QMan: An Adaptive End-to-End QoS Management Architecture

Sae-Whong Suthon, Hung-Keng Pung and LiFeng Zhou
Network Systems and Services Lab, School of Computing
National University of Singapore
3 Science Drive 2, Singapore 117543

Abstract — Recognizing the need to provide end-to-end QoS for performance sensitive applications, and the need to manage such QoS provisions in the presence of diverse application QoS requirements as well as heterogeneous network environment, we propose an adaptive end-to-end QoS management architecture – QMan, to address such issues. QMan, as one of the key components of OCTOPUS middleware, makes no presumption of the underneath QoS provisions for end-to-end QoS management, but can take advantage of such provisions if there are. QMan allows applications to exchange QoS requirements and negotiate for a suitable protocol stack configuration with the assistance from Dynamic Protocol Framework (DPF). During data transmission, QMan monitors QoS violations and performs QoS adaptations according to the rules defined through a combination of three possible conduits in an end-to-end fashion: network level, host level and application level. These rules are pre-determined according to application requirements or network policies and can be re-configured for customization of QoS adaptations during run-time. Performance evaluations of our implementation demonstrate the feasibility of QMan's design with satisfactory performance.

Keywords — QoS management; multimedia; adaptive QoS middleware; QoS policy; dynamic protocol configuration

I. INTRODUCTION

To provide end-to-end QoS provisions for advanced applications, it is widely accepted that support is needed at both end-hosts and networks. Recent advancements in Internet technology have brought forth several QoS models such as *Integrated Service* and *Differentiated Service*. However, these models are not expected to be widely deployed as common services in the foreseeable future due to various reasons. In contrast, best effort service class is still prevalent, which implies that Internet is more likely to provide best assure rather than guaranteed QoS.

In end-hosts, various QoS approaches have been proposed and discussed. These approaches can be categorized into reservation-based approach and adaptation-based approach according to their different resource management styles. Reservation-based approach employs admission control and resource reservation along the end-to-end communication path. However, current operating systems and networks are not ready to support these advanced QoS features, which makes reservation-based approach hard to realize practically. In contrast, adaptation-based approach proposes to operate in best-effort environment and manages QoS in a pure end-to-end fashion where QoS monitoring and adaptation are enforced at end-hosts throughout the lifecycle of the session to smooth the quality fluctuation and maintain the agreed QoS level. Adaptation-based approach requires no modification at network level and OS level thus makes itself more suitable to deploy over current non-real-time OS and best effort network environment.

In adaptation-based QoS research, progresses have been made in the directions of 'QoS-aware applications' and 'QoS middleware'. Most work done in the application layer is related to the transmissions of continuous media streams (e.g. variable bit rate codec, media compression, frame-dropping and layered encoding scheme), and hence is rather media specific and restrictive in certain application domains [1] [2] [3]. In recognition of these limitations, more active research efforts have been spent in the middleware approaches. QoS middleware approach is popular for at least two main reasons despite of its performance overhead: (1) the QoS solutions are likely to be independent of the network and OS platforms, and (2) the QoS controls can be transparent to applications.

However, QoS should be more than just delay and throughput as is considered by current projects. Nowadays, application programmers could be overwhelmed by the problems of heterogeneous run-time environments and resource diversity, which makes applications hard to be developed. To gracefully and efficiently manage QoS for multimedia applications, these issues also need to be handled. In this paper, we propose our design philosophy to restore the flexibility of applications. Instead of specifying the names of the services, applications may specify the desirable properties of the services. For example, instead of "RSVP", applications can specify "resource reservation". In bind-time, an available service that offers resource reservation can be selected. If the selected service fails to deliver at runtime, it can be quickly replaced by another service instead of dully reporting to the application for rescue. This design philosophy assists applications to regain their flexibility and ensure that applications can be deployed easily and rapidly in large scale.

0-7803-8783-X/04/$20.00 © 2004 IEEE

An adaptive QoS Management architecture (QMan) has been developed based on such a design philosophy. It enables end-to-end QoS management and control in both unicast and multicast settings. The remainder of this paper is organized as follows: Section II discusses related work; Section III describes the architecture of QMan in detail; Section IV explains QMan QoS management mechanisms; Section V evaluates the functionalities and performance characteristics of our current implementation. This is followed by a conclusion and future work in section VI.

II. Related Work

The open problems of providing Quality of Service have been addressed by various research efforts in the past years. In end-host QoS research, various projects have been proposed to provide QoS supports as middleware services. Agilos [11] is a middleware control architecture to determine application-aware adaptations. The main contribution of this project is the introduction of Fuzzy Control Model to make QoS adaptation decisions. $2K^Q$ [12] proposes a resource-aware service configuration model. Multi-tie QoS translation and discovery server have been designed for distributed QoS compilation and runtime instantiation. DaCaPo++ [7] is a middleware QoS project that integrates various functionalities, e.g., security and multicast. It supports a range of multimedia applications and automatically configures itself at start-up time to provide suitable communication protocols and multimedia oriented services that are adaptable to application needs. MCF [13] from the same research group offers flexible multipoint communication services through protocol configuration at start-up time. DJINN [8] and RWANDA [9] follow component-based approaches, which provide high level QoS re-configuration. Components can be stacked and re-stacked in a variety of ways to meet the communication demands of applications.

Although great strides have been made in current projects, there still remain some problems toward a better QoS middleware:

1. *Lack of meta-information model for specification of QoS requirements, adaptation rules and policy.*

Adaptive QoS frameworks, like DaCaPo++, DJINN, ADAPTIVE [15], and RWANDA offer the ability to re-configure their components. However, the reconfiguration needs guidance from applications when violations occur, since such guidance is not included in the QoS specification. As it is unlikely to anticipate all possible QoS violations at the design phase of an application, expensive software re-engineering process is needed to 'update' the QoS awareness of existing applications, including adding new codes and re-compilations of applications [10]. In our view, this tedious process should be de-coupled from the applications by defining a meta-information model. This is the approach taken in the design of QMan, with details to be presented later.

2. *Lack of integrated framework utilizing both end-host and network QoS capabilities in a flexible manner.*

A middleware system can deliver much better end-to-end QoS if it collaborates with the host OS and network QoS support. However, previous frameworks that provide flexible end-host adaptation, such as Agilos and $2K^Q$, did not address this essential issue. We believe an integrated framework that utilizes both end-host and network QoS capabilities is essential to end-to-end QoS provisions.

Motivated by the shortcomings above, QMan is designed to provide QoS management functions in an integrated, seamless and efficient manner. XML-based adaptation rules are included into QoS specification. Application designers can define their rules according to application characteristics, while network administrators define rules according to network policy. QMan allows these rules to be combined and assigned to the framework. The introduction of *communication component* and *management component* allow collaboration between end-host QoS management and network QoS management. Moreover, all management functions are extensible to support one-to-many and many-to-many communications.

III. Overview of QMan Architecture

An overview of the QMan architecture is shown in Figure 1. QMan consists of two principal components: *QoS Agent* and *QoS Manager*. The former resides in each end-host to collect local QoS information and holds references of available components, whereas the latter directs participating QoS Agents to perform QoS management operations via a *control channel*. A *data channel* between participating hosts can subsequently be established according to application requirements. The data channel is referred to as *Adaptive Communication Tunnel (AC Tunnel)*; whereas the protocol

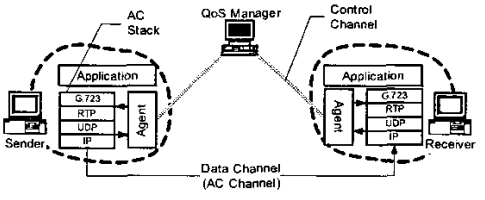

Figure 1: Overview of QMAN Architecture

stacks in the respective end-host is referred to as *Adaptive Communication Stack (AC Stack)*. Details of various key components in QMan are described below.

A. QoS Manager

QoS Manager is a distributed component that is responsible for overall QoS management of a session. A QoS Manager enters an *unbound* mode once created. It is subsequently *bound* when a sender attaches to it. This is the pre-condition for the QoS Manager to make sure there is at least one sender in a group session before any receivers can be accepted to join the group.

QoS Manager provides *Binding operation* to allow sender/receiver to create an AC Tunnel under its supervision. QoS requirements of AC Tunnel are specified as parameters of the binding operation. QoS Manager also extends an interface for applications to specify adaptation rules about the AC Tunnel; other functionalities provided by QoS manager include *Session Status Query* and *Session Instrument*, which is useful to interested third parties (e.g., authorized ISP) to shape an on-going session under regulation.

B. QoS Agent

QoS Agents reside in each participating end-host to schedule the local QoS management. QoS Agents hold references of *Communication Components (C-Components)* and *Management Components (M-Components)* in the same end-host. (Definitions of C-Components and M-Components can be found in next section). QoS Agents provide a set of APIs for applications to access QoS management functions. Methods provided by the QoS Agent include *Defining local QoS requirements and policy, Setting DataSource*[1] and *Getting DataSource*.

C. AC Stack and AC Tunnel

The outcome of the binding operation is an AC Stack, which is created at each end-host according to negotiated QoS requirements. Subsequently Adaptive Communication Tunnel (AC Tunnel) between end-hosts will be established for transmitting multimedia data from the sender to the receiver(s). AC Tunnel covers from sender's AC Stack, networks and along the way to receivers' AC Stack. In AC Tunnel, applications may specify QoS requirements in term of reliability, security, and etc.; data can be sent and received via *DataSource* object. AC Stack capitalizes on functions provided by *Dynamic Protocol Framework* (DPF) [5]. The AC Stack is richer in functionality than the Berkeley Socket where it can adapt itself to maintain the required QoS.

D. Control Channel

The control channel is mainly used for the exchange of control information and signaling messages between QoS Manager and QoS Agent. Inspired by the OMG Audio/Video Streaming Service specification, QMan separates the control channel from the data channel to ensure that additional control overhead will not degrade processing of media data [6]. In order to ensure the delivery of messages, the channel should be established over a reliable connection such as a TCP connection or a reliable lightweight connection with ARQ running over a unreliable UDP connection [4].

E. Communication Components (C-Component)

QMan regards a protocol component as its C-Component, which is composed of a dynamic executable code and a XML file. The executable code contains mechanism to process data and to generate default parameters automatically. In our implementation in Java, the execute code is Java byte-code. The XML file, which is called *QoS-Meta-Information-Based (QMIB)*, stores C-Component properties such as name, version and QoS attributes. As can be seen in Figure 2, the necessary information from QMIB and executable code of components will be translated into *ComponentDescripter* object, which is collected and stored in registry by *Component Registry Manager (CRM)*. Applications can quickly start-up based on such pre-loaded information.

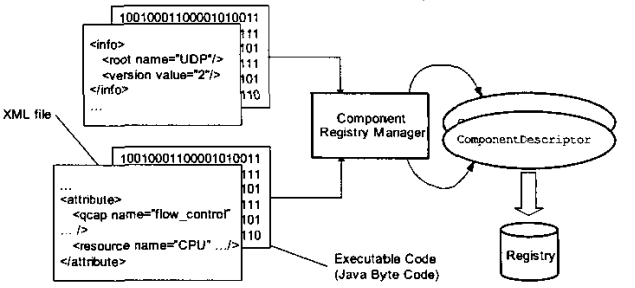

Figure 2: C-Components and Component Registry Manager

F. Management Component (M-Component)

M-Components are used to take advantage of QoS services provided by external entities such as the host operating systems (e.g., real-time OS) or network-level QoS support (e.g. DiffServ and RSVP). Each M-Component is designed to be a plug-in to QMan rather than a tightly coupled module. Consequently, QMan can function even without the presence of these management components. Two management components have been designed to enhance QoS functionalities currently: *Resource Manager* and *Connection Manager*. Resource Manager computes and reserves end-host/network resources before an AC Stack is created, provided such reservation is supported by host OS or QoS-aware network. Connection Manager (CM) manipulates group level QoS adaptations. For example, CM enables a group of users of similar QoS requirements to receive more or less layers of encoded data in layered encoding scheme via the *group joining* or *disbanding operations*. All C-Components and M-Components are transparent to applications. The combination of the adaptation of C-Components and the adaptation of M-Components form the main pillar in the QMan holistic approach to adaptive QoS support.

IV. QoS Management Functions

The sequence diagram presented in Figure 3 provides a clear illustration of the workflow of QMan QoS management in conjunction with DPF. Applications first specify QoS requirements and initiate binding operations to the QoS manager in order to construct an AC Tunnel. At the connection setup phase, QoS Agents retrieve information of protocol components available in each end-host from Component Registry. They then feed this information to QoS Manager, which will subsequently conduct the QoS negotiation operation. After that, QoS mapping is performed to obtain DPF specifications of a functional stack that meets the requirements. After the stacks have been synchronized (i.e. the

[1] DataSource, a term in Java Media Framework, is used to manage the transfer of media-contents. A DataSource encapsulates both the location of media and the protocol and software used to deliver the media. Once obtained, the DataSource object cannot be reused to deliver other medias.

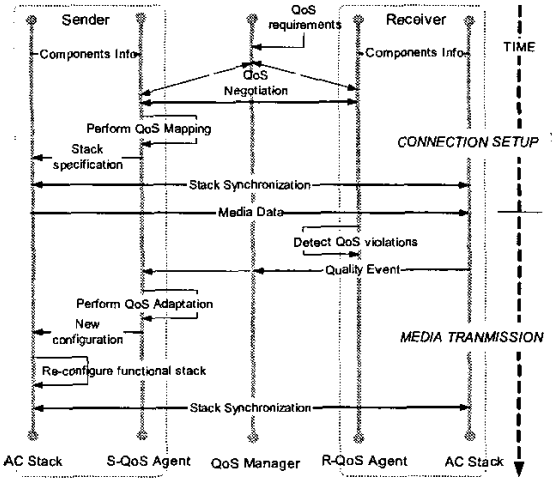

Figure 3: Workflow of QMan and DPF

stacks at both hosts are compatible), the sender and the receiver will proceed to transmit data via the constructed AC Stack.

During transmission, QoS Agent on the receiver side monitors whether the QoS level has been sustained or violated. A *Quality Event* is sent out to the QoS Manager when a QoS violation occurs. The QoS Manager will then conduct QoS adaptation according to adaptation rules and policies to derive a suitable new configuration of the AC Stack. Finally, the re-configuration of the functional stack is performed. Some key steps of QMan management functionalities are outlined below.

A. QoS Specification and Mapping

The QoS specification of QMan is organized in two tiers fashions: application level and middleware level. Applications can utilize the provided APIs to specify *Application Level Requirements (ALR)*, which will be translated automatically or manually into *Middleware Level Requirements (MLR)*. MLR specifies QoS in XML, and the resulting XML file, as mentioned, is known as a QMIB. For each level, there exists a corresponding set of QoS requirements to be met. Each requirement can be represented by a simple data structure consisting of a keyword of QoS parameter with minimum and maximum value. For example, the built-in QMan API *QoSMapper* (Figure 4) provides simplicity of usage with pre-defined keywords such as "reliability" and "bandwidth usage". Application developers can create application-specific GUI for QoS specification based on pre-defined QMan APIs, third party APIs or directly on top of MLR. These APIs should have their desirable QoS requirements translated into the corresponding MLR representations in XML (QMIB). It should be noted that MLR also allows new keywords to be defined which provides the flexibility to extend and customize the QMan framework.

```
QoSMapper mapper = new QoSMapper();
ProtocolStack stack = null;

mapper.setReliability(true);
mapper.setBandwidthUsage(mapper.BANDWIDTH_USAGE_
LOW);
mapper.setClassofService(mapper.TRAFFIC_CLASS_LOW);
mapper.setJitterControl(true);
mapper.setDelayBound(mapper.DELAY_REALTIME);
mapper.addPreferredComponent("IP");
stack = mapper.createStack();
```

Figure 4: Sample code of ALR

QoS Mapping is the process within each end-host to map higher-level application QoS requirements into middleware requirements. Consequently, QMan selects a set of protocol components and deploys them into a functional stack according to the application's QoS requirements. Current QMan implementation supports *pattern matching based QoS mapping*, which provides fast searching. One ALR keyword may be translated to either one or few of pre-defined MLR keywords. Components selection and DPF specification translation are needed to complete MLR to DPF specification mapping. Since MLR contains keywords indicating the required QoS capabilities of C-Components, such mapping can be done by retrieving corresponding C-Components from the Component Registry.

B. QoS Negotiation

Before invoking a binding operation, an application has to specify MLR to QoS Manager. If ALR is being used, it will be translated to MLR either manually or automatically because only MLR elements are exchanged between end-hosts. Hence, the application can define application-specific ALR without affecting management functions. In the binding operation, the first step to perform in a QoS negotiation is to finalize the QoS requirements between end-hosts. QMan conduct negotiation in two aspects: MLR negotiation and components negotiation.

1) MLR Negotiation

Technically, QoS Manager and QoS Agents negotiate to finalize a range of QoS parameter values. If the required range of a particular agent is different from the one selected by the QoS Manager, some compromises will have to take place. QMan supports three methods of MLR negotiation: *system driven* in which the decision is up to the QoS Manager; *sender driven* in which the negotiation takes place between the sender and the QoS Manager; and lastly *bilateral decision* in which sender, receiver and QoS manager are involved in the decision making.

2) Components Negotiation

Once a MLR have been confirmed, components that match MLR will be selected. Although the selection is done in QoS Mapping step, components negotiation ensures that all participants will support the selected components. In a network as technologically diverse as the Internet, each end node may have its own set of system and network resources available. As mentioned early, QMan utilizes a local Component Registry to keep track of the protocol components available for use. For example, in a system where IPv6 is not supported, an IPv6 component should not be listed in the local registry. Components negotiation is then conducted via the *stack*

synchronization mechanism of DPF. Thus we have solved the *service diversity* problem posed at the beginning of this paper.

C. QoS Monitoring and Adaptation

QoS monitoring is the process to detect QoS violations throughout the lifecycle of a session. Any piggyback-like protocols, such as RTP and SFP [6], can be deployed in our framework for QoS monitoring. In addition to monitor at AC Stack, M-Components may also act as monitoring units. For example, an M-Component that utilizes *Simple Network Management Protocol (SNMP)* can be deployed to obtain network status from network devices. When a violation is detected, QoS Monitor will generate a report to the QoS Manager, where the actually adaptation choice will be determined. This approach has the advantage of preventing QoS feedbacks explosion at the sender in case of multicast.

Upon the receipt of a QoS violations report, QoS Manager will consult *Adaptation Rules Table* to perform QoS adaptations. Adaptation rules are predefined by applications or other relevant parties and can be re-installed during runtime. Each adaptation rule consists of three parts: 'when to do adaptation', 'where to do adaptation' and 'how to do adaptation'. Adaptation Rules can be defined to tune performance of a particular protocol (e.g., increase buffer size in case of jitter) or even to modify AC stack. Figure 5 show an example of C-Component adaptation rule: a flow control based transport protocol (e.g., TCP) is to be replaced by another protocol without such feature, when the packet loss rate exceeds 20%. Upon occurrence of such violation, a suitable protocol (e.g., UDP) will be picked out to substitute the original protocol according to this rule.

```
<rule name="rule2" priority="2">
   <when>
     <condition param="loss_rate" bound="++0.2"/>
   </when>
   <where layer=octopus.protocol.plugin.transport">
     <how action="remove" param="flow_control"/>
          <how action="insert"
param="basic_transport"/>
   </where>
  </rule>
```

Figure 5: An XML-based rule to remove a C-Component

In addition to C-Component configuration and adaptation, applications can also specify adaptation rules to perform M-Component adaptations. Such adaptations will not interrupt transmission of media flow because adaptation is performed at external entities, not within the AC Stack.

```
<rule name="rule4" priority="4" description="modify
traffic class">
  <when>
    <condition param="throughput" bound="—80%">
  </when>
  <where
manager=octopus.qos.plugin.resource_manager">
    <how action="modify" param="traffic_class"
value="real_time"/>
  </where>
</rule>
```

Figure 6: An XML-based rule to modify M-Component

Figure 6 shows an M-Component adaptation example whether the Resource Manager will perform re-negotiation for a new traffic class when the throughput of the media flow is less than 80 percent of the required bandwidth. This adaptation is actually performed by the network components, such as DiffServ.

D. Group Level QoS Management

To address the heterogeneity and scalability issues of group QoS, we have extended the QMan framework presented so far to provide adaptive QoS support for group communication. The concept of *group layering* has been introduced where receivers are divided into groups so that all receivers in the same group have the same QoS requirements and share the same data flow. For each group, a QoS Manager is created to manage the session, the whole of which form a QoS management tree to balance loads of root manager and to prevent feedback explosion.

V. Implementation and Evaluations

We have implemented a prototype QMan to demonstrate the feasibility of our approach as well as to test the basic functionalities of the architecture. In our implementation, Java was chosen as the programming language due to its platform independent feature. The reflective API and dynamic class loading features of Java have also found to be essential in the implementation of DPF. The control channel between manager and agent is implemented using the *Remote Method Invocation (RMI)*, which enables Java-based application to include distributed functionality.

The configuration of the testbed is as follows: three personal computers, each equipped with a 2 GHz Pentium IV processor and 512MB memory, are used as end-hosts in our experiments. Microsoft Windows XP is the primary operating system used on these machines. Three additional machines, each equipped with a 400 MHz Pentium III processor and 256MB memory, are used as Linux routers. One of the routers supports traffic shaping [14]; the Token Bucket Filter are used as queuing discipline. End-hostss and routers are connected via 10Mbps hubs where each end-host resides in different subnet. Every end-host has ten C-Components: RTP, TCP, UDP, IP (unicast/multicast), G711 codec, G723 codec, GSM codec, MPEG codec, affine encryption/decryption and stop-and-wait flow control.

A. Functional Verification

Two programs have been written to facilitate the testing the functionality and measurement of the performance of the implemented framework: firstly, an end-host application which define the capabilities of local media devices and registers these devices to a JINI lookup service. Media devices can be discovered by the second application - a JINI control application.

We have then designed several application scenarios to test the functionalities of QMan:

i). Detection of QoS violation due to, for example, packets loss as the result of congestion at routers and the subsequent adaptation of AC stacks by (a) replacing its TCP stack with a UDP stack, (b) changing to a codec with a higher compression ratio (e.g., from MP3 to GSM) during runtime. These tests were conducted for unicast transmission.

ii). Create several multicast groups - audio conferencing, video multicast, and shared white-board – to test the operation of group joining and leaving as well as QoS adaptation.

We found QMan is functioning in the way as expected.

B. *Latency Evaluations*

We subsequently designed some experiments to measure the latency of QMan in different phases of operations: before establishing a network connection, the QoS requirements are mapped into MLR. After that, the QoS Manager conducts QoS negotiations to identify AC Stack for both sender and receiver. Finally, according to agreed meta-stack, QoS Agents invoke DPF to create actual protocol stacks. Time taken for various QMan operations is shown in Table I. The first sub-column under the time column shows the time taken in QoS management when a single QoS Manager is used; whereas the second one shows the time taken when a management tree (tree depth equals to 2) is used for multicast settings.

TABLE I. TIME-TAKEN IN QOS MANAGEMENT FUNCTIONS

Operation	Time (ms)	
	Single	Tree
Creation of QoS Manager (total)	855	849
Instantiation of QoS Manager	156	156
Analyzing adaptation table	485	483
Registration of QoS Manager	214	210
Creation of Sending-QoS-Agent (total)	1640	1646
Instantiation of QoS Agent	16	17
Analyzing components	432	438
Attaching to QoS Manager	1192	1191
Creation of Receiving-QoS-Agent (total)	454	446
Instantiation of QoS Agent	16	16
Analyzing components	438	430
Retrieval of QoS Manager Reference	1000	1030
Retrieval of Receiving-QoS-Agent Reference	1078	1022
QoS Binding (total)	13823	14905
Attaching leaf manager to tree	-	1600
QoS Negotiation	2219	2230
QoS Mapping	26	27
Initializing AC Stack	2880	2880
Opening AC Stack	8698	8668
QoS Adaptation (total)	7663	7682
Delay before notification of violation	2740	2744
Notification of violation	661	680
Looking up for action object	1	1
Delivering of action object	668	668
AC Stack adaptation	3593	3589

As can be seen from the table, the various latencies in multicast settings with the presence of QoS management tree are comparable to those in the case of a single QoS manager.

C. *Invocation delay of RMI*

As mentioned early, RMI is used to establish the control channel between QoS manager and QoS agents. Table II shows the invocation delay of RMI. In this delay measurement, a client simply invoked a remote method, passing difference sizes (16, 32, 64 ... 32768 bytes) of message to the sender. Measurements were taken under different network operational conditions. The first three series were performed in 10 Mbps, 100 Kbps and 50 Kbps connections (slightly overlapped with the line represents 100 Kbps) respectively, without other network program running. The connection bandwidths other than 10 Mbps are simulated by running traffic shaping at the intermediate Router. Another four series of experiment were performed under violated environment; for example, Violate1 was performed in a 100 Kbps connection, with another network program sending media data at the rate of 129 Kbps.

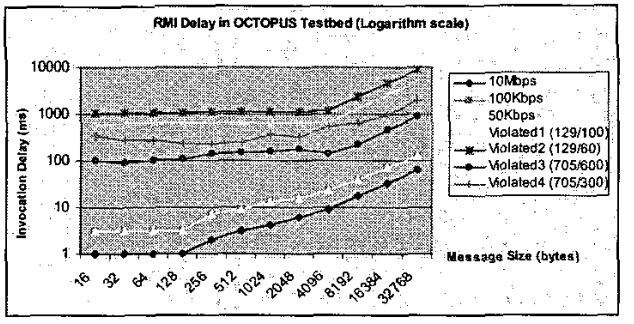

Table II: Handover time vs. notification delay

According to the results, RMI does not contribute significant additional delay to QoS functions at binding-time. As the largest control message is set to 4 Kbytes, RMI produces delay that is less than 10ms (under 10 Mbps network). The loading of configuration files, including available components and adaptation table, incurs around 430 ms and 480 ms of delay respectively, while registering and discovery services (due to JINI) take around 1 second each. However, the reflective API adopted in DPF and its messaging have contributed a significant delay to stack creation which suggests that further investigation and improvement of DPF performance is needed.

In the presence of QoS violations, RMI incurs higher delay where QMan has to issue QoS adaptation commands via the control channel. The receiver's QoS Agent will send the notifications of violations when detecting the violation within a specific time interval of 2.7 seconds. This is to reduce the chances of false raising an alarm on the violation (stability consideration). Notification of violations and delivering action objects to the sender take around 0.7 seconds each. However, processing of QMan takes less than 0.1 seconds because the

RMI has already produced the delay of approximate 0.6 seconds (see the delays of the upper four lines in Table II). Although notification has to be delivered from the receiver's QoS Agent to the leaf, the QoS manager will then forward it directly to the root manager, thereby reducing the delay incurred by half. Therefore, the notification takes 10 ms to be delivered from the leaf manager to the root manager. This is one of the benefits of separating the control path from the data path.

VI. CONCLUSIONS AND FUTURE WORK

A holistic approach for end-to-end QoS control and management has been adopted in the design of QMan. It has the following key features: (i) the ability to provide a high level QoS specification for applications, a QoS-Meta-Information-Based (QMIB) for the specification of QoS requirements and adaptation rules at middleware level, and the translation of the former to the latter; (ii) the ability to provide QoS monitoring and signaling between end-hosts, which allows the QoS requirements and the rules for adaptation to be configurable prior to sessions setup and re-configurable during sessions; (iii) component-based approach where C-Component and M-Component can be dynamically selected, configured and adapted; (iv) the ability to provide multiple strategies for QoS adaptation – from adaptation of hosts' protocol stacks and network QoS support to re-adjustments of QoS requirements by applications during runtime; (v) multicast QoS management tree which can be created to balance the working loads of QoS Manager and prevent feedback explosion.

Leveraging on Dynamic Protocol Framework, QMan offers greater functionality and flexibility in the end-to-end QoS management. Results of tests show that QMan does not introduce unacceptable overhead while offers enhanced functionalities for distributed multimedia applications. Future work can be carried out in several directions. Firstly, a study of DiffServ model and its integration into QMan is needed as the extension to current architecture. Secondly, the validation of adaptation rules should be examined since QMan allows different stakeholders (e.g., application, network administrator) to define rules and policies, which may potentially conflict with each other. Lastly, application QoS requirements should be modeled to prepare QMan with the ability to better support applications from various domains.

REFERENCES

[1] C. Aurrecoechea, A. T. Campbell, L. Hauw, "A survey of QoS architectures," *Mulitmedia Systems*, 1998

[2] N. Klara Nahrstedt, X. Dongyan, W. Duangdao, and L. Baochun, "QoS-Aware middleware for ubiquitous and heterogeneous environments," *IEEE Communications Magazine*, 2001

[3] X. Wang and H. Schulzrinne, "Comparison of adaptive Internet multimedia applications," *IEICE*, June 1999

[4] C.H. Zhang, T.K. Chin, K.Y. Koh, G.M. Ong, C.H. Peng, H.K. Pung, and S. Suthon, "OCTOPUS: A middleware for multimedia communication," *IMSA*, August 2002

[5] Sae-Whong Suthon, Geok Meng Ong and Hung Keng Pung, "An Adaptive End-to-End QoS Management with Dynamic Protocol Configurations", *IEEE International Conference on Networks (ICON '02)*

[6] Object Management Group (OMG), "Control and management of Audio/Video Streams", CORBATelecoms: Telecomm Domain Spec version 1.0, Group Management Protocol, Version 2", *RFC 2236*, November 1997

[7] B. Stiller, C. Class, M. Waldvogel, G. Caronni, D. Bauer and B. Plattner, "A Flexible Middleware for Multimedia Communication: Design, Implementation, and Experience", *IEEE JSAC: Special Issue on Middleware*, Vol. 17, No. 9, 1999

[8] Mitchell S., H. Naguib, G. Coulouris, and T. Kindberg, "A QoS Support Framework for Dynamically Reconfigurable Multimedia Applications", *Proc. DAIS'99*, June 1999

[9] Gerard Parr, Kevin Curran, "A Paradigm Shift in the Distribution of Multimedia", *Communications of the ACM*, Vol 43, No 6, June 2000

[10] G. Xiaohui, N. Klara, Y. Wanghong, W. Duangdao, X. Dongyan, "An XML-based QoS Enabling Language for the Web", *Journal of Visual Language and Computing, Special Issue on Multimedia Languages for the Web*, 13(1): 61–95, Feb.2002

[11] Baochun Li, Klara Nahrstedt, A Control-based Middleware Framework for Quality of Service Adaptations, *IEEE JSAC: Special Issue on Service Enabling Platforms*, Vol. 17, No. 9, pp. 1632-1650, September 1999

[12] K. Nahrstedt, D. Wichadakul, and D. Xu, "Distributed QoS Compilation and Runtime Instantiation," in *Proceedings of IEEE/IFIP International Workshop on Quality of Service 2000 (IWQoS 2000)*, June 2000

[13] D. Bauer, B. Stiller, B. Plattner, "Guaranteed multipoint communication support for multimedia applications," in *SYBEN'98 Broadband European Networks Conference*, Switzerland, May 1998

[14] B. Hubert, G. Maxwell, R. Mook, P. B Schroeder, J. Spaans, "Linux Advance Routing and Traffic Control HOWTO", 2001

[15] D. Schmidt and T. Suda, "Transport system architecture services for high performance communication subsystems," *IEEE JSAC*, vol. 11, no. 4, May 1993