similarly with a B party. The primary reason for making a distinction is so that when an instance of a typed **StreamEndPoint** is created the

flows will be plumbed in the right direction (i.e., an audio consumer **FlowEndPoint** in a **StreamEndPoint_A** will correspond to an audio producer **FlowEndPoint** in a **StreamEndPoint_B**. The choice of which end-point is the A party and which is the B party is entirely arbitrary. A call to connect() which will establish or 'bind' the stream can be made on either a **StreamEndPoint_A** or **StreamEndPoint_B**.

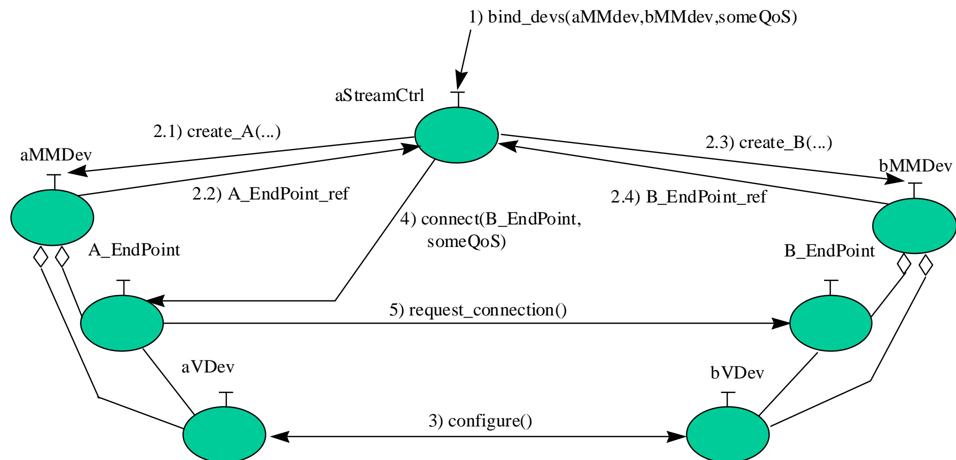

Figure 2-4 Establishing a stream (simplified)

Figure 2-4 illustrates what happens when the **bind_devs()** operation is called on a **StreamCtrl** object.

Step 1: A **StreamCtrl** object may be created by the application programmer to initiate a stream between two multimedia devices. To establish a stream between aMMDev and bMMDev, a call is made to **bind_devs()** on the **StreamCtrl** interface.

Step 2: The StreamCtrl asks aMMDev and bMMDev for a **StreamEndPoint** and **VDev** to support the stream by calling **create_A**(…,someQoS,…) on one and **create_B**(…,someQoS,…) on the other. The former call, for example, will return a **StreamEndPoint_A** object which is associated with **aVDev**. This is the point at which an **MMDevice** could decide that it can support no more connections and refuse to create the **StreamEndPoint** and **VDev**. A particular **MMDevice** might be specialized to only create A-end-points for a type of stream (for example, a microphone end-point and **VDev**) or only B-end-points (for example, a speaker end-point and **VDev**).

Step 3: This step involves the **aVDev** calling configuration operations on the **bVDev** and vice versa. This is the device configuration phase already discussed in "(Virtual) Multimedia Device Interface" on page 2-5.

Step 4: The actual stream is set up by calling **connect()** on the **A_EndPoint** with the **B_EndPoint** as a parameter. Stream end-points contain a number of flow end-point objects. A flow end-point object can be used to either pull information from a multimedia device and send it over the network or vice versa. In the light profile the **FlowEndPoint** objects are co-located with the StreamEndPoint objects and do not expose IDL interfaces.

Step 5: The **A_EndPoint** may choose to listen on transport addresses for the flows which terminate on the A side. It will then call **request_connection()**. Among the information passed across will be the transport addresses of the listening flows on the A side. The **B_EndPoint** will connect to the listening flows on the A-side and will listen on transport addresses for flows which terminate on the B-side. Among the information passed back by the **request_connection()** operation will be the transport addresses of the flows listening on the B side. The final stage is for the **A_EndPoint** to connect to the listening flows on the **B_EndPoint**.

So far, streams have always been shown between multimedia devices. It is possible that a programmer will want to stream data between objects in a distributed application which have nothing to do with multimedia devices. As an example, streaming updates of the positions of players in a distributed multi-player game. Where there is no analog of a multimedia device involved, asking the application programmer to specify a dummy multimedia device class is inappropriate and an unnecessary overhead. The framework makes it possible to have a stream existing independently of any multimedia devices. The stream will be established directly between two **StreamEndPoints** which have been created by the application programmer. This can be done by making the following call on the **StreamCtrl** interface:

```
boolean bind(in StreamEndPoint_A A_party,
             in StreamEndPoint_B B_party,
             inout streamQoS theQoS,
             in flowSpec theFlows)
```

The **bind()** call can also be used to set up multipoint streams in exactly the same way as a **bind_devs()** call.

2.2.5 Flow Endpoints, Flow Connections, and FDevs

Heretofore, this document has described connection setup using the light profile of the Audio/Video Streams specification. The full profile provides IDL interfaces for each individual flow (**FlowConnection**) and its endpoints (**FlowEndPoint**) allowing a greater degree of versatility in connection setup. In systems which support the full as opposed to the light streams profile, individual flow endpoints and flow connections (which connect flow endpoints) are accessible. This gives greater granularity of control over stream establishment and manipulation. Indeed, a stream can be established by simply setting up a set of **FlowConnections** and then grouping them under a **StreamCtrl.**

A flow endpoint supports an operation **is_fep_compatible()**. This allows a third party (usually a **FlowConnection**) to check whether two flow endpoints are connectable. A flow endpoint may support a selection of protocols and a selection of encoding/decoding formats. Two flow endpoints are compatible if they share a common protocol over which they can transport media, and if they share a common format for media encoding/decoding (e.g., MPEG). The **FDev** is exactly analogous to the **MMDevice** for streams. A **FlowConnection** can be used to bind **FDevs** in exactly the same manner as a **StreamCtrl** binds **MMDevices**. Whereas an

MMDevice creates a **StreamEndPoint** and a **VDev**, an **FDev** just creates a **FlowEndPoint**. The **FlowEndPoint** combines the functionality of a **VDev** and a **StreamEndPoint** in a single interface but only applies to a single flow.

The algorithm for connecting a set of flow endpoints can then be specialized for supporting a particular type of stream connection. For example, in Figure 2-3 on page 2-4 the top dangling flow endpoint in stream endpoint A might be a French language audio producer, while the next lowest flow endpoint might be the English language audio producer, and the other two might be video provider and talk-back audio consumer flow endpoints. Stream endpoint A is compatible with stream endpoint C, even though stream endpoint C has no French audio consumer. In a given request for a stream connection (or connection modification), a subset of flow endpoints may be selected.

In general, the compatibility rules for connecting two stream endpoints (A and C) are as follows:

- The stream endpoint types for both A and C must specify which flow endpoint types are included, along with the causality (producer or consumer) of each flow endpoint.
- When stream connection setup is requested, the client may select a subset of the flow endpoints to be included in the stream connection for a specified stream endpoint. For each flow endpoint selected in each stream endpoint (default is to try to connect all flow endpoints), the connection provided must determine if there is a compatible flow endpoint in the other stream endpoint using the flow compatibility rules outlined above.
- The two stream endpoints are compatible if they both contain at least one flow endpoint which is compatible with at least one flow endpoint in the other stream endpoint. In the case of one-to-many flow endpoint compatibility choices, a best fit algorithm could be used to select one-to-one associations of flow endpoints in the two stream endpoints. Flow endpoints in each stream endpoint, which have no compatible flow endpoints in the other stream endpoints are called dangling flow endpoints, and cannot participate in a stream connection involving those two stream endpoints.
- In order to set-up a stream connection there must be at least one pair of compatible flow endpoints in the connected stream endpoints. If this is not so, the stream connection will fail.
- A stream connection may be modified by requesting additional flow endpoints (included in the stream endpoint type) be added or dropped from an existing connection. In such cases, the stream controller attempts to connect the newly selected flow endpoints with an associated flow endpoint on the other side of the stream connection.

If a Media Streams implementation supports the full profile of operation, then making a bind call on **StreamCtrl** leads to the execution of the certain behavior for establishing whether the stream setup is possible. First consider the scenario where the **StreamCtrl** has been implemented by a vendor who has supplied the full profile, the **StreamEndPoint_A** is running on a platform which is supplied with a full profile implementation and **StreamEndPoint_B** is from a light profile implementation.

Because the **StreamCtrl** is a full profile implementation, it will attempt to bind the stream using the algorithm above. It will begin by querying both **StreamEndPoints** to see what flows they support **SEP_A->get_property_value**("**Flows**") and **SEP_B->get_property_value("Flows")**. It will then attempt to iterate through the flows on A and B by calling an operation **get_fep("flowName")** for each flow. On the **StreamEndPoint_A** (which is full profile) this will return an Object which can be narrowed to a **FlowEndPoint** but on the **StreamEndPoint_B** (light profile) this operation will raise the exception notSupported with reason "Full profile not supported." The **StreamCtrl** will then realize that this StreamEndPoint is a light profile and will proceed by using the light profile algorithm for stream establishment (i.e., it calls **connect()** on **StreamEndPoint_A** with **StreamEndPoint_B** as a parameter). In fact, the **connect()** operation works just as well from a B endpoint to an A endpoint as it does from an A to a B.

If, on the other hand, both **StreamEndPoints** were both full profile, then the **get_fep()** operation would succeed on both and the **StreamCtrl** would build a list of **FlowEndPoints** on the A side and on the B side. It will use the algorithm above to iterate through these lists matching **FlowEndPoints** on the A side to **FlowEndPoints** on the B side. Once it is established that two **FlowEndPoints** can be connected, the **StreamCtrl** creates a **FlowConnection** to bind them and calls **connect()** on the **FlowConnection** with the two **FlowEndPoints** as a parameter. For each pair of matching source/sink **FlowEndPoints**, the **FlowConnection** calls **set_peer()** on the producer **FlowEndPoint** with the consumer as a parameter. This has the same function as calling **set_peer()** on the **VDev**. The **FlowConnection** then calls **go_to_listen()** (see Figure 2-5) on the sink which returns a transport address. It then calls **connect_to_peer()** on the source using the transport address as a parameter. This scenario is depicted in Figure 2-5. This **StreamCtrl** implementation's flow matching algorithm may be designed to be overriden in derived classes so that the algorithm for matching flow endpoints is suited to a particular scenario. Note that the sink **FlowEndPoint** can be on either the A side or the B side. The bind call to **StreamCtrl** will be given the names of the flows to bind and it will first search the A side for a flow of this name to bind and then the B side.

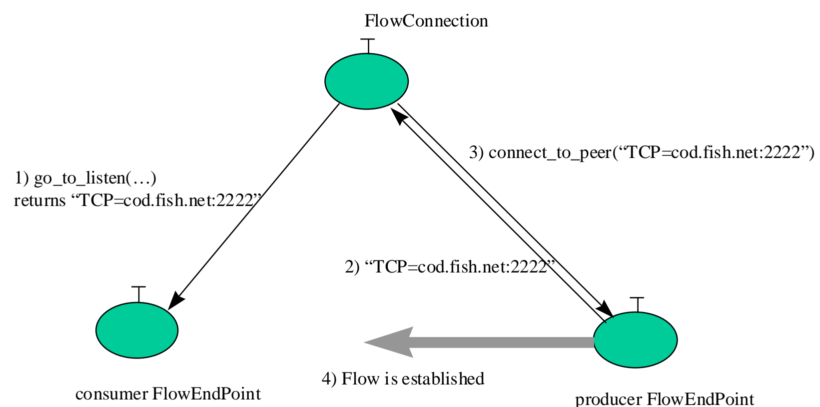

Figure 2-5 Flow establishment in full profile Media Streaming Framework

There are two important points to note about flows vs streams in the full profile:

- A stream endpoint can be used to group a collection of flow endpoints which can reside on different nodes. A **FlowEndPoint** can be added to a **StreamEndPoint** by calling **add_fep()** on a **StreamEndPoint**.
- A **FlowConnection** can be used exactly like a **StreamCtrl** except that it only controls one flow. A collection of **FlowConnections** can be set up and subsequently grouped under a **StreamCtrl** by calling **set_flow_connection()** on the **StreamCtrl** for each **FlowConnection**. Operations like stop and start applied to the **StreamCtrl** will be applied to all contained **FlowConnections**.

2.2.6 Properties of Streams

The Media Streaming Framework makes extensive use of properties to describe devices, streams, flows, and their endpoints. This allows any of these entities to be queried to establish their status and their compatibility constraints. "Device and Stream Parameters" on page 2-25 outlines some standard properties of stream entities and their associated values.

2.2.7 Quality of Service

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 B-8). 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.

Note – Please note the following convention: Where a range of values is acceptable, the desired value should be called "Name" and the acceptable value should be either "Name_Max" or "Name_Min."

2.2.8 Stream Specification

It is beneficial for a stream service implementation to provide a notation to specify the content of flows and their relative direction within a particular stream type. This notation, however, is beyond the scope of this specification. A de facto notation such as TINA-C ODL stream template [9], for example, could be used. It is expected that another RFP will be issued which will cover the area of stream typing notation and the related language mappings for typesafe insertion and extraction of data to and from a flow.

It should be possible to compile typesafe versions of the interfaces so far discussed by using a streams notation. For example, a stream of type X will always lead to the generation of a new IDL interface **X_StreamCtrl** which can only be used to bind X devices. The compiler would also generate the implementation code for the new IDL interfaces. Although the notation itself is not within the scope of this specification, the resulting generated IDL interfaces can be standardized. Consider, for example, a specification for a videophone stream with two flows containing audio and video, one in each direction. Compiling such a stream specification will result in an IDL file which will contain IDL for specializations of the following interfaces:

- **StreamCtrl (videophone_StreamCtrl)**
- **StreamEndPoint _A and _B (videophone_A, videophone_B)**
- **VDev (v_videophone)**
- **MMDevice (videophone)**

A Stream notation might typically define a stream as being a composition of different flows. Flows can also be strictly typed, as follows (for a flow type X):

- **FlowProducer (X_Producer)**
- **FlowConsumer (X_Consumer)**
- **FDev (F_X)**
- **FlowConnection (X_Connection)**

The generated IDL for the videophone definitions might look something like the following:

```
#include <MediaStreams.idl>

interface Videophone_StreamCtrl : StreamCtrl{
```

```
        ...
        boolean bind_videophone_devs(in videophone a_party,
                in videophone b_party,
                inout streamQoS the_qos,
                in flowSpec the_spec)
                raises(...);
        ...
};
etc.
```

The stream notation compiler will generate the source code for implementation classes corresponding to these interfaces (see Figure 2-6). In Figure 2-6 the classes labeled ‘user’ are generated as a result of the stream specification. By using a videophone class in place of an **MMDevice** class and a **videophone_StreamCtrl** class instead of **StreamCtrl**, the application programmer can catch gross errors, such as attempting to connect a videophone to a cd_player at compile time. The **StreamCtrl**, **MMDevice**, and **StreamEndPoint** classes all contain the property "Type" which allows them to be queried as to the type of stream they support. A value of empty-string indicates that no typing information is supplied. Each of the interfaces also contains a property "Flows" which contains the names of all the flows in the stream, device, or stream endpoint.

It is important to bear in mind that the classes **StreamCtrl**, **StreamEndPoint_A**, and **StreamEndPoint_B** are directly instantiable and can be used for general stream control and management. The use of specialized subtypes like **videophone_StreamCtrl** is merely a convenience for catching errors at compile time and making stream programming simpler.

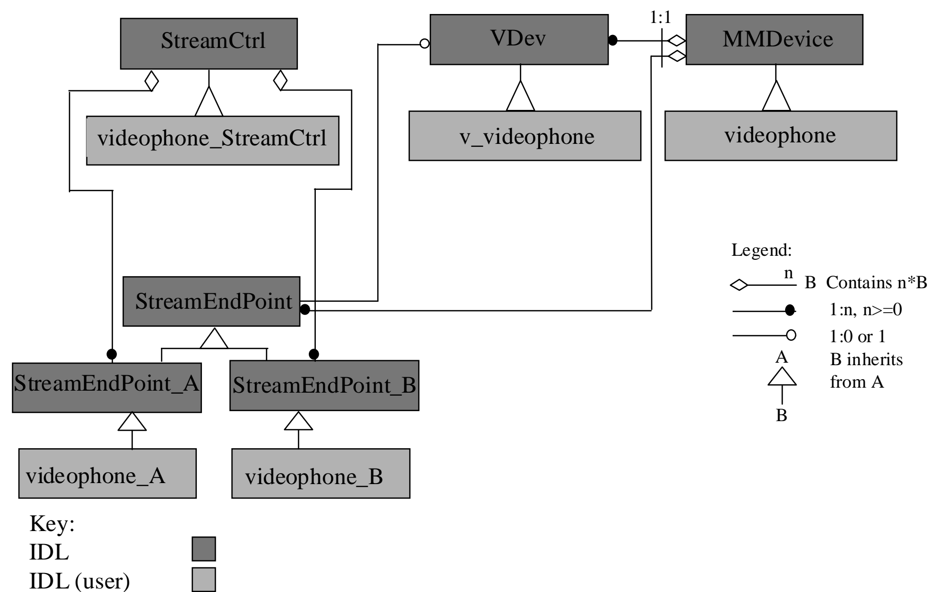

Figure 2-6 Relationships between a subset of objects in the media stream framework

2.2.9 Flow Protocol

There are a number of ways in which the type of transport used to stream data has a fundamental impact on the solution architecture. To understand why this is the case, it is necessary to realize that there are differences between the transport requirements of a regular ORB and the transport requirements for a multimedia stream. An ORB requires a reliable transport, which implicitly involves retransmission when a frame is dropped or in error. Unfortunately retransmission can seriously impact an isochronous stream by causing jitter. Furthermore, it can add unnecessary weight to the transportation of time-critical data if that data is being transmitted on a highly reliable network anyway. It is quite often the case that the timeliness of stream data is more important than whether it is completely correct all of the time. As long as the application is aware that there is a problem, it can deal with it in its own way. With this in mind there are three fundamental types of transport which should be supported by the framework:

- Connection-Oriented transport: This is provided by transports like TCP and required where completeness and reliability of data are essential.
- Datagram-oriented transport: This is frequently more efficient and lightweight if the application doesn't mind losing the occasional datagram and handling a degree of mis-sequencing. UDP is used by many popular Internet streaming applications. The framework must insert sequence numbers into the packets to ensure that mis-sequencing and packet-loss is detected.
- Unreliable connection-oriented transport: This is the type of service supplied by ATM AAL5. Messages are delivered in sequence to the endpoint but they can be dropped or contain errors. If there are errors, then this fact is reported to the application. There is no flow control unless provided by the underlying network (e.g., ABR).

In addition to the requirements placed on the framework by the various transport types, there are application level factors which will influence the way that flows are structured. These include the need for timestamping in some applications. Also, there may be a need for adding an indication of the source of a media packet. This is in cases where a stream endpoint may be receiving from multiple senders on an internet multicast connection. All this information will need to be transported in band. Such information can include:

- Sequence numbers
- Source indicators
- Timestamps
- Synchronization sourceR

Refer to RTP specification [5] for additional information.

There is no single transport protocol which provides all the capabilities needed for streamed media. ATM AAL5 is good but it lacks flow control so that a sender can overrun a receiver. Only RTP provides facilities for transporting the in-band

information above, but RTP is internet-centric and it should not be assumed that a platform must support RTP in order to take advantage of streamed media. Furthermore, none of the transports provide a standard way for transporting IDL typed information.

The only way to accommodate the various needs of multimedia transport over a multitude of transports is to define a simple specialized protocol which works on top of various transport protocols and provides architecture independent flow content transfer. This protocol will be referred to as the Simple Flow Protocol (SFP). There are two important points to note about SFP:

1. It is not a transport protocol, it is a message-level protocol, like GIOP, and is layered on top of the underlying transport. It is simple to implement.
2. It is not mandatory for a Media Streaming Framework implementation to support SFP. A flow endpoint which supports SFP can switch it off in order to communicate with a flow endpoint which does not support it. The use of the SFP is negotiated at stream establishment. This is discussed in “The StreamEndPoint_A and StreamEndPoint_B” on page 2-43. If the stream data is not IDL-typed (i.e., it is in agreed byte layout, for example MPEG) then, by default, SFP will not be used. This allows octetstream flows to be transferred straight to a hardware device on the network.

The SFP v1.0 protocol messages are specified as:

```
module flowProtocol{
enum MsgType{
    // Messages in the forward direction
    Start,
    EndofStream,
    SimpleFrame,
    SequencedFrame,
    Frame,
    SpecialFrame,
    // Messages in the reverse direction
    StartReply,
    Credit};

struct frameHeader{
    char magic_number[4]; // ‘=’, ‘S’, ‘F’, ‘P’
    octet flags;        // bit 0 = byte order,
                        // 1 = fragments, 2-7 always 0
    octet message_type;
    unsigned long message_size;// Size following this header
};

struct fragment{
    char magic_number[4]; // ‘F’, ‘R’, ‘A’, ‘G’
    octet flags; // bit 1 = more fragments
    unsigned long frag_number; // 0,..,n
    unsigned long frag_sz;
    unsigned long source_id;// Required for UDP multicast
```

```
};          // with multiple sources
struct Start{
    char magic_number[4]; // '=', 'S', 'T', 'A'
    octet major_version;
    octet minor_version;
    octet flags;  // bit 0 = byte order
};

// Acknowledge successful processing of
// Start
struct StartReply{
    octet flags;  // bit 0 = byte order, 1 = exception
};

// If the message_type in frameHeader is sequencedFrame
// the the frameHeader will be followed by this
// (See also RTP note)
struct sequencedFrame{
    unsigned long sequence_num;
};

// If the message_type is Frame then
// the frameHeader is followed by this
// See also RTP note
struct frame{
    unsigned long timestamp;
    unsigned long synchSource;
    sequence<unsigned long> source_ids;
    unsigned long sequence_num;
};

struct specialFrame{
    frameID context_id;
    sequence <octet> context_data;
};

struct credit{
    unsigned long cred_num;
};
};
```

As in GIOP, the SFP uses CDR encoding and obeys the same conventions. That is, in the frameHeader structure the magic number and flags are in network byte order. All subsequent fields are CDR encoded. All fields in the Start and StartReply messages are also in network byte order.

Note that for a simple frame there is no subsequent header, simply a stream of octets. The other message headers: **frameHeader, frame**, and **specialFrame** are followed by a stream of octets.

Normally if a **frameHeader** has the **message_type=SequencedFrame**, then it is followed by a **sequencedFrame** structure and if it has the **message_type=Frame**, then it is followed by a frame structure. The exception to this rule is when SFP is running over RTP. In this circumstance, the **sequencedFrame** and frame structures are not used since the values for **timestamp, synchSource, source_ids,** and **sequence_num** are embedded directly into the corresponding RTP protocol fields [5].

The typical structure of an SFP dialog for a point-to-point flow is shown in Figure 2-7. The dialog begins with a **Start** message being sent from source to sink. The source waits for a **StartReply** message. The **message_size** field in the frame header denotes the size of the message (including the headers) if no fragmentation is taking place. If fragmentation is being used, then message_size indicates the size of the first fragment including headers.

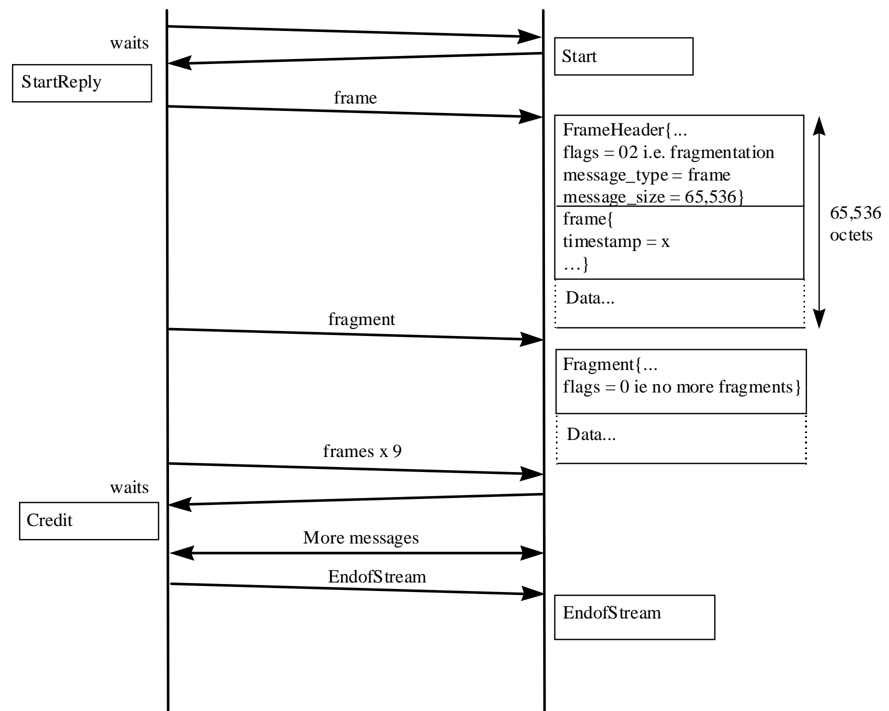

Figure 2-7 Typical SFP dialog

The sink of a flow may send a credit message to the source of the flow to tell it to send more data. The **cred_num** field will be incremented with each credit message sent. This facility may be used with protocols such as ATM AAL5 or UDP which have no flow control. “Compatibility of Flow Formats” on page B-6 describes the behavior of SFP in more detail. Note that the SFP dialog is much simpler on a multicast flow since no messages are sent in the reverse direction.

2.2.10 Examples for point-to-point streams

The code fragment below illustrates a simple stream binding (see Figure 2-8 on page 2-22):

```
// 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.

```
// C++
// 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:

```
// C++
// Details omitted
// 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 multimedia 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.

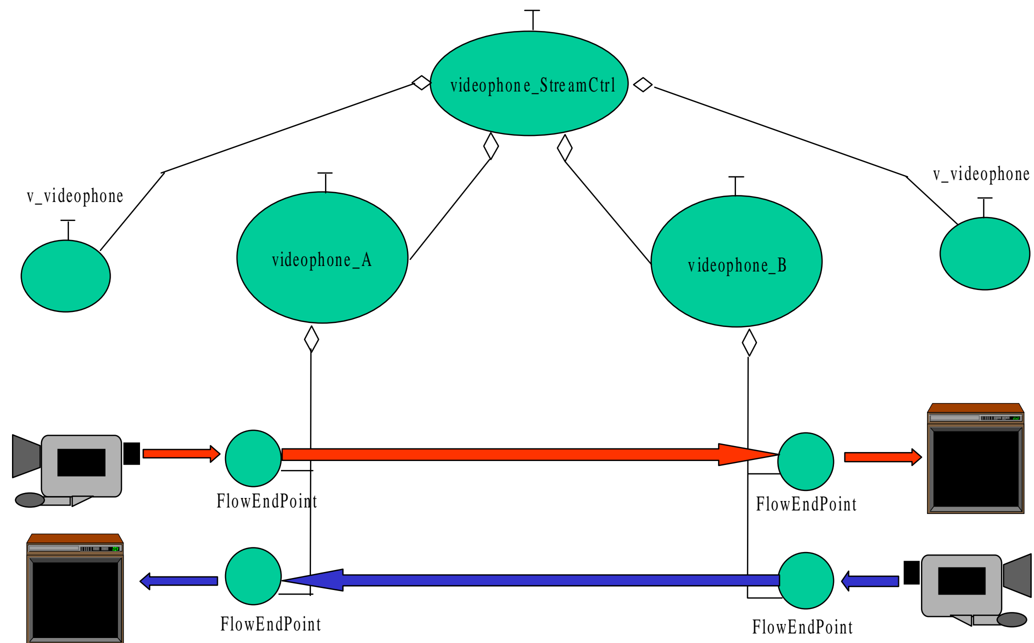

Figure 2-8 Overview of objects involved in a point-to-point videophone stream

2.2.11 Issues in Modifying QoS

One advantage of having a stream abstraction is that the QoS associated with that stream can be changed over time. One way in which this can happen is through the user directly requesting a quality of service change.

```
// C++
// Modifying streams
try{
      myStream->modifyQoS(newQoS);
}catch(QoSRequestFailed f){
   …
}
```

This is deceptively simple since changing the application level QoS can have complex effects. Any changes to application level QoS must therefore be informed to one or both of the **VDevs**. When it receives a **modify_QoS()** request, the **StreamCtrl** will assess which flows are involved and what directions those flows go. For example, there are three flows f1, f2, and f3. The f1 flow has direction in (by convention 'in' flows are towards the B party). The f2 and f3 flows have direction out. Since the flow f1 originates in the A party, the **StreamCtrl** passes the request modify_QoS(…,<f1>) to the **VDev** at the A end. It will then pass the request modify_QoS(…,<f2,f3>) to the **VDev** at the B end. By default, the **VDev** will simply pass the **modify_QoS()** request on to its associated **StreamEndPoint**. However, the application programmer who is subtyping a **VDev** may typically do the following upon receiving a request to modify QoS:

- Check that the QoS modification is valid.
- If policy dictates, then stop the stream.
- Send any re-configuration data needed to the peer device. The peer device might reject a change in the QoS.
- Call modifyQoS on the appropriate **StreamEndPoint_A**.
- Restart the stream if necessary.

It should be noted that today's protocols generally do not support QoS changes. Where a **FlowEndPoint** does not support QoS modification, the flow may be brought down and restarted with a new QoS by the **FlowEndPoint** object.

2.2.12 Issues in Multipoint Streams

Multicasting is a very important network technology since it will form the basis of conferencing and broadcasting applications. At the protocol level, multicasting minimizes the bandwidth and complexity required to send information to multiple stations on a network. There are two basic programming models for multicast networking: 1) the internet model and 2) the ATM model.

1. In the internet model, a party who wishes to multicast information finds a free multicast address and starts sending information on that address. Any party who wishes to listen 'tunes in' to that multicast address. This is the technique used by

the popular MBone applications. If more than one party wishes to broadcast packets to the address then there is nothing preventing this from happening. The underlying transport is UDP and so transmission is unreliable and no flow control is exercised.

2. The ATM-style involves setting up a Switched Virtual Circuit (SVC) to a destination station. The initiator (A party) can then request the local switch to add parties to that circuit creating a multicast tree with one root (source) and a number of leaves (sinks). The interface to the **StreamCtrl** hides these differences in the underlying programming model from the application programmer.

This section explains how multicast streams are used by the application. Consider a videoserver stream with two flows, audio and video. The code below shows how an application programmer can add a multicasting source (A party) to such a stream and subsequently add two B parties (sinks). It should be noted that in regular internet multicasting no one party is aware that other parties are listening to a multicast. This is different in the **Media Streaming Framework** because each B party is explicitly added to the list of B parties by the **StreamCtrl** and is given the appropriate multicast addresses to listen on. In the example below, all parties are joined prior to start() being called on the stream. It must be possible, however, to add and remove parties after start() has been called.

```
// C++
// Multicast stream binding
//
vidserver_ptr aVidsrc = …;
vidserver_ptr vidsink1, vidsink2;
…
// Declare the stream controller
vidserver_StreamCtrl_ptr myVid = …;

// Bind aVidsrc, vidsink1 and vidsink2
…
// Add the multicast root
myVid->bind_devs(aVidsrc,nilObject,someQoS,nilFlowSpec);

// Add a B party for receiving the a/v data
myVid->bind_devs(nilObject,vidsnk1,someQoS,nilFlowSpec);

// Start the stream
myVid->start(nilFlowSpec);

// Add another B party for receiving the a/v data
myVid->bind_devs(nilObject,vidsnk2,someQoS,nilFlowSpec);

cout << "Hit return to end multicast" << endl;
cin >> buf;
myVid->stop(nilFlowSpec);
myVid->unbind();
```

2.2.13 Extending Stream Management Functionality

The **StreamCtrl** interface provides no support for performing operations like rewind or fast forward on a flow. This kind of behavior depends on the functionality of the device that is the producer of the flow. In order to facilitate this kind of operation, each **FlowEndPoint** may be associated with 0 or 1 media controllers. In the light profile, the application programmer can access a flow's media controller through an object property of the **VDev** called "FlowNameX_related_mediaCtrl." In the full profile, the media controller can be accessed through the **FlowEndPoint** using the object property "Related_mediaCtrl." In both cases an Object reference is returned. This allows the application programmer to narrow the reference to the appropriate type of controller. In "Device and Stream Parameters" on page 2-25 a standard controller interface for stored media is provided.

Returning to the application programmer who wants to just rewind, s/he can do this by specifying a media controller for the device as follows:

```
// IDL
interface rewindable_control{
        void rewind();
        void stop_rewind();
        … etc.
};
```

The programmer then implements this and associates an instance with the appropriate flow endpoint. S/he can also extend the **StreamCtrl** interface to make include rewind functionality (see below). The implementation of **rewind()** will narrow the Object reference returned by getting a "Related_mediaCtrl" property to a **rewindable_control** reference.

```
//IDL
interface myStreamCtrl : StreamCtrl {
        void rewind(string flowname);
        void stop_rewind(string flowname);
};
```

2.2.14 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 "The A/V Streams Registration Space" on page B-8):

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 B-8). 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 B-8).

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 FlowEndPoint" on page 2-48 supplies a useful standard media control interface; whereas, Appendix C illustrates how DAVIC's DSM-CC UU commands could be used to perform media control.

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 (see also "The A/V Streams Registration Space" on page B-8):

- 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**

Further properties can be exposed through a registration process described later in Appendix A.

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>

2.3 IDL Interfaces

The purpose of this section is to specify the semantics of each IDL interface in more detail.

2.3.1 The Basic_StreamCtrl

The first and most important interface is the **StreamCtrl** interface. The **StreamCtrl** interface inherits from **Basic_StreamCtrl**.

```
struct SFPStatus{
        boolean isFormatted;
        boolean isSpecialFormat;
        boolean seqNums;
        boolean timestamps;
        boolean sourceIndicators;
};

interface Basic_StreamCtrl : PropertyService::PropertySet {

        // Empty flowSpec => apply operation to all flows
        void stop(in flowSpec the_spec) raises (noSuchFlow);
        void start(in flowSpec the_spec) raises (noSuchFlow);
        void destroy(in flowSpec the_spec) raises (noSuchFlow);

        boolean modify_QoS(inout streamQoS new_qos,
                in flowSpec the_spec)
                raises (noSuchFlow, QoSRequestFailed);
```

```
        // Called by StreamEndPoint when something goes wrong
        // with a flow
        oneway void push_event(
                in streamEvent the_event);

        void set_FPStatus(in flowSpec the_spec,
                in string fp_name,        // Only SFP1.0 currently
                                          // specified
                in any fp_settings)       // Currently SFP accepts
                                          // SFPStatus structure
                raises (noSuchFlow, FPError);

        Object get_flow_connection(in string flow_name)
                raises (noSuchFlow, notSupported);

        void set_flow_connection(in string flow_name,
                in Object flow_connection)
                raises (noSuchFlow, notSupported);
};
```

The property service is used to record the properties enumerated in the previous section. The properties are readonly.

When a point-to-point stream receives a **start()** operation it does two things:

1. It determines whether there is a related Media Control object for that flow. For example, it queries the FlowNameX_related_mediaCtrl property on the **VDev** in the light profile or queries the Related_mediaCtrl property on the relevant **FlowProducer** in the full profile.
2. It then calls **start()** on the media control object with a relative position of 0. It then calls **start()** on the A-party stream endpoint. The A-party end-point could do the following:
   - Calls **start()** on its incoming flows.
   - Calls **start()** on its peer B-party end-point.
   - Calls **start()** on its outgoing flows.

The **start()** operation takes a sequence of flow names (called a flowSpec) as a parameter. If the list is empty, then the operation is applied to all flows. This convention is used throughout these interfaces. The **stop()** operation works in exactly the opposite way to the **start()** operation. In multipoint streams, only the A-party is started and stopped, the B-party is always ready to read.

The **destroy()** operation tears down a stream. This includes tearing down the transport connections and deleting all MMDevice-created **StreamEndPoints** and **VDevs**.

The **modify_QoS()** operation changes the QoS associated with a stream. The operation of the **modify_QoS()** operation has already been discussed in “Issues in Modifying QoS” on page 2-23. If the resulting QoS was best effort, then it returns false and the QoS parameter’s out value will indicate the actual modification to the QoS.

The **push_event()** operation is used by **StreamEndPoints** or **VDevs** to inform the **StreamCtrl** of events that are happening on a flow. Events are just typed as properties. The following events are currently defined:

- {"FlowLost" string} where string is the name of the flow lost
- {"FlowReEstablished" string} where string is the name of the flow re-established.
- {"QoSChanged" string} where the string holds the name of the flow.

The **set_FPStatus()** operation allows the application programmer to explicitly control aspects of flow protocol for all flows in the stream. It is envisaged that setting the flow protocol parameters of a stream will seldom be done explicitly at this level but the hooks to allow it are there. There will be further discussion of this feature under the **StreamEndPoint** interface.

The **set_flow_connection()** operation will raise a notSupported exception in light profile implementations. In full profile implementations it will allow the declaration and installation of a particular **FlowConnection** object for flow. This is desirable where an application programmer may want a **FlowConnection** to have special, inherited functionality. For example:

```
interface specialFlowConnection : FlowConnection, MediaControl {};
```

The application programmer can declare an instance of **specialFlowConnection** to handle a particular flow (e.g., a video flow) and then call:

```
myStreamCtrl->set_flow_connection("video1",myspecialFlowConnection);
```

2.3.2 The StreamCtrl

The **StreamCtrl** interface for streams is specified below. Its purpose is to allow for point-to-point and point-to-multipoint bindings. The **StreamCtrl** interface was kept separate from Basic_StreamCtrl to allow the future possibility of other binding controllers which inherit from Basic_StreamCtrl. The **StreamCtrl** IDL follows:

```
interface StreamCtrl : Basic_StreamCtrl {

    boolean bind_devs(in MMDevice a_party, in MMDevice b_party,
        inout streamQoS the_qos,
        in flowSpec the_flows)
        raises (streamOpFailed, noSuchFlow, QoSRequestFailed);

    boolean bind(in StreamEndPoint_A a_party,
        in StreamEndPoint_B b_party,
        inout streamQoS the_qos,
        in flowSpec the_flows)
        raises (streamOpFailed, noSuchFlow, QoSRequestFailed);

    void unbind_party(in StreamEndPoint the_ep,
        in flowSpec the_spec)
        raises (streamOpFailed, noSuchFlow);
```

```
    void unbind()
        raises (streamOpFailed);
};

interface MCastConfigIf : PropertyService::PropertySet{

    boolean set_peer(in Object peer,
        inout streamQoS the_qos,
        in flowSpec the_spec)
        raises (QoSRequestFailed, streamOpFailed);

    void configure(in PropertyService::Property a_configuration);

    void set_initial_configuration(
        in PropertyService::Properties initial);

    // Uses <format_name> standardized by OMG and IETF
        void set_format(in string flowName, in string format_name)
            raises (notSupported);

    // Note, some of these device params are standardized by OMG
    void set_dev_params(in string flowName,
        in PropertyService::Properties new_params)
        raises(PropertyService::PropertyException,
            streamOpFailed);
};
```

The basic **bind_devs()** operation sets up a stream between **MMDevices: streamQoS** identifies the desired QoS for the stream. If it is not met, then the **bind_devs()** call returns false and the **streamQoS** parameter is changed to reflect the actual QoS of the stream. The **flowSpec** is used to name the subset of flows to bind. If the **flowSpec** is an empty sequence, then all flows are bound.

The **bind_devs()** call can be used in the following modes:

- **bind_devs(aDev,bDev,someQoS,flowSpec)**
  Bind two MMDevices, aDev and bDev with a stream
- **bind_devs(aDev,nilObject,someQoS,<f1,f2>)**
  Bind aDev as a multicast source with the flows f1 and f2. If aDev is already bound, then the effect is to add the flows f1 and f2 to the stream. For example, the A party will now be multicasting f1 and f2 as well as whatever flows it was multicasting previously. Any subsequent B parties that join with a nilflowSpec will automatically have access to these flows, but existing B parties will not.
- **bind_devs(nilObject,bDev,-,<f1,f2>)**
  Bind bDev as a multicast sink with the flows in flowSpec
  If bDev is already bound, then add flows from f1 and f2 (i.e., the bDev will now listen for multicast flows f1 and f2).

- **bind_devs(nilObject,nilObject,someQoS,<f1,f2>)**
  Add the flows f1 and f2 to all parties in an existing stream. In the multicast case, this means all existing A parties and B parties will multicast/receive the flows f1 and f2.

In multicast streams between **VDevs**, configuration information is distributed via the **MCastConfigIf** interface, shown above. The configuration commands (e.g., **set_format()**, **configure())** are sent by A party devices to an **MCastConfigIf**. The semantics of the configuration operations on the **MCastConfigIf** is the same as their equivalents on the **VDev** interface except that all configuration calls will be distributed by the **MCastConfigIf** to B parties connected to the stream. How this information is distributed to the B parties is not stipulated in this specification. It is possible (if multicast messaging is not available) for the **MCastConfigIf** object to make configuration calls on each of the B parties in turn.

Calls to **configure()** use parameters of type **Property**, for example:

```
{
"video_interlace"
True
}
```

When a B party joins it should receive some initial configuration information to tell it how the A party device is configured. The A party can set this initial configuration information through the **set_initial_configuration()** operation on **MCastConfigIf**.

The **bind()** operation is the equivalent of the **bind_devs()** operation for **StreamEndPoints**. Its modes of operation work in just the same way.

The **unbind()** operation is used to unbind parties from a stream or unbind flows. A sample of the different modes used are as follows:

- **unbind(aSEP,nilFlowSpec)**
  Unbind the StreamEndPoint aSEP
- **unbind(aSEP,<f1,f2>)**
  Unbind only the flows f1 and f2 from sSEP
- **unbind(nilObject,<f1,f2>)**
  Unbind flows f1 and f2 from all stream endpoints
- **unbind(nilObject,nilflowSpec)**
  Unbind the stream. Equivalent to calling **unbind()** which is equivalent to calling **destroy()**.

A stream notation compiler will generate specialized **StreamCtrl** interfaces which inherit from **StreamCtrl**. Consider the example definition previously given for the videophone stream. This would lead to the following code being generated for a specialized **StreamCtrl**:

```
Videophone_StreamCtrl : StreamCtrl{

    boolean bind_videophone_devs(in videophone a_party,
```

```
        in videophone b_party,
        inout streamQoS the_qos,
        in flowSpec the_flows)
        raises (streamOpFailed, noSuchFlow, QoSRequestFailed);

    boolean bind_videophone(in videophone_A a_party,
        in videophone_B b_party,
        inout streamQoS the_qos,
        in flowSpec the_flows)
        raises (streamOpFailed, noSuchFlow, QoSRequestFailed);

    boolean bind_videophone_A_party(in videophone_A a_party,
        inout streamQoS the_qos,
        in flowSpec the_flows)
        raises (streamOpFailed, noSuchFlow, QoSRequestFailed);

    boolean bind_videophone_B_party(in videophone_B b_party,
        inout streamQoS the_qos,
        in flowSpec the_flows)
        raises (streamOpFailed, noSuchFlow, QoSRequestFailed);
};
```

2.3.3 The StreamEndpoint

The **StreamEndPoint** is described in IDL as follows:

```
interface StreamEndPoint : PropertyService::PropertySet{

    void stop(in flowSpec the_spec) raises (noSuchFlow);
    void start(in flowSpec the_spec) raises (noSuchFlow);
    void destroy(in flowSpec the_spec) raises (noSuchFlow);

    boolean connect(in StreamEndPoint responder,
        inout streamQoS qos_spec,
        in flowSpec the_spec)
        raises (noSuchFlow, QoSRequestFailed, streamOpFailed);

    boolean request_connection(
        in StreamEndPoint initiator,
        in boolean is_mcast,
        inout streamQoS qos,
        inout flowSpec the_spec)
        raises (streamOpDenied, noSuchFlow,
            QoSRequestFailed, FPError);

    boolean modify_QoS(inout streamQoS new_qos,
        in flowSpec the_flows)
        raises (noSuchFlow, QoSRequestFailed);

    boolean set_protocol_restriction(in protocolSpec the_pspec);
```

```
    void disconnect(in flowSpec the_spec)
        raises (noSuchFlow, streamOpFailed);

    void set_FPStatus(in flowSpec the_spec,
        in string fp_name,
        in any fp_settings)
        raises (noSuchFlow, FPError);

    Object get_fep(in string flow_name)
        raises (notSupported, noSuchFlow);

    string add_fep(in Object the_fep)
    // Can fail for reasons {duplicateFepName, duplicateRef}
        raises (notSupported, streamOpFailed);

    void remove_fep(in string fep_name)
        raises (notSupported, streamOpFailed);

    void set_negotiator(in Negotiator new_negotiator);
    void set_key(in string flow_name, in key the_key);
    void set_source_id(in long source_id);
};
```

A **StreamEndPoint** may be associated with a **VDev**. If this is the case, then the **Related_VDev** property holds a reference to the **VDev**.

The **start()** and **stop()** and **destroy()** operations have already been discussed in relation to streams. Calling **destroy()** will have the effect of disconnecting all the flow transports rather than destroying the **StreamEndPoint** objects.

connect() and request_connection()

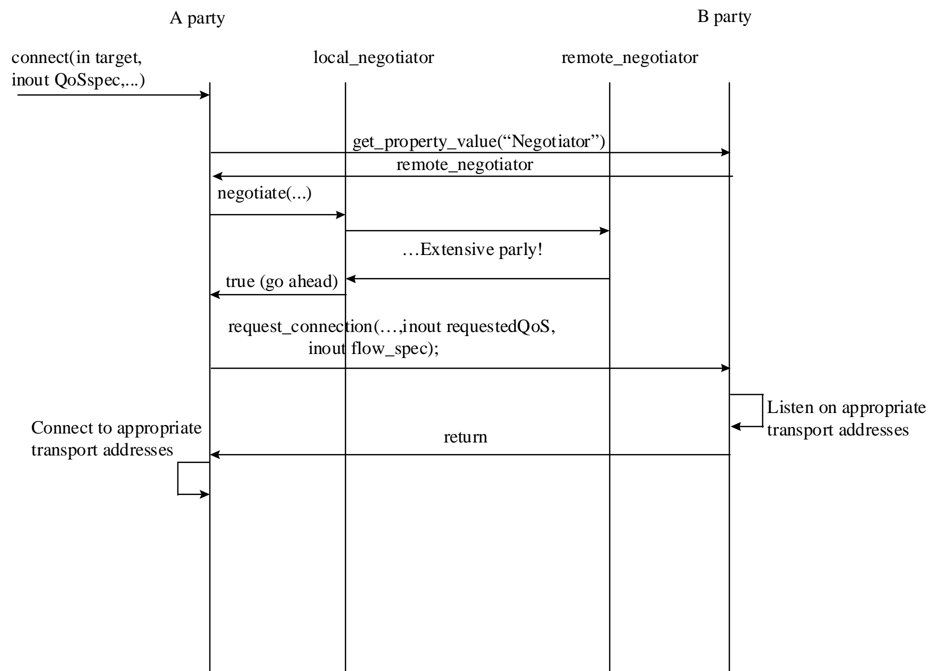

Figure 2-9 Message Sequence Chart for a connect () operation

The **StreamEndPoint** interface provides the **connect()** operation for connecting to a peer StreamEndPoint object. This operation is normally used in the 'light' profile of the stream component and is usually called from an A party to a B party. The 'full' profile, where available, will perform stream setup using **FlowEndPoint** interfaces. The workings of **connect()** are illustrated by the message sequence chart in Figure 2-9. The sequence in which an A end-point makes calls to the B end-point is very important since this sequence of operations must be followed in order for different implementations of **StreamEndPoint** from different vendors to interwork.

```
// Pseudo-code
boolean StreamEndPoint_A::connect(
   in StreamEndPoint_B_ptr remoteStreamEp,
   inout streamQoS aQoS,…)

   If there is no local negotiator, then
      there will be no negotiation
   else
      tmpAny = remoteStreamEp
         ->get_property_value("Negotiator");
      // extract peerNegotiator from tmpAny
      …
      myNegotiator->negotiate(peerNegotiator,
         aQoS);
```

```
Choose protocols based on what the remote end-point can sup-
port. If the QoS is specified at application level then
translate application-level QoS into network level QoS for
those protocols.

Set up listening flows where appropriate

Try{
      remoteStreamEp->request_connection(
         this,
         false       // Not multicast
         offeredQoSspec,// inout
         flow_spec   // inout
         );
}catch(…){
      Take appropriate action
   }
Connect to listening flows in the B-party where appropriate
using transport addresses from the flow spec

}
```

Flow Specification

An important point for interoperability is the format of the **flowSpec**. This is a crucial piece of data since it conveys the information needed to set up the transport connections which carry the flows. The **flowSpec** data type is a sequence of strings but each string must be formatted according to strict rules. Before describing the structure of these strings it is important to establish a basic rule:

> The **StreamEndPoint** which supports the flow consumer endpoint is not necessarily the **StreamEndPoint** which listens for the transport connection for that flow.

The specification is written in such a way that either the producer end or the consumer end can decide to listen for the transport connection which carries the flow. The structure of the **flowSpec** will indicate to the peer **StreamEndPoint** whether the A-party is already listening for a flow or whether it expects its B-party peer to listen for the flow.

For the critical **request_connection()** call, a **flowSpec** is either a forward flowSpec or a reverse flowSpec. A forward flowSpec is going from the caller to the callee (i.e., the the_spec parameter going in) and the reverse flow spec is going back from callee to caller (i.e., the_spec parameter's out value). The syntax is expressed below:

<flowSpec> ::= <flowName> ["\" <forwardFlowSpec> | <reverseFlowSpec>]
<forwardFlowSpec> ::= <direction> ["\" [<format_name>] ["\" <flowProto-col> "\"
[<address>] ["\" <address>]]]

<reverseFlowSpec> ::= [<address> ["\" <flowProtocol>]]

<direction> ::= "in" | "out"
<flowProtocol> ::= <flowProtocolName> ":" <version> [":" <flowProtocolOptions>]
<flowProtocolName> ::= "SFP" | …
<flowProtocolOptions> ::= <OptionTag> ["=" <OptionValue>] [";" <flowProtocolOptions>]
<OptionTag> ::= <AlphaNumericString>
<OptionValue> ::= <AlphaNumericString>
<address> ::= <carrier_protocol> ["=" <networkAddress> ":" <portList>]
<carrierProtocol> ::= "TCP" | "UDP" | "AAL5" | "AAL3-4" | "AAL1" | "RTP/UDP" | "RTP/AAL5" | "IPX" | …
<portList> ::= <portNumber> [";" <portList>]
<portNumber> ::= <intstring> // > 0
<version> ::= <major_version> "." <minor_version>
<major_version> ::= <intstring>
<minor_version> ::= <intstring>
<format_name> ::= <format_category> [":" <fname>]
<format_category> ::= "MIME" | "IDL" | "UNS"
<intstring> ::= stringified integer

The ellipsis (…) following the productions **<flowProtocolName>** and **<carrierProtocol>** indicate that further values for these can be registered with the OMG (see "The A/V Streams Registration Space" on page B-8). The currently recognized values for **<OptionTag>** are "Credit." The corresponding **<OptionValue>** will denote a positive, non-zero integer (e.g., **Credit=10**).

Example 1

An example of a simple forward flowSpec might be:
< "video1\out\MIME:video/MPEG" >.

This **flowSpec** tells the B party that the A party wants to establish one flow called **video1**. The flow is 'out'-bound so it flows from the B party to the A party. The format of the flow is video/MPEG. The B end-point will process this and might typically output something like the following:

<"video1\TCP=cod.fish.net:5678">

This confirms that the flow **video1** has been established and a TCP socket is waiting on transport address cod.fish.net:5678. Note that the sink of the connection does not have to be the party which goes into listen mode for the connection.

Example 2

In this example the forward flowSpec is:
< "audio1\out\\SFP:1.0:Credit=10" >.

That is, the flow **audio1** is to be established with the A party as the sink. No MIME format is given. The SFP v1.0 is to be used and the sink wishes to have 10 frames of credit for flow control. This implies that a protocol such as ATM AAL5, which does

not use flow control, has been chosen. The credit mechanism gives the sink of the flow a simple way to stop itself being overrun by the source. After every 10 frames (or whenever a timeout occurs), it must send a credit message back to the source using the SFP source, in turn, stops after every ten frames it has sent and waits for the receiver to acknowledge that it has caught up. The request_connection call may fail with exceptions that indicate that the B party does not support the formatting protocol or that it cannot use flow control. If successful the reverse flowSpec will look something like the following: **< "audio1\AAL5=..." >**.

Example 3

In this example the forward flowSpec is:
< "audio1\out\\SFP:1.0:Credit=10\AAL5=..." >

This is essentially the same as the above except that the A party is offering to listen for connection establishment. The reverse flowSpec will simply be **< "audio1" >**.

Example 4

In this example the forward flowSpec is:
<"simdata\out\UNS:user/simdata\SFP:1.0\\UDP=analogue:5432" >

This time the A party is the source of the flow. It has been decided already by the A party that the flow will be carried on UDP. The format name is user defined (i.e., unspecified) and indicates to the application that information of type user/simdata is being carried by the flow. The SFP:1.0 field indicates that the flow should use flow protocol SFPv1.0. No credit is sought because the A party is the source of the flow and only sinks can ask for credit. The address given in this context is the address of a reverse channel. A reverse channel is only needed when a datagram-oriented transport like UDP is in use. This reverse channel has two purposes:

1. For credit messages to be sent from the sink to the source, the sink will need an address to send the credit datagrams to.
2. It is used at the sink to filter out packets which are not coming from the flow source. Because UDP is not connection oriented, datagrams from other sources could potentially corrupt the flow unless this is done.

Typically, the return flowSpec might contain a request for credit (e.g., **< "simdata\UDP=digital:5417\SFP:1.0:Credit=20" >**). If, on the other hand, the reverse flowSpec is **< "simdata\UDP=digital:5417" >**, then no credit is being sought.

There is one exceptional circumstance which causes behavior to differ from the above. If the A party above had left the choice of protocol to the B party (i.e., the forward flowSpec is: **< "simdata\out\UNS:user/simdata\SFP:1.0>** and the B party chose UDP). In this case, the A party will need to supply the B party with a reverse channel address using another call to **request_connection()** with the forward flowSpec set to:

< "simdata\out\UNS:user/simdata\SFP:1.0\\UDP=analogue:5432" >

The B-party will not commence listening for packets from the A party until it has received this reverse channel address.

Other Functions Supported by StreamEndPoint

The **modify_QoS()** operation on this interface refers to modification of transport QoS rather than application QoS. If the transport supporting a flow does not support QoS modification, then it may tear down the flow and restart it.

The operations **get_fep()**, **add_fep()**, and **remove_fep()** are all implemented in the full profile specification but are not implemented in the light profile, where they raise notSupported exceptions with the value "Full profile not supported." The **get_fep()** operation returns a named **flowEndPoint** object. The **add_fep()** operation adds a named **FlowEndPoint** to the **StreamEndPoint**. The returned string indicates the name of the flow endpoint that was added. This is discovered through the **StreamEndPoint** querying the **FlowEndPoint's** "FlowName" property. If this property does not exist, then a name will be generated by the **StreamEndPoint**.

The operation **set_negotiator()** may be used by a stream endpoint to attach a negotiating object to the stream endpoint. Negotiator objects are used as a catch-all to implement special user defined behavior during stream setup. This negotiator object must be derived from the interface Negotiator, but will be highly specialized to a particular type of negotiation.

The **set_protocol_restriction()** operation is used to restrict the set of protocols which may be used by a stream endpoint when creating a stream. This knowledge can be used, for example, to stop the A party end-point selecting a set of protocols which aren't supported by the B party end-point. Suppose that the A end supports TCP and AAL5 and the B end supports AAL5 and UDP. By setting the restriction set in A to {"AAL5," "UDP"} the AAL5 will be chosen for connection set up since it is the only protocol in common with the A sides available set {"TCP," "AAL5"}. Using an empty list clears the restriction set. Strictly speaking every call to **bind_devs()** or **bind()** (in point-to-point cases) should include the behavior that the **StreamCtrl** queries one of the **StreamEndPoints** for its "AvailableProtocols" property and then calls **set_protocol_restriction()** on the other **StreamEndPoint** before calling **connect()** on that **StreamEndPoint**. In practice most application programmers will have their system configured on a single network so the protocol restriction checking will be turned off.

The **set_source_id()** gives the end-point a unique number which can be tagged onto in-band SFP data to identify where it has come from. This can be important in applications which use multicast UDP. For example, consider the case of a flow listening on a UDP multicast address. There are two A-parties sending data to this multicast address. The flow is receiving a sequence of frames but the only way it knows which A party sent which frame is by examining the source-id of the frame. The **StreamCtrl** ensures that each new A or B party added to a stream receives a unique source id to allow its frames to be uniquely identified on such a multicast address. A **VDev** will be given this unique source id by the **StreamCtrl** which can then set its stream endpoints to carry this identifier, if need be.

The **set_FPStatus()** operation results in flags being set for the flow protocol used by all flows. There are various types of format information which can be transmitted in band using SFP, these are: sequence numbers, timestamps, and source indicators. The sequence numbers are always sent for a UDP-based flow. The sourceIndicators are taken from the **source_id()** (see previous operation). This information can be conveyed by specifying **fp_name** as "SFP1.0" and **fp_settings** as a struct SFPStatus.

The **set_key()** operation can serve two functions. If public key encryption is being used, then this operation can be called by the **StreamCtrl** to set a shared secret key for the encryption of a flow in the stream. If public key encryption is being used, then this call is made by the **VDev** and informs the local **StreamEndPoint** of the public key of the peer **StreamEndPoint** (it is assumed that the local private key is known to the **StreamEndPoint**, this is considered an implementation detail). The public keys can be exchanged by **VDevs** during the configuration phase when **set_peer()** has been called. The **VDev** supports a standardized property, FlowNameX_PublicKey.

2.3.4 The StreamEndPoint_A and StreamEndPoint_B

The **StreamEndPoint_A** and **StreamEndPoint_B** classes are derived from **StreamEndPoint**. The **StreamEndPoint_A** interface is shown below:

```
interface StreamEndPoint_A : StreamEndPoint{

    boolean multiconnect(inout streamQoS the_qos,
        inout flowSpec the_spec)
        raises (noSuchFlow, QoSRequestFailed, streamOpFailed);

    boolean connect_leaf(in StreamEndPoint_B the_ep,
        inout streamQoS the_qos,
        in flowSpec the_flows)
        raises (streamOpFailed, noSuchFlow,
        QoSRequestFailed, notSupported);

    void disconnect_leaf(in StreamEndPoint_B the_ep,
        in flowSpec theSpec)
        raises(streamOpFailed, noSuchFlow);
};

interface StreamEndPoint_B : StreamEndPoint {

    boolean multiconnect(inout streamQoS the_qos,
        inout flowSpec the_spec)
        raises (streamOpFailed, noSuchFlow,
            QoSRequestFailed, FPError);
};
```

Multicasting Operations

The **StreamEndPoint** A and B interfaces support multicast operations such as **multiconnect()**, **connect_leaf()**. The **multiconnect()** operation is used with Internet-style multicast and **connect_leaf()** is used with ATM-style multicast. The sequence of events when an application programmer sets up a multicast connection is as follows:

1. The programmer calls **StreamCtrl->bind_devs(Adev,nil,...)**.
   - The **StreamCtrl** calls **A_SEP = Adev->create_A(...,out AVDev,...).**
   - It then creates **mc**, an instance of **MCastConfigIf** and calls **AVDev->setMCastPeer(mc,...)**.
2. When the application programmer calls **StreamCtrl->bind_devs(nil,Bdev,...)**, the **StreamCtrl** calls **B_SEP = Bdev->create_B(...,out BVDev,...)**, it then calls **mc->setPeer(BVDev,...)** followed by **A_SEP->connect_leaf(B_SEP,...)**
   - If this completes successfully, then **B_SEP** is bound to a multicast tree with **A_SEP** being the root.
   - If **connect_leaf()** returns an exception of type notSupported with reason "noMCastTreeSupported," then it retries but using **multiconnect()** instead.

The **multiconnect()** call returns a **flowSpec** with internet multicasting addresses for each of the flows. **StreamCtrl** calls **multiconnect()** on B_SEP using this **flowSpec.**

The **StreamEndPoint::disconnect()** and **disconnect_leaf()** operations are used to respectively tear down connection (internet multicast A party or any B party) and to remove a leaf from a multicast tree.

Specializations of StreamEndPoint A and B

The stream notation compiler will generate specialized interfaces for the endpoints of streams. For the videophone example these would be:

```
interface Videophone_A : StreamEndPoint_A {
};
```

Similarly for videophone_B:

```
interface Videophone_B : StreamEndPoint_B {
};
```

2.3.5 The MMDevice

The interface for **MMDevice** is:

```
interface MMDevice : PropertyService::PropertySet {
    StreamEndPoint_A create_A(
        in StreamCtrl the_requester,
        out VDev the_vdev,
        inout streamQoS the_qos,
```

```
            out boolean met_qos,
            inout string named_vdev,
            in flowSpec the_spec)
            raises(streamOpFailed, streamOpDenied, notSupported,
                QoSRequestFailed, noSuchFlow);

        StreamEndPoint_B create_B(
            in StreamCtrl the_requester,
            out VDev the_vdev,
            inout streamQoS the_qos,
            out boolean met_qos,
            inout string named_vdev,
            in flowSpec the_spec)
            raises(streamOpFailed, streamOpDenied, notSupported,
                QoSRequestFailed, noSuchFlow);

        StreamCtrl bind(in MMDevice peer_device,
            inout streamQoS the_qos,
            out boolean is_met,
            in flowSpec the_spec)
            raises (streamOpFailed, noSuchFlow, QoSRequestFailed);

        StreamCtrl bind_mcast(in MMDevice first_peer,
            inout streamQoS the_qos,
            out boolean is_met,
            in flowSpec the_spec)
            raises (streamOpFailed, noSuchFlow, QoSRequestFailed);

        void destroy(in StreamEndPoint the_ep, in string vdev_name)
            // ie VDev not found
            raises (notSupported);

        string add_fdev(in Object the_fdev)
            raises(notSupported, streamOpFailed);

        Object get_fdev(in string flow_name)
            raises(notSupported, noSuchFlow);

        void remove_fdev(in string flow_name)
            raises (notSupported, noSuchFlow);
};
```

The **create_A()** and **create_B()** operations return references to (A and B) **StreamEndPoint** objects each of which have an associated **VDev** object passed back in the out parameter the_**vdev**. The **namedDev** parameter allows the caller to specify a particular subsystem in the logical device (for example, "camera1"). This is an inout value so it can also be used by the **MMDevice** to tell the **streamCtrl** what the logical name of the endpoint is. This can later be used to expedite the process of destroying an endpoint. The **met_qos** parameter indicates whether the QoS of the returned virtual device meets the criteria specified in the ongoing QoS parameter. If the QoS was not met, then the actual QoS will be given on the out value of the_**qos** parameter. The

exception streamOpFailed can be raised as a result of badly formed parameters or a system failure. The streamOpDenied exception can be used to indicate that the device is overloaded and cannot take any more connections. If this is the case, then reason "Device overload" should be used. The notSupported exception is thrown to indicate that, if an A endpoint was requested only B endpoints are supported and vice versa. The reason will be given as "Device supports only A endpoints" or "Device supports only B endpoints."

The **bind()** operation creates a stream for binding one **MMDevice** to another. Calling **bind_mcast()** creates a multicast binding and adds the first **MMDevice** sink to the multicast stream.

The **destroy()** operation will remove a **StreamEndPoint** and its associated **VDev**. The **destroy()** operation can either use a **StreamEndPoint** reference or the string that was returned by the **named_vdev** parameter in the **create_A** or **create_B** operations.

In full profile implementations, an **MMDevice** can act as a container for a number of **FDevs**. In such cases, **create_A()** and **create_B()** will normally result in calls to the contained **FDevs** which will return **FlowEndPoints**. A **StreamEndPoint** will be created and the **FlowEndPoints** will be added to it by calls to **add_fep()**. Generally, no **VDev** will be created because the **FlowEndPoints** individually contain functions like **set_peer()**. To fill the **MMDevice** container with **FDevs** calls can be made to **add_fdev()**. Each **FDev** has a name (in its properties) and this is what is returned in a string. If the **FDev** has no name, then one is assigned by the **MMDevice**. This is analogous to **add_fep()** on **StreamEndPoint**. The **get_fdev()** operation returns a named **FDev**, while **remove_fdev()** removes an **FDev** from the **MMDevice**. All of these operations raise the exception notSupported with reason "Full profile not supported" in light profile implementations.

The stream notation compiler will generate the following specialized interface for the videophone example:

```
interface Videophone : MMDevice{

    Videophone_A create_videophone_A(
        in StreamCtrl the_ctrl,
        out v_Videophone the_vdev,
        inout streamQoS the_qos,
        out boolean met_qos,
        inout string named_vdev,
        in flowSpec the_spec)
        raises(streamOpDenied, streamOpFailed, notSupported,
            QoSRequestFailed, noSuchFlow);

    Videophone_B create_videophone_B(
        in StreamCtrl the_ctrl,
        out v_Videophone the_vdev,
        inout streamQoS the_qos,
        out boolean met_qos,
        inout string named_vdev,
```

```
            in flowSpec the_spec)
            raises(streamOpDenied, streamOpFailed, notSupported,
                QoSRequestFailed, noSuchFlow);

        Videophone_StreamCtrl videophone_bind(
            in videophone peer_device,
            inout streamQoS the_qos,
            out boolean is_met,
            in flowSpec the_spec)
            raises (streamOpDenied, streamOpFailed, notSupported);

        Videophone_StreamCtrl videophone_bind_mcast(
            in videophone first_peer,
            inout streamQoS the_qos,
            out boolean is_met,
            in flowSpec the_spec)
            raises (streamOpDenied, streamOpFailed, notSupported);
};
```

2.3.6 The VDev

The **VDev** abstracts the idea of a multimedia device which can be linked up to peers across a network. This is the interface which requires most effort on the programmer's part since the behavior of every device will differ. The IDL for **VDev** follows:

```
interface VDev : PropertyService::PropertySet{

    boolean set_peer(
        in StreamCtrl the_ctrl,
        in VDev the_peer_dev,
        inout streamQoS the_qos,
        in flowSpec the_spec)
        raises (noSuchFlow, QoSRequestFailed, streamOpFailed);

    boolean set_Mcast_peer(in StreamCtrl the_ctrl,
        in MCastConfigIf a_mcastconfigif,
        inout streamQoS the_qos,
        in flowSpec the_spec)
        raises (noSuchFlow, QoSRequestFailed, streamOpFailed);

    void configure(in PropertyService::Property the_config_mesg)
        raises(PropertyException, streamOpFailed);

// Uses <formatName> standardized by OMG and IETF
    void set_format(in string flowName, in string format_name)
        raises (notSupported);

// Note, some of these device params are standardized by OMG
    void set_dev_params(in string flowName,
        in PropertyService::Properties new_params)
```

```
        raises(PropertyException, streamOpFailed);

    boolean modify_QoS(inout streamQoS the_qos,
        in flowSpec the_spec)
        raises (noSuchFlow, QoSRequestFailed);
};
```

The application developer or multimedia device vendor must implement the behavior for **set_peer()** and **set_Mcast_peer()**. The purpose of these operations is to ensure that all flows originating in this **VDev** can be understood by the peer **VDev**. The programmer is free to implement this operation in any way s/he chooses. Typically the **VDev** will query the properties of its peer to see what formats it supports for its various flows. In cases where the Protection parameter is set in the QoS specification, it can also find out the public key for its peer flows. The configuration of the peer device will generally consist of calls to **set_format()** and **set_dev_params()** on the peer **VDev** or **MCastConfigIf**. Any type of configuration which can't be achieved with these calls can be done through setting miscellaneous properties using the catch-all **configure()** calls to the peer device and in the multicast case also calling **set_initial_configuration()**. The **set_peer()** call can also be used to get the public key of a peer device. This will be followed by a call to **set_key()** on the local **StreamEndPoint**.

The application programmer is free to reimplement the behavior of modify_QoS(). This has already been discussed in "Issues in Modifying QoS" on page 2-23.

The stream notation compiler generates the following videophone interface which specializes **VDev: interface v_Videophone : VDev{};**

2.3.7 The FlowEndPoint

The **FlowEndPoint** interface is only required to be supported in the full profile. The IDL for **FlowEndPoint** is:

```
interface FlowEndPoint : PropertyService::PropertySet
{
    boolean lock();
    void unlock();

    void stop();
    void start();
    void destroy();

    // Default is a nil object reference
    attribute StreamEndPoint related_sep;
    attribute FlowConnection related_flow_connection;

    FlowEndPoint get_connected_fep()
        raises (notConnected, notSupported);

    // syntax of fp_name is <flowProtocol>
```

```
        boolean use_flow_protocol(in string fp_name,
            in any fp_settings)
            raises (FPError, notSupported);

        // set_format() initializes ‘format’
        // as current media format e.g. MPEG.
        void set_format(in string format)
            raises (notSupported);

        void set_dev_params(
            in PropertyService::Properties new_settings)
            raises (PropertyException, streamOpFailed);

        void set_protocol_restriction(in protocolSpec the_spec)
        raises (notSupported);

        boolean is_fep_compatible(in FlowEndPoint fep)
          raises (formatMismatch, deviceQosMismatch);

        boolean set_peer(
            in FlowConnection the_fc,
            in FlowEndPoint the_peer_fep,
            inout AVStreams::QoS the_qos)
            raises (AVStreams::QoSRequestFailed,
                AVStreams::streamOpFailed);

        boolean set_Mcast_peer(
            in FlowConnection the_fc,
            in MCastConfigIf a_mcastconfigif,
            inout QoS the_qos)
            raises (QoSRequestFailed);
    };

    interface FlowProducer : FlowEndPoint
    {
        boolean connect_to_peer(inout AVStreams::QoS the_qos,
            in string address,
            in string use_flow_protocol) // syntax <flowProtocol>
            raises(failedToConnect,
                AVStreams::FPError, AVSTreams::QoSRequestFailed);

        string connect_mcast(inout QoS the_qos, out boolean is_met,
            in string address,
            in string use_flow_protocol)
            raises (failedToConnect, notSupported,
            FPError, QoSRequestFailed);

        string get_rev_channel(in string pcol_name);

        void set_key(in key the_key);
        void set_source_id(in long source_id);
```

```
};

interface FlowConsumer : FlowEndPoint
{
    string go_to_listen(
        inout AVStreams::QoS the_qos,
        in boolean is_mcast,
        in FlowProducer peer,
        inout string flowProtocol    // syntax <flowProtocol>
                                     // The out value contains SFP
                                     // version supported and
                                     // all options including
                                     // "Credit"
        raises(failedToListen, AVStreams::FPError,
            AVStreams::QoSRequestFailed);
};
```

The **lock()** and **unlock()** operations reserve a **FlowEndPoint** for use with a particular flow. This would typically be called as a result of an **add_fep()** operation on a **StreamEndPoint**. Once a **FlowEndPoint** is locked, it cannot be locked by another **StreamEndPoint** until it has been unlocked by the current owner (i.e., the **lock()** call will return false). This problem does not arise if **FDevs** are used to create **FlowEndPoints** (which is recommended).

Most of the other operations will be familiar from **StreamEndPoint** and **VDev**. The **is_fep_compatible()** checks whether this fep supports compatible protocols and formats to its peer fep.

The **connect_to_peer()** operation connects a source to sink which has already been put into listen mode through a call to **go_to_listen()**. The **get_rev_channel()** call is made as a result of a call to **go_to_listen()** where UDP-like unidirectional protocol has been chosen and SFP is being used. Since SFP sends messages in both directions, it needs a channel to send backward messages on such as Credit.

The **connect_mcast()** operation performs connection for multicast flows. A typical scenario is as follows. A call is made to **add_producer()** on the **FlowConnection**. This will map to a call to **connect_mcast()** on **FlowProducer** with an empty string for the address parameter. This call will return a string with the syntax **<address>** (see “Flow Specification” on page 2-39). If the **connect_mcast()** call returns a string with a full address (including protocol name, transport address, and port number) then internet-style multicasting is in use. The returned string will contain the multicast address of the flow. Any subsequent calls to **add_consumer()** on the **FlowConnection** will result in a call to **go_to_listen()** on the consumer with the multicast address as a parameter. If, on the other hand, a string was returned by the call to **add_producer()** which contains only the **<carrier_protocol>** part of the **<address>** syntax, then ATM-style multicast is in use. Subsequent calls to **add_consumer()** will use **set_protocol_restriction()** calls on the consumer with the name of the returned carrier protocol (e.g., "AAL5"). This will ensure that when **go_to_listen()** is called on the **FlowConsumer** (with **is_mcast** set to true) that it

will be listening on the right protocol. The **FlowConnection** will then call **connect_mcast()** on the **FlowProducer** with listening address returned by **go_to_listen()**.

A suitable stream notation compiler will produce the following interfaces for a flow of type X:

```
interface X_Producer : FlowProducer
{
};


interface X_Consumer : FlowConsumer
{
};"
```

The IDL for **MediaControl** is:

```
enum PositionOrigin {
    AbsolutePosition, RelativePosition, ModuloPosition
};

enum PositionKey {
    ByteCount, SampleCount, MediaTime
};

struct Position {
    PositionOrigin origin;
    PositionKey key;
    long value;
};

exception PostionKeyNotSupported { PositionKey key;};
exception InvalidPosition { PositionKey key;};
exception InvalidTransform {};

// MediaControl interface is similar to
// ControlledStream interface in MSS.
// It can be inherited by flow endpoints or
// FlowConnection interfaces.
interface MediaControl{

    exception PostionKeyNotSupported { PositionKey key;};

    Position get_media_position(
        in PositionOrigin an_origin,
        in PositionKey a_key)
        raises (PostionKeyNotSupported);

    void set_media_position(in Position a_position)
        raises (PostionKeyNotSupported, InvalidPosition);
```

```
    void start(in Position a_position)
        raises(InvalidPosition);

    void pause(in Position a_position)
        raises(InvalidPosition);
    void resumes(in Position a_position)
        raises(InvalidPosition);
    void stop(in Position a_position)
        raises(InvalidPosition);
};
```

A **MediaCtrl** interface can be associated with a **VDev** (using the FlowNameX_related_mediaCtrl property) or **FlowEndPoint** (using the Related_mediaCtrl property) and it may be multiply inherited by specializations of **FlowConnection** or **StreamCtrl**. For example:

```
interface myStreamCtrl : StreamCtrl, MediaControl{};
```

Instances of **myStreamCtrl** allow standardized control over the stored media replay. The importance of this type of scenario merits the inclusion of this interface as standard. Appendix C also deals with this subject.

2.3.8 The FlowConnection

This interface is only implemented by the full profile.

```
interface FlowConnection : PropertyService::PropertySet{
    void stop();
    void start();
    void destroy();

    boolean modifyQoS(inout AVStreams::QoS new_qos)
        raises (AVStreams::QoSRequestFailed);

    boolean use_flow_protocol(
        in string fp_name,
        in any fp_settings)
        raises (FPError, notSupported);

    oneway void push_event(in streamEvent the_event);

    boolean connect_devs(in FDev a_party, in FDev b_party,
        inout QoS the_qos);

    boolean connect(
        in FlowProducer flow_producer,
        in FlowConsumer flow_consumer,
        inout QoS theQoS)
        raises (formatMismatch, FEPMismatch, alreadyConnected);
```

```
    boolean disconnect();
    // The notSupported exception is raised where
    // flow cannot have multiple producers
    boolean add_producer(in FlowProducer flow_producer,
        inout QoS theQoS)
        raises (alreadyConnected, notSupported);

    boolean add_consumer(in FlowConsumer flow_consumer,
        inout QoS theQoS)
        raises (alreadyConnected);

    boolean drop(in FlowEndPoint target)
        raises (notConnected);
};
```

The **FlowConnection** is the flow-level analog of the **StreamCtrl**. A full profile implementation of the **StreamCtrl** which connects two or more stream endpoints which use a flow connection for each of the individual flows within a stream.

The following describes the typical sequence of events in using a full profile **StreamCtrl** to bind to two full profile stream endpoints:

- User A adds some FEPs to an instance of a full profile implementation of **StreamEndPoint_A (myA)** by calling **myA->add_fep(aFEP);**
- User B adds some FEPs to an instance of a full profile implementation of **StreamEndPoint_B (theirB)**.
- User C creates a **StreamCtrl** and calls

  aSC->bind(myA,theirB,someQoS,nilflowSpec)

  The **bind()** algorithm will find which pairs of **FlowEndPoints** are compatible between **myA** and **theirB** and create a **FlowConnection** for each pair of **FlowEndPoints**. Each flow connection within a stream can be individually accessed and manipulated.

A suitable stream notation compiler can generate the following code for a flow of type X:

```
interface X_Connection : FlowConnection {

    boolean connect_X_devs(in F_X a_party, in F_X b_party,
        inout QoS the_qos);

    boolean connect_X(
        in X_Producer flow_producer,
        in X_Consumer flow_consumer,
        inout QoS theQoS)
        raises (formatMismatch, FEPMismatch, alreadyConnected);

    // The notSupported exception is raised where
```

```
    // flow cannot have multiple producers
    boolean add_X_producer(in X_Producer flow_producer,
        inout QoS theQoS)
        raises (alreadyConnected, notSupported);

    boolean add_X_consumer(in X_Consumer flow_consumer,
        inout QoS theQoS)
        raises (alreadyConnected);
};
```

2.3.9 FDev

The **FDev** is only used in the full profile. The IDL is given below:

```
interface FDev : PropertyService::PropertySet {

    FlowProducer create_producer(
        in FlowConnection the_requester,
        inout QoS the_qos,
        out boolean met_qos,
        inout string named_fdev)
        raises(streamOpFailed, streamOpDenied, notSupported,
            QoSRequestFailed);

    FlowConsumer create_consumer(
        in FlowConnection the_requester,
        inout QoS the_qos,
        out boolean met_qos,
        inout string named_fdev)
        raises(streamOpFailed, streamOpDenied, notSupported,
            QoSRequestFailed);

    FlowConnection bind(in FDev peer_device,
        inout QoS the_qos,
        out boolean is_met)
        raises (streamOpFailed, QoSRequestFailed);

    FlowConnection bind_mcast(in FDev first_peer,
        inout QoS the_qos,
        out boolean is_met)
        raises (streamOpFailed, QoSRequestFailed);

    void destroy(in FlowEndPoint the_ep, in string fdev_name)
        // ie FDev not found
        raises (notSupported);
};
```

The **FDev** is exactly analogous in operation to the **MMDevice** for streams.

A stream notation compiler could generate the following specialization of **FDev** for a flow type X:

```
interface F_X : FDev {

    X_Producer create_X_producer(
        in X_Connection the_requester,
        inout QoS the_qos,
        out boolean met_qos,
        inout string named_fdev)
        raises(streamOpFailed, streamOpDenied, notSupported,
            QoSRequestFailed);

    X_Consumer create_X_consumer(
        in X_FlowConnection the_requester,
        inout QoS the_qos,
        out boolean met_qos,
        inout string named_fdev)
        raises(streamOpFailed, streamOpDenied, notSupported,
            QoSRequestFailed);

    X_Connection X_bind(in F_X peer_device,
        inout QoS the_qos,
        out boolean is_met)
        raises (streamOpFailed, QoSRequestFailed);

    X_Connection X_bind_mcast(in F_X first_peer,
        inout QoS the_qos,
        out boolean is_met)
        raises (streamOpFailed QoSRequestFailed);
};
```

2.4 Conformance Criteria

This section summarizes the various levels of conformance to the specification.

2.4.1 Light vs Full Profile

There are two main levels of specification: "light" and "full." The "light" implementation must support the following interfaces, as described in the text:

PropertyService::PropertySet, Basic_StreamCtrl, StreamCtrl, Negotiator, MCastConfigIf, StreamEndPoint, StreamEndPoint_A, StreamEndPoint_B, VDev, MMDevice.

In addition to these, the full profile must support the following interfaces, as described in the text:

FlowConnection, FlowEndPoint, FlowConsumer, FlowProducer, FDev, MediaControl.

The following operations in the light profile will raise the notSupported exception with reason "Full profile not supported."

- In StreamEndPoint: get_fep(), add_fep(), remove_fep()
- In Basic_StreamCtrl: get_flow_connection(), set_flow_connection()

2.4.2 Flow Protocol

Both light and full profiles may optionally support a flow protocol. SFP version 1.0 is specified in this document. A system which does not support SFP causes the exception FPError with reason "No flow protocol supported." If the version of SFP supported is different from that requested, then it uses reason "<flowProtocol> only supported." The following operations raise the FPError exception:

- In Basic_StreamCtrl: set_FPStatus()
- In StreamEndPoint: set_FPStatus(), request_connection()
- In FlowConnection: use_flow_protocol()
- In FlowEndPoint: use_flow_protocol()

2.4.3 Network QoS Parameters

It is mandatory for a streams implementation to support Network QoS parameter set 1 (as discussed in “Device and Stream Parameters” on page 2-25). It is optional to support Network QoS parameter set 2.

Complete OMG IDL — A

A.1 Full IDL

The following is the full IDL for this specification. All IDL compiles using the Orbix 2.2 IDL compiler on NT.

```
#include "PropertyService.idl"

module AVStreams{

struct QoS{
        string QoSType;
        PropertyService::Properties QoSParams;
};

typedef sequence<QoS> streamQoS;

typedef sequence<string> flowSpec;

typedef sequence<string> protocolSpec;

typedef sequence<octet> key;

// protocol names registered by OMG.
// e.g., TCP, UDP, AAL5, IPX, RTP

// This structure is defined for SFP1.0
// Subsequent versions of the protocol may
// specify new structures
struct SFPStatus{
        boolean isFormatted;
        boolean isSpecialFormat;
        boolean seqNums;
```

```
        boolean timestamps;
        boolean sourceIndicators;
};

enum flowState {stopped, started, dead};

enum dirType {dir_in, dir_out};

struct flowStatus{
        string flowName;
        dirType directionality;
        flowState status;
        SFPStatus theFormat;
        QoS theQoS;
};

typedef PropertyService::Property streamEvent;
exception notSupported {};
exception PropertyException {};
// An flow protocol related error
exception FPError { string flow_name; };

exception streamOpFailed{
        string reason;};
exception streamOpDenied{
        string reason;};
exception noSuchFlow{};
exception QoSRequestFailed{
        string reason;};

interface Basic_StreamCtrl : PropertyService::PropertySet {

        // Empty flowSpec => apply operation to all flows
        void stop(in flowSpec the_spec) raises (noSuchFlow);
        void start(in flowSpec the_spec) raises (noSuchFlow);
        void destroy(in flowSpec the_spec) raises (noSuchFlow);

        boolean modify_QoS(inout streamQoS new_qos,
            in flowSpec the_spec)
            raises (noSuchFlow, QoSRequestFailed);

        // Called by StreamEndPoint when something goes wrong
        // with a flow
        oneway void push_event(
                in streamEvent the_event);

        void set_FPStatus(in flowSpec the_spec,
                in string fp_name,      // Only SFP1.0 currently
                                        // specified
                in any fp_settings)     // Currently SFP accepts
                                        // SFPStatus structure
```

```
            raises (noSuchFlow, FPError);

        Object get_flow_connection(in string flow_name)
            raises (noSuchFlow, notSupported);

        void set_flow_connection(in string flow_name,
            in Object flow_connection)
            raises (noSuchFlow, notSupported);
};

interface Negotiator{
        boolean negotiate(in Negotiator remote_negotiator,
            in streamQoS qos_spec);
};

interface VDev;
interface MMDevice;
interface StreamEndPoint;
interface StreamEndPoint_A;
interface StreamEndPoint_B;

interface StreamCtrl : Basic_StreamCtrl {

        boolean bind_devs(in MMDevice a_party, in MMDevice b_party,
            inout streamQoS the_qos,
            in flowSpec the_flows)
            raises (streamOpFailed, noSuchFlow, QoSRequestFailed);

        boolean bind(in StreamEndPoint_A a_party,
            in StreamEndPoint_B b_party,
            inout streamQoS the_qos,
            in flowSpec the_flows)
            raises (streamOpFailed, noSuchFlow, QoSRequestFailed);

        void unbind_party(in StreamEndPoint the_ep,
            in flowSpec the_spec)
            raises (streamOpFailed, noSuchFlow);

        void unbind()
            raises (streamOpFailed);
};

interface MCastConfigIf : PropertyService::PropertySet{

        boolean set_peer(
            in Object peer,
            inout streamQoS the_qos,
            in flowSpec the_spec)
            raises (QoSRequestFailed, streamOpFailed);

        void configure(in PropertyService::Property a_configuration);
```

```
        void set_initial_configuration(
                in PropertyService::Properties initial);

        // Uses <format_name> standardized by OMG and IETF
        void set_format(in string flowName, in string format_name)
                raises (notSupported);

        // Note, some of these device params are standardized by OMG
        void set_dev_params(in string flowName,
                in PropertyService::Properties new_params)
                raises(PropertyService::PropertyException,
                        streamOpFailed);
};

interface StreamEndPoint : PropertyService::PropertySet{

        void stop(in flowSpec the_spec) raises (noSuchFlow);
        void start(in flowSpec the_spec) raises (noSuchFlow);
        void destroy(in flowSpec the_spec) raises (noSuchFlow);

        boolean connect(in StreamEndPoint responder,
                inout streamQoS qos_spec,
                in flowSpec the_spec)
                raises (noSuchFlow, QoSRequestFailed, streamOpFailed);

        boolean request_connection(
                in StreamEndPoint initiator,
                in boolean is_mcast,
                inout streamQoS qos,
                inout flowSpec the_spec)
                raises (streamOpDenied, noSuchFlow,
                        QoSRequestFailed, FPError);

        boolean modify_QoS(inout streamQoS new_qos,
                in flowSpec the_flows)
                raises (noSuchFlow, QoSRequestFailed);

        boolean set_protocol_restriction(in protocolSpec the_pspec);

        void disconnect(in flowSpec the_spec)
                raises (noSuchFlow, streamOpFailed);

        void set_FPStatus(in flowSpec the_spec,
                in string fp_name,
                in any fp_settings)
                raises (noSuchFlow, FPError);

        Object get_fep(in string flow_name)
                raises (notSupported, noSuchFlow);
```

```
        string add_fep(in Object the_fep)
        // Can fail for reasons {duplicateFepName, duplicateRef}
                raises (notSupported, streamOpFailed);

        void remove_fep(in string fep_name)
                    raises (notSupported, streamOpFailed);

        void set_negotiator(in Negotiator new_negotiator);
        void set_key(in string flow_name, in key the_key);
        void set_source_id(in long source_id);
};

interface StreamEndPoint_A : StreamEndPoint{

        boolean multiconnect(inout streamQoS the_qos,
                inout flowSpec the_spec)
                raises (noSuchFlow, QoSRequestFailed, streamOpFailed);
        boolean connect_leaf(in StreamEndPoint_B the_ep,
                inout streamQoS the_qos,
                in flowSpec the_flows)
                raises (streamOpFailed, noSuchFlow,
                QoSRequestFailed, notSupported);

        void disconnect_leaf(in StreamEndPoint_B the_ep,
                in flowSpec theSpec)
                raises(streamOpFailed, noSuchFlow);
};

interface StreamEndPoint_B : StreamEndPoint {

        boolean multiconnect(inout streamQoS the_qos,
                inout flowSpec the_spec)
                raises (streamOpFailed, noSuchFlow,
                        QoSRequestFailed, FPError);
};

interface VDev : PropertyService::PropertySet{

        boolean set_peer(
                in StreamCtrl the_ctrl,
                in VDev the_peer_dev,
                inout streamQoS the_qos,
                in flowSpec the_spec)
                raises (noSuchFlow, QoSRequestFailed, streamOpFailed);

        boolean set_Mcast_peer(in StreamCtrl the_ctrl,
                in MCastConfigIf a_mcastconfigif,
                inout streamQoS the_qos,
                in flowSpec the_spec)
                raises (noSuchFlow, QoSRequestFailed, streamOpFailed);
```

```
          void configure(in PropertyService::Property the_config_mesg)
                    raises(PropertyException, streamOpFailed);

// Uses <formatName> standardized by OMG and IETF
          void set_format(in string flowName, in string format_name)
                    raises (notSupported);

// Note, some of these device params are standardized by OMG
          void set_dev_params(in string flowName,
               in PropertyService::Properties new_params)
               raises(PropertyException, streamOpFailed);

          boolean modify_QoS(inout streamQoS the_qos,
               in flowSpec the_spec)
               raises (noSuchFlow, QoSRequestFailed);
};

interface MMDevice : PropertyService::PropertySet {
          StreamEndPoint_A create_A(
                    in StreamCtrl the_requester,
                    out VDev the_vdev,
                    inout streamQoS the_qos,
                    out boolean met_qos,
                    inout string named_vdev,
                    in flowSpec the_spec)
                    raises(streamOpFailed, streamOpDenied, notSupported,
                              QoSRequestFailed, noSuchFlow);

          StreamEndPoint_B create_B(
                    in StreamCtrl the_requester,
                    out VDev the_vdev,
                    inout streamQoS the_qos,
                    out boolean met_qos,
                    inout string named_vdev,
                    in flowSpec the_spec)
                    raises(streamOpFailed, streamOpDenied, notSupported,
                              QoSRequestFailed, noSuchFlow);

          StreamCtrl bind(in MMDevice peer_device,
                    inout streamQoS the_qos,
                    out boolean is_met,
                    in flowSpec the_spec)
                    raises (streamOpFailed, noSuchFlow, QoSRequestFailed);

          StreamCtrl bind_mcast(in MMDevice first_peer,
                    inout streamQoS the_qos,
                    out boolean is_met,
                    in flowSpec the_spec)
                    raises (streamOpFailed, noSuchFlow, QoSRequestFailed);

          void destroy(in StreamEndPoint the_ep, in string vdev_name)
```

```
                // ie VDev not found
                raises (notSupported);

        string add_fdev(in Object the_fdev)
                raises(notSupported, streamOpFailed);

        Object get_fdev(in string flow_name)
                raises(notSupported, noSuchFlow);

        void remove_fdev(in string flow_name)
                raises (notSupported, noSuchFlow);
};

};

// Additional IDL for full profile
#include "AVStreams.idl"

module AVStreams_Full{

exception protocolNotSupported{};
exception formatNotSupported{};
exception formatMismatch{};
exception FEPMismatch{};
exception alreadyConnected{};
exception invalidSettings{string settings;};
exception notConnected{};
exception deviceQosMismatch{};
exception failedToConnect{string reason;};
exception failedToListen{string reason;};

interface FlowProducer;
interface FlowConsumer;
interface FlowEndPoint;
interface FDev;

interface FlowConnection : PropertyService::PropertySet{
        void stop();
        void start();
        void destroy();

        boolean modify_QoS(

                inout AVStreams::QoS new_qos)
                raises (AVStreams::QoSRequestFailed);

        boolean use_flow_protocol(
                in string fp_name,
                in any fp_settings)
                raises (AVStreams::FPError, AVStreams::notSupported);
```

```
        oneway void push_event(in AVStreams::streamEvent the_event);

        boolean connect_devs(in FDev a_party, in FDev b_party,
                inout AVStreams::QoS the_qos)
                raises (AVStreams::streamOpFailed,
                AVStreams::streamOpDenied,
                AVStreams::QoSRequestFailed);

        boolean connect(
                in FlowProducer flow_producer,
                in FlowConsumer flow_consumer,
                inout AVStreams::QoS the_qos)
                raises (formatMismatch, FEPMismatch, alreadyConnected);

        boolean disconnect();

                // The notSupported exception is raised where
                // flow cannot have multiple producers
        boolean add_producer(in FlowProducer flow_producer,
                inout AVStreams::QoS the_qos)
                raises (alreadyConnected, AVStreams::notSupported);

        boolean add_consumer(in FlowConsumer flow_consumer,
                inout AVStreams::QoS the_qos)
                raises (alreadyConnected);

        boolean drop(in FlowEndPoint target)
                raises (notConnected);
};

interface FlowEndPoint : PropertyService::PropertySet
{
        boolean lock();
        void unlock();

        void stop();
        void start();
        void destroy();

        // Default is a nil object reference
        attribute AVStreams::StreamEndPoint related_sep;
        attribute FlowConnection related_flow_connection;

        FlowEndPoint get_connected_fep()
                raises (notConnected,
                AVStreams::notSupported);

        // syntax of fp_name is <flowProtocol>
        boolean use_flow_protocol(in string fp_name,
                in any fp_settings)
                raises (AVStreams::FPError, AVStreams::notSupported);
```

```
        // set_format() initializes 'format'
        // as current media format e.g. MPEG.
        void set_format(in string format)
                raises (AVStreams::notSupported);

        void set_dev_params(
                in PropertyService::Properties new_settings)
                raises (PropertyService::PropertyException,
                AVStreams::streamOpFailed);

        void set_protocol_restriction(in AVStreams::protocolSpec
                the_spec)
                raises (AVStreams::notSupported);

        boolean is_fep_compatible(in FlowEndPoint fep)
           raises (formatMismatch, deviceQosMismatch);

        boolean set_peer(
                in FlowConnection the_fc,

                in FlowEndPoint the_peer_fep,
                inout AVStreams::QoS the_qos)
                raises (AVStreams::QoSRequestFailed,
                        AVStreams::streamOpFailed);

        boolean set_Mcast_peer(
                in FlowConnection the_fc,
                in AVStreams::MCastConfigIf a_mcastconfigif,
                inout AVStreams::QoS the_qos)
                raises (AVStreams::QoSRequestFailed);
};

interface FlowProducer : FlowEndPoint
{

        boolean connect_to_peer(inout AVStreams::QoS the_qos,
                in string address,

                in string use_flow_protocol) // syntax <flowProtocol>
                raises(failedToConnect,
                AVStreams::FPError, AVStreams::QoSRequestFailed);

        string connect_mcast(inout AVStreams::QoS the_qos,
                out boolean is_met,
                in string address,
                in string use_flow_protocol)
                raises (failedToConnect,
                AVStreams::notSupported,
                AVStreams::FPError,
                AVStreams::QoSRequestFailed);
```

```
        string get_rev_channel(in string pcol_name);

        void set_key(in AVStreams::key the_key);
        void set_source_id(in long source_id);
};

interface FlowConsumer : FlowEndPoint
{

        // Needs to know its peer to choose its protocol correctly
        // Also to ask for a reverse channel for credit-based flow
        // control, if one is required
        string go_to_listen(
                inout AVStreams::QoS the_qos,
                in boolean is_mcast,
                in FlowProducer peer,
                inout string flowProtocol)// syntax <flowProtocol>
                raises(failedToListen, AVStreams::FPError,
                AVStreams::QoSRequestFailed);
};

interface FDev : PropertyService::PropertySet {
        FlowProducer create_producer(
                in FlowConnection the_requester,
                inout AVStreams::QoS the_qos,
                out boolean met_qos,
                inout string named_fdev)
                raises(AVStreams::streamOpFailed,
                        AVStreams::streamOpDenied,
                        AVStreams::notSupported,
                        AVStreams::QoSRequestFailed);

        FlowConsumer create_consumer(
                in FlowConnection the_requester,
                inout AVStreams::QoS the_qos,
                out boolean met_qos,
                inout string named_fdev)
                raises(AVStreams::streamOpFailed,
                        AVStreams::streamOpDenied,
                        AVStreams::notSupported,
                        AVStreams::QoSRequestFailed);

        FlowConnection bind(in FDev peer_device,
                inout AVStreams::QoS the_qos,
                out boolean is_met)
                raises (AVStreams::streamOpFailed,
                AVStreams::QoSRequestFailed);

        FlowConnection bind_mcast(in FDev first_peer,
                inout AVStreams::QoS the_qos,
```

```
                out boolean is_met)
                raises (AVStreams::streamOpFailed,
                AVStreams::QoSRequestFailed);

        void destroy(in FlowEndPoint the_ep, in string fdev_name)
                // ie FDev not found
                raises (AVStreams::notSupported);
};

enum PositionOrigin {
        AbsolutePosition, RelativePosition, ModuloPosition
};

enum PositionKey {
        ByteCount, SampleCount, MediaTime
};

struct Position {
        PositionOrigin origin;
        PositionKey key;
        long value;
};

exception PostionKeyNotSupported { PositionKey key;};
exception InvalidPosition { PositionKey key;};

// MediaControl interface is similar to
// ControlledStream interface in MSS.
// It can be inherited by flow endpoints or
// FlowConnection interfaces.
interface MediaControl{

        exception PostionKeyNotSupported { PositionKey key;};

        Position get_media_position(
                in PositionOrigin an_origin,
                in PositionKey a_key)
                raises (PostionKeyNotSupported);

        void set_media_position(in Position a_position)
                raises (PostionKeyNotSupported, InvalidPosition);

        void start(in Position a_position)
                raises(InvalidPosition);
        void pause(in Position a_position)
                raises(InvalidPosition);
        void resume(in Position a_position)
                raises(InvalidPosition);
        void stop(in Position a_position)
                raises(InvalidPosition);
};
```

```
};
```

Requirements for Control and Management of A/V Streams B

This appendix discusses how the issues raised in the RFP for Control and Management of A/V Streams were addressed by the architecture outlined in this document.

B.1 Topologies

The RFP required a solution to allow one-to-one, one-to-many, many-to-one, and many-to-many sources and sinks to be configured in the same stream binding. The architecture presented in this specification addresses point-to-point and point-to-multipoint configurations which can easily be used as the building blocks of many-to-many and many-to-one streams. For example, the application developer or third party supplier could define an interface **manyPt_Stream** which inherits from **Basic_StreamCtrl** and which supports an operation **join_party(in MMDevice newparty)**. The programmer may then choose to implement many-to-many streams using a video-bridge style device, as illustrated in Figure B-1.

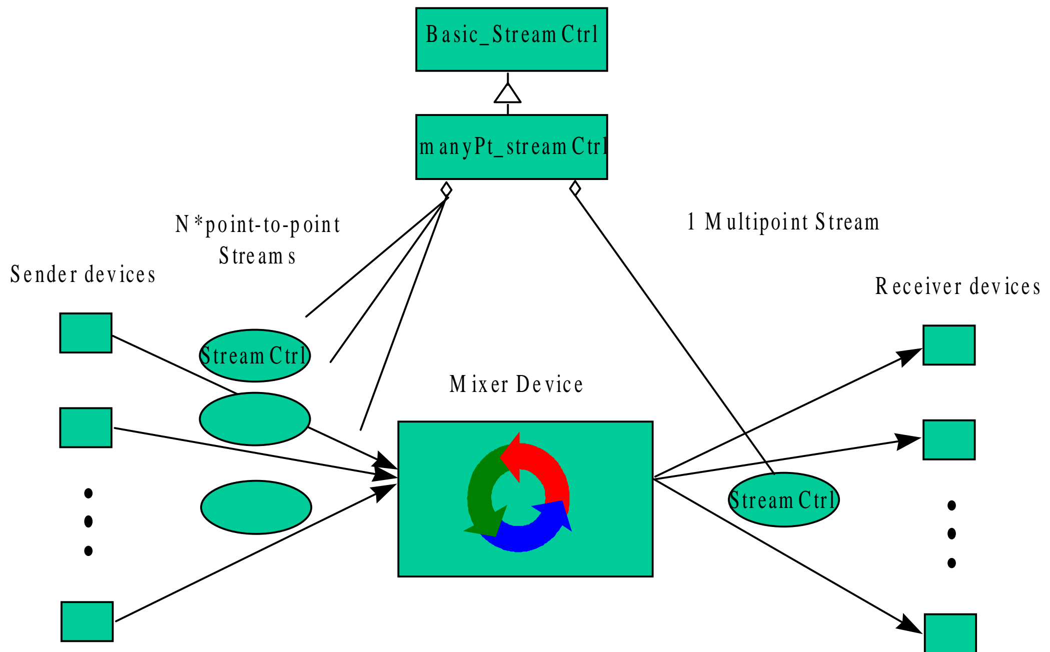

Figure B-1 Using a mixer device for creating multipoint-to-multipoint streams

When the application programmer calls to **stop() on manyPt_streamCtrl** this will result in **stop()** being called on all constituent streams.

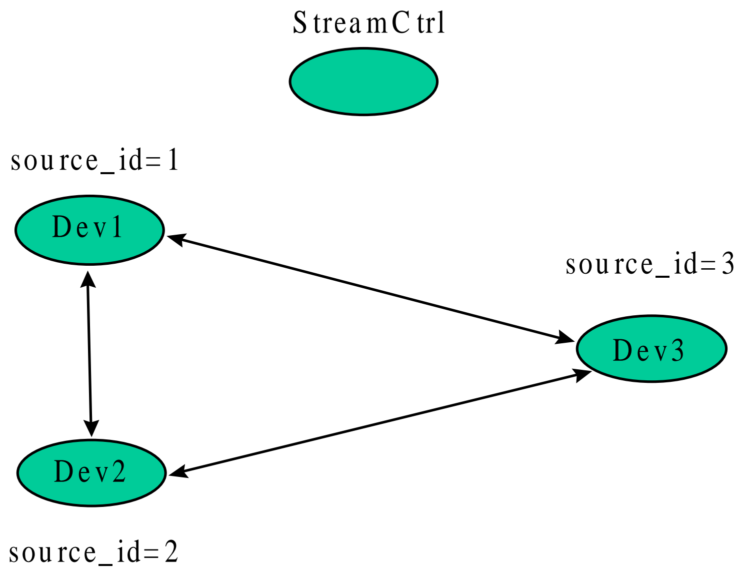

Figure B-2 Using internet multicasting to implement multipoint-to-multipoint streams

Using internet multicasting to support many-to-many streams (see Figure B-2) is even easier. The application programmer uses a call to **set_FPStatus()** to indicate that source-ids should be used. By configuring the A/V Streams implementation, s/he can ensure that UDP is the default protocol. The programmer subsequently calls:

- bind_devs(Dev1,nilObject,…)
- bind_devs(nilObject,Dev1,…)
- bind_devs(Dev2,nilObject,…)
- bind_devs(nilObject,Dev2,…)
- bind_devs(Dev3,nilObject,…)
- bind_devs(nilObject,Dev3,…)

All **StreamEndPoints** will receive data from all other stream endpoints. They will use source-ids to distinguish which frames are from which endpoints. The application must then perform its own mixing of the received data similar to other internet conferencing software.

The important point is that this specification allows multipoint-to-multipoint streams to be built but does not force the use of any one approach. This is a very important area where vendors and third parties can add value.

B.2 Multiple Flows

Multiple flows are an integral part of the solution presented in this specification. A stream may contain any number of flows and an instruction to stop or start can be applied to the stream as a whole or any subset of its flows. A stream endpoint contains a **FlowEndPoint** object for each flow within the stream; however, in the light version of the specification **FlowEndPoints** do not expose their IDL. In the light profile, flows information is accessed instead via the **StreamEndPoint** interfaces.

B.3 Stream Description and Typing

It is beneficial for a stream service implementation to provide a notation to specify the content of flows and their relative direction within a particular stream type. This notation, however, is beyond the scope of this specification. A de facto notation such as TINA-C ODL stream template, for example, could be used. It is expected that another RFP will be issued which will cover the area of stream typing notation and the related language mappings for typesafe insertion and extraction of data to/from a flow.

Although the notation for typing streams is beyond the scope of this document; nevertheless, SFP provides for transportation of IDL typed data. This specification has also described a scheme for creating strictly typed versions of the interfaces for **StreamCtrl, MMDevice, VDev, StreamEndPoint_A, StreamEndPoint_B, FlowConnection, FlowProducer, FlowConsumer,** and **FDev**. This allows typed streams to be dealt with in a standardized way without dictating the notation used to describe streams or the language mappings for inserting/extracting typed information to/from flows.

B.4 Stream Interface Identification and Reference

A reference to a stream is simply a reference to a **StreamCtrl** interface. A reference to a stream endpoint is a reference to a **StreamEndPoint**.

B.5 Stream Setup and Release

Stream setup and release are the main focus of this specification. Streams can be established flexibly using either the **MMDevice::bind()** operations or the **bind_devs()**, bind() family of operations in **StreamCtrl**. There is also support for releasing stream connections using **unbind(), destroy()**, or **unbind_party()**. Individual flows may also be setup and released in both the light and full profiles.

B.6 Stream Modification

One of the principal advantages of streams is that they can be modified flexibly during their lifetime. These modifications include changes to QoS using the **modify_QoS()** operations. An appropriate error is thrown whenever a QoS modification fails. It should be recognized that changes to application level QoS cannot be made in isolation of the **VDevs** being linked by a stream. A **VDev** may reject the QoS modification because it would overload the underlying multimedia device.

All streams can be modified during their lifetime through the addition or removal of individual flows.

Multipoint streams add the flexibility of being able to add or remove A or B parties.

B.7 Stream Termination

The RFP specified that it should be possible to terminate flow-endpoints "in hardware or software." This can be achieved using the architecture outlined in this specification by suitable implementation of the **StreamEndPoint** or **FlowEndPoint**. For example, a **StreamEndPoint_A** or **StreamEndPoint_B** derived object which overrides behaviors in the normal **StreamEndPoint_A** and **StreamEndPoint_B** could execute methods which signal directly to hardware devices on the network. Therefore, the **StreamCtrl** need not be aware of which flows are terminated in hardware and which are terminated in software.

B.8 Multiple Protocols

The specification was designed with multiple protocols in mind. This allows the solution to be applied, in principle, to any transport protocol, including ATM protocols and internet protocols. The specification makes explicit provision for RTP. The protocols to be used for carrying individual streams will be selected at runtime in the process of establishing a connection. The stream endpoint may be restricted in the choice of protocol by explicitly setting a protocol restriction

(**set_protocol_restriction()**). This can be used to prevent an A party from choosing protocols which a B party does not speak. The details of how different transports are abstracted internally to the end-points is an implementation issue.

B.9 Flow Synchronization

Inter-flow synchronization is not tackled within a stream. The position of this specification is that inter-flow synchronization is more properly an issue to be dealt with by the device endpoints. The SFP includes a timestamp field which can be used by the **VDev** implementation at the receiving end to perform synchronization between different flows by reconstructing timing relationships at the source.

B.10 Interoperability

Interoperability in this specification has a number of different aspects:

1. Naming of generated interfaces
2. Compatibility of **StreamEndPoint** implementations
3. Compatibility of SFP
4. Compatibility of common parameters, properties, and syntaxes

The last point is dealt with in Section 2.2.14, “Device and Stream Parameters,” on page 2-25 and in the A/V Streams registration space which is presented in Section 2.2.13, “Extending Stream Management Functionality,” on page 2-25. The syntaxes for common string parameters such as **<flowSpec>** are also defined in Section 2.3, “IDL Interfaces,” on page 2-31.

B.10.1 Naming of Generated Interfaces

These interoperability rules concern the way that a stream notation compiler will generate IDL interfaces. The rules are relatively simple and should be respected by all conforming stream-IDL compilers regardless of the notation which they use to represent streams.

If a stream interface is named X, then the generated IDL interfaces will be:

- **X_StreamCtrl** derived from **StreamCtrl**
- **X_A** and **X_B** derived from **StreamEndPoint_A** and **StreamEndPoint_B**
- **X** derived from **MMDevice**
- **v_X** derived from **VDev**

If a flow type is named Y, then the generated IDL interfaces will be:

- **Y_FlowConnection** derived from **FlowConnection**
- **Y_Consumer** and **Y_Producer** derived from **FlowConsumer** and **FlowProducer**

- **F_Y** derived from **FDev**

A full description of all the IDL generated for each of these is given in Section 2.3, “IDL Interfaces,” on page 2-31.

B.10.2 Compatibility of StreamEndPoints

It should be possible to set up a stream between two device objects even if the two device objects belong to two different implementations of the streaming service. This requires that implementations offer uniform behavior. This is particularly important between **StreamEndPoint** objects in the "light" profile of the service. In particular, the **connect()** call on the "light" **StreamEndPoint** must follow a particular sequence of calls to its peer **StreamEndPoint**. This sequence of operations between the "light" versions of the end-points is detailed in Section 2.3.4, “The StreamEndPoint_A and StreamEndPoint_B,” on page 2-43.

B.10.3 Compatibility of Flow Formats

Compatibility of flow data is not an issue where raw data in an agreed byte layout is being exchanged (e.g., a raw MPEG flow). At a higher level, the architecture provides for extensive negotiation between **VDevs** prior to stream setup which ensures that the source of a flow is producing information in a format which is understanding to the sink of the flow. This **VDev** negotiation is one of the most important aspects of this specification.

Where typed data needs to be transferred or where in-band information such as timestamping is needed, compatibility of flows formats will be assured through the (optional) use of the SFP defined in Section 2.2, “Architecture Overview,” on page 2-3. Figure B-3 depicts the state machine for the sending end of an SFP flow in the point-to-point case. The figure also depicts the state machine for the receiving end in the point-to-point case. The multipoint case is of course much simpler since no information ever travels in the reverse direction. Each transition is labelled by the condition which causes the transition to occur and the action which is performed when the transition is made. Note that the state machines have been slightly simplified in that the sending of fragments is not shown. Also, the machines implementing the protocol at each end will discard any messages which are received when they are not expected (e.g., if the sender gets a credit when it is not expected it will be discarded). The timeout value T2 should be set to be several times the duration of timeout of T1. This allows for several attempts by the other side to send their message.

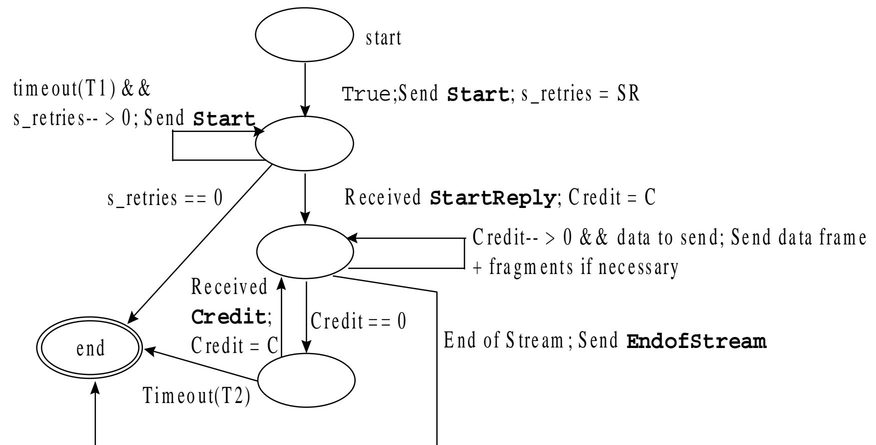

Figure B-3 The state machine for the SFP sender

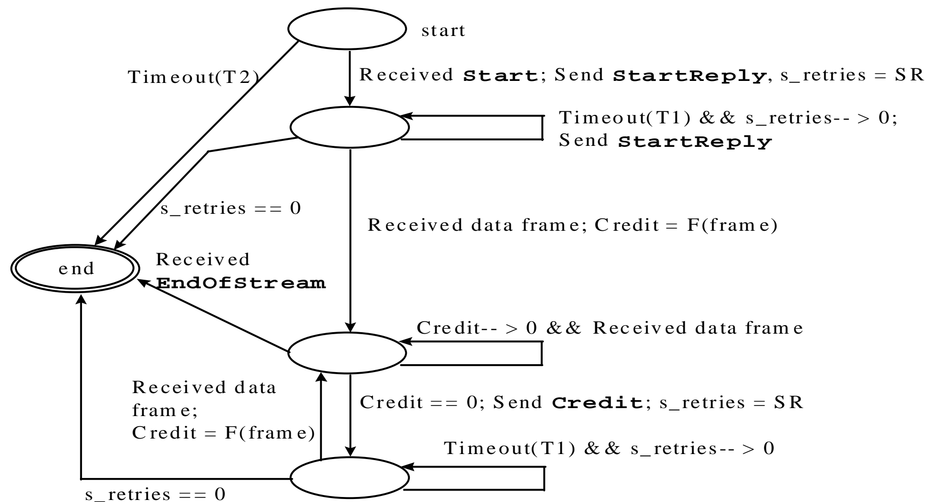

Figure B-4 The state machine for the SFP receiver

B.11 Security

One of the principal advantages of using a distributed streaming framework is that it allows uniform application of security policies to streamed media. Regular security mechanisms can be applied to operations invoked on any of the objects in the media streams framework. The mechanisms for encryption are beyond the scope of this specification; however, the hooks provided by the interfaces should allow encryption to

be applied to selected flows within a stream. The Protection attribute is a registered QoS parameter. In order to protect a flow the application programmer need only specify that the Protection QoS parameter is set to 1 (for encryption). The **set_key()** operation on the **StreamEndPoint** and **FlowEndPoint** then allows the flow/flows to be established or modified to use either public or private key encryption. Where public key encryption is to be used, Public keys can be exchanged via **VDev** negotiation (i.e., the **VDev** finds the value of FlowNameX_PublicKey on its peer and calls the **set_key()** operation with this value.

B.12 The A/V Streams Registration Space

The A/V Streams initial registration space is given below. This information will be maintained in a living document under the OMG Website. The A/V Streams registration space web page will contain instructions on how to register new values.

B.12.1 Carrier Protocol Names

<CarrierName> ::=

"TCP" |

"UDP" |

"AAL5" |

"IPX" |

"RTP/UDP" |

"RTP/AAL5"|

"AAL3-4" | "AAL1"

Other names may be registered via the OMG registration authority.

B.12.2 SFP

The **<flowProtocol>** syntax currently allows the following combinations of values:

<flowProtocolName> ::= "SFP"

The <version> for SFP is currently 1.0.

The only currently defined **<OptionTag>** is "Credit" which takes a non-zero positive integer **<OptionValue>**.

B.12.3 Media Format Categories and MIME Content-Types

<FormatCategory> ::=

"IDL" |

"MIME"

The Media-type registry for audio and video subtypes is currently very sparse. For the purposes of this specification we will temporarily define the following Media-types. These will be officially registered with the IETF in due course:

video/x-mjpeg, the MJPEG standard for video

The following media-types are also supported:

- audio/mpeg
- audio/x-wav, the Microsoft audio format
- audio/x-aiff
- video/x-msvideo, the Microsoft format (e.g., .avi files)
- video/x-mpeg2

B.12.4 QoS Parameters

Names for QoS parameter types and their associated IDL types

```
struct resolution{
    long horz,
    long vert,
};
```

"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_colourDepth"	short (e.g., 2, 4, 8, 12, 16, 24, 32)
"video_colorModel"	short 0 - RGB, 1 - YUV, 2 - HSV
"video_resolution"	struct resolution

The following parameters are also standardized for Network-level QoS (QoSType = "Network_QoS")

Parameter set 1 - Support = Mandatory

"ServiceType"	short, 0 = best effort, 1 = guaranteed, 2 = predicted
"ErrorFree"	boolean
"Delay"	long, milliseconds, desired delay

"Delay_Max"	long, max. acceptable delay
"Bandwidth"	long, bytes per second, desired bandwidth
"Bandwidth_Min"	long, bytes per second, min. acceptable bandwidth
"Bandwidth_Max"	long, max. offered bandwidth
"PeakBandwidth"	long, bytes per second, desired peak bandwidth
"PeakBandwidth_Min"	long, min. acceptable peak bandwidth
"PeakBandwidth_Max"	long, max. offered bandwidth
"TokenRate"	long value, bytes/second
"TokenRate_Min"	long value
"TokenRate_Max"	long value
"TokenBucketSize"	long value, bytes
"TokenBucketSize_Min"	long value
"TokenBucketSize_Max"	long value
"Jitter"	float, microseconds, bounds for delay variation
"Jitter_Max"	float, max. acceptable
"Cost"	float, dimensionless
"Cost_Max"	float, max. acceptable cost
"Protection"	short, 0= default, no encryption, 1= encryption level 1

Parameter set 2 - Support = Optional

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

B.12.5 Device Parameters

The following device parameters are standardized by this document. Similarities to QoS parameters are non-accidental.

"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 - YUV, 2 - HSV
"video_resolution"	- struct resolution

Relationship to DAVIC Work   C

C.1 Overview

The A/V Streams specification deals with a very different set of problems than those addressed by DAVIC's DSM-CC. CORBA is used for all aspects of stream establishment and management with an emphasis on ensuring that heterogeneous devices talking to each other across the stream are compatible. The specification also addresses: multiple protocols, unipoint and multipoint stream topologies and many different types of flow (not just MPEG). It is aimed at the off the shelf multimedia application programmer rather than those engineering Video on Demand services in particular.

Nevertheless, there are areas of overlap with DAVIC's DSM-CC User-User part. It is possible that a DAVIC stream could be managed as a flow within this architecture. In particular, an interface DAVIC_DSMCC_UU could be defined for DSM-CC User-User level operations. In particular, the DAVIC_DSMCC_UU could be used as the media controller for a VDev/FlowEndPoint and/or could be multiply inherited by a specialization of FlowConnection (i.e., interface davicFlowConnection : FlowConnection, DAVIC_DSMCC_UU;).

References D

[1] *Control and Management of A/V Streams*, OMG RFP, telecom/96-08-01

[2] *Integrating Multimedia Streams into a Distributed Computing System*, BJ Murphy and GE Mapp, ORL & Computer Laboratory, Cambridge.

[3] *IMA Recommended Practice, Multimedia Systems Services Part 1: Functional Specification,* September 1994.

[4] *Measuring the Performance of Object-Oriented Components for High-speed Network Programming*, Douglas C. Schmidt.

[5] *RTP: A Transport for Realtime Applications*, RFC1889, March 1996.

[6] *OMG RFP5 Submission, Trading Object Service*, orbos/96-05-06, Version 1.0.0, May 10, 1996.

[7] J. Postel, *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types,* RFC 2046, November 1996.

[8] C. Partridge, *A Proposed Flow Specification*, RFC 1363, September 1992.

[9] *TINA Object Definition Language (TINA-ODL) MANUAL*, PBL01, TINA-C, (www.tinac.com/deliverable/deliverable.htm), July 1996

Index

Symbols
(Virtual) Multimedia Device Interface 2-5

A
A/V Streams Registration Space B-8
A_parties 2-28
AAL5 2-42, 2-50
Active 2-31
add_fep() operation 2-50
add_producer() 2-50
Address 2-31
ATM AAL5 2-16, 2-19
ATM model 2-23
ATM-style multicast 2-50
audio_numChannels 2-29
audio_quantization 2-30
audio_sampleRate 2-29
audio_sampleSize 2-29
AvailableFormats 2-28
AvailableProtocols 2-30, 2-31

B
B_parties 2-28
Bandwidth 2-26
Bandwidth_Min 2-26
Basic stream configuration 2-4
Basic_StreamCtrl 2-31
bind() operation 2-7, 2-35, 2-46
bind_devs() 2-21
bind_devs() operation 2-34
bind_videophone_devs() 2-21
binding 2-5

C
Carrier Protocol Names B-8
CDR encoding 2-18
common device parameters 2-29
Compatibility of Flow Formats B-6
Compatibility of StreamEndPoints B-6
connect() and request_connection() 2-38
connect_mcast() operation 2-50
connect_to_peer() operation 2-50
Connection-Oriented transport 2-16
CORBA
  documentation set 1-2
Cost 2-26
Cost_Max 2-26
create_A() operation 2-45
create_B() operation 2-45
CurrentLoad 2-28, 2-29
CurrFormat 2-31
CurrProtocol 2-31

D
Damage 2-26
Damage_method 2-26
Datagram-oriented transport 2-16
Delay 2-25, 2-27
Delay_Cum 2-27
Delay_Max 2-26, 2-27
destroy() operation 2-32, 2-37, 2-46
Device and Stream Parameters 2-25
Device Parameters B-12
DevParams 2-31
Dir 2-28, 2-31
disconnect_leaf() operation 2-44
Duplication 2-26

E
ErrDamRatio 2-27
ErrDamRatio_Max 2-27
ErrLossRatio 2-27
ErrLossRatio_Max 2-27
ErrorFree 2-25
Establishing a stream (simplified) 2-8

F
FDev 2-9, 2-28, 2-54
Flow 2-28
flow connection 2-4, 2-9
Flow Devices 2-3
flow endpoint 2-4
Flow Endpoints 2-9
Flow Protocol 2-56
Flow Synchronization B-5
FlowConnection 2-28, 2-52
FlowEndPoint 2-30, 2-48
FlowName 2-30
FlowNameX_address 2-30
FlowNameX_availableFormats 2-28, 2-29
FlowNameX_currFormat 2-29, 2-30
FlowNameX_devParams 2-29
FlowNameX_dir 2-28, 2-29, 2-30
FlowNameX_flowProtocol 2-30
FlowNameX_PublicKey 2-28, 2-29, 2-30
FlowNameX_related_mediaCtrl 2-29
FlowNameX_SFPStatus 2-28, 2-29
FlowNameX_status 2-29, 2-30
FlowProtocol 2-31
Flows 2-4, 2-28, 2-29, 2-30
Flows and flow endpoints 2-3
Flows property 2-15
flows vs streams 2-12
flowSpec 2-39
Format 2-31
full profile 2-5

G
get_fep() operation 2-42
get_rev_channel() 2-50
go_to_listen() 2-50

I
IDL Interfaces 2-31
internet model 2-23
Interoperability B-5
is_fep_compatible() 2-50
Issues in Modifying QoS 2-23
Issues in Multipoint Streams 2-23
Ival_Const 2-26
Ival_Max 2-26

J
Jitter 2-26, 2-27

Index

Jitter_Max 2-26, 2-27

L
language 2-29
light profile 2-5
Light vs Full Profile 2-55
lock() operation 2-50
Loss 2-26

M
MaxFlows 2-29
MaxStreams 2-28
Media Format Categories B-8
Media Streaming Framework 2-7, 2-12, 2-21
MIME Content-Types B-8
MMDevice 2-6, 2-28
MMDevice interface 2-44
modify_QoS() operation 2-32, 2-42
Multimedia device 2-3, 2-5
Multiple Protocols B-4

N
Naming of Generated Interfaces B-5
Negotiator 2-30
Network QoS Parameters 2-56
Network_QoS 2-28

O
Object Management Group 1-1
  address of 1-2
octetstream flows 2-21

P
PeakBandwidth 2-26
PeakBandwidth_Min 2-26
peer_device 2-6
PeerAdapter 2-30
Playback_Max 2-27
Playback_Mode 2-27
plug and socket 2-5
point-to-point stream examples 2-20
Profiles 2-5, 2-55
Protection 2-26
ProtocolRestriction 2-30
PublicKey 2-29
push_event() operation 2-33

Q
QoS 2-27
QoS Parameters B-9
Quality of Service (QoS) 2-7

R
Related_mediaCtrl 2-31
Related_MMDevice 2-29
Related_StreamCtrl 2-30
Related_StreamEndPoint 2-29
Related_VDev 2-30
Reorder 2-26
RTP 2-16

S
Security B-7
ServiceType 2-25
set_flow_connection() operation 2-33
set_FPStatus() operation 2-33, 2-43
set_key() operation 2-43
set_Mcast_peer() operation 2-48
set_negotiator() operation 2-42
set_peer() operation 2-48
set_protocol_restriction() operation 2-42
SFP 2-17, 2-41, B-8
SFP dialog 2-19
SFPStatus 2-29, 2-31
Size_avg 2-26
Size_avg_span 2-26
Size_Max 2-26
Size_Min 2-26
source device 2-7
start() operation 2-32, 2-37
Status 2-27, 2-31
stop() operation 2-37
Stream connection compatibility rules can allow unconnected flow endpoints 2-4
Stream Description and Typing B-3
Stream endpoints 2-3
Stream Interface Identification and Reference B-4
Stream Modification B-4
Stream Setup and Release B-4
Stream Termination B-4
StreamCtrl 2-27
StreamCtrl interface 2-6, 2-33
StreamEndPoint 2-30, 2-36
  disconnect() operation 2-44
StreamEndPoint Interface 2-7
Streams 2-3
Switched Virtual Circuit (SVC) 2-24

T
TCP 2-42
TINA-C ODL stream template 2-14
TokenBucketSize 2-26
TokenBucketSize_Min 2-26
TokenRate 2-26
TokenRate_Min 2-26
transport types 2-16
Type 2-27
Type property 2-15

U
UDP 2-19, 2-41, 2-42
unlock() operation 2-50
Unreliable connection-oriented transport 2-16

V
VDev 2-29, 2-47
video_colorDepth 2-30
video_colorModel 2-30
video_framerate 2-30
video_resolution 2-30
Virtual Multimedia Devices 2-3, 2-5

W
Workahead_Max 2-27
Workahead_Mode 2-27