APPENDIX – A

Me-ADO-Q DESIGN USING UNIFIED MODELING LANGUAGE

A.1 Introduction
Chapter-VI provides the working of Me-ADO-Q scheme, while the appendix explores the development of distributed applications using the Common Object Request Broker Architecture (CORBA)[123]. Design of Me-ADO-Q approach and software utility to test the object-oriented method is discussed. Simulation software is developed to analyze the effectiveness of Me-ADO-Q. Software is implemented in Linux developed using Java v2.0. Testing is mainly limited to identify resources required for a Call, and effectives of Policy Control procedures [10,80].

A.2 Architectural Design
The architecture of Me-ADO-Q is as shown in Fig-A.1, which consists of four modules as Me-ADO-Q server, Resource Location Registry, and Call Session Manager and Agent Proxy Server. Each module is defined as Interface using IDL (Interface Description Language). A set of resources defined as resource components define the Interface.

![Diagram of Me-ADO-Q Software Stack Architecture]

Fig-A.1 Me-ADO-Q Software Stack Architecture
Me-ADO-Q Server is the primary module residing in gateway, acts as interface between call of user's and managing call component resources. This component handles various call types of users, identifies the type of request and passes the call request with required resource parameters to Call Session Manager. The functions of this module are:

1. Identifies Call Type and set of resource parameters to make a call
2. Cooperates with Policy manager module in handling call and resources
3. Works with Proxy server to create resource objects
4. Maintains the variation or changes in call.

Resource Location Registry works in tune with management of resource components used in a call. Resource components used for establishing a call are registered in RL_Table as well the MQ-Resource Direct Graph, discussed in Chapter VI notifies the resources in use. Functions are

1) Identifies the requested resources from Resource Table.
2) Negotiates with Agent Proxy Server to identify and allocate the resources.
3) Registers the allotted resource in RL_Table.
4) Updates in MQ-Resource Direct graph.

Call Session Manager handles various calls and its corresponding resources. In a conference media call, different users provide call to different users. The utilization of resources depends on type of call and terminal in use. Hence Call Session Manager works in Gatekeeper where some resources are being shared while some resources have to be specifically allotted for a call. Its functions are:
1) Handles information of different calls in state (State-less or State-full)
2) Manages different users
3) Provides information of resources in use
4) Any change in resource or modification in call is monitored and updated.

**Agent Proxy Server** controls sharing of network resources for various operations. Resources such as network bandwidth, buffer, and other components. Its functions are:

1) Creates network objects for set of resources in use.
2) Prepares Agents Objects for network resources
3) Responds on request call for resources from Me-ADO-Q server and negotiates on required set of resources.
4) Works with Call Session Manager

**A.2.1 Models and Descriptions – Class Diagram**

Class diagram is provided for three main modules, which define the interfaces and resource components. Fig-A.2 shows the class diagram for Stream Adapters and Terminals. Me-ADO-Q handles the properties for multiple communication devices as audio based IP Telephony, as well video based calls. Class diagram shows the abstract object definition properties for various interfaces as Stream Adapters, Stream Interfaces, Terminal Interface. End-Terminal Player components and intermediate resources are defined as abstract classes using IDL.
Terminal Interfaces are:

1. ITerminalDef - Terminal definitions, where definitions of end user terminals, which provide input and output for a media call are provided. This interface checks for the identity of terminal where corresponding input media type should be provided for similar media type.

2. IMediaTerm - Media types are defined for applications. This interface provides support for multiple media streaming types.

3. INetworkTerm - Definition of network parameters for controlling network. Parameters such as Buffer, Bandwidth, required link for transfer are determined by QoS Schemes such as Me-PLM, Me-TrffSchl and Me-TRAP-Q
4. **IUserTerm** - Parameters in application required for providing end-to-end QoS such as frame speed in video conferencing application, play-out capacity of buffer are considered.

5. **IOSTerm** - Parameters such as threading feature, cache memory, semaphore are considered.

```
Terminal Class
- Terminal_Type: DEVICE
- Terminal_Memory: BYTE
- Terminal_Cache: BYTE
- Terminal_VirtualMem: BYTE
- Terminal_OperatingSystem: STRING
- Terminal_VersionOS: WORD
- Terminal_ScrResolution: PIXEL

Network Class
- Network_Type: NETWORK
- Network_Bandwidth: BYTE
- Network_Structure: TOPOLOGY
- Network_Buffer: BYTE
- Network_TransferSpeed: BYTE

DEVICExMONITOR: 1
SPEAKER: 2
MIKE: 3

NETWORK
GATEWAY: 1
ROUTER: 2
GATEKEEPER: 3

HANDLER
Memory_Address: BYTE
ThreadNumber: BYTE
ProcessorIdentifier: INTEGER

MEDIATYPE
AUDIO: 1
VIDEO: 2

SYSTEM
Memory: BYTE
ThreadHandler: HANDLER
MemHook: HANDLER
SemaphoreAddress: BYTE

Fig-A.3 Media and Stream Interfaces
```

Interfaces used are:
1. ISystemStream
2. IApplStream
3. IMediaStream
4. INetworkStream
5. I TermStream
6. InetworkStream
7. IResourceClientCallback
8. INegotiateRegion
9. IResourceClientReq
The Stream Components and corresponding interfaces design shown in Fig-A.4 is represented in class approach below.

```cpp
class CMediaStream : public IMedia, // Interfaces
public IStreamAnalyzer
{
    public : // Implementation of abstract base class IMedia
        virtual void StreamBuffer()
        {
            // definition of Stream Buffer
        }
    virtual void PacketAnalyzer()
    {
        // method of analyzing the packets
    }
};
```
A.2.2 Deployment View

The deployment diagram of Me-ADO-Q is shown in Fig-A.5. The terminal component can be a terminal of the caller device or call receiver device.

A.2.3 Logical View

Fig-A.6 shows the Interaction diagram of Me-ADO-Q approach. The interaction between components of Me-ADO-Q residing in gateway server is carried out by Policy Manager module residing in Me-ADO-Q server.
The call made by user carries with it the terminal properties, and other intermediate request for resources. The request for resources being identified by Me-ADO-Q server is allocated on negotiation (implemented using ORB) with available resource list in Resource Location Registry. The identified resources are registered and returned to Me-ADO-Q server, which handles the call session properly such that expected quality is provided.

A.4 Implementation

CORBA's interface description language (IDL) is an important base for developing distributed applications. IDL describes the functional interface to the object, the type signature of the operations, which the object embodies, independently of the underlying programming language, operating system, and communication medium. Specifying only the functional interface allows distributed applications to be developed rapidly and transparently without regard to the underlying services. CORBA thus uses the traditional model of an interface that hides the implementation details. However, distributed applications based on CORBA's IDL operate acceptably only as long as resources are plentiful. Implementation has been tried with CORBA where objects are either local (in the client's address space) or within the same LAN as the client, because the system properties of these environments are stable, well understood, and resources are plentiful.

Implementation diagram shown in Fig-A.7 consists of a set of packages and multiple utilities, which form the software. The functionalities of this software module and software description used for implementation are provided in Table A.1.
The software utility Me-ADO-Q provides a way to initiate different kinds of communication sessions with multiple participants. The set up can be implemented using a client-server setup. The setup is designed in such a way that the components can be installed in gateway server or gatekeeper. The Me-ADO-Q Server module resides in each client end of call session, while Resource Location Registry and Agent Proxy Server are implemented in Gateway and Call Session Manager resides in Gatekeeper setup or Gateway based on media setup in an organization.

**Table-A.1 Software Requirement**

<table>
<thead>
<tr>
<th>Software Used</th>
<th>Java v2.0 - JDK 1.4.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform Required</td>
<td>Linux 9</td>
</tr>
<tr>
<td>Test Method adopted</td>
<td>Black Box Testing, Interface Test</td>
</tr>
<tr>
<td>Components Used</td>
<td>Remote Method Invocation</td>
</tr>
</tbody>
</table>

Policy Manager is implemented using ORB (Object Request Broker) architecture, while method of calling an object is called using RMI method.
The pseudo-code explains the Negotiation Process for resource allocation.

```plaintext
/* Forward declarations for classes used in the connection's parameters. */
interface Resource_client_callback;
interface Resource_negotiated_region;
interface Resource_client_require;

connection invReqResource (
    // 3 Parameters required for every connection
    in Resource_client_callback cl_call, // for client_callback
    in Resource_client_require cl_exp,   // for client_expectations
    out Resource_object_require ob_exp,  // for object_expectations
)
{
    /* Parameters specific to this connection, which can be used in predicates for negotiated and reality regions. */
    in double max_invoc m_p_s,
    in double max_idle sec

    client_callback interface Resource_client_callback
    object_callback interface Resource_object_callback
    client_expectations interface Resource_client_require
    object_expectations interface Resource_object_require

    /* Meta-level interfaces for contract Resource and negotiated regions are */
    Allocated: // Negotiate and check for conditions
        when client_expectations.throughput > 0 m_p_s and
        when client_expectations.throughput <= max_invoc m_p_s and
        when object_expectations.capacity >= max_invoc m_p_s

    Free:
        when client_expectations.throughput == 0 m_p_s and
        when object_expectations.capacity == 0 m_p_s

    // transition callbacks
    Allocated -> Free:
        object_callback->client_asleep()

    Free -> Allocated:
        object_callback->client_awake()
        client_callback->now_allocated()

    end // transition callbacks
}
end // negotiated regions

// reality regions for Allocated are separate
// reality regions for Free are separate
end // contract Resource
end // connection invReqResource
```
Negotiation code continued ...

separate reality regions for Resource :: Allocated:

Normal:
    when MeADOQ_condition . measured_throughput > 0 m_p_s and
    when MeADOQ_condition . measured_throughput <= max_invoc m_p_s
and
    when MeADOQ_condition . measured_capacity >= max_invoc m_p_s and
    when MeADOQ_condition . measured_idleness <= max_idle secs

Insufficient_resources:
    when MeADOQ_condition . measured_capacity < max_invoc m_p_s

Client_overlimit:
    when MeADOQ_condition . measured_throughput > max_invoc m_p_s

Client_asleep:
    when MeADOQ_condition . measured_idleness > max_idle_sec

/* precedences tell which reality regions are chosen if more than one predicate is true
precedence Normal, Client Asleep, Client Overlimit, No resources */
transitions callbacks:

    Normal → Insufficient_resources:

    // Warn the client that there isn’t enough capacity, even though we’re in negotiated region Allocated and thus there is supposed to be capacity.
    client_callback → warn_no_resources()

    // Tell the object to allocate more capacity (or lower expectations)
    object_callback → allocate_capacity(max_invoc)

Insufficient_resources → Normal:
    // Let the client know that it doesn’t have to hold its breath more
    client_callback → warn_enough_resources()

    any → Client_overlimit:
    // Let the client know it is exceeding its negotiated promise
    client_callback → warn_overlimit(max_invoc)
    any → Client_asleep:

    /* Let both the object and the client know that the client has gone asleep. One or both may reset their expectations (e.g., the client’s throughput or the object’s capacity), which could cause a renegotiation. */

    client_callback → warn_sleeping()
    object_callback → client_asleep()

end // transition callbacks
end // separate reality regions Resource :: Allocated