Simplifying Real-Time Multimedia Application Development Using Session Descriptions

Sarom Ing, Steve Rudkin

BT Laboratories, Martlesham Heath
Ipswich IP5 3RE
England

Abstract

This paper presents a novel approach that simplifies real-time multimedia communication applications development and service provision. Such applications are no longer required to code the creation and management of their real-time communication needs, but are only required to declare them to our middleware. These applications typically require media streams for audio and video, and data channels for control purposes. Moreover they should be able to adapt to available network and host resources. This is particularly important for multi-party applications operating in heterogeneous environments where each party may have different resources available to them. In addition the nature of the heterogeneity may vary over the lifetime of the session for example as network congestion varies or as the terminal resources are shared with other applications or users. A further problem, still, is that the application developer and service provider typically need to address security and charging requirements. The approach being proposed in this paper allows media streams and channels, QoS, security and charging requirements to be specified in a session description for use within the *Multi-party Multimedia Middleware* to invoke appropriate communication, QoS management, charging and security procedures.

Keywords: Session Description Protocol, Real-time Multimedia Communications, and Middleware.

1. Introduction

Three developments are expected to lead to an explosive growth in commercial multimedia multi-party applications on IP networks. Firstly multicast offers an efficient, scalable and robust means of supporting multi-party applications. Within the next few years a large proportion of the internet is expected to support native multicast. Secondly, the IETF's (Internet Engineering Task Force) work on differential services promises a number of classes of service offered over one IP network. This will greatly facilitate the transmission of real time traffic such as voice, video etc. required for multimedia services. Thirdly, the development of generic Ecommerce services will enable new applications to be offered on a commercial footing without the overhead of developing a service-specific ecommerce capability.

Multi-party applications include: conferencing, networked games, auctions and dealing, shared virtual worlds, as well as *broadcast-like* applications such as video and audio streaming. The current implementation process for multi-party multimedia applications is complex, lengthy and inflexible. This paper addresses three aspects of this complexity. First, development of these applications requires teams with skills in audio/video coding, network transport protocols, real-time programming, user interface design and integration techniques. Our middleware offers the application developer dynamically instantiable real-time stream components that can be simply selected by means of the session description. Second, further complexity arises because multi-party applications must be robust enough to operate in heterogeneous environments (with different terminal and network access capabilities). The nature of the heterogeneity may even vary over the lifetime of the session, for example as network congestion varies or as the terminal resources are shared with other applications or other users. Third, the application development process is further complicated by the need to address security and charging requirements. By including security and charging policies within the session description this complexity can be addressed by the middleware.

The Session Description Protocol (SDP) [1] is a product of the Multiparty Multimedia Session Control (MMUSIC) working group of the IETF. An SDP session description is intended to convey information about media streams[1] and channels in a multimedia session to allow recipients to participate in the session. We are proposing to use SDP announcements for conveying communication, security and charging requirements.

In the next section, we present an overview of SDP. We then present how SDP can be used to simplify the development of robust and flexible real-time multimedia applications. This is followed by a discussion of our prototype middleware design and implementation. Lastly, we contrast our system with related work, and offer conclusions and suggestions for further work.

2. Session Description Protocol (SDP)

SDP is a session description protocol for multimedia sessions. The purpose of SDP is to convey information about media streams in multimedia sessions to allow the recipients of a session description to participate in the session. Normally a session directory tool such as *Session Directory Rendezvous* (SDR) [5] is used to send SDP announcements. These announcements are sent by periodically multicasting an announcement packet on a well-known multicast address and port using the Session Announcement Protocol [6]. SDR is also able to receive announcements from the SAP multicast address and presents them to the users.

SDP is used extensively on the Mbone (which is the part of the internet that supports IP multicast) that permits efficient many-to-many communication. It is used quite regularly for multimedia conferencing. Such conferences usually have the property that tight co-ordination of conference membership is not necessary and anyone receiving the traffic can join the session (unless the traffic is encrypted). However SDP is meant to be general enough for use in tightly controlled sessions (e.g. multimedia conferencing) where participants are to be invited using the Session Initiation Protocol (SIP) [7] for example.

SDP announcements are entirely textual and consist of a session-level and optionally several media-level information. The session-level description provides such information as, the owner/creator details, session identifier, session name, time the session is active and zero or more media description. The media-level description contains the following information:

- Media type and transport address (e.g. audio 3456 RTP/AVP 0)
- Media title
- Connection information (e.g. IN IP4 224.2.17.12/127)
- Bandwidth information
- Encryption key
- Media attributes lines

SDR parses SDP announcements and presents sessions information to the user. When the user chooses to join a session, it parses the media information associated with the session to determine which application(s) to launch. For example it may launch *vic* [2] to handle a video stream, or *vat* [8] to handle an audio stream with an RTP/ format, or wb [3] to handle a shared data channel. It is possible but not very common due to the lack of applications to dynamically map a media format to applications that users are familiar with, for example a video format RTP/AVP may be handled by either *vic* or *Netshow*. This dynamic mapping will benefit end users, as it is possible to enable them to choose their favourite application. It is important to note that once *SDR* has launched an application, the application needs to handle the complexity of establishing and managing its communication channels. It would make sense to provide support for this so that multimedia applications do not have to handle this complexity. The following section will describe how we provide support for this.

3. Declarative Programming

Real-time multimedia communications applications are characterised by their needs to send and receive media streams. Currently, for such applications to receive media streams, they must create suitable

[1]A stream is a channel for continuous data types (e.g. audio or video), whereas a channel can be used for both discrete and continuous data types.

communication channels (e.g. TCP sockets), receive packets of data for those channels, and decodes the data before rendering audio or video signals at the receivers' speakers and monitors. In addition, these applications may have the need to maintain the quality of the audio and video signals in the face of varying resource availability (e.g. network and hosts resources). Furthermore, in the case that there are not adequate resources to handle all media streams, the applications may decide to prioritise these streams. Application programmers have to deal with these issues at the network and operating systems level.

In our declarative approach, real-time communications needs are specified using simple declarative statements. These statements define the number of media channels required, their type (e.g. audio or video), their format (e.g. .au or H263), their connection information, and their quality of service. These statements can also specify the relative priorities of different media channels. SDP can be used for some of the above purposes, however we have also found it necessary to extend SDP to cope with the specification of quality of service and priorities of media streams.

Declarative statements in the form of SDP session description can be processed by our middleware, which is responsible for instantiating the appropriate communication systems components. The approach can also be extended to cover security and charging issues, which may be required by real-time multimedia communications applications. The middleware aims to simplify the development of multi-party multimedia applications by saving the application developer from having to worry about setting up communication channels, robustness in the face of time varying heterogeneity, or charging and security. These are discussed in turn below.

Establishing Communication Channels

We have adopted a declarative approach to simplify the development of real-time multimedia communication applications. Applications can thereby avoid the complexity of creating and managing real-time communication channels. Instead they declare which channels they require, and it is the middleware that handles their creation and management (see Figure 1).

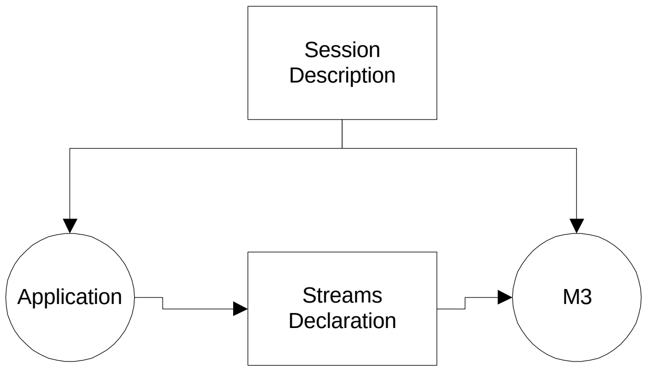

Figure 1 Declarative Programming Model

Currently, when a session directory tool such as *SDR* launches an application such as *vic*, it passes on just the transport address specified in a session description. *Vic* is a monolithic application that needs to deal with the creation of a video *streamgroup*, furthermore it is statistically configured to support just a single video streamgroup (the concept of a streamgroup will be described later). In our approach, a *SDR* would pass the whole session description to our multi-party multimedia middleware (*M3*), which in turns parses the session description and determines that a particular multimedia application is required (e.g. *SharedTV*, see section 5). *M3* then launches *SharedTV* and passes on a list of media streams specified in the session description on. *SharedTV* can parse this list and requests *M3* dynamically to create a subset or all of the streams in the list (as described in the next section). Our approach envisages that an application can use the media streams specification to dynamically instantiate streams implemented as part of a generic middleware.

The dynamic aspect of this approach offers following benefits: providing a suitable means exists for distributing updated session descriptions during the life time of a session, the application can dynamically create and terminate communication channels as required. The power of this simple idea is only apparent when the full range of multiparty multimedia applications is considered. Consider for example a shared virtual world with an audio stream, position data channel and orientation data channel for each room or zone within the virtual world. Zones may include a variety of media sources, which can only be sensed within a certain range. Each such source would require a separate channel or stream. Also it may be possible to establish small or private conversations transmitted on a different audio stream from that of the room. As a user moves around the world, the user's application would continually be changing the number and type of communication channels that it was using.

Robustness

We are also concerned that multimedia applications can be deployed in an environment with different network and host capabilities. Consider a multimedia conferencing application that requires an audio and a video channel. On those hosts with adequate computing and network bandwidth, all these channels could be established. However, on those hosts without adequate resources, one may wish to have both channels but at a lower quality. Unfortunately, this solution may degrade both channels and may render the application unusable, thus it may be better for the application to establish just the audio channel.

These possibilities may be handled by a middleware or be hard coded in the program. The advantages of using a middleware to provide support for the different possibilities are: to produce a robust implementation by reducing the program complexity, and to produce a more stable program as it does not need to be changed because more policies are required.

We have used a *Session QoS Policy* to describe the previously mentioned possibilities. This policy provides guidelines to the middleware for the creation and management of the session's media channels. The policy comprises of a mandatory list and an ordered optional list. The mandatory list specifies those channels that are crucial for the proper functioning of an application. The optional list is an ordered list and specifies those channels that are not crucially required, when resources are scarce then media streams at the head of the list is created first. Consider the previous multimedia conference, the audio channel might be essential whereas the video and shared whiteboard data channels might be optional. In addition the shared whiteboard data channel might be more important than the video channel so that if there is network congestion then degrading video quality would be preferred.

Charging and Security

Service providers want to be able to offer new multimedia services without the overhead of developing a service specific electronic commerce capability. This is leading to the growth of generic Ecommerce services like electronic payment services [9] and trusted third parties [10]. By including charging and security policies within the session description it becomes straightforward to invoke such services from our middleware. The Session Directory tool automatically passes these policies to the middleware so the application developer does not have to be concerned with the charging and security issues.

The advantages are clear. The application developer can leave the middleware to implement charging and security procedures. The service provider simply selects appropriate charging and security procedures through the charging and security policies. This means that the application developer and the service provider can focus on higher level aspect of their application or service. Moreover the applications can be used in a variety of different service contexts (e.g. internet or intranet, charged or free). The model also would allow innovative new service providers to enter the market easily and communities of users to establish their own sessions without a service provider in any traditional sense.

We have described how we intend to use SDP to facilitate the development of robust, flexible and extensible multimedia applications. In the next section, we shall describe our prototype implementation of the *M3* system, note however that the current prototype does not yet support the charging and security policies.

4. Multi-party Multimedia Middleware (M3) Architecture

A session directory tool will contact *M3* to start a session and passes to *M3* a session description. *M3* parses the description to determine which applications are required and launches them. These applications then interact with M3 whenever it wishes to create media channels. *M3* is responsible for creating and managing the quality of service for those channels and may also contact other services such as the authentication, accounting and payment services. In this prototype we have only dealt with media channels creation and managing their quality of service, and these aspects will be described below.

The *M3* architecture (grey area) is primarily peer to peer, that is to say that each host will communicate directly with one another rather than going via a central server. Each host is structured as shown in Figure 2.

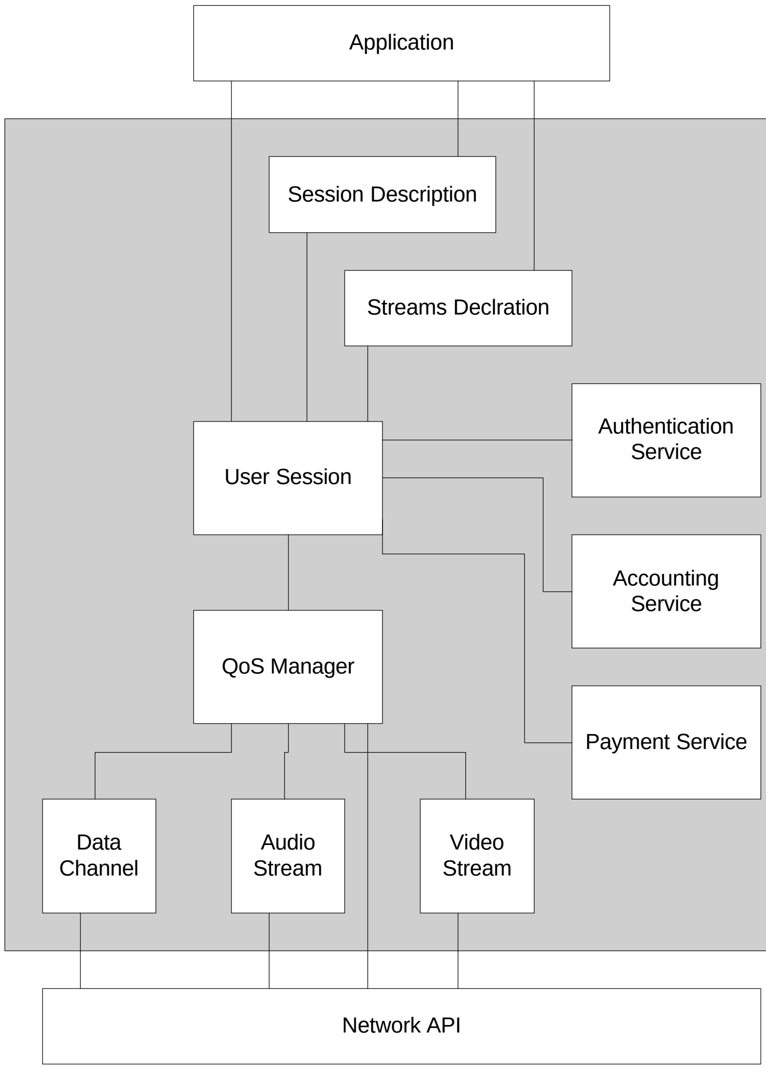

Figure 2 Applications Architecture

Figure 2 shows an application interacts directly with three objects: the *Session Description*, *Stream Declaration* and the *User Session*. These and other key objects are described below.

Session Description Object

Our session directory tool (SDTool) parses session announcements into *Session Description* objects. These objects provide convenient methods to access attributes of a session (e.g. the session name, owner, start time, a list of media streams, Session QoS Policy, and etc.), please refer to Figure 3 below. The media streams list comprises of media media objects. These objects have convenient methods for accessing such attributes as their identifiers, multicast address and port, quality of service mechanisms (e.g. RSVP, layered encoding) and their associated specification (i.e. for layered encoding a list of multicast address, port numbers and bandwidth value are specified). The Session QoS Policy object provides methods to access media identifiers according to priority rules contained within the *Session QoS Policy* object.

When a user wishes to join a session, the *SDTool* contacts the *User Session* object to start that session and passes the appropriate *Session Description* object on.

User Session Object

The *User Session* parses the given *Session Description* object to determine which application to launch. It does so by inspecting stepping through the media list and inspecting the media client (e.g. *sharedTV*, *Netshow* and etc.). If no particular media client is specified then it inspects the media format (e.g. *mpeg*, *au*, *dvi* and etc.). *The User Session* then launches an appropriate application that can handle the particular format.

An application such as the *sharedTV* application is implemented using the *M3* middleware. Such an application interacts with the middleware for the creation and management of communication channels. The application interacts with the middleware via an API: e.g. joining and leaving a session, adding and removing media channels to and from a session, getting membership and statistical information of a session (see Figure 3 below).

When the application wishes to create an initial set of media channels, it sends a *join session* request to the *User Session*. It passes as argument a *Streams Declaration* object that it creates after having parsed the list of media streams given to it by the *User Session* object. The *Streams Declaration* object contains a list of *media* identifiers to be created and a *Session QoS Policy*. The policy specifies the relative priorities of these media. The *User Session* uses the list of media identifiers to generate a list of media objects. This list together with the Session QoS policy is then passed to the QoS Manager for creation.

The application may wish to add more channels or remove existing channels. It does so by sending the *add/delete channel* request respectively to the *User Session*.

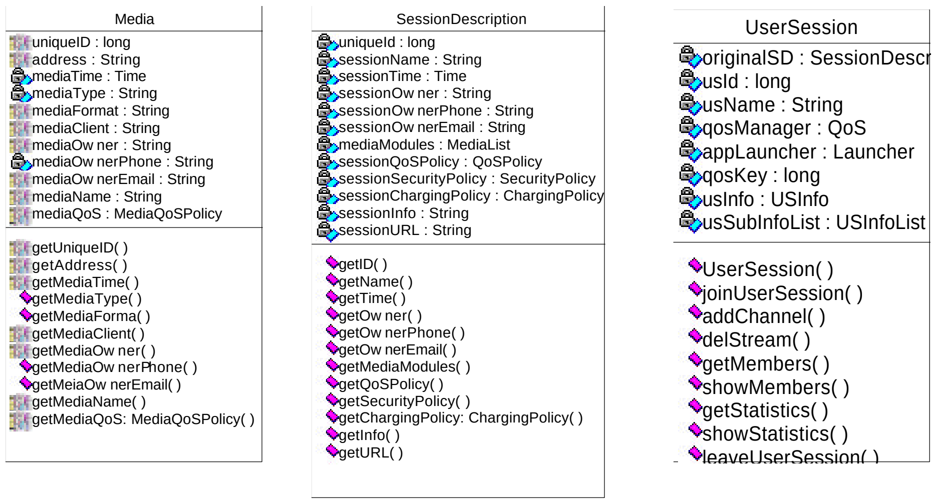

Figure 3 Key Classes

QoS Manager Object

The *QoS Manager* accepts from the *User Session* a request to create a set of communication channels. A list of media objects and a *Session QoS Policy* object are given to the *QoS Manager* as arguments. In our current implementation, the *Session QoS Policy* comprises a mandatory list and an ordered optional list. The mandatory list is an ordered list and specifies those channels that are not crucially required. When resources are scarce, media streams at the head of the list are created first.

The QoS Manager will use the policy object when creating media streams: it will first verify that it has enough resources (e.g. network bandwidth, processing power, and devices) to create all media streams in the mandatory list. If there are not adequate resources, it will throw a *No Resource Exception.* It will then check to determine whether there are enough resources to handle all the optional media streams. If there are not enough resources for this then it will throw a *Partial Resource Exception*, and creates only those channels that can be handled. If there are adequate resources to create all the optional media streams as well as well as the mandatory streams, it will create all streams and passes references of these streams back to the *User Session*.

When the *QoS Manager* creates the audio and video media streams, it uses the *Media QoS Policy* to determine which mechanism (e.g. *RSVP*, or *layered encoding*) to use. The current implementation only supports layered encoding and this is explained below.

Audio Media Stream

An audio media stream or sometimes called *streamgroup* is made up of a number layers (or *RTP/RTCP* sessions). Figure 5 below illustrates an audio stream that comprises of three audio encoded layers, and each layer in turns comprises an RTP and an RTCP session. The audio quality gets progressively better as more layers are added to the base layer (layer 1), for more details please refer to [13].

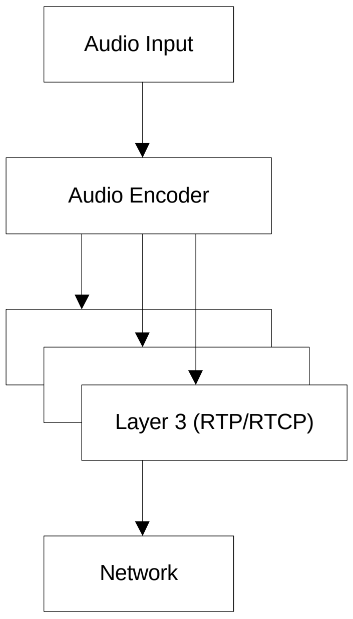

Figure 4 Audio Stream Group Encoder

The above figure illustrates the encoding end of the audio stream (i.e. the server end). At the receiver end the QoS manager is responsible for creating the audio *streamgroup* decoder (see Figure 5 below). The QoS Manager determines the maximum number of layers, RTP/RTCP addresses and ports, and bandwidth from the *Media QoS Policy* object. It will then create appropriate number of layers depending on the resource available and the Session QoS Policy as described previously. Once the decoder is created, the QoS Manager will monitor RTCP statistics and local host resource usage to add/remove layers appropriately.

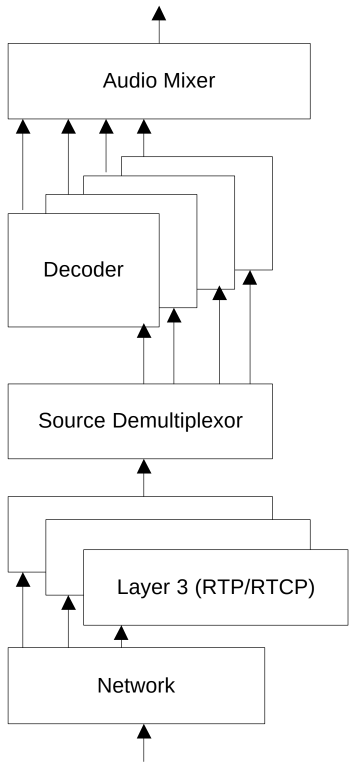

Figure 5 Audio Stream Group Decoder

Video Media Stream and Data Channel

The Video Stream Group Encoder has the same architecture as the Audio Stream Group Encoder. The Video Stream Decoder differs from the Audio Stream Group Decoder in that the former does not have a mixer.

5. Example Applications

Two applications have been developed using the *M3* architecture. These are the *sharedTV* and a *virtual world sharedTV* applications. The *sharedTV* application enables users to come together in a conference to watch television. That is, users are able to receive audio and video streams from a server, and they are also able to communicate with one another via an audio channel. The video server sends out layered audio and video streams. The *sharedTV* client enables users to receive these streams and manually add/remove audio and video layers to/from the streams. Our experience in building this application demonstrates that using the *M3* middleware the application can easily create and manage media streams. In addition the application can easily enable users to choose the quality of service of different media streams. The application also demonstrates the flexibility of the resulting application from a service provider's point of view. The video server can be configured to run with a low or high quality by changing the number of layers sent, this only requiring in a change in the session description and not the application itself.

The *virtual world sharedTV* application provides a 3D user interface so that the television can be viewed from different perspectives, and users will be able to see each other's representation in a 3D virtual world. Hence this application requires the same media streams for the television channel as the previously mentioned application, however in addition it uses the data channels to communicate the positions and orientations of users' avatars. The additional insight gained (in terms of middleware usage) from developing this application is that the application can easily create multiple logical data channels. It is envisaged that later version of this application will exploit other functionality of the middleware - such as dynamic sessions - as it becomes available.

6. Related Work and Conclusions

The Session Directory Rendezvous (*SDR*) tool receives session descriptions and launches applications such as *vic*, *vat* and *wb*. It passes to these applications just the technical information for concerning their media streams or channels. It is then up to the applications to then create and manage those streams or channels. In our approach, we pass the session description to multimedia applications and the *M3* middleware. In this way, the applications can use the description to declare their communication needs, whilst the middleware can take care of matching those needs to the underlying implementation. In addition, the middleware can handle other related requirements for the session, e.g. security and charging. This declarative approach to configuring the middleware is being advocated by many researchers [11, 12]. These researchers have invented their own particular language for this description. We have chosen to extend or modify the IETF's SDP protocol for declaring communication needs, charging and security policies. We believe that our approach facilitates the implementation of robust and flexible multi-party, multi-media applications. We are encouraged by our early experience in building our *sharedTV* and *virtual world sharedTV* applications.

There are a number of possible future directions that we intend to take: we are currently looking into extending SDP to specify security and charging policies. This in turns requires the extension of the *M3* functionality to support security and charging. We also intend to explore how complex sessions[2] can be supported: that is how to specify a complex session, how to parse the session description and how the middleware manages the existence of multiple sub-sessions. We also intend to explore how dynamic sessions[3] can be supported: that is to explore how updates to a session description can be delivered to interested parties, and how these updates are to be handled by the middleware. Lastly but not least we are intending to investigate how terminals can automatically adapt the number of layers being received.

7. Acknowledgements

The authors would like to thank members of the Distributed Systems Group at BT Laboratories, in particular Ian Fairman and Mike Rizzo reviewing this paper. We would also like to thank Dave Dalby for his help with the implementation of the audio and video streamgroup components, and Kashaf Khan for his useful initial insights into the use of *M3* for building the *virtual world sharedTV* application.

References

[1] M. Handley and V. Jacobson, *"SDP: Session Description Protocol"*, IETF MMUSIC Working Group RFC 2327, April 1998.

[2] S. McCanne and V. Jacobson, *"Vic: a flexible framework for packet video"*, in Proceedings of the ACM Multimedia '95, November 1995.

[3] The whiteboard wb, ftp://ftp.ee.lbl.gov/conferencing/wb

[4] S. Shenker, D. Clark, D. Estrin and S. Herzog, "Pricing in Computer Networks: Reshaping the Research Agenda", SIGCOMM Computer Communication Review, Volume 26, Number 2, April 1996.

[2] A complex session comprises of more than one sub-session. For example, a technical conference may comprise of multiple tracks, each of these track corresponds to a sub-session.
[3] A dynamic session is one in which its description changes during its life time. For example, a secure session, in which access to its media streams are protected by an encryption key, this key may change from time to time.

[5] M. Handley, "The Session Directory Tool SDR", http://mice.ed.ac.uk/archive.srd/html.

[6] M. Handley, "SAP: Session Announcement Protocol", Internet Draft, draft-ietf-mmusic-sap-00.txt, November 1996.

[7] J. Rosenberg, E. Schooler, H. Schulzrinne and M. Handley, "SIP: Session Initiation Protocol", draft-ietf-mmusic-sip-08.txt, August 1998.

[8] S. Casner and S. Deering, "First IETF Internet Audiocast", Computer Communications Review, July 1992.

[9] P. A. Putland and J. Hill, "Electronic Payment Services", BT Technical Journal, Volume 15, Number 2, April 1997.

[10] P. J. Skevington and T. J. Hart, "Trusted Third Parties in Electronic Commerce", BT Technical Journal, Volume 15, Number 2, April 1997.

[11] R. Balter, L. Bellissard, F. Boyer, M. Riveill, J. -Y. Vion-Dury, "Architecturing and Configuring Distributed Application with Olan", Middleware '98, IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing, July 1998.

[12] A. Zarras and Valerie Issarny, "A Framework for Systematic Synthesis of Transactional Middleware", Middleware '98, IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing, July 1998.

[13] M. Nilsson, D. Dalby and J. O'Donnell, "Layered Audivisual Coding for Multicast Distribution on IP Networks", submitted to IEEE Multimedia Systems'99, June 1999.