An Efficient Delivery Middleware - (DMIFWare) For MPEG-4 Applications: Design, Implementation and Experience

Deepak Jaiswal, Rajiv Chakravorty, Rajesh Babu
Multimedia Codec Group, Sasken Communication Technologies Limited, Bangalore, India
[djai | rajivc | rajeshb]@sasken.com

Abstract

This paper describes the novel design of a delivery framework for MPEG-4 applications. The delivery model for MPEG-4 also called as DMIF (as recently standardized by ISO/IEC) is used as plug-ins for wide variety of distributed real-time applications right from video conferencing to interactive virtual games. Our design is in the form of a communication middleware that offers manifold advantages including increased efficiency, dynamic transport channels with QoS support and dedicated functionality towards underlying network stacks. MPEG-4 DMIF delivery middleware (termed here as DMIFWare) integrates three key delivery technologies namely broadcast (e.g. HDTV), disk delivery (e.g. DVD, CD) and remote network delivery. In this paper we describe the design and implementation of a middleware for remote network delivery. Several issues have been taken into consideration so as to include design features like flexibility, modularity, QoS support, programmable and interoperable transport stack along with issues that are pertinent to an efficient DMIFWare design. Design features of DMIFWare were also compared with other existing middleware design. The features incorporated within DMIFWare were found to be quite satisfactory, even tough we believe that a seamless integration of other delivery technologies (disk delivery, broadcast delivery) into our existing design will enhance the DMIFWare outlook.

1 Introduction

MPEG-4 [8][9][10][11][12] is a new standard from ISO/IEC for coding natural and synthetic audiovisual data in the form of audiovisual objects that are arranged into an audiovisual scene by means of a scene description. It provides the end-users a high level of interaction with content, within the limits set by the *author*. The various operations that users are allowed:

1. Change the viewing/listening point of a scene (by navigation)
2. Drag media objects of a scene to different locations
3. Trigger cascade of events by clicking on a specific object (for e.g. start, stop, pause, rewind)
4. Multi-lingual track facility to select the desired language

MPEG-4 therefore offers a whole spectrum of services that includes complete integration of production, distribution and content access across several delivery paradigms. In addition, it also offers to the service provider the information about an application's use of resources in a given network domain. This in turn helps the service providers to apply suitable billing policies.
At the network level, information can be interpreted and translated into the appropriate native signaling messages corresponding to a transport protocol stack. DDSP[1]protocol is used for such mapping purposes.

---

[1] Default DMIF signaling protocol

MPEG-4 also provides a generic QoS parameter set to map different types of media streams. To guarantee QoS support for the application it is necessary to have an efficient mapping across each of these QoS parameter set with respect to the underlying network QoS parameters.

In future, wireless technology will be required to support delivery of multimedia services to mobile terminals with QoS guarantees. The networks supporting these terminals will have a high degree of fluctuating support towards Quality of Service (due to packet loss, fading effects etc.). Since MPEG-4 supports dynamic addition and deletion of transport channels (depending upon availability), a middleware supporting it should posses adequate flexibility to manage QoS even in an environment prone to high degree of QoS fluctuation. Therefore, a delivery framework that supports all kinds of heterogeneous network stacks in addition to providing QoS support is what is deemed necessary in any delivery model.

The paper describes an efficient design of a delivery framework for MPEG-4 applications. The design is flexible in order to incorporate all types of current and future delivery technologies (including wireless domains). Design efficiency is high since it offers a low response time essential to provide multimedia services even to a resource constrained terminal (in terms of CPU, memory etc.). Depending upon the requirements of the user application, network channels can be dynamically reconfigured, added or released. Each such channel will have an appropriate QoS parameter set depending upon the nature of the specific audiovisual objects; so as to provide QoS support to the application. However, support to QoS in various networks is still devoid of the desired QoS guarantees. Internet is one such example that comprises of various heterogeneous networks where QoS guarantees are absent. In near future, QoS support to the Internet is expected from various quarters such as *IntServ*[2] (RSVP) [14] and/or *DiffServ*[3] [16] – *MPLS*[4][15]. In this paper, we will discuss about design modifications at the delivery layer that can ensure QoS at the network level using one of these techniques.

The paper is structured as follows: The next section provides a brief overview of the MPEG–4 DMIF standard. It provides the MPEG-4 DMIF overview, explains the MPEG-4 system architecture, DMIF computation model and also the architectural framework of DMIF in the form of a middleware (DMIFWare). Section 3 gives a description of the DMIFWare system along with implementation specific details of the design. Here it describes issues related to DMIFWare functionality, DAI/DNI primitives, signaling aspects and also elaborates on design optimizations related to DMIFWare. In section 4, we evaluate the MPEG-4 QoS at each levels (by using a layered model) and discuss how DMIFWare can be used to provide network QoS by negotiating resources from the network. Some important features like group delivery support (i.e. multicasting) and issues related to security are discussed in section 5. Other middleware architectures are discussed in section 6 and a feature/functionality-based comparison is performed to ratify our DMIFWare design. Finally, open issues are discussed that are pertinent to an enhanced DMIFWare design applicable for the future.

2 MPEG-4 DMIF Overview

The DMIF architecture is such that the applications that rely on DMIF for communications do not have to be concerned with the underlying communication methods. The implementation of DMIF takes care of the details of delivery technology thereby presenting a common interface to the application.

An application accesses data through the DMIF-Application Interface (DAI) as shown in Figure 1. This is irrespective of whether such data comes from a broadcast/local storage/remote server. A delivery layer implementation allows the concurrent presence of more than one DMIF instance. Each DMIF instance is delivery unaware and responsible for managing a given delivery technology. When requesting a specific service, the application supplies a Uniform Resource Locator (URL) that allows the delivery layer to determine the appropriate DMIF instance to activate.

---

[2] Integrated Services QoS model (uses RSVP protocol)

[3] Differentiated Services

[4] Multi Protocol Label switching

The DMIF instance will then translate the originating application requests into specific actions[5] or translate it into messages[6], thereby taking care of the specific delivery technology. Similarly, data entering the terminal is uniformly delivered to the originating application through the DAI. The DAI allows the DMIF user to group elementary streams into services and to specify the QoS requirements for the desired elementary streams.

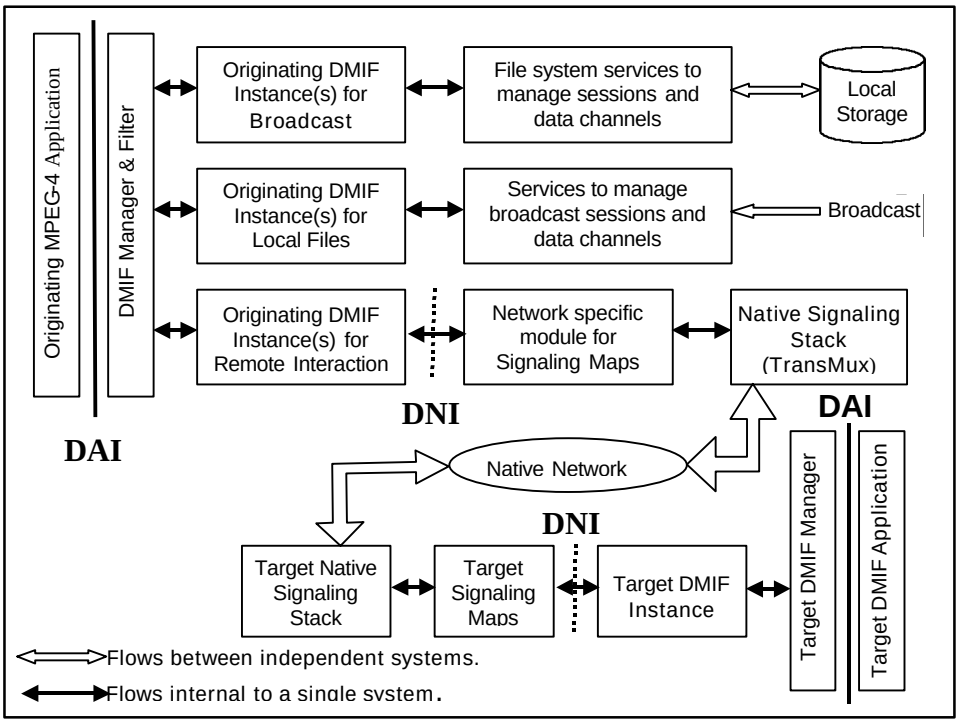

Figure 1: Architecture of Delivery Multimedia Integration Framework

DMIF separates the common features that should be implemented in each DMIF instance from the details of the delivery technology. In the case of interactive networks, DMIF specifies a logical interface (the DMIF-Network Interface (DNI)) between a hypothetical module implementing the common features and the network specific modules. The DNI specifies the information flow that should occur between peers of such hypothetical modules. The network specific module specifies the mapping of the DNI primitives into signaling messages.

In case of broadcast and local storage scenarios, the model is simpler and no internal interface has been identified. Conceptually, each DMIF instance interacts with a module which implements the features of a target DMIF peer as well as those of the target application. This implies that, in this case the DMIF instance is not unaware of the application making use of it.

2.1 MPEG-4 System Architecture : An Overview

The system architecture for MPEG-4 can be effectively defined using the three layers i.e., compression layer, synchronization layer and underlying delivery layer. Figure 2 shows such an abstraction. The compression layer processes individual audiovisual media streams independent of the underlying delivery technology. MPEG-4 can achieve compression using efficient encoding from a few kbps to multiple Mbps. The output of this layer is the audiovisual media streams organized into elementary streams. The compression layer is only media aware. The interface between this layer and the syncronization later is termed as Elementary Stream Interface (ESI). Creation of distributed and integrated content presentations necessitates establishment of

[5] To be taken for the broadcast media or the local file system.

[6] To be delivered to the target application.

relationships between the elementary streams, in addition to enabling synchronization between them. The synchronization layer is therefore both media and delivery unaware.

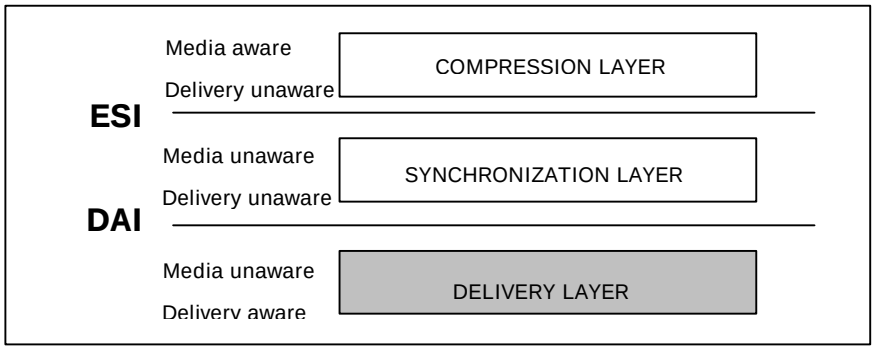

Figure 2: MPEG-4 System Architecture

The delivery layer in MPEG-4 is primarily an integration framework. It provides a transparent access to delivery of content irrespective of the underlying techology used by offering content location independent procedures for establishing MPEG-4 sessions and access to data channels. The DDSP signaling protocol interacts with the lower (network) layer so as to map the messages appropriately to be sent over the network to a remote interactive peer.

2.2 DMIF Computation Model

The DMIF computation model provides a high level view of a service activation and start of data exchange between two DMIF peers. Figure 3 shows the end-to-end signaling communication path that is established for transfer of control messages. The basic necessity of a computation model here is to identify the need of a user plane and a control plane.

The computation model for MPEG-4 DMIF is described below:

1. The originating application requests the activation of a service to its local DMIF Layer: a communication path between the originating application and its local DMIF layer is established in the control plane (C1 - plane).

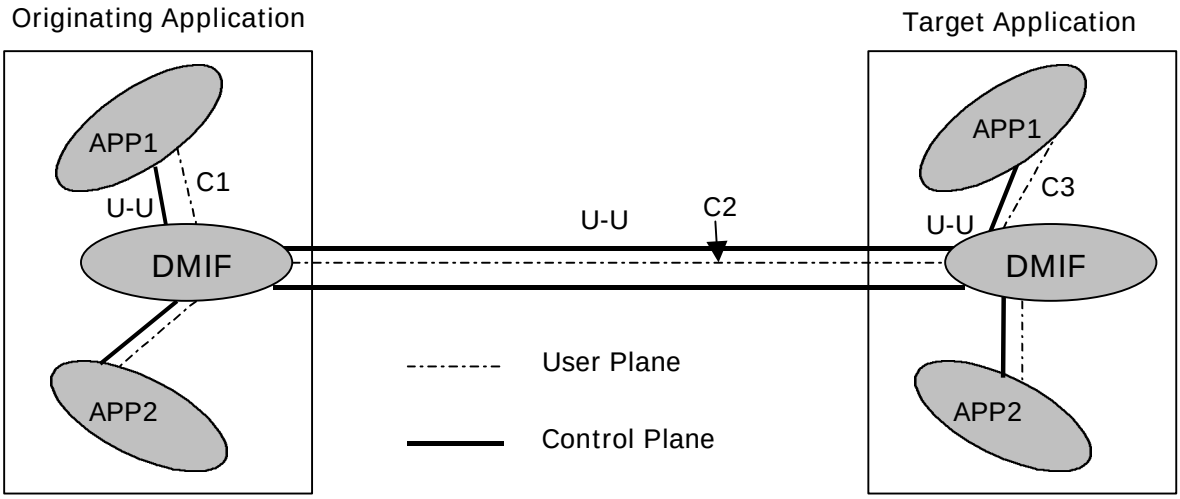

Figure 3: DMIF Computation Model

2. The originating DMIF then establishes a network connection with the target DMIF peer: a communication path between the originating DMIF peer and target DMIF peer is established in the control plane (C2-plane).

3. The target DMIF peer identifies the target application and forwards the service activation request: a communication path between the target DMIF peer and the target application is established in the control plane (C3-plane).
4. The peer applications create channels requests flowing through paths C1, C2 and C3 plane. The user plane (U-U) will exchange the actual data by the applications.

DMIF therefore allows the concurrent presence of one or more DMIF instances, each one targeted for a particular delivery technology, in order to support multiple delivery technologies (broadcast, local storage, remote interactive) within a given terminal. Multiple delivery technologies may be activated by the same application that in turn can seamlessly manage data transfer across each of these delivery technologies.

2.3 DMIFWare Architectural Framework

Our design of the DMIF interface forms part of a (communication) middleware that is end-system based and located between the network access and applications (Figure 4). The middleware has access to all the underlying network resources. As the application does not need to care about differences in transport mechanisms used, properties and especially semantics of different transport mechanisms are hidden away. The level of abstraction provided by the transport modules is a part of the configured communication protocol. The DMIFWare resides once per workstation (or terminal), while multiple applications might use the middleware. The DAI translates the request into appropriate actions to be taken with respect to the broadcast network, local file system or into messages delivered to the target application.

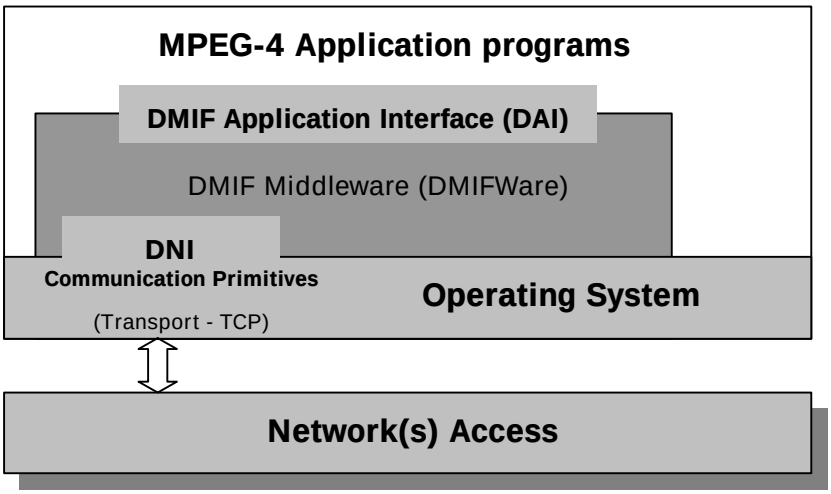

Figure 4: DMIF Middleware (DMIFWare) Architecture

In case of remote interactive network, the network side of the DMIFWare uses the DNI (DMIF Network Interface) primitive to invoke the control message signaling. DNI is a set of primitives to be invoked whenever a control message needs to be generated from the native signalling stack (NSS) of the DMIF to be sent over the network. The native signaling stack uses TCP as the transport for exchanging the control messages across the network. Here the communication primitives are assumed to be a part of operating system. However unlike control messages which use TCP channels, data is sent over the UDP channels as it is a real time data. This is compliant to the requirements of any real time application; to eliminate any retransmission overhead for data packets that are lost during the transfer.

3 DMIFWare System Design

We have extended the multimedia middleware design beyond traditionally layered communication architectures that avoid common design pitfalls in multimedia communications, such as reduced efficiency and flexibility. We have also maintained considerable modularity to integrate and support any other key design features that needs to be incorporated within DMIFWare in the near future.

3.1 DMIFWare Functionality

Our design of the DMIFWare is object-oriented. Each of the functional entity is implemented in a C++ class. Figure 5 depicts the various modules of the DMIF layer along with their functionality. All the components involved during the creation of a service along with their inter-relationship are based on the computation model discussed in section 2.2. The architecture therefore encompasses a signaling plane (control plane) and a data-plane. The following explains the functionality of each of the modules in brief within DMIFWare:

- **DMIF Filter and Manager:** The main responsibility of this module is to manage (initialize, de-initialize, configure, reset) the various modules in the DMIF layer. For each DAI primitive called by the application layer, it forwards the parameters passed to the appropriate DMIF module. Control messages and data packets received from the interacting peer terminal are forwarded to the appropriate application.
- **DMIF Instance:** This module handles all the communication that takes place with remote peers using a particular delivery mechanism. The functionality of DMIF instance are listed below:
  1. Manage the service session provided to application layer
  2. Manage network wide session with the remote peer using DNI calls
  3. Map service session to network session to optimize network resources

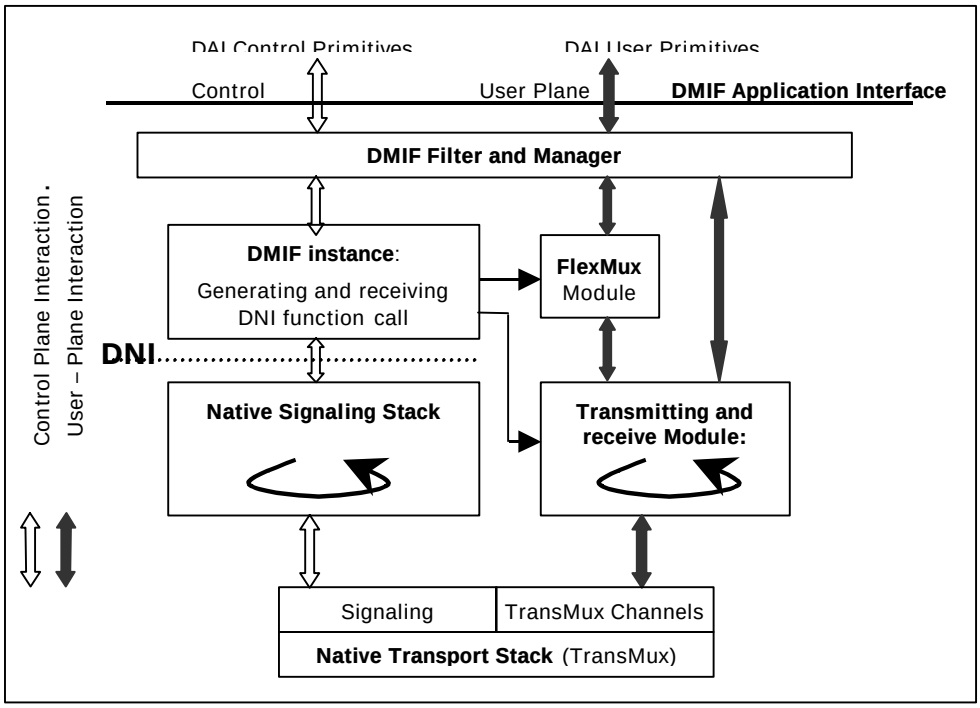

Figure 5: DMIFWare Architectural Framework

  4. Map Elementary Stream channels either to FlexMux or TransMux[7] streams based on QoS requirements
  5. Translate elementary streams QoS requirements to actual network resources (network QoS)
- **FlexMux Module:** It allows grouping of elementary streams with a low multiplexing overhead. Multiplexing is used to group elementary streams with similar QoS requirements to reduce network connections and bandwidth requirements.
- **Native Signaling Stack (NSS):** This module is responsible for generating control messages (signals) using a DNI call based on DDSP protocol. It also receives control messages from the remote peer and

[7] Transport layer multiplexing (Transmux) of audiovisual streams

maps them to other modules of the DMIFWare for further handling. It explicitly uses a thread called as *ReadThread* to manage the signaling channels. This module is described in section 3.5

- **Transmit and Receive Module:** It uses a third thread termed as *Transmit and Receive Thread* for management of data. This module receives data packets either from the FlexMux module or directly from the application layer using the DAI primitive. The data packets received from the higher module is transmitted over the TransMux channel (UDP based) provided by the native network. The data packets received over the TransMux channels are appropriately forwarded either to the FlexMux module or to the application layer.
- **Native Transport Stack:** This stack is responsible for communication over the actual native network connecting the two remote peers. Presently the DMIFWare design uses TCP/UDP (native transport stack) as the delivery technology for control message signaling and data transfer respectively.

3.2 DMIF Application Interface (DAI)

DAI is the interface visible to the application programmers at the end-systems for the DMIFWare. It is responsible for hiding all the transport details from the application user. The interface supports passage of both data streams as well as control message flow from the DMIFWare. DAI primitives enable the user to specify the QoS requirements for each of the data streams that allow the mapping of the application QoS requirements to the corresponding transport channels. The user requests made across the DAI are also translated to the corresponding DNI requests. The DAI is composed of three different types of primitive classes:

- Service Primitives – allows management of service sessions (attach and detach).
- Channel Primitives – allows management of data channels (add and delete).
- Data Primitives – serves the purpose of transferring data through channels.

The following table describes different primitives in DAI in order to request a certain service or establish desired data channels across any delivery technology including a remote interactive peer.

Primitives	Description
DA_ServiceAttachReq	Opens a service session to the specified destination (target DMIF - in case of remote interactive network)
DA_ServiceDetachReq	Closes the service session.
DA_ChannelAddReq	Requests the addition of a new data channel for the application within a service session
DA_ChannelDeleteReq	Requests for the deletion of a data channel
DA_UserCommand	Used by the application to send application level command (e.g., PLAY, STOP, PAUSE, and RESUME) to the peer application.
DA_Data	It is used to send user -to- user application data from originating DMIF to the destination (target DMIF – in case of remote interactive network)

Table 1: DMIF Application Interface

3.3 DNI – DMIF Network Interface

To allow flexibility in the design, the DNI separates the Native Signaling Stack (NSS) from the rest of the DMIF module. The idea here is to de-couple the NSS module from other DMIF modules since NSS is largely depended upon underlying transport protocol. Our design considers the conventional Internet transport protocol stack i.e. TCP for control signaling. DMIF instances can be added for various delivery technologies (e.g. AAL5/ATM etc.) depending upon the requirements. Thus, DNI models the signaling message exchange between DMIF peers irrespective of the type of delivery mechanism.

The following primitives (along with their description) are used as a part of DNI:

Primitives	Description
DN_SessionSetupReq	Request to set up a network session with the remote DMIF peer
DN_SessionReleaseReq	Request to release an existing network session
DN_ServiceAttachReq	Request to attach to the desired service of a network session.
DN_ServiceDetachReq	Request to detach an existing service from the network session
DN_TransMuxSetupReq	Request for starting a new TransMux channel
DN_TransMuxReleaseReq	Request to release an existing TransMux channel
DN_TransMuxConfigReq	Request to reconfigure one or more TransMux channels previously established inside a network session
DN_ChannelAddReq	Request to add new data channels for the application (channel added by the target DMIF)
DN_ChannelAddedReq	Request to add new data channels for the application (channel added by the originating DMIF)
DN_ChannelDeleteReq	Request to delete the existing data channels
DN_UserCommand	Request to send application level command (e.g., PLAY, STOP, PAUSE, and RESUME) to the remote DMIF peer.

Table 2: DMIF Network Interface (DNI) Primitive Description

As the table illustrates, many of the primitives in the DNI are similar to the primitives used in DAI. However, some other primitives like *DN_SessionSetupReq/ReleaseReq* and *DN_TransMuxSetupReq/ ReleaseReq* are in addition to that already present in the DAI. The DNI therefore provides an interface to separate *Native Network Stack* (NSS) module from the rest of the DMIF that are functionally independent of an underlying network stack. This technique provides large design flexibility and reusability in terms of system components.

3.4 Control – Plane Message Signaling in DMIF

The control messaging across two DMIF peers should take place at the C-plane before the actual application level data transfer happens. To invoke any control message across the network, the user application should first invoke one of the DAI primitives. The contexts of these message exchange across the two peer depends upon three types of primitive (i.e. service primitive, channel primitive or data primitive) that can be invoked. The MPEG-4 DMIF standard specifies the syntax of the control messages to any request made for a given service. The request can be for a service invocation from a remote peer or else related to a new channel addition or existing channel deletions. All the DAI primitives the application user can invoke are described in section 3.2. For brevity, we look into two examples that involve signaling issues across two DMIF peers. The first scenario explains signaling involved in initiating a service across two DMIF peers while the second one describes a channel addition by an originating DMIF peer. (For other scenarios refer [11]).

3.4.1 Service Initiation in DMIF

The originating DMIF initiates a service session by calling the *DA_ServiceAttachReq* primitive of DAI interface consisting of the URL along with upper layer information (Step A) (refer Figure 6). If a network session to the remote peer already exists, it will skip (Step B) and will directly invoke the *DN_ServiceAttachReq* from the DNI primitives (step D). If the network session to that peer does not exist, it will invoke a *DN_SessionSetupReq* primitive using the DNI primitive (Step B). This request will reach the target DMIF; it now has the knowledge of the new network session and then replies (Step C) back to the

originating DMIF with a *DN_SessionSetupRsp* primitive. Once the new network session is established, the originating DMIF then requests (Step D) for the desired service using DN_ServiceAttachReq of the DNI primitive. Once the target DMIF gets the service request, it invokes a callback function DN_ServiceAttachInd (Step E) to indicate the target DMIF application user about the requested service. The target application interprets the data and replies with a DA_ServiceAttachCnf (Step F) back to the target DMIF.

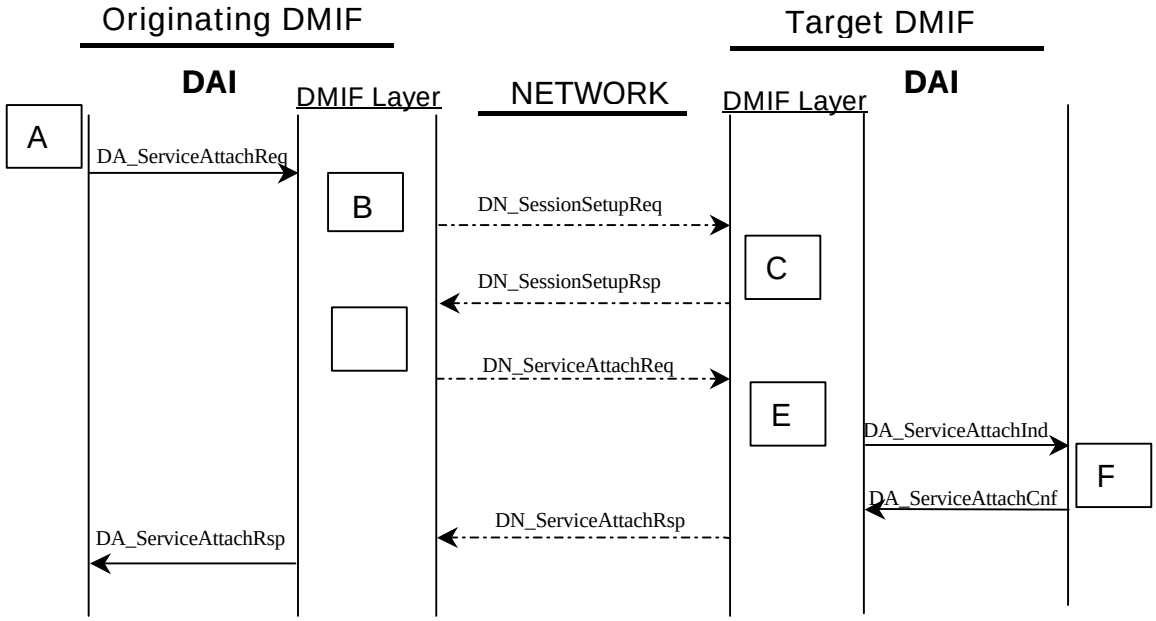

Figure 6: Service initiation in a remote interactive DMIF (ISO/IEC – 14496/6)

The target DMIF then invokes the DN_ServiceAttachRsp and sends the response message to the originating DMIF. The originating DMIF in turn invokes the DA_ServiceAttachRsp callback function to provide a response to the request made by the application. However, before making a service initiation request the originating DMIF must have the URL (from session directory, email etc.) and also the requested service registered at the target DMIF.

3.4.2 Dynamic Channel Addition by Originating DMIF

The addition of channels requires two preconditions:

1. Service between the originating and target applications has been initiated successfully.
2. Location of the source stream is available from previous interaction.

Figure 7 shows the scenario for channel additions by the originating DMIF. The originating application passes a *DA_ChannelAddReq* to indicate (Step A) that it requires data channels. QoS parameter, direction and other application level information characterize each of the channels. The DMIF layer inspects the network resources against the request made (by checking the media stream QoS metrics) and if the network resources are not sufficient, invokes (Step B) the *DN_TransMuxSetupReq* function from the DNI primitive. The target DMIF receives the request, associates it to network resources of a particular network session. It then replies (Step C) with the *DN_TransMuxSetupRsp* that sends the response back to the originating DMIF. The originating DMIF creates a channel and sends this channel information to the target DMIF using *DN_ChannelAddedReq* function (Step D) of the DNI.

The target DMIF receives the request that in turn invokes the *DA_ChannelAddInd* (indication) function (Step E) and informs the target application with all the channel information sent by the originating DMIF. The target application interprets the application information and replies (Step F) with a response code in the form of *DA_ChannelAddCnf* function to the target DMIF. The target DMIF then sends back a response message using *DN_ChannelAddedRsp* function with DMIF. This message then reaches the originating DMIF that informs the originating application using *DA_ChannelAddRsp* function to indicate that the given channel have been added.

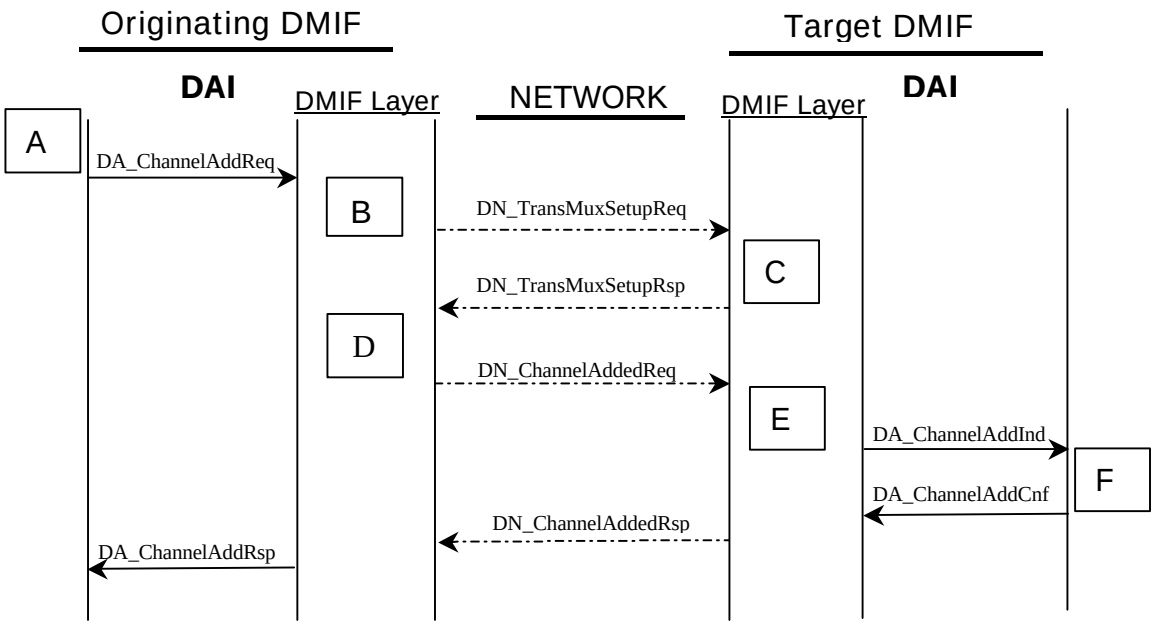

Figure 7: Addition of channels by the originating DMIF (ISO/IEC –14496/6)

The other case of message signaling involves addition of channels by the target DMIF peer. Similarly the deletion of channels can also take place by making appropriate signaling requests. All such signaling issues related to MPEG-4 -DMIF has been already discussed in the standard [11].

3.5 Native Signaling Stack (NSS)

The basic functionality of this module is to implement the DDSP protocol in order to enable MPEG-4 multimedia communication over the Internet. At the originating DMIF, DDSP protocol generates control messages to be sent over the network.

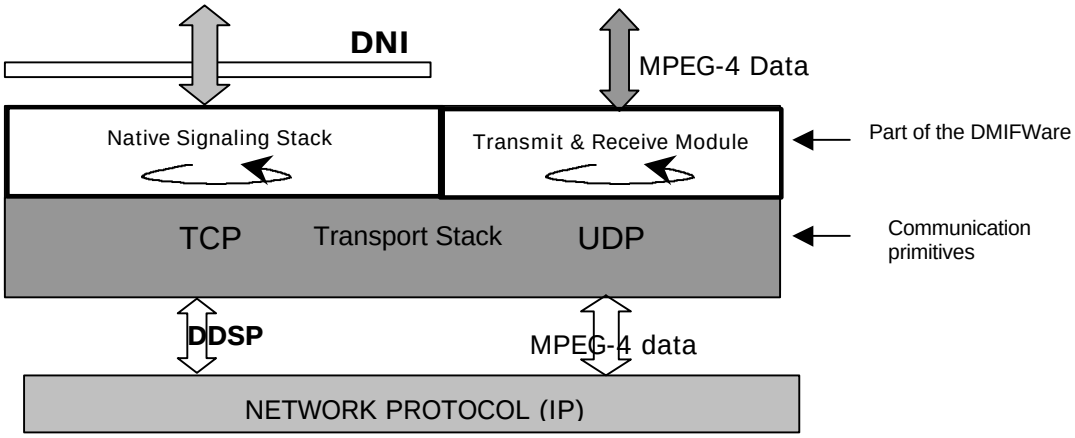

Figure 8: Location of Native Signaling Stack (NSS)

Invocation of any of the DNI primitive generates a control message over the network to a remote destination. The signaling can be for session setup, session release, service initiation, service release, channel addition or channel deletion which are compliant to the signaling strategy as discussed in earlier section.

Figure 8 shows the position of NSS along with the transport stack. Any of the primitives (as in section 3.3) can be invoked to generate the required control message to be sent over the network. The NSS signaling based on DDSP protocol that adheres to the reference DMIF standard [11].

3.6 Design Optimizations in DMIFWare

The complete DMIFWare is implemented as a 32-bit interface (for DAI and DNI) that uses the C++ environment based on Microsoft Visual C++ compiler (v6.0). DMIFWare runs under the Windows NT 4.0 OS (also extendible to Windows 95 platform) in a Pentium machine.

DMIFWare functionality is divided according to its usage in three different threads. The idea here is to reduce the number of threads so as to decrease response time. We have carefully optimized the usage of threads and made sure that none of the threads are blocked in any of the function calls within DMIFWare. The first thread (Thread-1) (see Figure 9) is the main *application* thread that the application user invokes for any of the DAxx_Req (DAI) primitives. This is the normal thread that runs through DMIFWare to finally invoke the corresponding DNxx_Req of DNI primitive. By invoking a DNxx_Req primitive, a DSxx_Req control message is sent over the network. The other thread (Thread - 2) called as *ReadThread* is created when the NSS module is instantiated that accepts any connect request made from a remote peer and also receives a control (signaling) message that arrives after the connection is established.

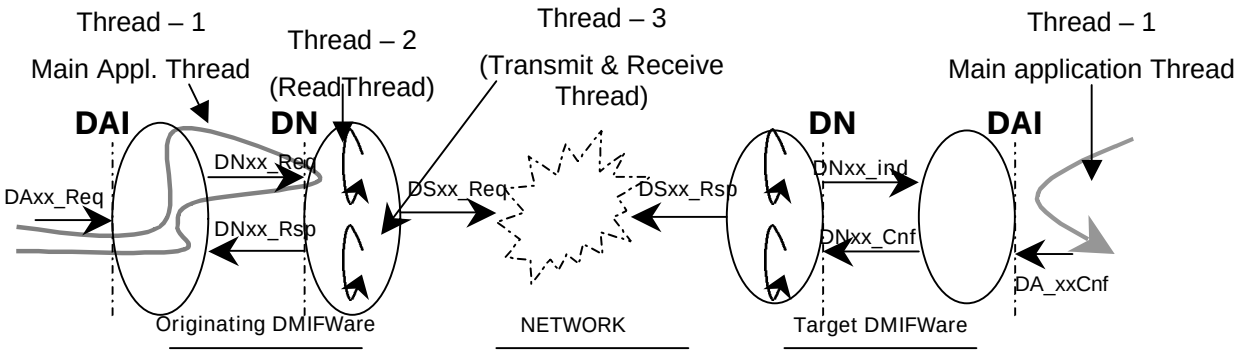

Figure 9: Thread Usage and Call Sequence in DMIFWare

When the *ReadThread* receives the control message, it forwards it to the appropriate message handler (not shown) to service the request. This request made to the message handler invokes the DNxx_ind (indication) function and which calls a callback function that puts this message into a queue. The main *application* thread reads the message posted in the queue and processes each of the messages in succession. After processing a message, the thread then invokes the corresponding DAxx_Cnf (confirmation) function that goes all the way to invoke DNxx_Cnf and generates DSxx_Rsp (response) message to be sent up over the network to the originating DMIF. Finally, the originating DMIF gets back the DSxx_Rsp (response) message that completes the request. A third thread in Transmit and Receive module is used to send and receive data. This thread called as the *Transmit and Receive Thread* is used to manage data channels i.e. sending and receiving of data from remote peers. However, this thread is not related to the control signaling aspects of DMIFWare. The obvious advantage here is the separation of data and controls path that in turn provides high efficiency along the data path [21].

Other types of design optimizations include reducing the time to copy packets (especially the larger ones) using the C-library call `memcpy`. In fact we have reduced the usage of `memcpy` to almost nil in our design. This results into significant improvements in the response time of the DMIFWare interfaces. The maximum throughput that governs the performance of the DMIFWare depends to a large extent on optimizing the usage of threads and simultaneously reducing the memory accesses. DMIFWare also optimizes the use of objects (e.g. synchronization objects (mutexes) etc.) inherent to an applied operating system. Synchronization objects have been used only when synchronization between the threads was found to be necessary.

DMIFWare exchanges control data between two remote applications using winsock2 API. Winsock2 API primarily provides a protocol-independent transport interface that is fully capable of supporting emerging networking capabilities including real-time multimedia communications [7].

4 QoS Support for DMIFWare

To achieve an end-to-end guaranteed Quality of service (QoS) along multimedia communication paths for distributed multimedia applications, we need to provide services and protocols in the end-points and networks which understand the mapping issues related to QoS at each of these levels [6]. QoS mapping at each level is discussed here to determine their interrelationships with DMIFWare that in turn provides a complete QoS management architecture for MPEG-4 system.

4.1 Mapping QoS in MPEG-4

As illustrated in Figure 10 MPEG-4 system includes the application and the delivery middleware. Our middleware is in between the MPEG-4 application and the underlying transport protocol stack. The QoS is distributed across each of these layers. In other words if we consider end-point layered QoS architecture [4][5], we can separate QoS into *application QoS* (e.g., 25 frames/sec), *System* QoS (e.g. 50 ms period/cycle) and *network* QoS (e.g.16 Mbps). In addition *User* QoS (also called perceptual QoS e.g. CD quality) is also specified to complete the QoS mapping. DMIFWare falls into the category of S*ystem* QoS. Hence parameters that relate to CPU (or memory) such as computation time, cycle time and CPU utilization directly affects the system level QoS. DMIFWare being a part of system QoS should necessarily have an efficient design to optimize over these parameters. This in turn can significantly improve system level QoS.

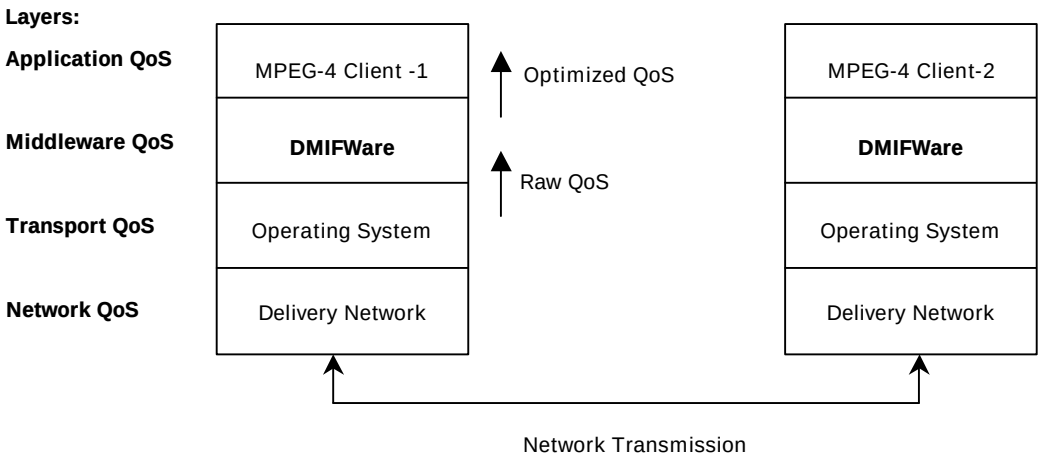

Figure 10: QoS mapping in MPEG-4 System

4.2 Application Level QoS

The application level QoS in MPEG-4 mostly constitutes the QoS applicable at the compression layer. The compression layer is responsible for encoding and decoding of elementary streams that in turn specifies the processing requirements. There are several parameters affecting QoS in MPEG-4 at this level (for e.g. frame size and frame rate, number of frames per Group of Pictures (GOP), compression pattern (I, P, B) etc.). MPEG-4 compression layer performs encoding that enables bandwidth requirements from few Kbps to couple of Mbps. The application level QoS in MPEG-4 is therefore highly varying and mostly dependent upon the application requirements.

4.3 System Level QoS

All sorts of reusable components between application and the underlying network decide the system level QoS. These include the operating system, filters, device drivers and all additional middleware components. Middleware QoS and transport QoS both add up to give the resultant system QoS. In our design, DMIFWare as well as the underlying transport stack both in combination will specify the system level QoS. Hence any additional component incorporated (e.g. authentication and encryption) within DMIFWare will directly increase the CPU requirements and thereby degrade QoS at the system level. An efficient middleware design

is therefore a must for optimization of system level QoS. In case of DMIFWare, we have optimized the use of all types of components (e.g. thread usage, memory copy etc) that can significantly reduce the QoS at this level.

4.4 Network Level QoS

All levels of QoS discussed earlier specify only the end-system QoS. Specifying QoS at the network level requires a QoS broker that can arbitrate for QoS at that level. To accomplish this we should involve protocols that can communicate the application requirements using explicit admission control to all the elements in a network. This is possible by using resource reservation protocol (RSVP). RSVP is a working protocol of the IntServ QoS model for the Internet. To ensure QoS guarantees in a network, all network elements (routers) should support RSVP. Microsoft's winsock2 API provides the required interface to negotiate parameters such as bandwidth and latency to achieve the desired service levels of QoS for a given communication service. Our framework to implement "network QoS" integrates winsock2 GQoS[8] [12] that provides QoS support at the network level using RSVP. Currently, the facility to support RSVP using GQoS is available only in Windows 98 and Windows NT 5.0 beta machine. Therefore, our DMIFware integration paradigm to support "network QoS" is available only with terminals supporting these OS'es.

4.4.1 Winsock2 GQoS

The QoS enabled service provider in these terminals (NT 5.0 and Widows 98) is called as QoSSP (QoS Service Provider) which is a part of the operating system [7].

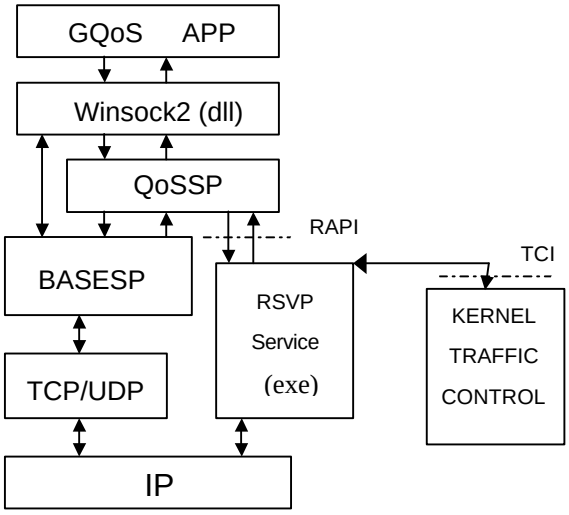

Figure 11: Architecture of Generic QoS (GQoS)

It is a layer on top of the base service provider (BASESP) (Figure 11) which invokes resource reservation protocol (RSVP) to signal the network to do resource allocation so as to satisfy QoS requests made by the applications. QoSSP also configures the kernel traffic control in compliance with the QoS negotiated with the network.

The QoSSP intercepts and handles all QoS related Winsock2 API calls. The RSVP service process implements the RSVP protocol for network signaling. The QoSSP interacts with RSVP service via the RAPI[9] interface. Kernel traffic control is activated indirectly by QoSSP via the RSVP service process, which uses the Traffic Control interface (TCI).

[8] generic quality of service

[9] RSVP Application Programming Interface

4.4.2 Integrating DMIFWare with RSVP enabled winsock2 GQOS

To integrate DMIFWare with winsock2 GQoS, use of QoS enabled data structures and API calls are required. A QoS aware application can specify the QoS parameters of its sent and received traffic in the *FlowSpec* structures within a QoS structure. (Refer Appendix-A for all QoS enabled data structures)

The *flowspec* may then be included with QoS-related calls (for e.g. **WSAConnect**, **WSAJoinLeaf**, **WSAAccept**, and **WSAIoctl (SIO_SET_QOS)** [[13], see appendix-A]). Each of these QoS-related calls specifies a particular socket to which the call applies. Thus, QoS is invoked relative to a particular socket. A socket, on which QoS has been invoked, is said to be a *QoS Socket*.

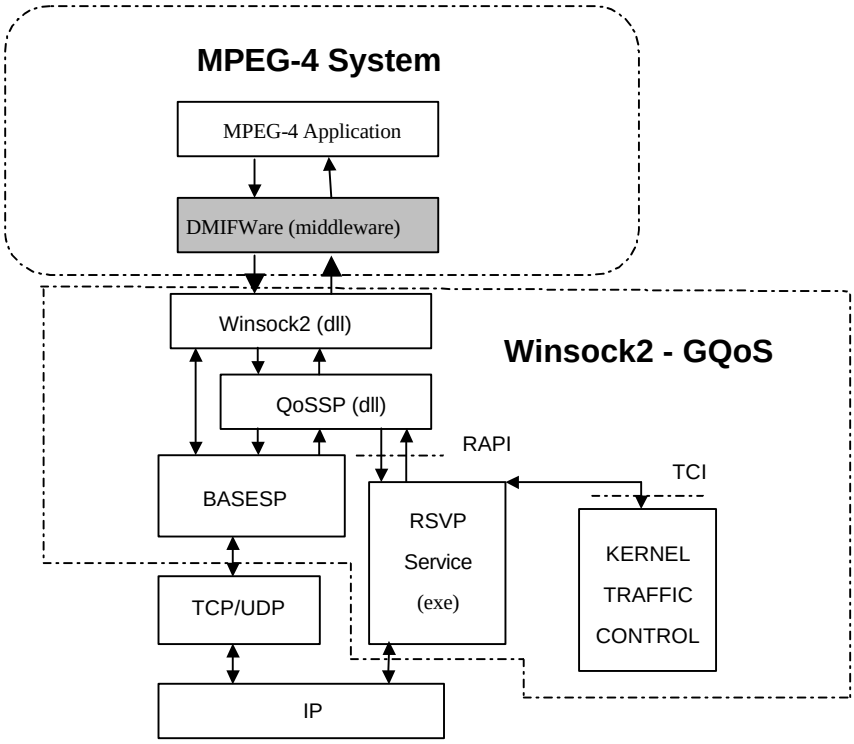

Figure 12: Integrating DMIFWare with Winsock2 – GQoS [Win'NT(5.0)/Win'98]

To enable GQoS for RSVP, the DMIFWare should request for the *QoS sockets* (see Figure 12). All the QoS related calls are mapped to the flow specifications [Appendix-A] according to the requirements set by the user application. Depending upon the MPEG-4 application, DMIFWare will decide which types of service (guaranteed, controlled load or best effort) would be suitable for the application. This decision can be based upon the application traffic characteristics, performance requirements and the user preference. Since MPEG-4 application provides real time data, they should be mapped either to controlled load service or guaranteed service. RSVP signaling is invoked only if the type of service is set to either guaranteed service or controlled load service. Triggering of the RSVP control messages (*path* and *resv)* begins only after the sender has invoked one of the QoS-related calls along with the socket unambiguously bound to a local address.

In case a network is not RSVP capable, the QoSSP can be used in a pass-through mode, in which case it will invoke local traffic control functionality (TCI) without RSVP signaling.

4.5 Network Resource Optimization using DMIFWare

DMIFWare provide considerable flexibility in terms of number of transport channels; their bandwidth and QoS requirements. Such requirement is essential in the case of MPEG-4 applications that calls for dynamic establishment of data channels with varying QoS parameters. In Figure 10, we find that the QoS available to DMIFWare from the underlying transport layer is the raw QoS. DMIFWare negotiates the right amount of QoS available from these lower layers by mapping that many numbers of data channels to corresponding transport channels that are required for the application.

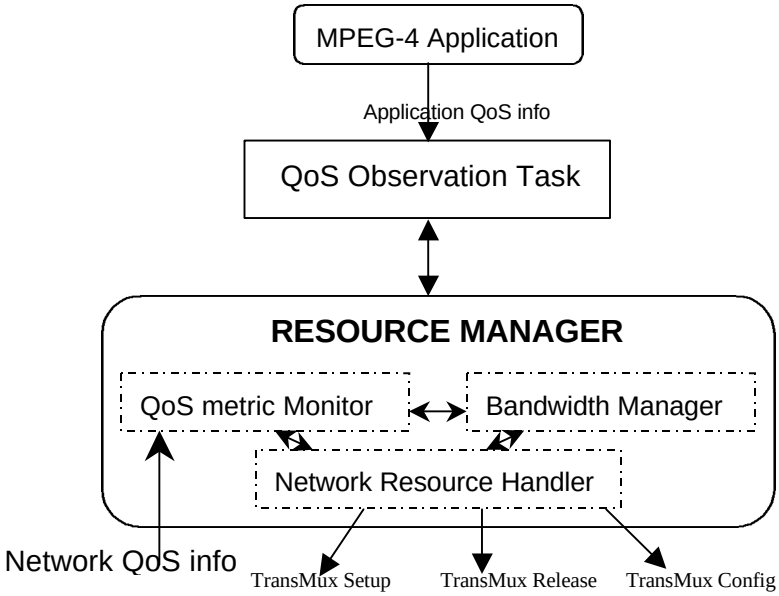

Figure 13: Resource Management process in DMIFWare

In other words, this type of dynamic support from DMIFWare enables a distributed application to optimize the use of available network resources. Figure 13 shows the resource management process in DMIFware. A resource manager accepts the application QoS information and arbitrates for the desired QoS from the network. The bandwidth manager regularly updates the bandwidth available from the underlying network. Depending upon the application QoS requirements and network resource availability, it will first try to reconfigure one or more TransMux channels previously established inside a network session using *DN_TransMuxConfigReq* of the DNI primitives. If the re-configuration process does not satisfy the application QoS requirements, then it will set up new transport channels (using *DN_TransMuxSetupReq*) that can handle the application QoS requirements. In the same way, a network resource handler can also decide to release existing transport channels (using *DN_TransMuxReleaseReq)* in order to efficiently manage the network resources.

Moreover, QoS specification for each of the channels carrying the streams can be mapped appropriately to the QoS parameters of the underlying network technology; an advantage which is available only from DMIFWare.

5 Other Support Features

There are other supporting features that are equally important for any middleware architecture. Multiparty communication is a core feature that should be supported in any distributed communication architecture. Security and authentication also forms part of any successful middleware design. We present a brief discussion on these issues so that these features can be incorporated into our present design of DMIFWare.

5.1 Multiparty Communication using DMIFWare

MPEG-4 DMIF standard has also specified the extensions to support multiparty communication also termed as group delivery (multicast) technology [20]. While the extension does not require any modification of DAI, it does require an extension to DNI. In addition to the three types of primitives (service, channel and data), DNI should also incorporates identification[10] and state[11] primitives. A multiparty group session consists of a DMIF group-signaling channel for state information distribution and one or more group transport channels to deliver the application information.

[10] which allows for identification of content producers (i.e. source request and id source response)

[11] allows distribution of content producer states (i.e. source request and source state response)

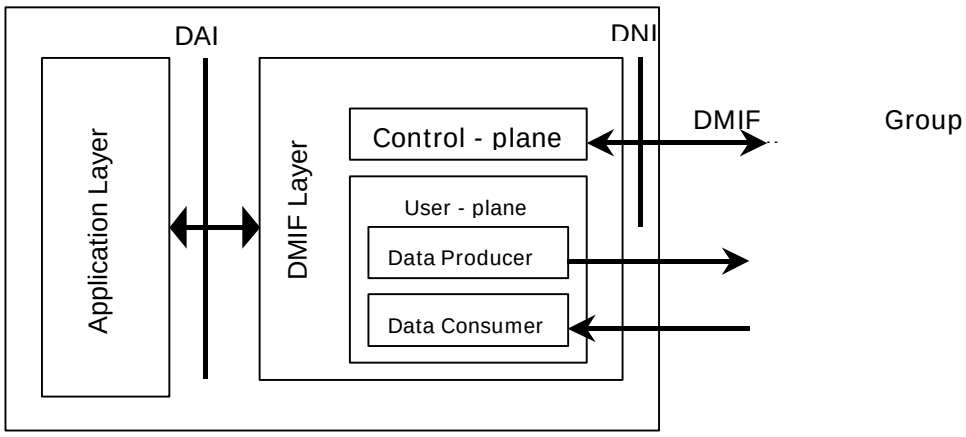

Figure 14: Multicast support (group delivery) in DMIFWare

There are certain additions needed in the DMIFWare design in order to enable multiparty communication. A separate DMIF instance that supports group session (to enable multicast) needs to be incorporated in the DMIFWare module (see Figure 14). A group session can be identified by a muticast address and a corresponding port number. To enable multipoint communication, a single sender (also called as DPDT – Data Producer DMIF Terminal) explicitly joins the DMIF group session registering a session by itself. This session is known to all other terminals joining as receivers (termed as DCDT – Data Consumer DMIF Terminal). To join the flow (using the URL), each receiver should listen to the group signaling channel. The receiver DCDT that joins the group should avoid making any explicit request to the sender but make use of the information that was just sent or requested by another DCDT. None of the DCDT terminals are aware (through explicit DMIF-DMIF signaling) about other DCDT attached to the group session.

For control information distribution we have planned to use TCP multicasting while data transfer takes place using UDP/IP multicasting. For other network options like ATM we can use AAL5/ATM multicasting. We have looked into this issue considerably and expect it to involve it in the next phase of our DMIFWare design.

5.2 Security/Authentication Issues for DMIFWare

DMIFware does not inherently provide any authentication or encryption functions. MPEG-4 provides an environment where handling of high volume multimedia data streams is fairly common (especially the one involving multiparty communication). As such providing even a fairly decent level of security will amount to considerable use of the CPU power. This in turn has significant effect on the QoS at the middleware level. Moreover all such algorithms providing encryption have an extra overhead. Therefore the amount of security provided should be based on demand mostly to cater to the requirements of efficient middleware architecture. In any case, we have left considerable flexibility within our DMIFWare design to incorporate security and authentication functions whenever required.

6 Selective Comparision of Related Work

DMIFWare provides a comprehensive systems approach for providing QoS based multimedia services to MPEG-4 applications. However, the performance of such a middleware is vital in case of multimedia - specific domains where requirements for QoS guarantees are high. There have been several approaches to providing QoS support in a middleware; each of them requires a detailed characterization of applications and specification of communication requirements at different levels. We characterize each of these approaches in terms of the their overall features and functionality.

To do this, we compare DMIFWare with other important and existing middleware architectures. There are several such middlewares [1][2][3][5][17][18][19] available; we will necessarily consider the one's which

justifies our present discussion. First, we take the Lancaster QoS- Architecture [18] which was meant to define QoS at the end-systems. The second such middleware architecture called as *DaCaPo++* [1] has comparable features to DMIFWare. In addition, we will also discuss a third model i.e. the xbind's DMIF delivery model [3] meant specifically for MPEG-4 applications. Our comparison across these three architectures along with DMIFWare is strictly based on their features and functionality. A performance specific evaluation requires availability of evaluation parameters for a middleware (e.g. middleware overhead in terms of memory, Interface (API) throughput etc.) that enables a comparison across each of these models (except for *DaCaPo++* where performance related data is available). Therefore we have restricted ourselves to a feature based "selective comparison" technique between each of these models.

To emphasize the core issues that are combined with DMIFWare:

- Application Quality of Service (QoS) Specification
- Flexibility of a middleware architecture
- Network Level QoS Support using RSVP (NT v 5.0 & Windows 98)
- Effective transport protocol stack configuration
- Inherent support to local delivery and broadcast delivery technology (In progress).
- Dynamic mapping of transport channels to data channels along with QoS support.

However, other related work in this direction have also focussed significantly to issues that DMIFWare presently does not support or else the work is still in progress. A brief comparison of DMIFWare and other related work is highlighted in the following table:

CRITERIA	QoS-A	DaCaPo++	Xbind	DMIFWare
Middleware Flexibility	No	High	Medium	High
Portability	UNIX	UNIX	WinNT	WinNT/95
Signaling Stack	No	Flexible	Flexible	Flexible
QoS (Appl. Level)	Yes	Yes	Yes	Yes
QoS adaptation	Yes	Yes	Yes	Yes
QoS (Network Level Support using RSVP)	No	No	No	Yes
Multicast Support	No	Yes	No	In progress
Synchronization	No	Yes	Yes	Yes
Security	No	Yes	No	No
FlexMux Functionality	No	No	Yes	Yes
Middleware Overhead	High	Medium	Meduim	Low
Dynamic Channel Allocation	No	No	Yes	Yes
Local delivery and Broadcast Support	No	No	Not Specified	In Progress

Table 3: Comparing DMIFWare with other communication middleware architectures

Many architectures have attributed security and authentication features to an efficient middleware design. Altough *DaCaPo++* incorporates such features, other designs have overlooked this functionality. In case of DMIFWare, a related discussion was presented in section 5.2 altough our present design does not support such functionality. To enable multiparty communication, multicast support is required. Here again

DaCaPo++ architecture scores over other middleware architectures. We have briefly explained how we will be providing multiparty communication support in a DMIFWare. This will be incorporated in the next phase of our design. Xbind's model offers functionalities similar to that of our DMIFWare design, meant for MPEG-4 delivery. But the model lacks features like security and authentication and does not support multiparty communications. All such features are coupled in our DMIFWare design in addition to support for QoS. Even tough all the architectures provide end-host support for QoS, they however do not enable side-by-side QoS support from the underlying network. DMIFWare can negotiate for QoS guarantees from the network. It can reserve resources (banwidth) from the network by negotiating with the network elements (routers) for desired resources all along the path. The functionality to reserve resources using RSVP protocol has not been provided in any of the delivery architectures, other than DMIFWare.

7 Open Issues in DMIFWare design

There are open issues in the DMIFWare design that needs resolution. We discuss only the important ones and mention our approach towards any enhancement in future for the DMIFWare design.

1. The design of the DMIFWare was only limited to native stack options for IP networks. We used here a transport option of TCP/UDP for control message signaling and data channels respectively. Extension of our design towards support for delivery technologies like ATM, wireless (especially for mobile multimedia terminals) should also be envisaged.
2. Extension of DMIFWare to support local delivery and other broadcast technologies are currently in progress. Work in this direction is in the completion phase and it is expected to be soon incorporated into our existing DMIFWare design.
3. MPEG-4 standard does not specify any provision for security/authentication. Some proposals in the MPEG-4 standards before arriving to a general consensus about these issues can be particularly useful when implementing these features within DMIFWare.
4. Performance evaluation of the middleware (in terms of middleware overhead based on memory and CPU, Interface throughput, response time of control message signalling etc.) should be carried out to ratify the DMIFWare design.

8 Summary and Conclusion

In this paper, we detailed the design of the multimedia communication middleware (DMIFWare) architecture used specifically for the delivery of multimedia data. DMIF Application Interface (DAI) offers the required degree of transparency between user applications and the delivery technology supported by middleware. It hides all the complexity of the underlying transport details and makes available to the MPEG-4 application user a set of primitives that can be invoked irrespective of the delivery technology.

DMIFWare integrates all types of delivery technologies as specified in the MPEG-4 standard, in addition to optimizing QoS not only at the DMIFWare (middleware) level but also at the network level. It supports Resource Reservation Protocol (RSVP) to guarantee resources from the underlying network using the GQoS winsock2 API.

DMIFWare integrates group delivery support i.e. multicasting (point–to-multipoint communication) feature by justifying the need for certain additions in the present model. The multicast feature intends to minimize the signaling to distribute the state of the DMIF group session and at the same time to allow dynamic join and leave of DMIF terminals to the DMIF group session. Moreover, the implementation also supports dynamic creation and release of transport channels that provide efficient management of network resources. Such type of features can provide different levels of QoS guarantees at all types of environments (depending upon resource availability).

Our group at SAS has already implemented the beta version of the QoS-aware middleware (DMIFWare) used for multimedia communication. However, the final implementation will incorporate all the open issues;

issues that are presently under development (multiparty communication, local delivery and broadcast support, etc.) and those partially addressed (security/authentication, other transport stack options, billing etc.). Future DMIFware will provide much more enhanced features necessary to carry out the next generation multimedia communication.

9 Acknowledgements

We gratefully acknowledge the support of Dr. P. G. Poonacha, Dr. B. Singh and Vineet Govil who are the main architect in leading our implementation towards MPEG-4 DMIFWare design. Many of our group colleagues (Multimedia Technology Group) at SAS constantly provided useful comments and feedback during the DMIFWare design. The authors would like to thank all of them for their invaluable and constructive comments.

10 References

[1] Bukhard Stiller, Christina Claus, Marcel Waldvogel, Germano Caronni, Daniel Bauer, Bernhard Plattner, “The Design and Implementation of a Flexible Middleware for Multimedia Communications Comprosing Usage Experience”, Institute of Communication Networks, ETHZ, Zurich, TIK-Report No. 54, July 1998. Source: http://www.tik.ee.ethz.ch/tik/research/publications/Publications.html

[2] Baochun Li, Klara Nahrstedt, “Configurable Adaptors for Multimedia Delivery – an End System Middleware Solution” Technical Report UIUCDCS-R-97-2018, Department of Computer Science, University of Illinois at Urbana-Champaign, July 1997. Source: http://cairo.cs.uiuc.edu/papers.html

[3] J. F. Huard, A.A. Lazar, K. S. Lim, and G.S. Tselikis "Realizing the MPEG-4 Multimedia Delivery Framework," IEEE Network Magazine, pp. 35-45, November/December 1998. Source: http://www.xbind.com/pub.html

[4] Klara Nahrstedt, Hao-hua Chu and Srinivas Narayan “QoS aware Resource Management for distributed Multimedia Applications” accepted to Journal on High-Speed Networking (Special Issue on Multimedia Networking). Source: http://cairo.cs.uiuc.edu/papers.html

[5] Klara Nahrstedt, Baochun Li “A Control – Based Middleware Framework for Quality of Service Adapations” to appear in IEEE Journal of Selected Areas in Communications, Special Issue on Service Enabling Platforms, 1999. Source: http://cairo.cs.uiuc.edu/papers.html

[6] J. F. Huard and A.A. Lazar, "A Programmable Transport Architecture with QOS Guarantees," IEEE Communications Magazine, Vol. 36, No. 10., pp. 54-62, October 1998. Source: http://www.xbind.com/pub1.html

[7] Windows Sockets 2, Protocol Specific Annex, Revision 2.0.3, May 10, 1996. Source: ftp://ftp.microsoft.com/bussys/winsock/winsock2/wsanx203.doc

[8] ISO/IEC 14496-1 International Standard “MPEG-4 Systems”. source: http://www.cdt.luth.se/~rolle/mpeg4.html

[9] ISO/IEC 14496-2 International Standard “MPEG-4 Vedio”. Source: source: http://www.cdt.luth.se/~rolle/mpeg4.html

[10] ISO/IEC 14496-3 International Standard “MPEG-4 Audio” Source: source: http://www.cdt.luth.se/~rolle/mpeg4.html

[11] “Information Technology - Generic Coding of Moving Pictures and Associated Audio Information Part 6: Delivery Multimedia Integration Framework,” ISO/IEC CD 14496-6, May 1998.source: http://www.cdt.luth.se/ ~rolle/mpeg4.html

[12] ISO/IEC 14496- 5 International Standard “MPEG-4 Reference Software”. Source: source: http://www.cdt.luth.se/ ~rolle/mpeg4.html

[13] Yoram Bernet, Jim Stewart, Raj Yavatkar, Dave Anderson, Charlie Tai, Bob Quinn, Kam Lee, Windows Networking Group “ Winsock Generic QoS Mapping” (draft – under development), Version 3.1, September 1998. Source: ftp://ftp.microsoft.com/bussys/winsock/winsock2/gQoS_spec.doc

[14] R. Braden, L. Zhang, S. Berson, S. Herzog, S. Jamin “Resource Reservation Protocol – Version 1 Functional specification” IETF RFC 2205, September , 1997 (Related: 2210/2211/2212/2215). Source: http://erf.fh-koeln.de/rfc.html

[15] Xipeng Xioa and Lionel M. Ni “Internet QoS: A Big Picture”, IEEE Network, pp 8-18, March/April 1999

[16] Welter Weiss, “ QoS with differentiated services”, BELL-LABS technical journal (Packet Networking), Volume 3, Number 4, October-November 1998.

[17] Campbell, A.T., "Mobiware: QOS-Aware Middleware for Mobile Multimedia Communications" , 7th IFIP International Conference on High Performance Networking, White Plains, NY, April, 1997. Source: http://comet.ctr.columbia.edu/~campbell/andrew/publications /publications.html

[18] Campbell, A.T., Coulson G., and D. Hutchison, "Supporting Adaptive Flows in a Quality of Service Architecture", ACM/Springer Verlag Multimedia Systems Journal , Special Issue on QoS Architecture, Vol. 6 No. 3, pg. 167-178, May 1998

[19] J.Robert Ensor and Sudhir R. Ahuja “Communication Middleware for Multi-Party Multimedia Applications” BELL-LABS Technical journal, Winter 1997. Source: http://www.lucent.com/ ideas/perspectives/bltj

[20] ISO/IEC JTC 1/SC 29/WG 11 N 2720 (date: 99-04-01) ISO/IEC 14496-6: 1999/PDAM 1 “Information technology – Very-low bitrate audiovisual coding – Part 6: Delivery Multimedia Integration Framework (DMIF)”

[21] Douglas Comer, David Stevens “Internetworking with TCP/IP” Volume III , Client Server Programing and Applications, ISBN-81-203-0928-6.

Appendix -A

QOS-Related Data Structures and API Calls

Note: The provision for Generic-Quality of Service (GQOS) support is available only in Windows NT 5.0 (beta machine) or Windows 98. For detailed description of QoS related data structures and related API calls (which involve RSVP), one can refer [14])

To open a socket supporting QOS:

- Enumerate the available protocols using either **WSAEnumProtocols()** or **WSAIoctl(SIO_GET_QOS).**
- Loop through the returned list of protocols looking for a protocol that supports QOS. Do this by checking if the *XP1_QOS_SUPPORTED* flag is set in **dwServiceFlags1** in each WSAPROTOCOL_INFO structure.
- When a protocol is found that supports QOS, call **WSASocket()** passing a pointer to that WSAPROTOCOL_INFO structure. **Also be sure to set the WSA_FLAG_OVERLAPPED flag so that the socket is created in overlapped mode. The RSVP service provider requires an overlapped socket.**

In Winsock2, a QOS-aware application can specify the QOS parameters of its sent and received traffic in the *SendingFlowspec* and *ReceivingFlowspec* structures within the QualityOfService (QOS) structure:

```
typedef struct _QualityOfService
{
  FLOWSPEC   SendingFlowspec;   /* flow spec for data sending */
  FLOWSPEC   ReceivingFlowspec; /* flow spec for data receiving */
  WSABUF     ProviderSpecific;  /* provider specific stuff */
} QOS;
```

Where WSABUF contains a pointer to a provider specific buffer that may be used by the application to supply additional QOS control objects to QOSSP. The Flowspec, which contains a set of token bucket parameters and a service type specification, is defined as:

```
typedef struct _flowspec
{
    int32        TokenRate;            /* In Bytes/sec    */
    int32        TokenBucketSize;      /* In Bytes        */
    int32        PeakBandwidth;        /* In Bytes/sec    */
    int32        Latency;              /* In microseconds */
    int32        DelayVariation;       /* In microseconds */
    SERVICETYPE  ServiceType;          /* Service Type    */
    int32        MaxSduSize;           /* In Bytes        */
    int32        MinimumPolicedSize;   /* In Bytes       */
} FLOWSPEC;
```

Each flowspec muct be related to any of the following QoS related call:

- **WSAConnect()**

```
int WSAConnect (
       SOCKET s,
       const struct sockaddr FAR * name,
       int namelen,
       LPWSABUF lpCallerData,
       LPWSABUF lpCalleeData,
       LPQOS lpSQOS,
```

```
        LPQOS lpGQOS
    );
```

- **WSAJoinLeaf()**

```
    SOCKET WSAJoinLeaf (
        SOCKET s,
        const struct sockaddr FAR * name,
        int namelen,
        LPWSABUF lpCallerData,
        LPWSABUF lpCalleeData,
        LPQOS lpSQOS,
        LPQOS lpGQOS,
        DWORD dwFlags
    );
```

- **WSAAccept(lpfnCondition)**

```
    SOCKET WSAAccept (
    SOCKET s,
        struct sockaddr FAR * addr,
        LPINT addrlen,
        LPCONDITIONPROC lpfnCondition,
        DWORD dwCallbackData
    );
```

A prototype of the **condition function** is as follows:

```
    int CALLBACK ConditionFunc(
        IN LPWSABUF lpCallerId,
        IN LPWSABUF lpCallerData,
        IN OUT LPQOS lpSQOS,
        IN OUT LPQOS lpGQOS,
        IN LPWSABUF lpCalleeId,
        OUT LPWSABUF lpCalleeData,
        OUT GROUP FAR * g,
        IN DWORD dwCallbackData
    );
```

- **WSAIoctl(SIO_SET_QOS)**

```
    int WSAIoctl (
        SOCKET s,
        DWORD dwIoControlCode,
        LPVOID lpvInBuffer,
        DWORD cbInBuffer,
        LPVOID lpvOUTBuffer,
        DWORD cbOUTBuffer,
        LPDWORD lpcbBytesReturned,
        LPWSAOVERLAPPED lpOverlapped,
        LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionROUTINE
    );
```

The prototype of the completion routine is as follows:

```
    void CALLBACK CompletionRoutine(
        IN DWORD dwError,
        IN DWORD cbTransferred,
        IN LPWSAOVERLAPPED lpOverlapped,
        IN DWORD dwFlags
    );
```