Paving the First Mile for QoS-dependent Applications and Appliances

Mohamed El-Gendy† Abhijit Bose† Seong-Taek Park‡ Kang G. Shin†

†Department of EECS
The University of Michigan
Ann Arbor, MI 48109-2122
{*mgendy,abose,kgshin*} *@eecs.umich.edu*

‡Digital Convergence Center
Samsung Electronics Co.
Seoul, South Korea
stpark@samsung.com

Abstract— **Paving the first mile of Quality-of-Service (QoS) support has become essential for full deployment and utilization of QoS proposals in the Internet. Most of existing efforts have been made to provide network services and control without paying much attention to how applications can use these services. In this paper we design and implement a two-tier architecture, called *QoS Gateway* (QoSGW), that acts as an interface between application QoS requirements and network-provided QoS capabilities. The QoSGW is to support small embedded network devices that rely on network-provided QoS. Our architecture, in its full version, is composed of two key components: (i) an agent that resides on the end-host and provides an adequate interface for QoS-dependent applications, and (ii) a QoS manager that provides an interface to network services for the agents. Using these two components enhances generality and scalability in providing QoS support for Internet applications and end-devices. The QoSGW is intended to promote QoS deployment and facilitate construction of QoS-aware access networks.**

I. Introduction

The past few years have witnessed the convergence of digital technologies such as television, telephony, publishing, and computers. This convergence has yielded the emergence of many *QoS-dependent* applications on the Internet [1], that require certain service quality from the underlying network such as throughput guarantee, reliability, timeliness, and guaranteed delivery. For example, a wide variety of home devices and computing facilities, such as personal desktop computers, voice over IP (VoIP) phones, home telemetry and automation devices, home entertainment (mostly multimedia), shared data storage devices, as well as others, are getting connected via a Home Area Network (HAN) in what is called "smart homes." A HAN, as well as other examples of customer networks, are usually connected to the external network via cable modems, digital subscriber line (xDSL), ISDN, optical fiber, IEEE 802.11 wireless, as well as other standards. Due to the wide range and the dynamic operation of applications running on these devices, they require different network resources, and these requirements vary from local (within a HAN or customer network) to global (end-to-end) resources. Additionally, different physical media connecting the customer network to the external network affect the overall performance of such applications. Maintaining a satisfactory QoS level for applications in such varying network connectivity and complex application interactions poses a significant challenge that requires intelligent resource allocation and admission control as well as adequate QoS support to handle diverse link media and dynamic link quality associated with each medium.

On the other hand, with the introduction of new and different multi-service network technologies to the Internet, it becomes a tedious job to upgrade every application in order to accommodate the new technology. Moreover, managing service contracts with different network service providers as well as resource usage accounting, increase complexity in customer network management and end-host devices. Therefore, an abstraction layer is needed to provide an interface between the QoS-enabled network and QoS-dependent applications. This abstraction layer hides all the complexity of Service Level Agreement (SLA) management, QoS mapping, admission control, and even data plane traffic handling such as packet marking and policing.

Previous approaches depend on either putting all of these functionalities in the end-host operating system (such as IBM AIX [2] and Windows 2000 [3]) or deploying hardware devices, called *QoS appliances* [4–6]. The former approach does not address the QoS needs of other architectures, like small-memory devices and embedded systems, and also creates heavy dependency on the end-host operating systems. This has the disadvantage of relying on the QoS support "canned" in the underlying operating system, hence yielding inflexible solutions and limiting deployability. On the other hand, the latter approach provides only bandwidth management for different applications traffic. However, it does not provide enough flexibility to export network services up to the application level. Usually, some kind of mapping or translation is required between the application QoS requirements and available network services. The intelligence found in current QoS appliances is not sufficient enough to provide such QoS mapping.

To address the above challenges and, at the same time, to provide QoS accessibility and support to QoS-dependent applications (or QoS applications for short) and devices, we present a two-tier architecture, called *QoS Gateway* (QoSGW), that acts as a QoS mediator between these applications/devices and the underlying network QoS infrastructure. The QoSGW also acts as an abstraction layer that hides the complexity of the QoS-enabled network from QoS devices and applications. This has the advantages of shielding the device or the end-host application from the details of the underlying QoS infrastructure and facilitating both management and deployment. Our architecture provides a middleware QoS support for applications, which does not depend on the end-host operating system and does not assume QoS-aware applications.[1] It also provides a transparent support for current QoS applications without the need to modify their source code or operation. We employ the Differentiated Services (DiffServ) [7] network architecture as the underlying QoS infrastructure, but the approach is general enough to be used with other QoS infrastructures. We implemented a prototype system using the Linux operating system to prove the functionality and evaluate the performance of our proposed approach.

This paper is organized as follows. In Section II we present the basic design and operation of the QoSGW architecture and discuss the design considerations. Section III details the components of the architecture. We evaluate the architecture prototype in Section IV, and we contrast our approach with others in Section V. Finally, the paper concludes with Section VI.

II. Basic Architecture and Operation

The QoSGW architecture is composed of two components: an agent that resides on each end-host, called *QoS Agent*, and a manager working for each group of agents, called *QoS Manager*. Depending on the size and capability of the supported device or host, the *QoS Agent* is to be installed[2] to manage applications traffic. The *QoS Agent* provides a transparent QoS support for end-host applications by intercepting their network connections and directing QoS requests to the *QoS Manager* so that it can assign suitable service classes for applications traffic. The host agent also provides Application Programming Interfaces (APIs) to register applications for QoS support. These APIs basically add a QoS profile for each application to be run on the host in the host database, which defines all supported applications on that host. The QoS profiles are used to map application QoS requirements into their equivalent network-level QoS representations that are later used by the *QoS Manager* in assigning service classes. This two-level QoS mapping done to the application requirements allows more flexibility and degrees of freedom in meeting application QoS requirements through available network services. The host agent keeps track of running QoS applications on the host as well as their QoS profiles.

QoS Managers, on the other hand, process QoS requests sent by *QoS Agents* and allocate network resources and suitable service classes to applications traffic. The managers keep track of the SLA with the network service provider and apply appropriate policies and admission control to applications traffic accordingly. They keep two types of SLAs: application SLA (*appSLA*), and network SLA (*netSLA*). Application SLA carries per-flow specific parameters. Each *appSLA* has a mapping to one *netSLA*, which, in turn, defines the associated network service class. Network SLAs are negotiated with the network service provider (ISP), but the negotiation process is outside of the scope of this paper. Applications traffic is mapped to network service classes through packet marking.

Once the *QoS Manager* has been instructed to support a particular application traffic, it starts the admission control process that checks resource availability as well as policy rules. It also verifies the feasibility of meeting the e2e QoS requirements for the requested flow through the specified service class. This process can be summarized as follows: the manager compares the requested QoS against the available service classes defined in the list of *netSLAs*. If a class matches, then the capacity of that class is examined to see whether there is enough bandwidth to accommodate the new request. If there is enough capacity, then the manager starts an e2e QoS verification procedure that involves a signaling protocol like RSVP [8]. The result of this e2e verification procedure determines if the request can be admitted or not. If the request is admitted, the manager installs a new *appSLA*, which defines QoS parameters for that request, and starts installing traffic classifiers and traffic markers for the traffic flows.

Finally, a reply is sent back to the host's *QoS Agent* identifying the result of the admission and the QoS mapping process. Then, the application can either start sending traffic under the specified network service class or choose to modify its QoS requirements if the original specifications could not be met by the network. In the absence of *QoS Agent*, as in the case of small and embedded devices, direct device/application support is provided in the manager to replace the functionality of the agent. Figure 1 shows a general architecture of using *QoS Agents* and *QoS Managers* to establish services for applications running on hosts as well as on small and embedded devices.

A. Design Considerations

One important feature of our approach is that it is software-based architecture, which can be easily reconfigured and upgraded whenever there is any change in the working environment or the network support. This way, it acts as a virtual QoS stub that hides significant changes in the external network from the local network, hosts, and applications. Listed below are some of the design considerations and unique features of the QoS Manager/Agent approach:

- **Isolation:** The QoSGW provides an abstraction level that hides the complexity of the underlying network QoS infrastructure. It provides a uniform interface for applications to external networks such as IntServ, DiffServ, or even IEEE 802 network QoS (Figure 2 illustrates this).
- **Closeness and transparency to applications:** The use of an agent that resides on the host and interacts closely with QoS applications has the virtue of capturing the

[1]QoS-aware applications are the ones that use QoS APIs to convey their QoS requirements.

[2]Small embedded devices will not be able to accommodate a QoS Agent.

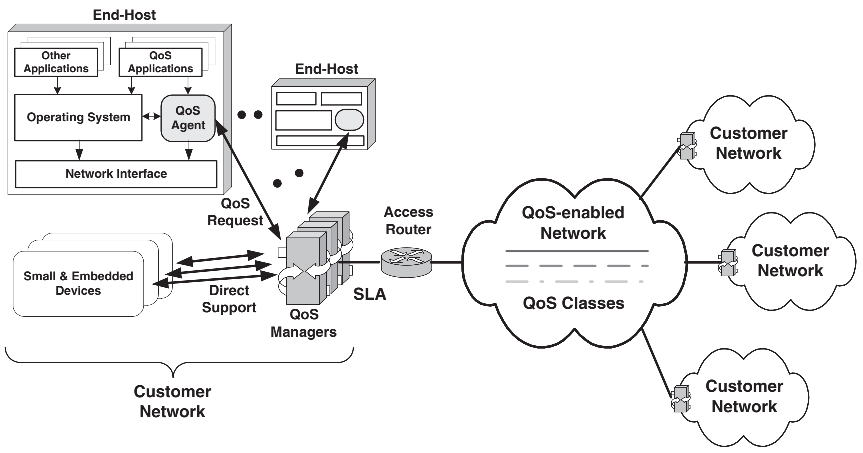


Fig. 1. Basic architecture for the QoS Gateway

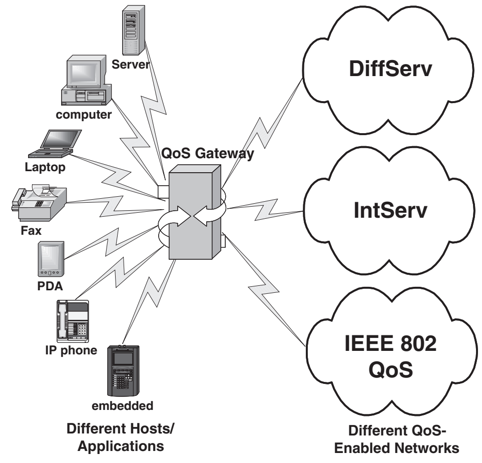


Fig. 2. Abstract interface to heterogeneous devices and networks

application requirements accurately. At the same time it provides a transparent QoS support (through a middleware) without the need to modify the applications' source code or the underlying end-host operating system.

- **Flexibility:** By contrast, the use of stand-alone *QoS managers* provides support to small and embedded QoS-dependent devices that cannot accommodate host agents such as PDAs, IP phones, and others. Also the ability to support different underlying networks is an important consideration in the architecture design.
- **Easy administration:** Administration of a few QoSGWs in a customer network is relatively easier than managing every host and device in the network. This includes policy, resource, or SLA updates. Moreover, controlling the network policy from a trusted device, *QoS Manager*, is securer than distributing the policy among various trusted and untrusted devices and applications.
- **Easy upgrade:** Separating functionality inside the QoSGW is very important to facilitate the upgrade and update of the architecture components in order to support future and new network QoS frameworks. Both entities of the QoSGW follow a modular design that allows replacement, addition, or even removal of some of the functionality and protocols involved in the operation.

III. QoSGW Design and Functions

We first detail the design of *QoS Agent*, specifying the main building blocks, functions, and their interactions, and then present the detailed design of *QoS Manager*.

A. QoS Agent

The main functions of a *QoS Agent* is to capture applications' QoS requirements and other traffic characteristics, perform initial QoS mapping, and then communicate the application requirements to the *QoS Manager*. The *QoS Agent* is designed as a middleware that resides on the end-host and enables applications running on this host to access QoS in the network. Using a middleware design eliminates the need for modifications of the host operating system. The *QoS Agent's* functionality is divided into two parts, a QoS module (*QoSmod*), which works in the operating system's kernel space,[3] and a user-space daemon, which is called QoS daemon (*QoSd*). The daemon and the module communicate via a special channel or a device driver.

The *QoSmod* intercepts applications' network connections and contacts the daemon to perform QoS mapping, and admission control with the *QoS Manager*. The application has to be registered with the host agent in order to intercept its network connections. Specific APIs are used to register new applications' QoS profiles in the host database. Before storing the profile in the database, the agent maps the application's QoS requirements to an intermediate form that is composed of bandwidth, delay, jitter, and loss. These parameters are called "network-level QoS." This is the first level of QoS mapping done to the application requirements. Figure 3 shows the main building blocks of the *QoS Agent* as well as their interactions, and in what follows, we give details of the functionality of each building block.

1) Socket-Call Capturing Module: QoSmod provides a transparent interface to registered QoS applications through intercepting their socket system calls. For each intercepted socket call, the module notifies the user-space daemon, *QoSd*, and delivers important information about the call to the daemon. This information includes the type of the call, the calling application name, the process identifier, network addresses, ports, and protocol type. This information is used by *QoSd* to locate the QoS profile of the application in the database and performs QoS mapping. The network addresses and ports as well as protocol type are sent within the QoS request to the *QoS Manager* to be used in traffic classifiers built for this application traffic on the manager. The *QoSmod* blocks the application (similar to waiting for a device) until a reply

[3]A kernel module in our Linux implementation.

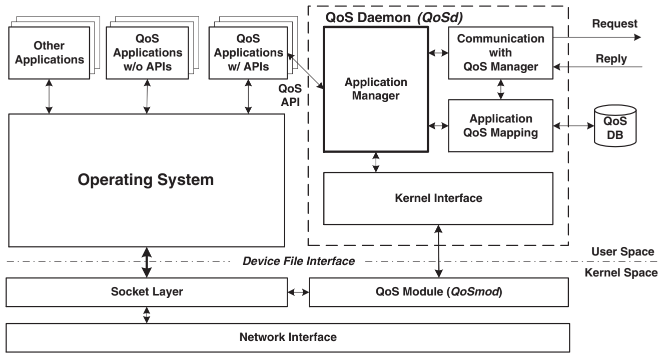


Fig. 3. *QoS Agent* main building blocks

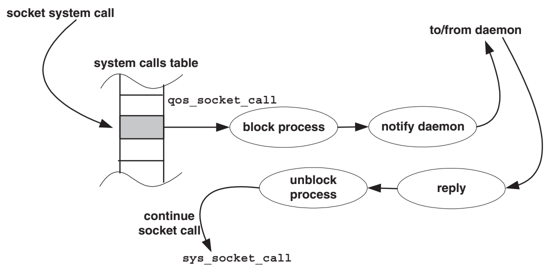


Fig. 4. *QoSmod* operation

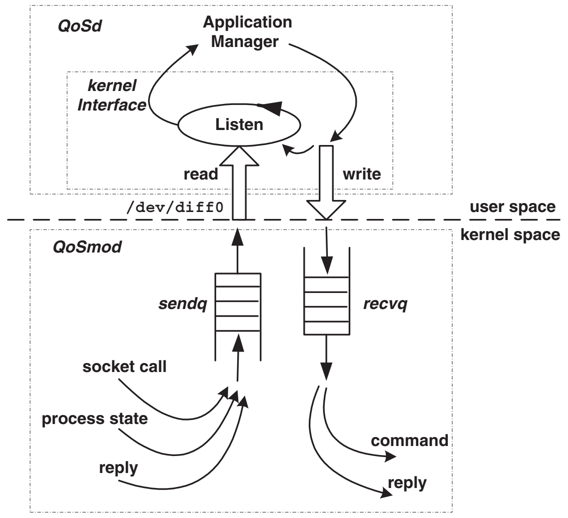


Fig. 5. Kernel interface

comes back from the daemon [9]. Once a reply is received from the daemon, the module decides whether to continue the connection normally (if the application's QoS can be met) or to drop the connection (if the application's QoS is rejected). Then, the module unblocks the application with either normal or error code. Figure 4 shows how *QoSmod* processes a system call from a QoS application.

In our Linux-based prototype, the module communicates with the user-space daemon through a special character device file called `/dev/diff0`. The module, when loaded, attaches itself to this device file, and provides read, write, poll, as well as other necessary functions to serve the file in the kernel space [9, 10]. The module also creates two message queues, a "send" queue (`sndq`) and a "receive" queue (`recvq`), for sending and receiving messages to and from the *QoSd*, respectively.

The daemon, on the other side, listens to the device file waiting for any message sent from the kernel module. Messages sent to the daemon are either reporting a socket call, informing the daemon of a process state change, or replying to a request from the daemon. A message reporting a socket call carries the user and the process identifiers of the calling process. It also carries the necessary information about the socket call like the protocol family (e.g., AF_INET), the type of the call (e.g., accept, connect, send), the protocol (e.g., TCP, UDP), and the `sockaddr` structure that have the addresses and port numbers of the two ends of the socket call. A message reporting a process state change contains the process identifier, and the current status of the process. Finally, a reply message to a daemon request contains the result of executing such a request.

Messages received from the daemon contain a message type, process identifier, size, and a command from the application manager. There are two types of commands, either terminating a certain process (or application) or replying to a socket call message from the kernel module. Upon receiving a terminate command or a terminate reply, the kernel module terminates the process immediately. Figure 5 illustrates the kernel interface in both the daemon and the kernel module and shows how they work together.

2) Application Interface and Management: The *QoSd* provides a simple API, called `RequestQoS()`, to specify application's QoS requirements to *QoS Agent* and hence to the *QoS Manager*. The API supports three functions, `addQoS`, `modQoS`, and `delQoS`. The first function, `addQoS`, is used to create a new QoS profile for an application's traffic and store it in the host's QoS database. The profile contains information about the name of the application, its type, number of flows generated by the application, traffic profiles of the flows, required QoS for each flow, user authentication information, and a unique identifier. Application's QoS requirements are mapped first to their equivalent network-level QoS before being stored in the host database. We refer to this procedure as "application registration." Figure 6 illustrates the main steps for registering an application with *QoS Agent*, where the *QoS Manager* is contacted only to find a matching network service class. However, the manager does not install traffic controls until the application starts sending traffic as will be shown later. After successful creation of the application's QoS profile, it can be modified or deleted altogether using `modQoS` and `delQoS` functions, respectively. The API functions can be used by QoS-aware applications or a manual configuration interface for other legacy QoS applications.

The Application Manager keeps track of registered QoS applications running on the host. When an application starts, *QoSmod* intercepts the application's socket call and notifies the Application Manager while blocking the application. If the application is registered in the QoS database, then the corresponding QoS profile is fetched and a QoS request is built to be sent to the *QoS Manager*. The *QoS Manager* is contacted to perform admission control, network service class selection, *appSLA* installation, and traffic control. When the QoS Manager replies back to the agent, the Application Manager adds this application to the active list and unblocks the application to continue normal socket call processing. This procedure is called "application invocation." If the application is not registered, a normal socket call will continue without contacting the *QoS Manager*. When an active application terminates,[4] the Application Manager notifies *QoS Manager* to delete the associated *appSLA* and free the resources allocated to the application's flows. Figure 7 illustrates the application invocation procedure.

3) QoS Mapping: QoS applications may have their own definitions for QoS and one of the main functions of *QoS Agent* is to translate these various definitions to a common QoS form to be sent to the *QoS Manager* and hence map it to a specific network service class. The QoS mapping module takes care of implementing different mapping algorithms for this translation process. The module also keeps track of the QoS database stored on the host, which records all QoS applications that can run on this host. Records in this database carry a common form for applications QoS which is referred to as the "canonical" QoS. This canonical form specifies QoS requirements for bandwidth, delay, jitter and loss. Specifically, it consists of a committed information rate (CIR), a peak rate (PIR), mean delay, maximum delay, jitter, and loss. We believe that any application's QoS can be mapped to this finite set of parameters using the appropriate algorithms. The QoS database carries also information about the traffic characteristics of these applications as well as information about their users. The latter information enables the agent to treat the same application differently depending on the user. We use the term "QoS profile" to describe an entry in the QoS database. A typical application's QoS profile consists of the application name (as a unique identifier), user and group identifiers,[5] flow identifiers (addresses and port of the traffic), protocol, traffic profile, and canonical QoS object.

In this paper we only give an example of this mapping process; the full set of mapping algorithms is out of the scope of this paper. Consider a video transmission application that wishes to transmit frames at a rate of 30 frames/sec. Each frame can be sent at two different resolutions, a high resolution of 1280×1024 and a low resolution of 1024×768, and the application can tolerate the loss of 10% of frames every 30 consecutive frames.[6] At the other end, frames have to be played back at a specific speed and according to a play-out timer. If a frame is late by $1/2$ of a frame time, it is considered as lost, and is not played back. Consider also that the video received is used to activate a control action based on its content, and this control action has to be taken within 100 msec.[7] The algorithm for this type of applications is straightforward. The CIR is the $rate$ (in bits/sec) for transmitting lower resolution frames, and the PIR is the rate for transmitting higher resolution frames. The delay would be equal to the control delay (100 msec). The e2e jitter equals $1/2 \times frame\_size/rate$ and the loss rate would be a $(1/10 \times frame\_size)/(30 \times frame\_time)$ in bits/sec, where $frame\_time$ is the $frame\_size$ divided by the $rate$. This shows an example of how to extract the canonical QoS form from the application's QoS specifications. However, the job is not always easy like this example, and it can be a very complex operation and may also involve the user's perspective, which is still an open issue.

QoS requests sent from the *QoS Agent* to the *QoS Manager* contains a canonical QoS object specifying the required application's QoS after being mapped, and a traffic profile description of the application traffic flows. The traffic profile follows the Dual Leaky Bucket (DLB) traffic model and includes the average rate, peak rate, burst size, average packet size, and maximum packet size of the flow. In addition to the QoS and the traffic profile, the agent sends traffic identifiers to the manager to be used in the traffic classifiers installed on the manager.

B. QoS Manager

The *QoS Manager* acts as a QoS server for the set of agents in the local customer network, and provides an interface for applications to access various network QoS facilities and service classes. As shown earlier in Figure 1, the *QoS Manager* serves two types of end-hosts: hosts running *QoS Agents* as well as hosts and devices that do not have QoS support, e.g., small embedded devices. In the latter case, the *QoS Manager* performs all the *QoS Agent*'s functions.

For flexibility and upgradability, the *QoS Manager* is composed of two planes, a control plane (called *Gateway*) and a data plane (called *Diff Agent*). The *Gateway* handles high-level QoS control functions such as SLA management, QoS mapping, admission control, and network QoS signaling. These functions are independent of the underlying network environment and can work with various QoS network infrastructures. The *Diff Agent*, which handles traffic marking, traffic classification, and traffic control, is network-specific and tailored to match the supported QoS network infrastructure such as DiffServ or MPLS. These two components communicate with each other using standard socket APIs. Therefore, these components can be located on different machines for fault-tolerance and scalability.

Our current architecture uses the DiffServ framework as the underlying QoS network infrastructure, and hence, the *Diff Agent* is DiffServ-specific. Figure 8 illustrates the main

[4]*QoSmod* notifies the daemon with application closing connection.

[5]We assume a UNIX-based or other multi-user system.

[6]This depends on the codecs used to send frames.

[7]Video sensors are examples of such a situation.

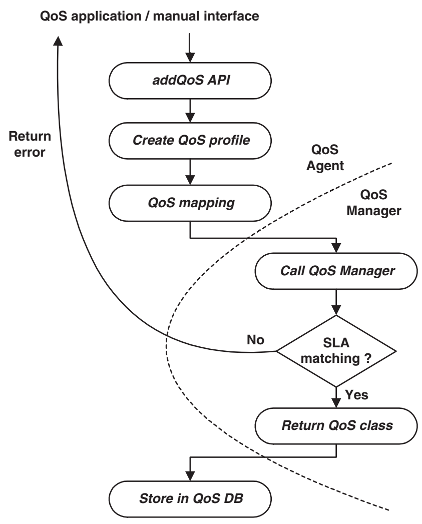


Fig. 6. Application registration procedure

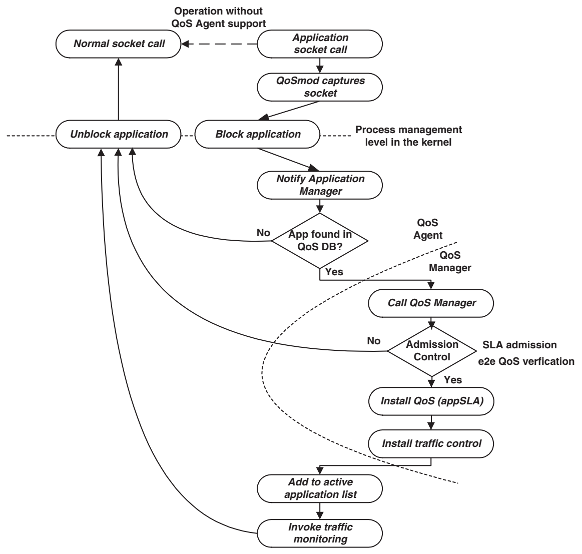


Fig. 7. Application invocation procedure

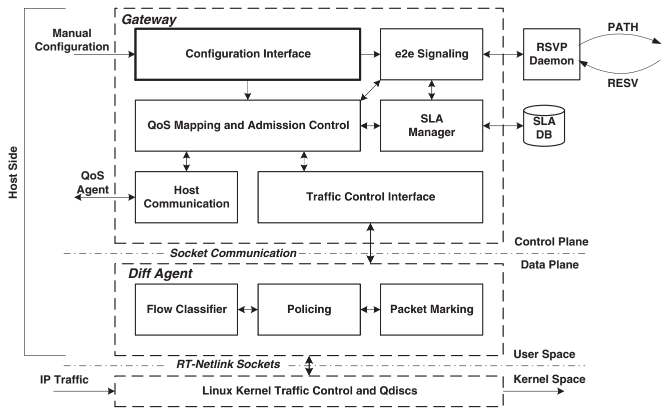


Fig. 8. *QoS Manager* main blocks

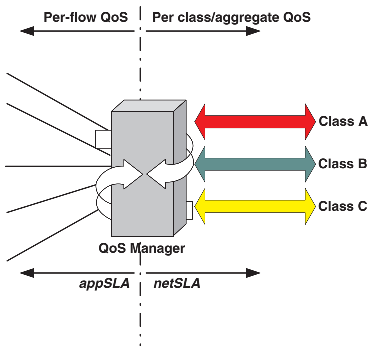
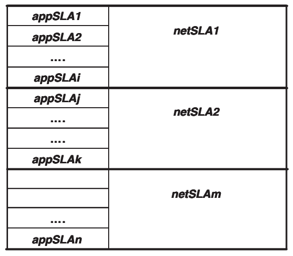


Fig. 9. Network SLA (*netSLA*) and Application SLA (*appSLA*)



appSLA1	netSLA1
appSLA2	
....	
appSLAi	
appSLAj	netSLA2
....	
....	
appSLAk	
	netSLAm
....	
appSLAn	

Fig. 10. *netSLA* ↔ *appSLA*

building blocks of the *QoS Manager* as well as their interactions, and the following subsections will give details of the functionality of each building block.

1) SLA Management: The *QoS Manager* acts as an interface between the offered network service classes and the application's QoS requirements, and manages SLA with the network service provider. It maps application requirements to appropriate network service classes in the SLA. As mentioned in Section II, the *QoS Manager* keeps track of two types of SLAs, network SLA (*netSLA*) for each network service class, and per-flow SLA (*appSLA*) for each active QoS application as illustrated in Figure 9. This is the second level of QoS mapping, which maps *appSLAs* to *netSLAs* and is illustrated in Figure 10.

Each *netSLA* entry consists of a unique identifier, a DiffServ service class, a list of DSCPs (each with 8 bits) associated with this service class, allocated bandwidth, available bandwidth, and a bandwidth sharing indicator (for bandwidth borrowing from other classes). Entries for *netSLAs* are stored in a database "SLA DB" as shown in Figure 8. On the other hand, each application flow is associated with an *appSLA* entry. Each entry consists of a unique identifier, a flow identifier (containing source, destination, ports, and protocol type), a traffic profile of the flow, application's QoS in the canonical form, and a pointer to the containing *netSLA*. As application flows are admitted, *appSLAs* are created for each flow and mapped to suitable *netSLAs*. Available bandwidths in *netSLAs* are adjusted according to subscribing applications.

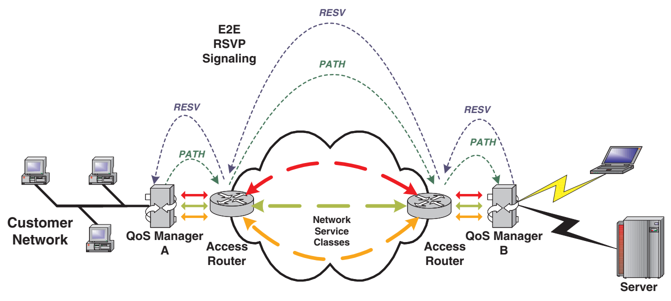


Fig. 11. E2E QoS verification via RSVP

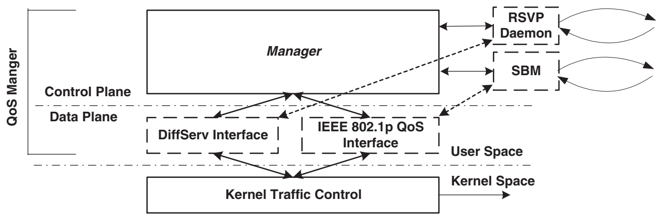


Fig. 12. Flexibility in *QoS Manager* Design

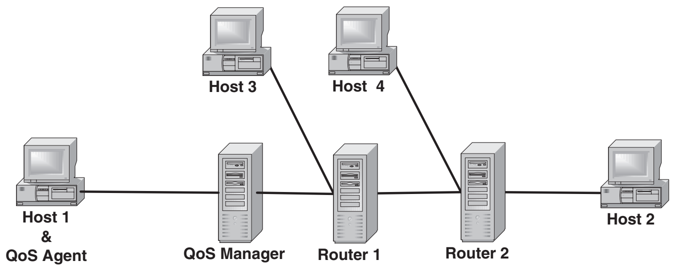


Fig. 13. The evaluation network

2) QoS Mapping and Admission Control: Typically, network service providers specify services classes in their SLAs in terms of capacity (amount of bandwidth available), latency (delay), delay variation (jitter), supported packet or frame sizes, availability (percentage of time available), reliability (sometimes referred to as loss rate), and schedule of operations. The *QoS Manager* maps the QoS of the application (*appSLA*) to one of the available services in the SLA (*netSLA*) by matching the values of its canonical QoS parameters to the service specifications. A simple example is the mapping of an application's QoS to one of the available service classes in a DiffServ-based network. For an *appSLA* that requires delay and jitter guarantees, it is assigned to a suitable Expedited Forwarding or EF-based service [11]. For an *appSLA* that has requirements on bandwidth and loss, it is assigned to a suitable Assured Forwarding or AF-based service [12]. By "suitable" we mean matching the values of individual QoS parameters with the values of the service specifications. In most cases, this mapping process is simpler than the first level of application's QoS mapping done by the host agent, and mapping algorithms for other types of networks can be found in [1, 13]. For small embedded devices, QoS mapping is done solely in the *QoS Manager* by manual programming. It simply involves creation of an *appSLA* for the device traffic that specifies the required QoS and the traffic-generation behavior. Then, QoS mapping and admission control are invoked to assign the device traffic to a network service class with matching e2e QoS. Mapping algorithms for different devices will be covered in a future paper.

The manager admits the *appSLA* if there is enough available bandwidth in the corresponding *netSLA*. The manager also starts an e2e admission control process, called "e2e QoS verification" using a well-known signaling protocol, such as RSVP [8, 14]. The manager invokes the RSVP admission test by communicating with a co-located RSVP daemon using RAPI [15]. We assume that RSVP processing along an e2e path is similar to that in [13, 16]. It is important to note that the use of RSVP in our architecture is for admission control only and not for resource reservation. Figure 11 illustrates the process of e2e QoS verification.

3) Traffic Control and Marking: After admitting a flow, the manager installs traffic classifiers and traffic policers based on the flow identifier and traffic profile in the flow's *appSLA*, respectively. Packet markers, such as those in [17, 18], are installed for the traffic flows using the DSCP assigned to the flow from the corresponding *netSLA*.[8] When the flow terminates, the manager deletes the corresponding traffic controls and markers and frees their resources.

4) Flexibility in QoS Manager Design: One of the QoSGW design considerations is flexibility to work with different underlying network frameworks. Figure 12 shows this flexibility in separating the control plane from the data plane in the *QoS Manager*. We can use multiple different data planes (e.g., DiffServ-specific, IntServ-specific, IEEE 802) that work with different underlying networks. However, the control plane remains unchanged. Moreover, the e2e signaling module in the *QoS Manager* can be replaced by a compatible module that uses network-specific signaling protocols (e.g., RSVP, Subnet Bandwidth Manager or SBM). If the manager is connected to multiple networks at the same time, then it can switch between different data planes.

IV. Evaluation

To demonstrate the effectiveness of the QoSGW in providing QoS for applications, we conducted several experiments using the prototype we built. We use a simple network testbed shown in Figure 13. Linux-based PCs are used in the testbed, where all PCs are 600 MHz Pentium III with 256 MB RAM and running Linux kernel 2.4.2 with QoS and traffic control enabled. The links between all PCs are 100Mbps Ethernet point-to-point. Host 1 is running an application, which sends traffic to Host 2 passing through *QoS Manager*, Router 1, and Router 2. Both Routers 1 and 2 are DiffServ-enabled routers implementing a simple Expedited Forwarding (EF) PHB based on priority scheduling. Host 1 is running *QoS Agent* that

[8]Assuming the DiffServ network architecture.

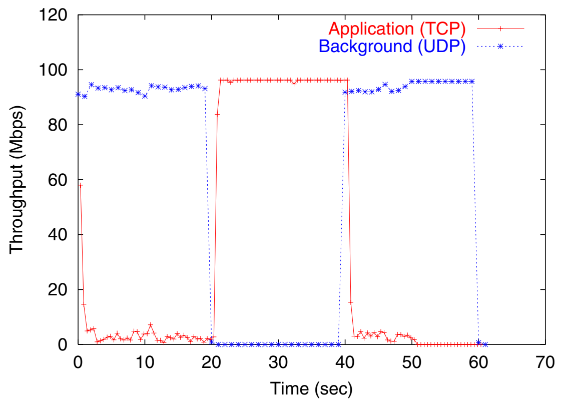


Fig. 14. TCP throughput – QoSGW deactivated

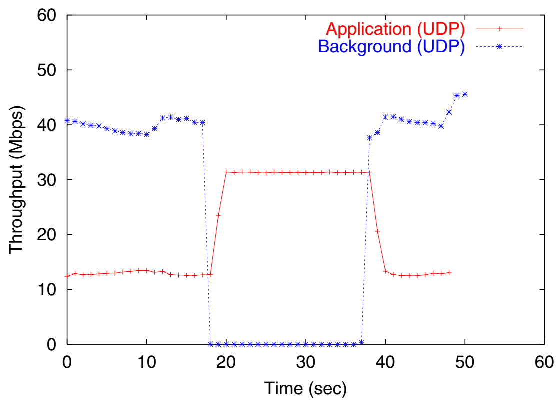


Fig. 16. UDP throughput - QoSGW deactivated

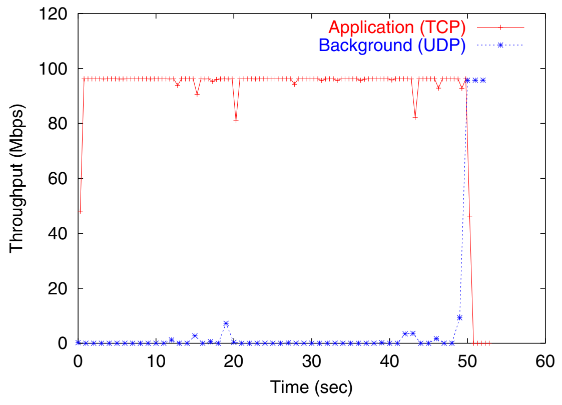


Fig. 15. TCP throughput – QoSGW activated

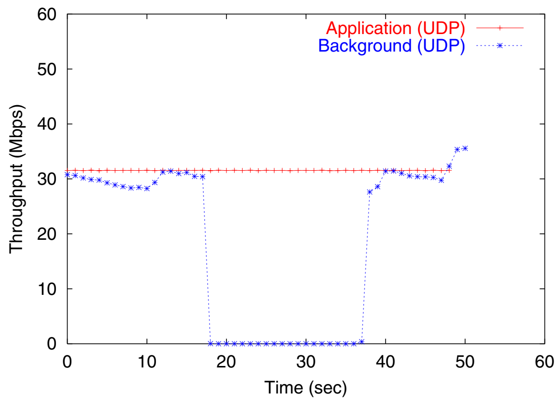


Fig. 17. UDP throughput - QoSGW activated

has the application's profile in the QoS database. When the *QoS Manager* is active, the application's traffic is to be mapped to an EF-based service defined in the manager's SLA database; otherwise, the application's traffic will be treated as best-effort. In addition to Host 1, both Hosts 3 and 4 are sending best-effort traffic (as background) to Host 2 that enter through Routers 1 and 2, respectively, sharing the path and link bandwidth with the application's traffic. We use ON/OFF UDP background traffic with 20 sec ON and 20 sec OFF periods, and each experiment is executed for a duration of 50 sec.

A. *Protection of Important Application Traffic*

In the first set of experiments, the application starts sending TCP traffic, and we compare the two cases of activating and deactivating the QoSGW. Figure 14 shows the performance of the application's traffic in terms of achieved throughput when the gateway is deactivated. As one can see, the application's traffic is not protected, and hence, suffers severely from the background traffic during the ON period. The loss here is 100%. In Figure 15, the gateway is activated and marks the application's traffic as EF traffic, and hence it does not suffer from any loss.[9]

[9]We can see small glitches at times 20 and 45 sec when the ON/OFF traffic changes mode, but this is negligible.

In the second set of experiments, the application sends real-time UDP traffic at a constant rate of 30 Mbps. In Figure 16, we plot the throughput of the application's traffic and the background traffic when the gateway is deactivated. It is clear that the application could not achieve more than 13 Mbps during the ON period of the background traffic as no protection is available. In Figure 17, we activate the gateway and as a result, the application can achieve the full 30 Mbps target rate without being affected by the background traffic. Not only throughput but also jitter and loss are affected. In Figure 18, we compare the jitter with and without the gateway. Without the gateway, and during the ON period of the background traffic, the application's jitter is between 0.5 and 0.6 msec, while during the OFF period it is between 0.2 and 0.45 msec. This unpredictability in jitter is undesirable in real-time applications. When the gateway is activated, the application's jitter is almost constant at one level and unaffected by the background traffic. The application's loss is plotted in Figure 19 in the two cases. The application suffers no loss when the gateway is activated, in accordance with Figure 17.

B. *Overhead*

We now consider the overhead in the gateway operation in supporting application's traffic. The first overhead is associated with application registration using the `RequestQoS()` API.

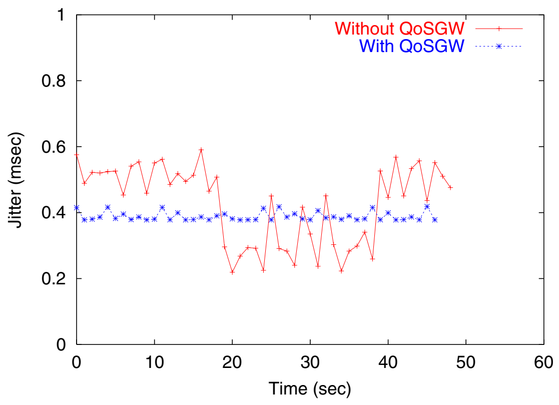


Fig. 18. Comparing jitter with and without the QoSGW

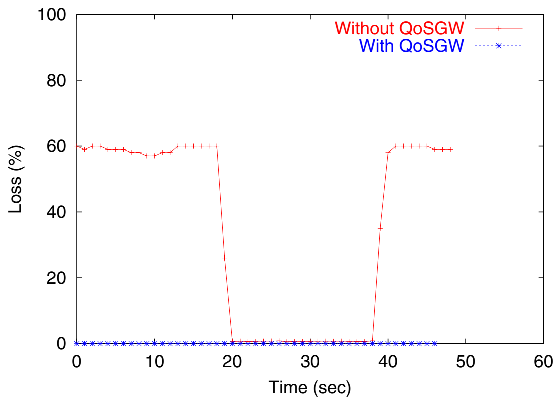


Fig. 19. Comparing loss with and without the QoSGW

This is usually done once for each application running on the host and can be performed as part of the application installation. Thus, no major overhead in this step. Second, at application startup time, the *QoS Agent* intercepts the application's connection and contacts the *QoS Manager* for QoS processing. This process involves a message from the kernel module to the daemon, followed by QoS profile lookup, then a network message from the agent to the manager, followed by a sequence of function calls to SLA and admission control. Without accounting for the e2e RSVP QoS verification, the manager replies back to the agent with an admission result message and installs the required traffic controls (this involves messages between the *Gateway* and the *Diff Agent*), then the agent unblocks the application and traffic starts transmission. Although this involves a number of messages to process the QoS request, only two of them are significant: the network messages between the agent and the manager.[10]

We measured the overhead caused by all these operations which was about 0.22 sec. Although the prototype is not optimized for performance, we find the setup time is not a major overhead, especially when it is executed only once at the application startup. Measuring the performance of RSVP signaling is outside the scope of this paper. The only overhead left to account for is the overhead of marking the application's traffic on the egress interface of the manager. This process is done inside the Linux kernel's traffic control [19] code and it adds only a call to `ipv4_get_dsfield()` function which changes the DS-field in the IP headers of each packet. This adds an insignificant overhead in the packets' path. Overall, the overhead of the QoSGW is insignificant compared to its added value.

[10]Assuming both components of the manager are on the same machine.

V. Related Work

A similar architecture to the QoSGW was presented in [20], which is called the QoS Broker. The QoS Broker orchestrates resources at the end-points and coordinates resource management across layer boundaries. Their architecture is tailored to ATM and manages application, network, and system QoS parameters. The QoS Broker architecture shares some common features with the QoSGW, such as hiding network configuration and operational details from applications and higher layers, and the use of a module-structured design, but the main difference is the use of two-tier architecture in the QoSGW. This has the advantages of easy upgrade, support for heterogeneity, resource aggregation, and security.

Two examples of QoS-enabled Residential Gateway (RG) are presented in [21, 22]. The first system mainly controls bandwidth allocation for different devices in a HAN. The authors classified home-generated traffic into seven categories based on bandwidth requirements. The second system works on layer 2 (MAC) and layer 3 (network), and is compliant with existing standards such as IEEE LAN Subnet Bandwidth Manager (SBM), and RSVP. The QoSGW architecture provides a more general architecture in terms of functionality and deployment environment.

Providing QoS APIs through the socket layer has been presented in [23] and called QoSockets. The authors introduce an extension to the UNIX socket interface that support e2e application QoS management. Although they share a similar goal with our architecture, the main disadvantage of their approach is the lack of transparent support. For the applications to take advantage of the new socket calls, they have to be reprogrammed to use the QoS APIs. Our architecture provides a transparent support for both types of applications, those that are not using the APIs and those that use the APIs.

In [24], a QoS Library Redirection (QLR) approach was presented on Microsoft Windows systems. The authors use a similar idea of intercepting socket calls from the Import Address Table (IAT) on Windows, and redirect the calls to add packet marking to the traffic before starting. However, this work is considered a subset of our architecture as no consideration for QoS mapping, SLA management, or admission control were presented in QLR. The QoSGW architecture provides a complete integrated system, not just APIs, for providing QoS support to applications.

VI. Conclusions

In this paper we presented a QoSGW architecture for providing QoS support to QoS-dependent applications and devices in customer networks. The architecture is composed of two main components or tiers: an agent that resides on the end-host and provides an adequate interface to QoS-dependent applications, and a QoS manager that provides an interface to network QoS capabilities for the agents. Our system acts as a QoS mediator between applications/devices and the underlying network QoS infrastructure. It also acts as an abstraction level that hides the complexity of the QoS network from devices and applications and provides easy management and deployment for QoS. Using the two-component approach provides better generality and scalability. We have built a prototype on Linux operating system implementing the main functionality of the overall design to test the operational correctness and to demonstrate the importance of our system in supporting real-time applications. We are planning to study different QoS mapping algorithms to be used within the QoSGW as this is one of the most important functionalities of the architecture. A system like ours will promote QoS deployment and facilitate building QoS-aware customer networks.

Acknowledgment

The work reported in this paper was supported in part by Samsung Electronics and the ONR under Grant N00014-99-0465.

References

[1] D. Verma, *Supporting Service Level Agreements on IP Networks*. Macmillan Technology Series, 1999.

[2] A. Mehra, D. Verma, and R. Tewari, "Policy-Based DiffServ on Internet Servers: The AIX Approach," *IEEE Internet Computing*, September-October 2000.

[3] "Windows 2000 Features in Support of Differentiated Services," July 2000, white paper, http://www.microsoft.com/TechNet/win2000/win2ksrv/diffserv.asp.

[4] Packeteer, "Packetshaper," http://www.packeteer.com.

[5] A. Communications, "Netenforcer," http://www.allot.com.

[6] S. Networks, "Qosworks," http://www.sitaranetworks.com.

[7] S. Blake, D. Black, M. Carlson, E. Davis, Z. Wang, and W. Weiss, "An Architecture for Differentiated Services," IETF, RFC 2475, December 1998.

[8] L. Zhang *et al.*, "RSVP: A New Resource ReSerVation Protocol," *IEEE Networks*, September 1993.

[9] D. Bovet and M. Cesati, *Understanding the Linux Kernel*. O'Reilly, 2001.

[10] A. Rubini and J. Corbet, *Linux Device Drivers*. O'Reilly, 2001, 2nd ed.

[11] B. Davis *et al.*, "An Expedited Forwarding PHB (per-hop behavior)," IETF, RFC 3246, March 2002.

[12] J. Heinanen, F. Baker, W. Weiss, and J. Wroclawski, "Assured Forwarding PHB Group," IETF, RFC 2597, June 1999.

[13] Y. Bernet *et al.*, "A Framework for Integrated Services Operation over DiffServ Networks," IETF, RFC 2998, November 2000.

[14] J. Wroclawski, "The Use of RSVP with IETF Integrated Services," IETF, RFC 2210, September 1997.

[15] R. Braden and D. Hoffman, "RAPI – An RSVP Application Programming Interface (Version 5)," IETF," Work in progress, Internet-draft, draft-ietf-rsvp-rapi-01.ps, February 1999.

[16] J. Wroclawski and A. Charny, "Integrated Service Mappings for Differentiated Services Networks," IETF," Work in progress, Internet-draft, draft-ietf-issll-ds-map-01.txt, August 2001.

[17] J. Heinanen and R. Guérin, "A Two Rate Three Color Marker," IETF, RFC 2698, September 1999.

[18] M. El-Gendy and K. G. Shin, "Equation-Based Packet Marking for Assured Forwarding Services," in *Proceedings of IEEE INFOCOM'02, New York*, June 2002.

[19] W. Almesberger, "Linux Network Traffic Control - Implementation Overview," in *Proceedings of the 5th Annual Linux Expo*, May 1999.

[20] K. Nahrstedt and J. Smith, "The QoS Broker," *IEEE Multimedia Magazine*, vol. 2, no. 1, Spring 1996.

[21] B. Lei, A. Ananda, and T. Teck, "QoS-aware Residential Gateway," in *Proceedings of the 27th Annual IEEE Conference on Local Computer Networks (LCN'02)*, 2002.

[22] D. Bansal, J. Bao, and W. Lee, "QoS-Enabled Residential Gateway Architecture," *IEEE Communications Magazine*, April 2003.

[23] P. Florissi, Y. Yemini, and D. Florissi, "QoS Sockets: a new extension to the sockets API for end-to-end application QoS management," *Computer Networks*, vol. 35, no. 1, January 2001.

[24] W. Hwang *et al.*, "An Approach of QoS Library Redirection Method for DiffServ in Microsoft Windows Systems," in *Proceedings of IEEE GLOBECOM'01, San Antonio, TX*, November 2001.