Streams and Sockets in DTU-RTMM

Robin Sharp Hans-Henrik Løvengreen
Edward Todirica
Department of Information Technology
Technical University of Denmark

Version 1.4, September 2000

Abstract

This document describes the concepts used in the stream layer in the DTU-RTMM distributed multimedia systems architecture, and presents standard interfaces for accessing this layer from the session layer which coordinates the transmission of data among the sites involved in the operation of a virtual seminar room.

1 Introduction

In a virtual seminar room, data representing video pictures, still images, sound and other information are passed around among the sites taking part in the seminar, and are presented in real time to the users at these sites to give them the impression of taking part in a seminar or other discussion. In the DTU-RTMM project, which aims at creating an implementation of such a virtual seminar room, the computer system at each site is organised in a layered architecture, as illustrated in Figure 1.

Communication between the computer system at any particular site and its environment, in the form of audio, video or network communication, ultimately takes place via the hardware adaptor cards which are controlled by the drivers in the operating system. We denote the combination of a peripheral unit with its driver a *system component*. In a distributed, interactive multimedia system, such as DTU-RTMM, it is necessary to pass data between the system components, so that for example encoded audio and video can be passed to the network (and vice versa), data can be passed between the network system component in one site to the network system component in another site, and so on. The Stream Layer offers facilities to create and maintain logical connections between system

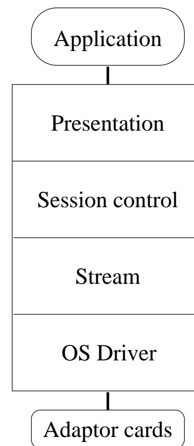

Figure 1: RTMM system architecture

components for this purpose. The Session Layer then coordinates the setting up of those connections which are necessary for particular applications such as the Virtual Seminar Room. Finally, the Presentation Layer provides facilities to implement the user interface, through which audio, video and control information is presented to the users.

This report focuses on the Stream Layer and Session Layer and the way in which they interact. Each layer is presented first in an abstract manner and then in terms of a programming interface which gives access to the facilities offered by the layer in question. The programming interface is documented using the `nuweb` literate programming system, which permits the code and documentation to be extracted directly from a single source.

2 Stream Layer Concepts

The Stream Layer offers the Session Layer facilities for creating and maintaining logical connections through which data may be passed.

2.1 Streams, Sockets and Connections

A *stream* is a logically related sequence of data units which pass between two system components, starting at the *source* of the stream and finishing at its *destination*. To pass a stream between system components, a *connection* must be set up between these components.

A *socket*, through which the stream passes, is used to identify the particular starting point of a stream in the source system component and the particular ending point in the

destination system component. Each socket has a unique identification and is associated with a particular system component. Sockets permit unidirectional flow of data. Sockets through which data pass out of the system component are distinguished from those through which data pass into the component: The socket in the source from which the stream starts is denoted an *OutSocket* and the socket in the destination at which the stream ends is denoted an *InSocket*. For illustrative purposes, these are conveniently designated by graphical symbols as shown in Figure 2.

Figure 2: An OutSocket (left) and an InSocket (right)

When initially created, a connection carries a stream between two sockets: one *OutSocket* and one *InSocket*, as illustrated in Figure 3. The sockets may be associated with the same or different system components. The flow of the stream to the destination *InSocket* can be switched on and off; this is indicated in the figures by a *valve*. When the connection is initially created, the flow is switched *off*.

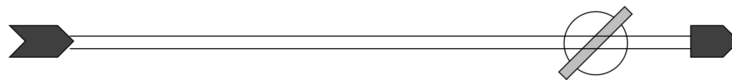

Figure 3: A stream passing on a connection between two sockets

By setting up a connection between an *OutSocket* which is the source of an existing stream to one or more further *InSockets*, a stream may be passed to multiple *InSockets*, as illustrated in Figure 4. The *InSockets* may be associated with the same or with different system components acting as destinations for the stream. Whereas it is in this way possible for

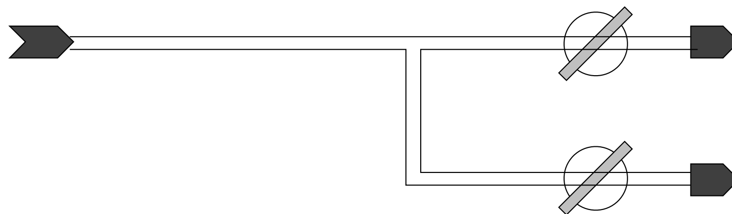

Figure 4: A stream passing between multiple sockets

an *OutSocket* to be the source for a stream with multiple destinations, it is not possible for the same *InSocket* to be the destination for several streams.

A connection passing between two sockets offers a certain *Quality of Service (QoS)* to the streams which it carries. The parameters of the QoS include the bandwidth, delay, jitter and other properties of the connection, and are expected to be compatible with the QoS required for the stream concerned (see Section 2.4 below).

2.2 Socket types

Both *InSocket*s and *OutSocket*s may be of two types:

Local sockets, which are associated with streams that carry data between system components on the same site.

Network sockets, which are associated with streams that carry data between system components on different sites. These system components will in practice always be network system components.

A typical system involving several streams, some with local sockets and some with network sockets is shown in Figure 5. In this example, an audio stream (stream of encoded audio data) originating from the physical microphone could be passed from the Audio System Component to the Network System Component on the Source Site, and from there to the Network System Component on the Destination Site, and then on to the Audio System Component on the Destination Site for presentation on the physical loudspeaker there.

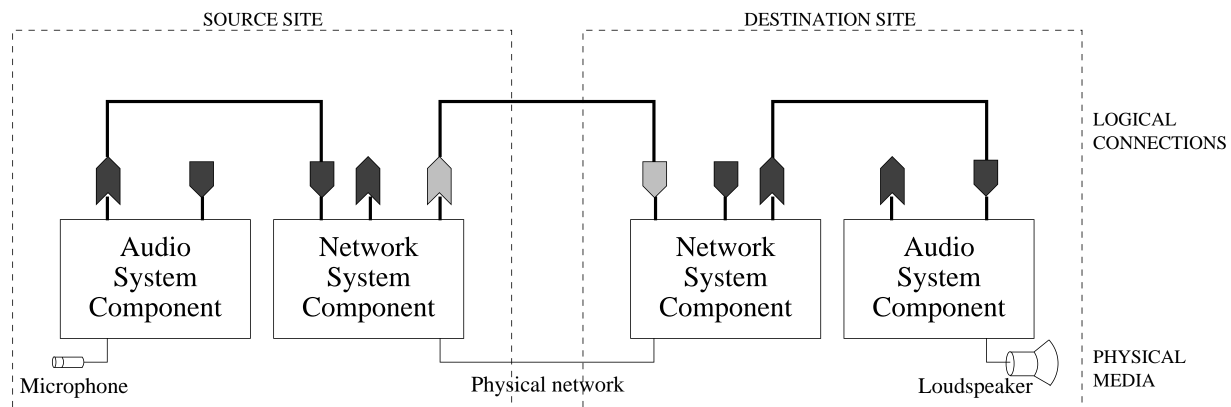

Figure 5: A system with connections between local and network sockets

Local sockets: Network sockets:

Note that the logical connections associated with streams for all types of sockets appear in the Stream Layer, but that they may make use of physical connections, typically via a network, which involve the Driver Layer and associated physical adaptor cards.

2.3 Relaying of Streams

In the version of DTU-RTMM described in this document, streams are considered as passing directly between a source and one or more destination system components. The

destination system component may then be responsible for *relaying* the stream to another system component, as illustrated in the example of Figure 5.

The relaying system component may pass the stream on unaltered, or may apply some *transformation* to the stream. In the latter case, the system component is considered to generate a new stream, for which it is the originating system component. The sequence of connections which through which an unaltered stream is passed is said to make up the *path* followed by the stream, and the component which is the final destination of the stream is denoted the stream's *sink*. Relaying is the concern of the designer of the system component concerned. The current version of the system does not include facilities for directly setting up streams on an end-to-end basis between system components which cannot directly communicate with one another; this is the case if they are on different sites and not directly connected by a network.

2.4 Stream States

A stream has a number of *state components* which can be read and set:

- The *Quality of Service* required for the stream, including in particular its average and maximum delay and jitter, its bandwidth and its error rate, as measured from the original source of the stream to its final destination.
- The *type-specific attributes* of the stream, which describe properties which are relevant to the type of stream concerned, such as the original volume of an audio stream, the title, frame size and original brightness of a video stream and so on.
- The *transmission state* of the stream, which is the the subset of the *InSocket*s associated with the stream which are open for passing data. An *InSocket* may be *open* or *closed*. When it is closed, data passed into it at the *OutSocket* in the source system component will just be ignored.
- The *input data pointer* of the stream, which describes how many data units have been passed into the *OutSocket* in the source of the stream.

Since the QoS and attributes are properties of the stream, they can only be meaningfully set on the site containing the system component from which the stream originates. Adjustment of presentation parameters such as the volume of the audio stream or brightness of the video stream at a given destination site are local matters which lie outside the scope of this document.

2.5 Stream Slots

Data passed into and out of sockets are carried in structures known as *stream slots*. Each stream slot consists of a buffer area, together with information about where in this area actual data and protocol control information are to be found. In the general case, the buffer area looks as illustrated in Figure 6. The aim of this is to permit easy implementation

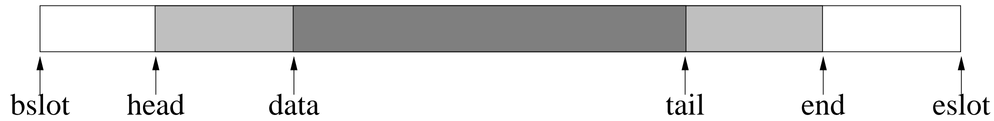

Figure 6: Layout of buffer area in a stream slot

of layered protocol architectures, where protocol control information in a given layer is appended before and after the data passed down from the layer above. The buffer area runs from *bslot* to *eslot*. Valid data (payload) for the device considered occupy the area between *data* and *tail*. Control information can be appended in the *header* area before *data* and in the *trailer* area after *tail*. At any instant, the current extent of the header area runs from *head* to *data*, and the current extent of the trailer area from *tail* to *end*.

3 Formalisation of Streams and Sockets

A more formal specification of what is required of the streams and sockets is presented here in a VDM-like notation describing the data domains involved and the criteria used for delimiting legal objects.

A stream is identified by a unique identifier, $id \in StrId$, and is described by a group of properties:

3.1 $Streams \ = \ StrId \rightarrow StreamProps$
3.2 $StrId \ \subseteq \ \mathsf{token}$

The properties of a stream are its required quality of service, $qos \in QoS$, its attributes, $attrib \in Attributes$, its type, $styp \in StrType$ and the set of connections currently carrying the stream, *state*, which is a possibly empty subset of the domain of *Connection*:

3.3 $StreamProps \ = \ QoS \times Attributes \times StrType \times Connection\text{-}\mathsf{set}$

The required quality of service of a stream is described by a mapping between unique identifiers and QoS parameters with numerical values:

3.4 $QoS \ = \ QoSId \rightarrow QosVal$

3.5 $QoSId \subseteq$ token
3.6 $QosVal = \mathcal{R} \mid \mathcal{Z} \mid \mathcal{N}$

The attributes of a stream are described by a mapping between unique identifiers and parameters of various types:

3.7 $Attributes = AttrId \rightarrow AttribVal$
3.8 $AttrId \subseteq$ token
3.9 $AttribVal = \mathcal{R} \mid \mathcal{Z} \mid \mathcal{N} \mid \mathcal{B} \mid$ token $\mid \ldots$

The type of a stream may be Audio, Video, etc.:

3.10 $StrType = Audio \mid Video \mid \ldots$

A connection is described by six components: the identifier for the associated stream, $sid \in StrId$, the OutSocket, $os \in OutSocket$ from which the connection originates, the InSocket, $is \in InSocket$ at which it ends, the transmission state of the connection, $open \in \mathcal{B}$, which is true if the connection is open for passing the stream and false otherwise, the budgetted quality of service parameters for the connection, $qosb \in QoS$ and the current values of these quality of service parameters, $qosv \in QoS$:

3.11 $Connection = StrId \times InSocket \times OutSocket \times \mathcal{B} \times QoS \times QoS$

A socket is identified by a unique identifier $id \in SockId$, and is associated with a system component, a socket type and a possibly empty set of connections:

3.12 $Socket = SockId \rightarrow SockProps$
3.13 $SockId \subseteq$ token
3.14 $SockProps = Component \times SockType \times Connection$-set

The type of a socket may be a network InSocket, network OutSocket, local InSocket or local OutSocket:

3.15 $SockType = InNet \mid OutNet \mid InLoc \mid OutLoc$

InSockets and OutSockets are disjoint subsets of sockets whose socket types are restricted in the obvious way:

3.16 $InSocket \subseteq Socket$
3.17 $OutSocket \subseteq Socket$
3.18 $InSocket \cap OutSocket = \emptyset$
3.19 is-wf-$InSocket(sid \mapsto sp) \stackrel{\text{def}}{=}$
let mk-$SockProps(comp, styp, cs) = sp$ in
$(styp = InNet) \vee (styp = InLoc)$

$$3.20\quad \textsf{is-wf-}OutSocket(sid \mapsto sp) \stackrel{\textsf{def}}{=}$$
$$\textsf{let mk-}SockProps(comp, styp, cs) = sp \textsf{ in}$$
$$(styp = OutNet) \vee (styp = OutLoc)$$

A system component is identified by a unique identifier $id \in CompId$, and is associated with a possibly empty set of InSockets, a possibly empty set of OutSockets, a possibly empty set of streams originating within the system component, a mapping which describes the current association between streams and the InSockets of the component, and a mapping which describes the current association between streams and the OutSockets of the component:

$$3.21\ Components = CompId \rightarrow Component$$
$$3.22\ CompId \subseteq \textsf{token}$$
$$3.23\ Component = InSocket\textsf{-set} \times OutSocket\textsf{-set} \times StrId\textsf{-set} \times Strmapi \times Strmapo$$
$$3.24\ Strmapi = StrId \rightarrow InSocket$$
$$3.25\ Strmapo = StrId \rightarrow OutSocket$$

A well-formed set of components is such that (1) For each component, the range of the Stream/InSocket mapping is a subset of the component's InSockets and the range of the Stream/OutSocket mapping is a subset of the component's OutSockets, (2) For each component, the domain of the Stream/OutSocket mapping is a subset of the streams originating in the set of components, and (3) For any two different components, their sets of InSockets are disjoint, their sets of OutSockets are disjoint and the sets of streams which they generate internally are disjoint:

$$3.26\quad \textsf{is-wf-}Components(comps) \stackrel{\textsf{def}}{=}$$
$$\forall cid \in \textsf{dom}\ comps \cdot$$
$$\textsf{let mk-}Component(iss, oss, sids, smapi, smapo) = comps(cid) \textsf{ in}$$
$$\textsf{rng}\ smapi \subseteq iss\ \wedge\ \textsf{rng}\ smapo \subseteq oss\ \wedge$$
$$\forall sid \in \textsf{dom}\ smapo \cdot (\exists cid' \in \textsf{dom}\ comps \cdot$$
$$(\textsf{let mk-}Component(iss', oss', sids', smapi', smapo') = comps(cid') \textsf{ in}$$
$$sid \in sids')) \wedge$$
$$\forall cid' \in \textsf{dom}\ comps \cdot$$
$$(\textsf{let mk-}Component(iss', oss', sids', smapi', smapo') = comps(cid') \textsf{ in}$$
$$(cid \neq cid') \Rightarrow$$
$$((iss \cap iss' = \emptyset) \wedge (oss \cap oss' = \emptyset) \wedge (sids \cap sids' = \emptyset))\ \ )$$

A system is composed of system components and connections:

$$3.27\ System = Components \times Connection\textsf{-set}$$

A well-formed system is composed of system components, $comps \in Components$, and connections, $cs \subseteq Connection$, such that the set of components is well-formed, and for each connection, $c \in cs$, (1) The stream identifier identifies a stream originating from

a component in the system, (2) The InSocket and OutSocket belong to components in the system, (3) Whereas the InSocket may be the origin of several connections, a given connection may only be associated with a single stream, and (4) The budgetted and current QoS settings refer to the same QoS parameters:

$$
\begin{array}{l}
3.28 \quad \textsf{is-wf-}\mathit{System}(\mathit{comps}, \mathit{cs}) \stackrel{\textsf{def}}{=} \\
\qquad \textsf{is-wf-}\mathit{Components}(\mathit{comps}) \wedge \\
\qquad \forall c \in \mathit{cs} \cdot \\
\qquad\quad (\textsf{let mk-}\mathit{Connection}(\mathit{sid}, \mathit{is}, \mathit{os}, \mathit{open}, \mathit{qosb}, \mathit{qosv}) = c \textsf{ in} \\
\qquad\quad\; (\exists t \in \textsf{dom } \mathit{comps} \cdot \\
\qquad\qquad \textsf{let mk-}\mathit{Component}(\mathit{iss}, \mathit{oss}, \mathit{sids}, \mathit{smap}) = \mathit{comps}(t) \textsf{ in} \\
\qquad\qquad\; \mathit{sid} \in \mathit{sids}\ ) \\
\qquad\quad\; \wedge (\exists t' \in \textsf{dom } \mathit{comps} \cdot \\
\qquad\qquad\; \textsf{let mk-}\mathit{Component}(\mathit{iss}', \mathit{oss}', \mathit{sids}', \mathit{smap}') = \mathit{comps}(t') \textsf{ in} \\
\qquad\qquad\quad \mathit{is} \in \mathit{iss}'\ ) \\
\qquad\quad\; \wedge (\exists! t'' \in \textsf{dom } \mathit{comps} \cdot \\
\qquad\qquad\; \textsf{let mk-}\mathit{Component}(\mathit{iss}'', \mathit{oss}'', \mathit{sids}'', \mathit{smap}'') = \mathit{comps}(t'') \textsf{ in} \\
\qquad\qquad\quad \mathit{sid} \in \textsf{dom } \mathit{smap}'' \wedge \\
\qquad\qquad\quad \mathit{os} \in \mathit{oss}'' \wedge \\
\qquad\qquad\quad \mathit{os} = \mathit{smap}''(\mathit{sid})\ ) \\
\qquad\quad\; \wedge (\forall c' \in \mathit{cs} \cdot \\
\qquad\qquad\; \textsf{let mk-}\mathit{Connection}(\mathit{sid}', \mathit{is}', \mathit{os}', \mathit{open}', \mathit{qosb}', \mathit{qosv}') = c' \textsf{ in} \\
\qquad\qquad\quad (\mathit{is} = \mathit{is}') \Leftrightarrow (\mathit{sid} = \mathit{sid}')\ ) \\
\qquad\quad\; \wedge (\textsf{dom } \mathit{qosb} = \textsf{dom } \mathit{qosv})\ )
\end{array}
$$

4 Concrete Stream Layer Interfaces

The Stream Layer interfaces are here defined in terms of C++ class definitions. A set of class definitions describe the concepts associated with streams and connections and another set the concepts associated with sockets.

The class definitions make use of definitions from the C++ Standard Template Library and the C++ String Library, and require the inclusion of header files for the STL `map`, `vector` and `string` types. The definitions are documented using the `nuweb` system, enabling the header files to be extracted directly from the documentation as described in Appendix A.

4.1 Debugging, Errors and Exceptions

To ease the task of debugging, a number of conventions have been introduced. Compilation of code which is only to be included and executed in a debugging version of the program can

be controlled by the DEBG variable which is 0 by default, and is set to 1 if the compilation parameter -DDEBUG is used. When DEBG is 1, an output stream debg is defined as being identical to cout, and can be used as the destination for debugging messages, for example:

```
debg << "this is a debugging message" << endl;
```

The variable DEBG can be used in the usual way to ensure conditional compilation of code which is only to be included in the debugging version:

```
if (DEBG)
cout << "This code is only executed in the debugging version" << endl;
```

The required definitions are as follows:

⟨debugging information 10a⟩ ≡

```
//defining a function for printing debugging messages if DEBUG is defined
#ifdef DEBUG
#define DEBG 1
#else
#define DEBG 0
#endif

#define debg if (DEBG) cout
◇
```

Macro defined by scraps 10ab.
Macro referenced in scrap 43.

Many of the classes provide a show method which displays information about the method variables or similar. The show method appends the output to the stream myshow, which will be directed to cout if the -DSHOW compilation parameter is used. The required definitions are:

⟨debugging information 10b⟩ ≡

```
//defining a function for printing messages if SHOW is defined
#ifdef SHOW
#define SHW 1
#else
#define SHW 0
#endif

#define myshow if (SHW) cout
◇
```

Macro defined by scraps 10ab.
Macro referenced in scrap 43.

Several of the classes may throw exceptions, which by convention are defined in a separate class, for example such that exceptions generated in class `XXX` are described in the class `XXX_error`. Objects of the exception class are parameterised by a value of an enumeration type which describes the error and a possibly empty string which can be used to generate error messages. Each exception class provides a method `showError` which displays the error message on standard output, using the string if it is not empty or a default error message if the string is empty.

4.2 Streams and Connections

Objects of the `Connection` class describe logical connections which can be used for passing streams between two sockets. The class provides methods for creating, destroying and manipulating the state of such connections. The Connection class relies on auxiliary classes whose dependencies require them to be defined in the following order:

⟨connection class 11⟩ ≡
    ⟨connection error class 13⟩
    ⟨connection class definition 12⟩
    ◇

Macro referenced in scrap 43.

The main definition of the Connection class is as follows:

⟨connection class definition 12⟩ ≡

```
class Connection
{ // Variables **************************************************************
  public:
    string      streamID;               // Stream identifier
    InSocket*   inSocket;               // Ref. to InSocket object
    OutSocket*  outSocket;              // Ref. to OutSocket object
    QoS*        budgetqos;              // Budget values for QoS parameters
    QoS*        currentqos;             // Current QoS parameter values
    void*       priv;                   // Hidden variables used in the
                                        //   implementation.
  protected:
    bool        open;                   // true if connection open

  // Methods   **************************************************************
  public:
    // Constructors and (alias for) destructor
    Connection(string strID, InSocket* is, OutSocket* os)
                                        throw (Connection_error);
    Connection(string strID, InSocket* is, OutSocket* os, QoS* qosb)
                                        throw (Connection_error);
    void        disconnect();

    // Methods for getting and setting transmission state of connection.
    bool        getConnState();       // Get transmission state
    void        openConnState();      // Set InSocket open
    void        closeConnState();     // Set InSocket closed

    void        show();
}; ◇
```

Macro referenced in scrap 11.

When a connection is initially created, it is closed for transmission of data, and the set of current values for QoS parameters, `qosv`, are all set to 0. If a set of budget values for QoS parameters, `qosb`, is not supplied in the constructor, a default set of budget values is taken from the stream object identified by `streamID`. There is no default constructor with a void argument list.

Errors in handling the setting up of connections cause exceptions described by objects of the class `Connection_error`, which is described as follows:

⟨connection error class 13⟩ ≡

```
class Connection_error
{ public:
    enum Connerrtype {unmatched_strID, not_sink};
  // Variables ***************************************************************
    Connerrtype e_type;
    string      nam;
  // Methods *****************************************************************
    Connection_error(Connerrtype e);          // Constructor initialises
                                              //   e_type to e and nam to
                                              //   the empty string
    Connection_error(Connerrtype e, string s);// Constructor initialises
                                              //   e_type to e, and nam to s
    void showError();
         // Displays on the standard output an error message associated
         //   with the errortype e.  If string s was not empty when the
         //   error was thrown, then s is displayed, otherwise a default
         //   error message is displayed.
         // This method is independent of the -DSHOW compilation parameter.
};
◇
```

Macro referenced in scrap 11.

The individual errors are identified by the elements of the enumeration type `Connerrtype`, of which the following are currently defined:

Error	Caused by...
`unmatched_strID`	Attempt to create a connection between two sockets which are associated with different streams
`not_sink`	Attempt to open a connection for which the input socket is not connected to a sink.

Objects of the `Stream` class describe named streams. The class provides methods for getting access to the objects which describe properties, such as the required end-to-end QoS and other attributes, of such streams. Methods associated with the QoS and Attributes objects make it possible to get and set the values for these properties.

⟨stream class 14⟩ ≡

```
class Stream
{ public:
    enum StrType  { AUDIO, VIDEO, WBOARD }; // Possible types of stream
  // Variables ************************************************************
  public:
    string      streamID;              // Identifier for stream
  protected:
    QoS*        qos;                   // (End-to-end) QoS for stream
    Attributes* attribs;               // Attributes for stream
    StrType     stype;                 // Type of stream
  public:
    void*       priv;                  // Hidden variables used in the
                                       //   implementation.
  // Methods   ************************************************************
  public:
    // Constructor
    Stream(string id, StrType s);      // Initialises streamID=id, stype=s;
                                       //   uses defaults for other members

    // Methods for getting access to QoS and other attributes
    QoS*        getQoS();              // Returns ref. to QoS object
    Attributes* getAttribs();          // Returns ref. to Attributes object
    StrType     getStrtype();          // Returns stream type
};
◇
```

Macro referenced in scrap 43.

As noted previously, the required QoS and other attributes are associated with the stream, and so it only makes sense to attempt to change these properties on the site from which the stream originates.

Objects of the class StreamState describe the dynamic aspects of the states of the local streams on a given site. Streams are identified by their `streamID` component. Methods are provided to retrieve the list of connections which the stream passes through, to derive the list of open InSockets which the stream passes through and to add new connections:

⟨stream dynamic state 15a⟩ ≡

```
class StreamState
{ protected:
    map <string,vector<Connection*>,less<string> > connmap;
                                 // Mapping between local stream identifiers
                                 //   and lists of connections
    void*       priv;            // Hidden variables used in the
                                 //   implementation.

  // Methods   *************************************************************
  public:
    // Constructors *******************************************************
    StreamState();               // Default constructor sets up empty
                                 //   mapping between ids and connections.
    ~StreamState();              // Default destructor
    // Methods for getting and setting components of the dynamic state *****
    void        getConnections(string strID, vector<Connection*>& clist);
                                 // Appends list of Connections through
                                 //   which stream with identifier strID
                                 //   passes to clist
    void        addConnection(string strID, Connection* & conn);
                                 // Adds conn to list of connections for
                                 //   stream with identifier strID
    void        getOpenSockets(string strID, vector<InSocket*>& opensl);
                                 // Appends list of open InSockets through
                                 //   which stream with identifier strID
                                 //   passes to opensl
    void        show();          // Displays member variables of the class.
                                 // To exploit this, the -DSHOW compilation
                                 //   parameter must be used when compiling.
};
◇
```

Macro referenced in scrap 43.

4.3 Sockets

The base class for all socket classes is the abstract class `Socket`, defined by:

⟨socket classes 15b⟩ ≡

```
class Connection;  // to be defined later!
class SysComp;     // to be defined later!

class Socket
{ // Variables *************************************************************
  public:
    enum SockType { INNET, OUTNET, INLOC, OUTLOC };

  protected:
```

```
    SysComp*            comp;               // Component which the socket
                                            //   belongs to
    Connection*         conn;               // Connection associated with
                                            //   socket
    string              socketID;           // Socket identifier
    SockType            socketType;         // Socket type
  public:
    void*               priv;               // Hidden variables used in the
                                            //   implementation.
  // Methods   *************************************************************
  public:
    virtual void        open();             // Creates and opens socket
    virtual void        open(string locid);// Creates and opens socket
                                            //   with given identifier
    virtual void        close();            // Closes and destroys socket
    virtual SysComp*    getComponent();     // Returns ref. to component
                                            //   associated with socket
    virtual void        setComponent(SysComp* comp);
                                            // Sets the component
                                            //   associated with socket
    virtual Connection* getConn();          // Returns ref. to connection
                                            //   associated with socket
    virtual void        setConn(Connection* c);
                                            // Sets the connection
                                            //   associated with socket
    virtual string      getSocketID();      // Returns socket identifier on
                                            //   local system
    virtual void        setSocketID();      // Sets the socket identifier
    virtual SockType    getSocketType();    // Returns the socket type
    virtual Stream*     getStream();        // Returns ref. to the stream
                                            //   associated with socket
    virtual QoS*        getQoSParams();     // Returns ref. to (budget) QoS
                                            //   parameters of associated
                                            //   stream

    // Functions made available for debugging purposes.
    void                show();  // Displays member variables of the class.
                                 // To exploit this, the -DSHOW compilation
                                 //   parameter must be used when compiling.
};
◇
```

Macro defined by scraps 15b, 17ab, 18, 19, 20.
Macro referenced in scrap 43.

The socket identifier identifies the socket uniquely within the component on the local system. The syntax of socket identifiers is given in Section 6.2 below.

InSockets and OutSockets are both Sockets. At the interface to the Stream Layer, it is possible to put data into an OutSocket and get data out of an InSocket. Data are carried

in structures known as *stream slots*, whose properties are described in Section 4.4 below.

⟨socket classes 17a⟩ ≡

```
class InSocket : virtual public Socket
{ // Variables ***************************************************************
  public:

  // Methods   ***************************************************************
  public:
    virtual CSlot* get() = 0;             // Gets one or more stream slots
                                          //   containing data
};

class OutSocket : virtual public Socket
{ // Variables ***************************************************************
  public:

  protected:
    Bufpool* freeSlots;                   // Pointer to buffer pool object

  // Methods   ***************************************************************
  public:
    virtual void put(CSlot* cs) = 0;      // Sends one or more stream slots
                                          //   containing data
};
◇
```

Macro defined by scraps 15b, 17ab, 18, 19, 20.
Macro referenced in scrap 43.

Network sockets are also Sockets, which on creation have an socket identifier valid on the local site. The remote socket to be connected to when a network connection is set up is specified in a method of the class OutNetSocket which is derived from the NetSocket class, as described below.

⟨socket classes 17b⟩ ≡

```
class NetSocket : virtual public Socket
{ // Variables ***************************************************************
  public:

  // Methods   ***************************************************************
  public:
    virtual void open(string locSocketID); // Creates and opens socket
                                          //   with the local socket id
                                          //   locSocketID
};
◇
```

Macro defined by scraps 15b, 17ab, 18, 19, 20.
Macro referenced in scrap 43.

Network InSockets are derived from Network sockets and InSockets. Network OutSockets are derived from Network Sockets and OutSockets. On creation, both these types of socket are given an local socket identifier valid on the site where they are created. Objects of the NetOutSocket class perform a listening function to detect requests for connection coming from remote sites. To set up the actual network connection when a request arrives, the `connect` method in the NetOutSocket is used, specifying the socket identifier on the remote site which is to be connected to.

⟨socket classes 18⟩ ≡

```
class InNetSocket : public InSocket, public NetSocket
{ // Variables *************************************************************
  public:

  // Methods   *************************************************************
  public:
    InNetSocket(string locSocketID);       // Constructor creates and opens
                                           //   InSocket with local socket
                                           //   id locSocketID
    void         open(string locSocketID); // Creates and opens InSocket
                                           //   with local id locSocketID
    CSlot*       get();                    // Gets one or more stream slots
                                           //   containing data
    void         close();                  // Closes and destroys socket
};

class OutNetSocket : public OutSocket, public NetSocket
{ // Variables *************************************************************
  public:

  // Methods   *************************************************************
  public:
    OutNetSocket(string locSocketID);      // Constructor creates and opens
                                           //   socket with local socket id
                                           //   locSocketID
    void         open(string locSocketID); // Opens OutSocket with local
                                           //   id locSocketID, and listens
                                           //   for connection requests
    void         connect(string remSocketID);
                                           // Connects to an InNetSocket
                                           //   which has requested the
                                           //   stream provided by this
                                           //   OutNetSocket
    void         put(CSlot* cs);           // Sends one or more stream slots
                                           //   containing data
    void         close();                  // Closes and destroys socket
};
◇
```

Macro defined by scraps 15b, 17ab, 18, 19, 20.
Macro referenced in scrap 43.

Local sockets are also Sockets, but their derived classes have no means of handling network connections.

⟨socket classes 19⟩ ≡

```
class LocSocket : virtual public Socket
{ // Variables ************************************************************
  public:

  // Methods   ************************************************************
  public:
    virtual void open(string locSocketID); // Creates and opens socket with
                                           //   local socket id locSocketID
};
◇
```

Macro defined by scraps 15b, 17ab, 18, 19, 20.
Macro referenced in scrap 43.

Local InSockets are derived from Local sockets and InSockets. Local OutSockets are derived from Local Sockets and OutSockets.

⟨socket classes 20⟩ ≡

```
class InLocSocket : public InSocket, public LocSocket
{ // Variables ************************************************************
  public:

  // Methods   ************************************************************
  public:
    InLocSocket(string locSocketID);       // Constructor creates and opens
                                           //   InSocket with local socket
                                           //   id locSocketID
    void         open(string locSocketID); // Creates and opens InSocket
                                           //   with local id locSocketID
    CSlot*       get();                    // Gets one or more stream slots
                                           //   containing data
};

class OutLocSocket : public OutSocket, public LocSocket
{ // Variables ************************************************************
  public:

  // Methods   ************************************************************
  public:
    OutLocSocket(string locSocketID);      // Constructor creates and opens
                                           //   OutSocket with local socket
                                           //   id locSocketID
    void         open(string locSocketID); // Creates and opens OutSocket
                                           //   with local id locSocketID
    void         put(CSlot* cs);           // Sends one or more stream slots
                                           //   containing data
};
◇
```

Macro defined by scraps 15b, 17ab, 18, 19, 20.
Macro referenced in scrap 43.

4.4 Stream Slots

The stream slot structures used for passing data into and out of sockets are instances of the class `CSlot`. The variables in objects of this class are as follows:

⟨stream slot class definition 21⟩ ≡

```
class CSlot
{ friend class Bufpool;

  // Variables *************************************************************
  public:
    unsigned char* data;        // Pointer to first octet of data
    unsigned int   dlen;        // Number of octets of data in slot
  private:                      // Pointers to:
    unsigned char* bslot;       //   First octet in buffer area
    unsigned char* head;        //   First octet of header in slot
    unsigned char* tail;        //   First octet of trailer in slot
    unsigned char* end;         //   First unoccupied octet in slot
    unsigned char* eslot;       //   First octet after buffer area
    CSlot*         nextslot;    //   Next slot in list, if any
  public:
    void*          priv;        // Hidden variables used in the
                                //   implementation.
◇
```

Macro defined by scraps 21, 23b, 24ab.
Macro referenced in scrap 25.

It is a requirement that, at all times:

$$\texttt{bslot} \leq \texttt{head} \leq \texttt{data} \leq \texttt{tail} \leq \texttt{end} \leq \texttt{eslot}$$

Likewise, by definition, $\texttt{tail} - \texttt{data} = \texttt{dlen}$. All the methods of the class must maintain these invariants. Errors in handling CSlot objects cause exceptions described by objects of the class `CSlot_error`, which is defined as follows:

⟨stream slot error 22⟩ ≡

```
class CSlot_error
{ public:
    enum Sloterrtype { buffer_full, buffer_empty,
                       header_ovfl, trailer_ovfl, data_ovfl,
                       header_unfl, trailer_unfl, data_unfl,
                       expand_head, expand_data };

  // Variables *************************************************************
  public:
    Sloterrtype e_type;
    string      nam;
  // Methods ***************************************************************
  public:
    CSlot_error(Sloterrtype e);            // Constructor initialises
                                           //   e_type to e and nam to
                                           //   the empty string
    CSlot_error(Sloterrtype e, string s);  // Constructor initialises
                                           //   e_type to e, and nam to s
    void showError();
         // Displays on the standard output an error message associated
         //   with the errortype e.  If string s was not empty when the
         //   error was thrown, then s is displayed, otherwise a default
         //   error message is displayed.
         // This method is independent of the -DSHOW compilation parameter.
};
◇
```

Macro referenced in scrap 25.

The individual errors are identified by the elements of the enumeration type `Sloterrtype`, of which the following are currently defined:

Error	Caused by...	Such that...
`buffer_full`	Adding data	$\texttt{data} < \texttt{bslot} \lor \texttt{tail} > \texttt{eslot}$
`buffer_empty`	Removing data	$\texttt{data} > \texttt{tail}$
`data_ovfl`	Adding data	$\texttt{data} < \texttt{head} \lor \texttt{tail} > \texttt{end}$
`data_unfl`	Removing data	$\texttt{tail} < \texttt{data}$
`header_ovfl`	Adding header	$\texttt{head} < \texttt{bslot} \lor \texttt{head} > \texttt{data}$
`header_unfl`	Removing header	$\texttt{head} > \texttt{data}$
`trailer_ovfl`	Adding trailer	$\texttt{end} < \texttt{tail} \lor \texttt{end} > \texttt{eslot}$
`trailer_unfl`	Removing trailer	$\texttt{end} < \texttt{tail}$
`expand_head`	Expanding header area	Header area overwrites data area
`expand_data`	Expanding data area	Data area overwrites trailer area

The pointers correspond in the natural way to the quantities illustrated in Figure 6, with the convention that pointers to the ends of areas always point to the first octet *after* the area concerned, i.e. the beginning of the next area, if any. Thus, for example, `end` points

to the first octet of the free area after the trailer and `eslot` to the first octet after the end of the buffer area.

Some default sizes for the header and trailer fields are set up by the following definitions:

⟨stream slot defaults 23a⟩ ≡

```
#define DEFAULTSIZE 1000 //Default size used to allocate the buffer from a
                         //CSlot class. In this buffer is stored the data form
                         //the header,tail,and data regions.
#define DEFAULTHEAD 50   //Default size allocated to the header in case no
                         //value is specified for the header length in the
                         //constructor of the CSlot class
#define DEFAULTDATA 0    //Default size allocated to the header in case no
                         //value is specified for the data length in the
                         //constructor of the CSlot class
#define DEFAULTTAIL 10   //Default size allocated to the tail in case no
                         //value is specified for the tail length in the
                         //constructor of the CSlot class
```

◇

Macro referenced in scrap 25.

The constructor and destructor methods of the class are as follows:

⟨stream slot class definition 23b⟩ ≡

```
  // Methods   *************************************************************
  public:
    // Constructors and destructors
    CSlot();                     // Default constructor uses default maximum
                                 //   header and trailer lengths with data
                                 //   length 0.
    ~CSlot();                    // Default destructor.
    CSlot(unsigned int dl);      // Uses default maximum header and trailer
                                 //   lengths with data length dl.
    CSlot(unsigned int hl, unsigned int dl, unsigned int tl);
                                 // Uses header length hl, trailer length tl
                                 //   and data length dl.
```

◇

Macro defined by scraps 21, 23b, 24ab.
Macro referenced in scrap 25.

The following methods can be used to check and/or modify the available space in the stream socket before adding data, headers or trailers:

⟨stream slot class definition 24a⟩ ≡

```
// Methods for checking and changing available space
unsigned int headRoom();     // returns (head - bslot)
unsigned int tailRoom();     // returns (eslot - end)
unsigned int dataRoom();     // returns (eslot - data - deftlen)
                             //   where deftlen is the default trailer
                             //   length.
void expandHead(int hl) throw( CSlot_error );
                             // Increments head, data, tail and end by
                             //   hl, causing exception if buffer area
                             //   contains data or trailer.
void expandData(int dl) throw( CSlot_error );
                             // increments tail and end by dl, causing
                             //   exception if buffer area contains
                             //   trailer.
```

◇

Macro defined by scraps 21, 23b, 24ab.
Macro referenced in scrap 25.

The class also includes a number of methods for adding headers and trailers to the stream slot before passing the slot into the socket, and for stripping them from the stream slot after receiving the slot from the socket:

⟨stream slot class definition 24b⟩ ≡

```
// Methods for adding and removing headers and trailers
void appendToTail(unsigned char* source, unsigned int dl)
             throw( CSlot_error );
                             // Copies dl octets of data from source to
                             //   area starting at end, and increments
                             //   end by dl.
void appendToHead(unsigned char* source, unsigned int dl)
             throw( CSlot_error );
                             // Copies dl octets of data from source to
                             //   area ending at head, and decrements
                             //   head by dl.
void appendToData(unsigned char* source, unsigned int dl)
             throw( CSlot_error );
                             // Copies dl octets of data from source to
                             //   area starting at data+dlen, and
                             //   increments dlen by dl.
void removeFromTail(unsigned char* dest, unsigned int dl)
             throw( CSlot_error );
                             // Copies dl octets of data to dest from
                             //   area ending at end, and decrements
                             //   end by dl.
void removeFromHead(unsigned char* dest, unsigned int dl)
             throw( CSlot_error );
                             // Copies dl octets of data to dest from
                             //   area starting at head, and increments
```

```
                                //   head by dl.
    void removeFromData(unsigned char* dest, unsigned int dl)
                throw( CSlot_error );
                                // Copies dl octets of data to dest from
                                //   area ending at data+dlen, and
                                //   decrements dlen by dl.
    void strip() throw( CSlot_error );
                                // Strips all headers and trailers from
                                //   the slot, leaving head=data and
                                //   tail=end.
    void clean();               // Frees the slot so it can be reused.

    // Functions made available for debugging purposes.
    void                show(); // Displays member variables of the class.
                                // To exploit this, the -DSHOW compilation
                                //   parameter must be used when compiling.
};
◇
```

Macro defined by scraps 21, 23b, 24ab.
Macro referenced in scrap 25.

Note that `appendToHead` and `removeFromTail` are defined in terms of the pointer to the end of the area concerned. In accordance with the convention given previously, this pointer points to the first octet *after* the area concerned.

The CSlot class relies on auxiliary classes whose dependencies require them to be defined in the following order:

⟨stream slot class 25⟩ ≡
⟨stream slot defaults 23a⟩
⟨stream slot error 22⟩
⟨stream slot class definition 21, ... ⟩
◇

Macro referenced in scrap 43.

Pools of stream slots are described by objects of the class `Bufpool`, which is described as follows:

⟨buffer pool class definition 26⟩ ≡

```
class Bufpool
{ // Variables ************************************************************
  private:
    CSlot* buffers;             // Linked list of buffer slots
    int    nbufs;               // Number of buffer slots
    void*  priv;                // Hidden variables used in the
                                //   implementation
  // Methods **************************************************************
  public:
    // Constructors ******************************************************
           Bufpool(int n) throw ( Bufpool_error );
                                // Initialise pool with n default slots,
                                //   throwing exception if not enough space
           Bufpool(int n, unsigned int hl, unsigned int dl, unsigned int tl)
                          throw ( Bufpool_error );
                                // Initialise pool with n slots with
                                //   header length hl, data length dl
                                //   and trailer length tl, throwing
                                //   exception if not enough space
    // Methods for reserving and releasing stream slots *******************
    CSlot* getbuf()       throw ( Bufpool_error );
                                // Get stream slot from pool, throwing
                                //   exception if no slots left in pool
    void   releasebuf(CSlot* cs);// Return stream slot to pool
    int    bufsleft();          // Returns number of slots left in pool
};
◇
```

Macro referenced in scrap 27b.

Errors in handling Bufpool objects cause exceptions described by objects of the class `Bufpool_error`, which is defined as follows:

⟨buffer pool error 27a⟩ ≡

```
class Bufpool_error
{ public:
    enum Bufpoolerrtype { pool_empty, no_space };

  // Variables *************************************************************
  public:
    Bufpoolerrtype e_type;
    string     nam;
  // Methods ***************************************************************
  public:
    Bufpool_error(Bufpoolerrtype e);       // Constructor initialises
                                           //   e_type to e and nam to
                                           //   the empty string
    Bufpool_error(Bufpoolerrtype e, string s);
                                           // Constructor initialises
                                           //   e_type to e, and nam to s
    void showError();
         // Displays on the standard output an error message associated
         //   with the errortype e.  If string s was not empty when the
         //   error was thrown, then s is displayed, otherwise a default
         //   error message is displayed.
         // This method is independent of the -DSHOW compilation parameter.
};
◇
```

Macro referenced in scrap 27b.

The individual errors are identified by the elements of the enumeration type `Bufpoolerrtype`, of which the following are currently defined:

Error	Caused by...
`pool_empty`	Attempt to get buffer from empty pool
`pool_nospace`	Insufficient space to create pool of requested size

The Bufpool class relies on auxiliary classes whose dependencies require them to be defined in the following order:

⟨buffer pool class 27b⟩ ≡

```
⟨buffer pool error 27a⟩
⟨buffer pool class definition 26⟩
◇
```

Macro referenced in scrap 43.

When stream slots are passed between system components, it is the responsibility of the originating system component to allocate space, and the responsibility of the destination system component to pass information back to the originating component when the stream

slot has been emptied, so that the originating component can return the slot to the relevant pool. Mechanisms for this purpose are not specified in the document.

4.5 Quality of Service Parameters

All streams and connections have Quality of Service parameters associated with them. The objects describing individual streams or connections are all instances of the class `QoS`. This makes use of the STL template class `map` to implement a mapping between identifiers for QoS parameters and their types and values. In this way, any set of named parameters with numerical values can be used to describe the QoS for a particular stream or connection.

The QoS class relies on two auxiliary classes, and the dependencies of these classes require them to be defined in the following order:

⟨stream QoS class 28a⟩ ≡
  ⟨QoS parameter 30⟩
  ⟨QoS parameter error 31⟩
  ⟨QoS class definition 28b, ... ⟩
  ◇

Macro referenced in scrap 43.

The main definition of the QoS class is as follows:

⟨QoS class definition 28b⟩ ≡

```
class QoS
{ // Variables ************************************************************
  public:
    map<string,QoSelem*,less<string> > qvals;
                                 // Mapping between names and (type,value)
                                 //   pairs of the QoS parameters
  // Methods **************************************************************
  public:
    // Constructor and destructor
            QoS();               // Creates  QoS object with empty map
            ~QoS();              // Destroys QoS object
```

◇

Macro defined by scraps 28b, 29.
Macro referenced in scrap 28a.

⟨QoS class definition 29⟩ ≡

```
        // Methods for getting and setting QoS values
        void    getQoSvalues(vector<QoSelem*>& v);
                                    // Appends all current (type,value) pairs
                                    //   for QoS parameters from qvals to v
        void    selectQoSvalues(vector<string>& naml, vector<QoSelem*>& v)
                                             throw ( QoSparm_error );
                                    // Appends current (type,value) pairs for
                                    //   QoS parameters whose names are given
                                    //   in naml to v;
                                    // Throws exception if one or more of the
                                    //   named parameters are not defined.
        QoSelem*     getQoSvalue(string nam) throw ( QoSparm_error );
                                    // Returns current (type,value) pair for
                                    //   the QoS parameter with name nam.
                                    // Throws exception if parameter with
                                    //   this name not defined.
        void         setQoSvalue(string nam, QoSelem* tvpair)
                                             throw ( QoSparm_error );
                                    // Sets (type,value) for the QoS parameter
                                    //   with the name nam.
                                    // Throws exception if parameter with
                                    //   this name not defined.
        void         putQoSvalue(string nam, QoSelem* tvpair);
                                    // Sets (type,value) for the QoS parameter
                                    //   with the name nam, inserting a new
                                    //   element in the map if no entry with
                                    //   the given name exists.
    };
    ◇
```

Macro defined by scraps 28b, 29.
Macro referenced in scrap 28a.

The type and value of each QoS parameter is described by an instance of the class `QoSelem`. This uses a `union` type to permit the value field of the QoS parameter to have any of the numerical types `int`, `long int`, `unsigned`, `float` and `double`, together with a type tag, which indicates the type which has currently been allocated to the value field:

⟨QoS parameter 30⟩ ≡

```
class QoSelem
{ public:
    enum qtype_id  { Z, L, N, R, D };

  // Variables *************************************************************
    qtype_id typ;                // Tag for actual type of QoS parameter;
                                 //   Default type = int (code Z)
    union   p_vals {int      i;  // Possible value fields
                    long int j;
                    unsigned k;
                    float    f;
                    double   g;} qval;
  // Methods ***************************************************************
     // Constructors: a default and one for each possible type of value
          QoSelem();             // Default: typ=Z, qval.i=0
          QoSelem(int z);
          QoSelem(long int l);
          QoSelem(unsigned n);
          QoSelem(float r);
          QoSelem(double d);
};
◇
```

Macro referenced in scrap 28a.

The sets of QoS parameters currently defined for audio and video stream QoS objects are as follows:

Identifier	Audio	Video	Type	Description
`maxbandwidth`	+	+	`float`	Maximum bandwidth (bits/s)
`avgbandwidth`	+	+	`float`	Average bandwidth (bits/s)
`maxburstbw`	+	+	`float`	Maximum burst bandwidth (bits/s)
`maxdelay`	+	+	`long int`	Maximum delay (100ns)
`avgdelay`	+	+	`long int`	Average delay (100ns)
`maxjitter`	+	+	`int`	Maximum jitter (100ns)
`avgjitter`	+	+	`int`	Average jitter (100ns)
`errorrate`	+	+	`float`	Error rate (error bits/bit)

The sets of QoS parameters which describe the QoS targets and current properties of connections are to be determined.

Errors in handling QoS objects cause exceptions described by objects of the class `QoSparm_error`, which is defined as follows:

⟨QoS parameter error 31⟩ ≡

```
class QoSparm_error
{ public:
    enum QoSerrtype {undef_QoSparm, muldef_QoSparm, unachiev_QoSparm };

  // Variables ************************************************************
    QoSerrtype e_type;
    string     nam;
 // Methods **************************************************************
    QoSparm_error(QoSerrtype e);          // Constructor initialises
                                          //   e_type to e and nam to the
                                          //   empty string
    QoSparm_error(QoSerrtype e, string s); // Constructor initialises
                                          //   e_type to e, and nam to s
    void showError();
         // Displays on the standard output an error message associated
         //   with the errortype e.  If string s was not empty when the
         //   error was thrown, then s is displayed, otherwise a default
         //   error message is displayed.
         // This method is independent of the -DSHOW compilation parameter.
};
◇
```

Macro referenced in scrap 28a.

The individual errors are identified by the elements of the enumeration type `QoSerrtype`, of which the following are currently defined:

Error	Caused by...
`undef_QoSparm`	Reference to undefined QoS parameter
`muldef_QoSparm`	Insertion of QoS parameter which is already in map
`unachiev_QoSparm`	Attempt to set unachievable value for QoS parameter

The member `nam` of the `QoSparm_error` object is expected to be the string which identifies the QoS parameter, reference to which caused the error.

4.6 Stream Attributes

Many streams have attributes in addition to classical QoS parameters. The class of attributes is defined in a manner analogous to that of QoS parameters, and relies on some auxiliary classes which have to be defined in the following order:

⟨stream attributes class 32a⟩ ≡

```
⟨attribute element 33⟩
⟨attribute element error 34⟩
⟨attributes class definition 32b⟩
◇
```

Macro referenced in scrap 43.

⟨attributes class definition 32b⟩ ≡

```
class Attributes
{ // Variables ***************************************************************
  public:
    map<string,Attrelem*,less<string> > avals;
                                 // Mapping between names and (type,value)
                                 //   pairs of the attributes
  // Methods ****************************************************************
  public:
    // Constructor and destructor
            Attributes();        // Creates Attributes object with empty map
            ~Attributes();       // Destroys Attributes object
    // Methods for getting and setting attribute values
    void    getAttrvalues(vector<Attrelem*>& v);
                                 // Appends all current (type,value) pairs
                                 //   for attributes in avals to v.
    void    selectAttrvalues(vector<string>& naml, vector<Attrelem*>& v)
                                          throw ( Attrib_error );
                                 // Appends current (type,value) pairs for
                                 //   attributes with names in naml to v;
                                 // Throws exception if one or more of the
                                 //   named parameters are not defined.
    Attrelem*    getAttrvalue(string nam) throw ( Attrib_error );
                                 // Returns current (type,value) pair for
                                 //   the Attribute with name nam.
                                 // Throws exception if parameter with
                                 //   this name not defined.
    void         setAttrvalue(string nam, Attrelem* tvpair)
                                          throw ( Attrib_error );
                                 // Sets (type,value) for the Attribute
                                 //   with the name nam.
                                 // Throws exception if parameter with
                                 //   this name not defined.
    void         putAttrvalue(string nam, Attrelem* tvpair);
                                 // Sets (type,value) for the Attribute
                                 //   with the name nam, inserting a new
                                 //   element in the map if no entry with
                                 //   the given name exists.
};
◇
```

Macro referenced in scrap 32a.

The type and value of each attribute is described by an instance of the class `Attrelem`. This uses a `union` type to permit the value field of the attribute to have any of the numerical types `int`, `long int`, `unsigned`, `float` and `double`, or the non-numerical types `bool` or `string`, together with a type tag, which indicates the type which has currently been allocated to the value field:

⟨attribute element 33⟩ ≡

```
class Attrelem
{ public:
    enum atype_id  { Z, L, N, R, D, B, TOK };

  // Variables ***************************************************************
    atype_id typ;                // Tag for actual type of attribute;
                                 //   Default type = int (code Z)
    union   p_vals {int      i;  // Possible value fields
                    long int j;
                    unsigned k;
                    float    f;
                    double   g;
                    bool     t;
                    char*    s;} aval;
  // Methods ****************************************************************
     // Constructors: a default and one for each possible type of value
            Attrelem();          // Default: typ=Z, aval.i=0
            Attrelem(int z);
            Attrelem(long int l);
            Attrelem(unsigned n);
            Attrelem(float r);
            Attrelem(double d);
            Attrelem(bool b);
            Attrelem(char* tok);
};
◇
```

Macro referenced in scrap 32a.

The sets of attributes currently defined for audio and video streams are as follows:

Identifier	Audio	Video	Type	Description
`imageheight`		+	`int`	Height of image (pixels)
`imagewidth`		+	`int`	Width of image (pixels)
`volume`	+		`int`	Volume of audio
`brightness`		+	`int`	Brightness of video
`title`		+	`string`	Title of video

Errors in handling attributes cause exceptions described by objects of the class `Attrib_error`, which is defined as follows:

⟨attribute element error 34⟩ ≡

```
class Attrib_error
{ public:
    enum Attrerrtype {undef_Attrib, muldef_Attrib, unachiev_Attrib };

  // Variables ************************************************************
    Attrerrtype e_type;
    string      nam;
  // Methods **************************************************************
    Attrib_error(Attrerrtype e);           // Constructor initialises
                                           //   e_type to e and nam to the
                                           //   empty string
    Attrib_error(Attrerrtype e, string s); // Constructor initialises
                                           //   e_type to e, and nam to s
    void showError();
         // Displays on the standard output an error message associated
         //   with the errortype e.  If string s was not empty when the
         //   error was thrown, then s is displayed, otherwise a default
         //   error message is displayed.
         // This method is independent of the -DSHOW compilation parameter.
};
◇
```

Macro referenced in scrap 32a.

The individual errors are identified by the elements of the enumeration type `Attrerrtype`, of which the following are currently defined:

Error	Caused by...
`undef_Attrib`	Reference to undefined attribute
`muldef_Attrib`	Insertion of attribute which is already in map
`unachiev_Attrib`	Attempt to set unachievable value for attribute

The member `nam` of the `Attrib_error` object is expected to be the string which identifies the attribute, reference to which caused the error.

4.7 System Components

Objects of the class `SysComp` describe System Components. The definition of this class makes use of auxiliary classes whose dependencies require them to be defined in the following order:

⟨system component class 35a⟩ ≡
    ⟨system component error 42⟩
    ⟨system component definition 35b, ... ⟩
    ◇

Macro referenced in scrap 43.

The member variables of such objects define the sets of InSockets and OutSockets, the set of streams which can be generated by the component concerned, the mappings between streams and sockets, and the (possibly empty) sets of functions which generate, consume and transform streams in the component. The member variables are:

⟨system component definition 35b⟩ ≡

```
class SysComp
{ // Variables *************************************************************
  protected:
    vector<InSocket*>      insocks;     // Set of InSockets
    vector<OutSocket*>     outsocks;    // Set of OutSockets
    vector<string>         genStreams;  // Identifiers for generable streams
    map<string,InSocket*>  strmapi;     // Association between stream ids
                                        //   and InSockets
    map<string,OutSocket*> strmapo;     // Association between stream ids
                                        //   and OutSockets
    map<Socket*,string>    sockmap;     // Inverse mapping giving assoc.
                                        //   between Sockets and stream ids
    vector<pair<InSocket*,OutSocket*> > iomap;
                                        // List of internal connections
                                        //   between InSockets and
                                        //   OutSockets.
    vector<string>         actStreams;  // Identifiers for active streams
    vector<genfunc>        sources;     // Vector of pointers to the
                                        //   functions which generate data
                                        //   for the streams in this
                                        //   component
    vector<genfunc>        sinks;       // Vector of pointers to the
                                        //   functions which consume data
                                        //   from the streams in this
                                        //   component
    vector<genfunc>        transforms;  // Vector of pointers to available
                                        //   transformation functions for
                                        //   this component
  public:
    void*                  priv;        // Hidden variables used in the
                                        //   implementation.
```
◇

Macro defined by scraps 35b, 36, 37, 38, 39, 40, 41.
Macro referenced in scrap 35a.

Objects of the class provide methods for obtaining information about the streams which the component can generate and streams which are currently actively being generated,

and also make it possible to activate and deactivate streams which can be generated by the component. The constructors and the methods for dealing with streams which the component can generate are defined as follows:

⟨system component definition 36⟩ ≡

```
// Methods   ************************************************************
public:
  // Constructors ********************************************************
  SysComp();                  // Default constructor: insocks, outsocks
                              //   genStreams are all empty vectors;
                              //   strmapi, strmapo, iomap are empty maps
  SysComp(vector<string>& gs); // Sets the set of identifiers for
                              //   generable streams to gs, and all other
                              //   variables to "empty" defaults
  ~SysComp();                 // Default destructor.  Removes component

  // Methods for dealing with streams which the component can generate
  void        getStreams(vector<Stream*>& sl );
                              // Returns list of generable streams in sl
  void        getActiveStreams(vector<Stream*>& sl);
                              // Returns list of active streams in sl
  void        activateStream(string strID)   throw ( Component_error );
                              // Activates generation of stream with
                              //   identifier strID.
                              // Throws exception if stream with this
                              //   identifier is undefined or not local
  void        deactivateStream(string strID) throw ( Component_error );
                              // Deactivates generation of stream with
                              //   identifier strID
                              // Throws exception if stream with this
                              //   identifier is undefined or not local
  bool        existsGenStream(string strID);
                              // True if a generated stream with id
                              //   strID exists, otherwise false.
  bool        existsActStream(string strID);
                              // True if an active stream with id
                              //   strID exists, otherwise false.
  string      existsInSockmap(Socket* sock);
                              // Returns the name of the stream
                              //   associated with this socket.
                              // If there is no name associated
                              //   then return NULL
```
◇

Macro defined by scraps 35b, 36, 37, 38, 39, 40, 41.
Macro referenced in scrap 35a.

Both the methods `getStreams` and `getActiveStreams` return a list of streams, which can be searched to find streams of interest in a particular context.

A further group of methods is available for associating new sockets with the streams passing into or out of the component and for obtaining information about which sockets are associated with named streams:

⟨system component definition 37⟩ ≡

```
// Methods for creating new local sockets within the component ***********
OutSocket*   createOutSocket(string locid);
                             // Creates and returns reference to a new
                             //   OutSocket with local socket id locid
                             //   in the component
InSocket*    createInSocket(string locid);
                             // Creates and returns reference to a new
                             //   InSocket with local socket id locid
                             //   in the component
// Methods for creating new network sockets within the component *********
OutNetSocket* createListenSocket(string strID, string sockID);
                             // Creates and returns reference to a new
                             //   network OutSocket with socket
                             //   identifier sockID, which can supply
                             //   the stream with identifier strID
                             //   to clients
InNetSocket* createReceiver(string sockID);
                             // Creates and returns reference to a new
                             //   network InSocket with socket
                             //   identifier sockID
// Methods for dealing with the mappings between streams, InSockets
//   and OutSockets
InSocket*    getInSocket(string strID)      throw ( Component_error );
                             // Returns reference to the InSocket
                             //   via which the stream with identifier
                             //   strID enters the component
                             // Throws exception if stream with this
                             //   identifier is not an incoming stream
OutSocket*   getOutSocket(string strID)     throw ( Component_error );
                             // Returns reference to the OutSocket
                             //   via which the stream with identifier
                             //   strID leaves the component
                             // Throws exception if stream with this
                             //   identifier is not an outgoing stream
void         connectInToOut(string strID, int transformID, void* params,
                            InSocket* is, OutSocket* os)
                                            throw ( Component_error );
                             // Makes internal connection to route the
                             //   incoming stream with identifier strID
                             //   from the InSocket  given by is
                             //   to   the OutSocket given by os,
                             //   using the transformation whose number
                             //   is transformID with parameters params.
                             //   If transformID = 0, the default
```

```
                                    //   (identity) transformation is used.
                                    // Throws exception if stream with this
                                    //   identifier is not an incoming stream
                                    //   or if refs. to sockets are invalid or
                                    //   transformation number is not defined.
        OutSocket*   existsInIomap(InSocket* sock);
                                    // Checks to see if the InSocket sock
                                    //   belongs to an internal connection.
                                    // If so, then a reference to the
                                    //   OutSocket at the other end of
                                    //   the connection is returned.
                                    // If not, NULL is returned.
        InSocket*    existsInOimap(OutSocket* sock);
                                    // Checks to see if the OutSocket sock
                                    //   belongs to an internal connection.
                                    // If so, then a reference to the
                                    //   InSocket at the other end of
                                    //   the connection is returned.
                                    // If not, NULL is returned.
    ◇
```

Macro defined by scraps 35b, 36, 37, 38, 39, 40, 41.
Macro referenced in scrap 35a.

The class also contains methods for reading the content of named hardware registers within the component:

⟨system component definition 38⟩ ≡

```
        // Methods for dealing with hardware registers in the system component ***
        unsigned int getHWProperty(string regid);
                                    // Retrieve content of hardware register
                                    //   with identifier regid
    ◇
```

Macro defined by scraps 35b, 36, 37, 38, 39, 40, 41.
Macro referenced in scrap 35a.

The identifiers used to identify individual registers are not described in this document.

Finally, the class contains a number of useful functions for setting up the sources for streams originating wthin the component and for introducing transformations to create new streams by modifying existing streams which arrive at the component:

⟨system component definition 39⟩ ≡

```
// Methods for setting up the input sources *****************************
void         source(int sourceID, string streamID, void* params,
                    OutSocket* socketID)
                                        throw (Component_error);
                              // Sends the stream generated by the source
                              //   identified by sourceID and identified
                              //   by streamID to the OutSocket given by
                              //   socketID.
                              // Throws exception if source with number
                              //   sourceID is not defined or if stream
                              //   does not have identifier streamID.
void         setupSource(int sourceID, void* params)
                                        throw (Component_error);
                              // Sets up the source parameters.
                              // For each source there are different
                              //   parameters, so for generality's sake
                              //   the type is given as void*.
                              // Throws exception if source with number
                              //   sourceID is not defined
int          registerSource(genfunc func);
                              // Registers the Source function func as a
                              //   source for this component, and returns
                              //   the source number
genfunc      retrieveSource(int sourceID);
                              // Retrieves the source function corres-
                              //   ponding to the source number sourceID
                              // Returns NULL if source with this number
                              //   is not defined
```

◇

Macro defined by scraps 35b, 36, 37, 38, 39, 40, 41.
Macro referenced in scrap 35a.

⟨system component definition 40⟩ ≡

```
// Methods for setting up sinks ****************************************
void        sink(int sinkID, string streamID, void* params,
                        InSocket* socketID)
                                    throw(Component_error);
                            // Starts a thread corresponding to
                            //    sinkID which receives a stream
                            //    identified by streamID from the
                            //    InSocket given by socketID
                            // Throws exception if dest. with number
                            //   sinkID is not defined or if stream
                            //   does not have identifier streamID.
void        setupSink(int sinkID, void* params)
                                    throw(Component_error);
                            // Sets up the sink parameters.
                            // For each source there are different
                            //   parameters, so for generality's sake
                            //   the type is given as void*.
                            // Throws exception if dest. with number
                            //   sinkID is not defined
int         registerSink(genfunc func);
                            // Registers the Sink function func
                            //    as a sink for this component,
                            //    and returns the sink number
genfunc     retrieveSink(int sinkID);
                            // Retrieves the sink function
                            //   corresponding to the sink
                            //   number sinkID
                            // Returns NULL if sink with this
                            //   number is not defined
```

◇

Macro defined by scraps 35b, 36, 37, 38, 39, 40, 41.
Macro referenced in scrap 35a.

⟨system component definition 41⟩ ≡

```
    // Methods for setting up transformations ********************************
    void        setupTransform(int transformID, void* params)
                                                throw(Component_error);
                                // Sets up the transformation parameters.
                                // For each source there are different
                                //   parameters, so for generality's sake
                                //   the type is given as void*.
                                // Throws exception if transform with number
                                //   transformID is not defined
    int         registerTransform(genfunc func);
                                // Registers the transformation function
                                //   func and returns the transformation
                                //   number
    genfunc     retrieveTransform(int transformID);
                                // Retrieves the transformation function
                                //   corresponding to the transformation
                                //   number transformID
                                // Returns NULL if transform with this
                                //   number is not defined

    // Auxiliary methods
    void        show();         // Displays member variables of the class.
                                // To exploit this, the -DSHOW compilation
                                //   parameter must be used when compiling.
    void        removeStream(string s);
                                // Removes all occurrences of the stream s
                                //   from the data structures for SysComp.
};
◇
```

Macro defined by scraps 35b, 36, 37, 38, 39, 40, 41.
Macro referenced in scrap 35a.

Errors in handling system components cause exceptions described by objects of the class `Component_error`, which is defined as follows:

⟨system component error 42⟩ ≡

```
class Component_error
{ public:
    enum Comperrtype {undef_stream, nonlocal_stream,
                      not_instream, not_outstream,
                      invalid_socket,
                      source_undefined, sink_undefined,
                      transform_undefined, unmatched_strID};

  // Variables ***************************************************************
    Comperrtype e_type;
    string      nam;
  // Methods *****************************************************************
    Component_error(Comperrtype e);           // Constructor initialises
                                              //   e_type to e and nam to
                                              //   the empty string
    Component_error(Comperrtype e, string s); // Constructor initialises
                                              //   e_type to e, and nam to s
    void showError();
         // Displays on the standard output an error message associated
         //   with the errortype e.  If string s was not empty when the
         //   error was thrown, then s is displayed, otherwise a default
         //   error message is displayed.
         // This method is independent of the -DSHOW compilation parameter.
};
◇
```

Macro referenced in scrap 35a.

The individual errors are identified by the elements of the enumeration type `Comperrtype`, of which the following are currently defined:

Error	Caused by...
`undef_stream`	Reference to stream whose identifier is unknown
`nonlocal_stream`	Reference to stream which is not generable in local component
`not_instream`	Reference to stream as an incoming stream when it is not
`not_outstream`	Reference to stream as an outgoing stream when it is not
`invalid_socket`	Reference to a socket which is not currently valid for the component
`source_undefined`	Reference to a source which has not been defined for this component
`sink_undefined`	Reference to a sink which has not been defined for this component
`transform_undefined`	Reference to a transformation which has not been defined for this component
`unmatched_strID`	Attempt to make internal connection between sockets associated with different streams

The member `nam` of the `Component_error` object is expected to be the string which identifies the stream, reference to which caused the error.

4.8 Class relationships

The class definitions given above are provided in a single header file, `streams.h`, where their dependencies require them to be defined in the following order:

```
"streams.h" 43 ≡
    // Header file for DTU-RTMM Streams and Sockets
    // Version 1.4      Robin Sharp       June 2000
    // ******************************************
    // *** Generated file.       Do not edit.   ***
    // ******************************************

    #include <list>
    #include <string>
    #include <vector>
    #include <map>
    #include <queue>
    #include <semaphore.h>
    #include <stdio.h>

    #ifndef _STREAMS_
    #define _STREAMS_

    ⟨debugging information 10a, ... ⟩

    // genfunc is a type used for generic functions, as required
    //   the pthread package used to create threads.
    typedef void* (*genfunc)(void*);

    ⟨stream slot class 25⟩
    ⟨buffer pool class 27b⟩
    ⟨stream QoS class 28a⟩
    ⟨stream attributes class 32a⟩
    ⟨stream class 14⟩
    ⟨socket classes 15b, ... ⟩
    ⟨connection class 11⟩
    ⟨stream dynamic state 15a⟩
    ⟨system component class 35a⟩
    #endif // _STREAMS_
    ◇
```

5 Session Layer Concepts

The Session Layer is responsible for setting up suitable connections for distributing the streams generated by the system components on the various sites. The Session Layer offers facilities for registering and deregistering system components and the streams which they generate. The detailed implementation of the registry used for this purpose will not be described in this document.

Streams are identified in the Session Layer by *Stream Identifiers*, each of which is constructed from a *Site Identifier*, which identifies the physical computer system in which the stream originates, together with a *Local Stream Identifier*, which uniquely identifies the stream within this system. Likewise, system components are identified by *Component Identifiers*, which are constructed from a *Site Identifier* and a *Local Component Identifier*.

The general procedure for setting up a logical connection for distributing a stream is based on the principle that the site, say Site A, in which the stream originates makes knowledge of this stream available to all other systems. Another site, say Site B, which wishes to receive the stream must then send a request to the originating site, A, asking for the stream to be directed to B. Both A and B are expected to set up any necessary internal connections between their network system components and other physical system components acting as the ultimate source or destination for the stream, as indicated in Figure 5.

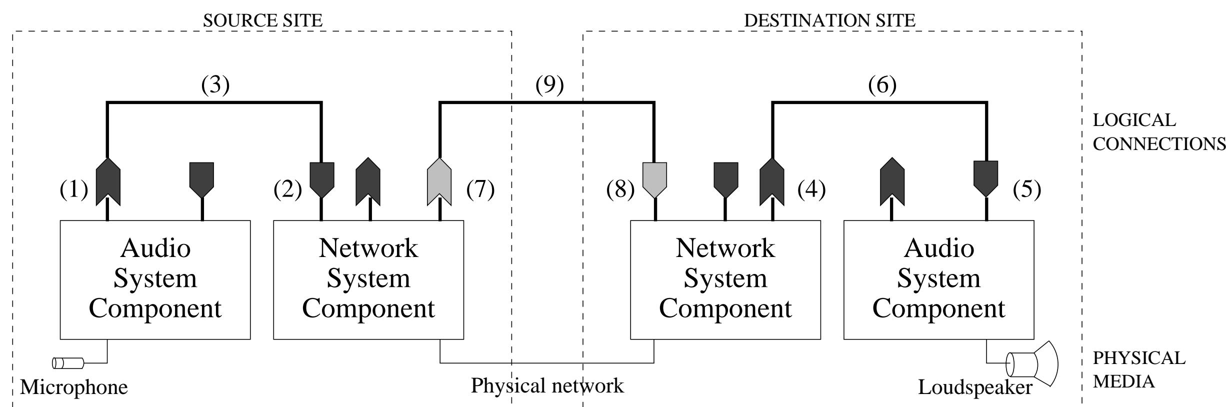

Figure 7: Setting up connections for distributing a stream

Informally, the procedure can be described, with reference to Figure 7, as follows:

- On the originating site, each system component which can act as the originator of a stream is assumed to have registered this stream when the system component is started. The Session Layer software is assumed to retrieve information about available streams from the registry. For each available stream which is interesting in the context of the application, the Session Layer software on the originating site will:

1. Create a Local OutSocket, say *los*, for the System Component in which the stream originates.
2. Create a Local InSocket, say *netis*, for a suitable network System Component in the context of the application.
3. Create a Connection between *los* and *netis*.

☐ Inform all other sites that the stream originating from the originating System Component is available.

In terms of the methods defined for system component objects, an example of a code sequence for setting up the required connection is as follows:

```
// Set up data structure with identifiers for locally generated streams
vector<string> locstr_ids;
locstr_ids.insert(ss.end(), "AudioStream2");

// Set up local system components
Syscomp*    network  = new Syscomp();
Syscomp*    audio    = new Syscomp( locstr_ids );

// Set up local sockets and make connection
OutSocket*  los      = audio->createOutSocket("AudioStream2");
InSocket*   netis    = network->createInSocket();
Connection* locaudio = new Connection("AudioStream2", los, netis);

// Set up network socket which can supply the stream
OutSocket*   nos      = network->createListenSocket("AudioStream2","socknam");
```

The mechanism for informing other sites about the existence of the stream lies outside the scope of this document.

- Any other site which receives knowledge of an available stream which is interesting in the context of the application will:

4. Create a Local OutSocket, say *netos*, for a suitable network System Component.
5. Create a Local InSocket, say *lis*, for the System Component which is the ultimate destination for the stream.
6. Create a Connection between *netos* and *lis*.

☐ Send a request to the originating site, asking for the stream to be distributed.

In terms of the methods defined for system component objects, an example of a code sequence for setting up the required connection is as follows:

```
// Set up data structure with identifiers for locally generated streams
vector<string> locstr_ids;
locstr_ids.insert(ss.end(), ..... );

// Set up local system components
Syscomp*    network  = new Syscomp();
Syscomp*    audio    = new Syscomp( locstr_ids );

// Set up network socket which can receive the stream
InSocket*   nis      = network->createReceiver("AudioStream2","socknam");

// Set up local sockets and make connection
InSocket*   lis      = audio->createInSocket("AudioStream2");
OutSocket*  netos    = network->createOutSocket();
Connection* locaudio = new Connection("AudioStream2",lis,netos);
```

The mechanism for discovering that another site can offer a given stream lies outside the scope of this document.

- On receipt of a request for distribution of a stream, the originating site will set up a network connection (9) between a free OutSocket (7) on its network System Component and a free InSocket (8) on the requesting system's network System Component, and will open this connection for transmission of data.

The software in the source and destination systems will decide when to open their respective local internal connections, (3) and (6), for transmission. This decision will depend on the needs of the application.

6 Concrete Session Layer Interfaces

The Session Layer interfaces are here defined in terms of C++ class definitions. At present, only one class is defined specifically for use in the Session Layer: the class Registry. Objects of this class describe System Components and the streams originating in these components, and provide methods for registering new components and their generated streams in a global registry accessible to all systems.

"registry.h" 47 ≡

```
class Registry
{ public:
    enum registry_error { registry_full,          registry_empty,
                          component_already_reg,  stream_already_reg,
                          component_notin_reg,    stream_notin_reg };
  // Variables ************************************************************
  public:

  // Methods   ************************************************************
  public:
    // Constructors and destructors
    Registry();    // Sets up registry with no components or streams
    ~Registry();   // Removes registry

    // Methods for registering and deregistering components and streams
    void registerComp(SysComp* comp, vector<Stream*>& slis)
                 throw( registry_error );
                                // Registers component comp with streams
                                //   given in the list slis.
    void deregisterComp(SysComp* comp)
                 throw( registry_error );
                                // Deregisters component comp.
    addStream(SysComp* comp, Stream* str)
                 throw( registry_error );
                                // Registers new stream str in component
                                //   comp.
    removeStream(SysComp* comp, Stream* str)
                 throw( registry_error );
                                // Deregisters existing stream str in
                                //   component comp.
};
```

◇

The methods of this class may raise exceptions of type `registry_error`, an enumerated type, whose elements have the following significance:

Error	Caused by attempt to...
`registry_full`	Register information when registry is full
`registry_empty`	Deregister information when registry is empty
`component_already_reg`	Register already registered component
`stream_already_reg`	Register already registered stream
`component_notin_reg`	Deregister unregistered component
`stream_notin_reg`	Deregister unregistered stream

6.1 Concrete Syntax for Stream Identifiers

As stated in Section 5 above, a Stream Identifier is composed of a Site Identifier together with a Local Stream Identifer. By convention, the Local Stream Identifier incorporates an identifier for the type of the stream, so that the syntactic form of Stream Identifiers is defined by:

```
StreamID      ::= [ SiteID "." ] LocalStreamID
LocalStreamID ::= StreamType "Stream" StreamSuffix
SiteID        ::= Identifier
StreamType    ::= "Audio" | "Video" | "WhiteBoard" | ...
StreamSuffix  ::= Number | Identifier
Number        ::= DecDigit DecDigit*
Identifier    ::= Letter IdentChar*
DecDigit      ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
Letter        ::= "a" |...| "z" | "A" |...| "Z"
IdentChar     ::= Letter | DecDigit | "_" | "-" | "&" | "'"
```

Examples of Stream Identifiers are therefore:

```
AudioStream2
AudioStreamLeft
VideoStreamMouse_Hole
IMM.VideoStream24
Tour_d'Eiffel.AudioStreamMixte
```

If the `SiteID` element is omitted from `StreamID`, the StreamID is assumed to be one which is valid locally on the site concerned.

6.2 Concrete Syntax for Socket Identifiers

As stated in Section 5 above, a Socket Identifier is composed of a Site Identifier together with a Local Socket Identifer. By convention, the Local Socket Identifier is an unsigned number, so that the syntactic form of Socket Identifiers is defined by:

```
SocketID          ::= ComponentID "." LocalSocketID
ComponentID       ::= [ SiteID "." ] LocalComponentID
SiteID            ::= Identifier
LocalComponentID ::= "Audio" | "Video" | "Slides" | ...
LocalSocketID     ::= Number
Number            ::= DecDigit DecDigit*
Identifier        ::= Letter IdentChar*
DecDigit          ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
Letter            ::= "a" |...| "z" | "A" |...| "Z"
IdentChar         ::= Letter | DecDigit | "_" | "-" | "&" | "'"
```

Examples of Socket Identifiers are therefore:

```
Video.24
Tour_d'Eiffel.Audio.117
IT.Slides.12345
```

If the `SiteID` element is omitted from `ComponentID`, the ComponentID is assumed to be one which is valid locally on the site concerned.

Appendix A Extracting Code and Documentation

The documentation in this report is generated using the `nuweb` literate programming system originally developed by Preston Briggs. This enables code and documentation to be extracted from a single source file, thus encouraging the use of meaningful annotations and facilitating the task of ensuring that modifications to the code are reflected in the documentation.

Portions of code are in `nuweb` known as *scraps*. A scrap may be specified as directly forming part of a named file with source text, or may be defined as making up the body of a named *macro*, which can be inserted within another scrap, as for example in the macro **`QoS parameter error`** on page 30:

⟨QoS parameter error 18⟩ ≡

```
class QoSparm_error
{ public:
    enum QoSerrtype {undef_QoSparm, muldef_QoSparm };

  // Variables ***********************************************************
    QoSerrtype e_type;
    string     nam;
 // Methods *************************************************************
    QoSparm_error(QoSerrtype e, string s); // Constructor initialises
                                          //   e_type to e, and nam to s
};
◇
```

Macro referenced in scrap 14.

As can be seen, macros are given intelligible names explaining the purpose of the code (here **`QoS parameter error`**). The same file name or macro name may be used for several scraps. The body of the file or macro then consists of the bodies of the individual scraps concatenated in the order in which they appear in the text. Scraps are numbered consecutively by the system for reference purposes – for example, the scrap shown here is scrap 18, and is referred to from scrap 14. The symbol ◇indicates the end of the body of the scrap.

The source file for this report is `streams.w` To generate files containing code and a LaTeX source for the documentation, use the `nuweb` system:

```
nuweb -n streams.w
```

If only code files are to be extracted, use the command:

```
nuweb -t streams.w
```

This will generate the header files `streams.h` and `registry.h` described in this document. If only the LaTeX source is required, use the command:

```
nuweb -on streams.w
```

Processing the LaTeX source with LaTeX will produce this report.

Appendix B Index of Symbols

The numbers in this index, which is generated by the `nuweb` system, refers to the *scraps* in which the symbols are defined or referred to. Definitions are indicated by underlined numbers, and references by plain numbers.

activateStream: 36.
addConnection: 15a.
addStream: 47.
appendToData: 24b.
appendToHead: 24b.
appendToTail: 24b.
Attrelem: 32b, 33.
Attrerrtype: 34.
Attributes: 14, 32b.
Attrib_error: 32b, 34.
atype_id: 33.
B: 33.
bslot: 21, 24a.
Bufpool: 17a, 21, 26.
Bufpoolerrtype: 27a.
Bufpool_error: 26, 27a.
bufsleft: 26.
clean: 24b.
close: 15b, 18.
Comperrtype: 42.
Component_error: 36, 37, 39, 40, 41, 42.
connect: 18.
connectInToOut: 37.
Connection: 12, 15ab.
Connection_error: 12, 13.
Connerrtype: 13.
createInSocket: 37.
createListenSocket: 37.
createOutSocket: 37.
createReceiver: 37.
CSlot: 17a, 18, 20, 21, 23a, 23b, 26.
CSlot_error: 22, 24ab.
D: 20, 30, 33.
data: 17a, 18, 20, 21, 23ab, 24ab, 26, 35b, 41.
deactivateStream: 36.
DEBG: 10a.
DEFAULTDATA: 23a.
DEFAULTHEAD: 23a.
DEFAULTSIZE: 23a.
DEFAULTTAIL: 23a.
deregisterComp: 47.
dlen: 21, 24b.
end: 14, 21, 24ab, 37.
eslot: 21, 24a.
existsActStream: 36.
existsGenStream: 36.
existsInIomap: 37.
existsInOimap: 37.
existsInSockmap: 36.
expandData: 24a.
expandHead: 24a.
freeSlots: 17a.
get: 17a, 18, 20.
getActiveStreams: 36.
getAttribs: 14.
getAttrvalue: 32b.

getAttrvalues: 32b.
getbuf: 26.
getComponent: 15b.
getConn: 15b.
getConnections: 15a.
getInSocket: 37.
getOpenSockets: 15a.
getOutSocket: 37.
getQoS: 14.
getQoSParams: 15b.
getQoSvalue: 29.
getQoSvalues: 29.
getSocketID: 15b.
getSocketType: 15b.
getStream: 15b.
getStreams: 36.
getStrtype: 14.
head: 21, 24ab.
headRoom: 24a.
InLocSocket: 20.
InNetSocket: 18, 37.
InSocket: 12, 15a, 17a, 18, 20, 35b, 37, 40.
L: 30, 33.
LocSocket: 19, 20.
N: 30, 33.
NetSocket: 17b, 18.
nextslot: 21.
open: 12, 15a, 15b, 17b, 18, 19, 20.
OutLocSocket: 20.
OutNetSocket: 18, 37.
OutSocket: 12, 17a, 18, 20, 35b, 37, 39.
put: 17a, 18, 20.
putAttrvalue: 32b.
putQoSvalue: 29.
QoS: 12, 14, 15b, 28a, 28b, 29, 30, 43.
QoSelem: 28b, 29, 30.
QoSerrtype: 31.
QoSparm_error: 29, 31.
qtype_id: 30.
R: 30, 33.
registerComp: 47.
registerSink: 40.
registerSource: 39.
registerTransform: 41.
Registry: 47.
releasebuf: 26.
removeFromData: 24b.
removeFromHead: 24b.
removeFromTail: 24b.
removeStream: 41, 47.
retrieveSink: 40.
retrieveSource: 39.
retrieveTransform: 41.
selectAttrvalues: 32b.
selectQoSvalues: 29.
setAttrvalue: 32b.
setComponent: 15b.
setConn: 15b.
setQoSvalue: 29.
setSocketID: 15b.
setupSink: 40.
setupSource: 39.
setupTransform: 41.
show: 12, 15a, 15b, 24b, 41.
showError: 13, 22, 27a, 31, 34, 42.
SHW: 10b.
sink: 40.
Sloterrtype: 22.
source: 24b, 39, 40, 41.
Stream: 12, 14, 15b, 36, 47.
streamID: 12, 14, 39, 40.
StreamState: 15a.
strip: 24b.
StrType: 14.
SysComp: 15b, 35b, 36, 41, 47.
tail: 21, 23a, 24ab.
tailRoom: 24a.
TOK: 33.
transform: 41.
Z: 30, 33.