QoSOS: AN ADAPTABLE ARCHITECTURE FOR QoS PROVISIONING IN NETWORK OPERATING SYSTEMS

Marcelo F. Moreno, Carlos de S. Soares Neto, Antônio Tadeu de A. Gomes, Sérgio Colcher, and Luiz Fernando G. Soares

Abstract - The increasing demand for distributed multimedia applications makes evident the need for end-to-end quality of service (QoS) provisioning. Particularly, operating systems, despite their location at end systems, switches or routers, must guarantee that resources under their control are adequately managed to fulfill the application requirements. This work proposes an architecture for adaptive QoS provisioning on network operating systems (QoSOS), focusing mainly on the packet queuing subsystem. The development of such architecture came after an analysis of solutions currently found in the literature and the perception of their functional similarities. QoSOS allows the reuse of common functions and the definition of an internal organization that is equivalent in different systems. In order to demonstrate how QoSOS can be applied in a real QoS provisioning scenario, the paper describes the modeling and implementation of an adaptable Intserv support, focusing on the management of the output queues of the Linux operating system. The architecture instantiation is based on few modifications introduced into the standard Linux kernel, that adds some desirable features such as runtime service adaptation.

Keywords: QoS, network operating systems, resource reservation, adaptability, frameworks.

Resumo - A crescente demanda por aplicações multimídia distribuídas torna evidente a necessidade da provisão de qualidade de serviço (QoS) de maneira fim-a-fim. Particularmente, sistemas operacionais, sejam eles localizados nas estações finais, comutadores ou roteadores, devem garantir que os recursos sob seu controle sejam gerenciados adequadamente de forma a preencher as necessidades das aplicações. Este trabalho propõe uma arquitetura para provisão de QoS em sistemas operacionais de rede (QoSOS), focando principalmente o subsistema de enfileiramento de pacotes. A construção de tal arquitetura foi originada a partir de análises das soluções atualmente encontradas na literatura e da percepção de suas semelhanças funcionais. QoSOS permite o reuso de funções comuns e a definição de uma organização interna equivalente em diferentes sistemas. A fim de demonstrar como a arquitetura QoSOS pode ser aplicada em um cenário real de provisão de QoS, o artigo descreve a modelagem e implementação de um suporte adaptável ao modelo Intserv, focando no gerenciamento das filas de saída de pacotes do sistema operacional Linux. A instanciação da arquitetura é baseada em algumas modificações introduzidas no kernel padrão do Linux, adicionando algumas características desejáveis como a adaptação de serviços em tempo de execução.

Palavras-chave: QoS, sistemas operacionais de rede, reserva de recursos, adaptabilidade, frameworks.

Authors are with the Department of Informatics of the Pontifical Catholic University of Rio de Janeiro, Rio de Janeiro, RJ, Brazil. E-mails: {moreno, csalles, atagomes, colcher, lfgs}@inf.puc-rio.br.

1. INTRODUCTION

In order to provide end-to-end QoS required by distributed multimedia applications, resource management on the whole operation environment is needed. Indeed, QoS provisioning requires the implementation of several tasks both in the end-systems and in the communication provider, including its switches and routers. In the end-systems, resources controlled by the operating system, like CPU, memory and communication buffers, must be adequately managed in order to ensure that the coexistence of various applications will not cause individual QoS violations. Within the service provider, each switch or router operating system must provide the same functionalities, besides the management of the communication channels at their many input/output ports.

QoS provisioning has become even harder since new requirements, imposed by new application types and new codification techniques, have emerged. In fact, the rapid and inexpensive deployment of services with new quality-of-service (QoS) requirements has become essential to telecommunications operators. As mentioned in [1], services should be *adapted* to new QoS demands through smooth changes in their communication and processing infrastructure.

The specification of a new service can involve the choice of scheduling, admission and classification algorithms, as well as other configuration parameters such as the tasks that will be part of the communication protocol stack or the description of the system initial state for the QoS provisioning (e.g. initial partitioning of resources for each application class). For this sake, diverse high-level adaptability abstractions have been proposed in the literature [2] (e.g. reflection, open signaling, active networks etc). These abstractions usually rely on switches and end systems that can be explicitly programmed during network operation, demanding, therefore, an operating system with sufficient flexibility. Nonetheless, the variety of available network operating systems (NOSs) hampers the deployment of such high-level adaptability abstractions.

One of the key issues of this paper is to discuss and propose an adequate support for QoS provisioning and service adaptability that can be built in a general purpose NOS. With this goal, we present a generic architecture for QoS provisioning on network operating systems, named QoSOS. The development of such generic architecture came after an analysis of solutions currently found in the literature and the perception of their functional similarities.

The QoSOS definition and specification came from the specialization and extension of an earlier work [3] that defined a set of frameworks for QoS provisioning in generic processing and communication environments. Like its predecessor, QoSOS allows the reuse of common functions and the definition of an internal organization that is equivalent in different subsystems. As a consequence, resource orchestration mechanisms for the whole system, including end-systems and the communication provider, are facilitated. [4] refers to frameworks as semi-finished architectures that capture design decisions common to a particular domain. Usually, various parts of a framework cannot be previewed, so they should be left incomplete or "subject to variations". These parts – the so-called hot spots – allow the instantiation of frameworks to specific needs. In this sense, QoSOS can be thought as a framework, or better, a set of frameworks, as will be presented.

For its most part, the paper shows how some of the framework hot spots described in [3] can be specialized in order to accommodate the QoS provisioning techniques in network operating systems. Other hot spots are still left open in order to allow the runtime configuration of specific mechanisms like resource scheduling and admission control. The paper also extends the basic generic frameworks, incorporating a model for hot-spot automatic adaptations into QoSOS. The adaptation model covers important issues, as adaptation security, system integrity and consistence maintenance.

Although the application of frameworks to model adaptations is not innovative [5, 6], our approach introduces a quite different viewpoint. Our frameworks are based on a conceptual model – the Service Composition Model (SCM) [7] – that provides generic adaptability abstractions. SCM features a nested organization of basic service elements bringing out the definition of recurrent frameworks. This paper shows how this characteristic of SCM was used to model adaptable QoS mechanisms for NOSs in a rather generic way.

With regard to QoS provisioning, we have already shown [3] that the same basic QoS mechanisms are recurrent on several parts of a system, with minor differences related to subsystem idiosyncrasies. By modeling all these mechanisms with the same frameworks and making explicit annotations to identify and specify hot spots, we could homogenize their adaptation interfaces, thus providing designers with a simple end-to-end QoS orchestration model. In this paper, we will show that our orchestration model also takes into account the relationship between processing and communication resources during QoS provisioning within NOSs.

A framework can be defined using a programming language, a specification modeling language, such as UML [8], a formal language, or any combination of these languages. The QoS provisioning frameworks for generic processing and communication environments were first described using UML. In a recent work we made use of *architecture description languages* (ADLs) to describe the resource orchestration *meta-services*[1] of those frameworks, which include *QoS negotiation* and *QoS tuning*, which are discussed in this paper. ADLs allow the performing of formal verification tests that proof many system properties, besides the definition of precise and ambiguity-free semantics for QoS provisioning. However, both the use of UML and ADLs on the specification of QoS provisioning architectures may results in complex specifications. In order to simplify the understanding of the architectural description by means of QoS provisioning-related notations, we have recently proposed a *domain-specific language* (DSL) [9] named LindaQoS [10]. Section 3.3 shows how LindaQoS represents the QoS negotiation meta-service in QoSOS architecture. Also, in order to present the instantiation of the frameworks for the Linux network subsystem in more detail, the paper will also make use of UML in Section 4.

The Linux operating system was chosen to illustrate a QoSOS use case. Although characterized by the common drawbacks of general-purpose operating systems regarding QoS, Linux can be modified to give support to adaptable QoS provisioning mechanisms. Moreover, the Linux system was chosen not only because its source availability but also due to the high interest of its use (already manifested by the community) for the development of low cost routers[2].

The focus of our prototype relies on the Linnx traffic control subsystem, which provides basic functionalities to rearrange packets and flow policing in the output network queues. However, it is important to emphasize that the QoSOS instantiation for the management of communication buffers is just a part of a complete solution for the problem of QoS in NOS, as will be seen in Section 2.

The paper is organized as follows. Section 2 discusses QoS provisioning in network operating systems, showing the relationship between processing and communication functions. Section 3 describes the frameworks that make up QoSOS. The framework instantiations for the Linux prototypes are presented in Section 4. Section 5 presents some related work, while Section 6 is reserved for conclusions and future work.

2. QoS ISSUES IN NETWORK OPERATING SYSTEMS

Services with end-to-end QoS support require the management of the underlying resources so that the communication and processing user requirements can be guaranteed during all service operation. Therefore, each involved subsystem (computer networks, operating systems and their subsystems) will have a portion of responsibility on the requested QoS, and must have its own mechanisms capable of redistributing this portion among their managed shared resources. This recursive process for QoS responsibility distribution is named QoS orchestration.

QoS orchestration is supposed to occur even inside the OS, among its processing and communication mechanisms. We can relate the processing and communication OS subsystems to two different kinds of flows that OS must deal with:

[1]Meta services act upon the main service and its elements, allowing adaptations. Examples of common meta services include the signaling mechanisms and the routing protocols.

[2]For example, the Linux Router Project, available at <http://www.linuxrouter.org/>

- *Data flow*: sequence composed of data packets transmitted among processes or threads. This transmission can occur among different machines, through network links, or inside the same machine, through buffers or function call parameters (for example, in the communication between entities of adjacent protocol stack levels);
- *Instruction flow*: sequence composed of commands interpreted by microprocessors. They can define an executing application or a routine for some system control, such as a protocol entity.[3]

These two flows are closely related regarding QoS provisioning. In a service operation environment with QoS support, it is necessary to guarantee that the data flow will receive the required QoS all the way from its source to the destination. This includes the path from the application to the network link interface within the same machine, in the case of an end system. In the case of a router or a switch, the edges of the path are the input link interface and the corresponding output link interface.

In several stages along these paths, data flows and processing demands are strongly related. Typically, instruction flows should be processed in order to comply with the QoS requirements of the corresponding data flows. In Figure 1 we have adopted an extended queuing network modeling notation [11] to illustrate this relationship inside an OS. According to the model, a running process waits to be scheduled until it is granted a quantum of processing time. Provided that the process includes a protocol entity and that all the protocol-related instructions have already been executed by the time its quantum finishes, some data from this process will have been conveyed to other protocol entities. This is represented by the token creation that liberates the data packet communication. This conveying will be either through communication buffers (in the case of a vertical communication between adjacent layers) or through a network link. The data transmission time (t_s) between two protocol entities will be either the time for buffer copying and/or pointer updating or the link propagation time.

In Figure 1, we show each protocol entity or application running in a separate process that is either dedicated to a specific data flow or shared by all of them. Nevertheless, we can abstract that some processes comprise the execution of as many protocol entities and applications as desired. Within such processes, there is no need for buffer copying among the protocol entities or applications; usually, there is at most some kind of pointer updating. For example, in general-purpose OSs, such as Linux, several protocol entities run together in the kernel address space as a single process. In this case, there is buffer copying only between the application and the kernel and then between the kernel and the network link interface.

In order to provide QoS, the OS must divide (orchestrate) the QoS provisioning responsibility between its processing and communication subsystems. As usual, the QoS orchestration may also recur within the communication OS subsystem. In fact, many orchestration scenarios are possible, according to the internal processing architecture of the OS communication subsystem. The frameworks for QoS provisioning must take all these scenarios into account.

[3]Although not very usual yet, instruction flows can also be transmitted between machines.

2.1 SERVICE ADAPTATION

As previously mentioned, services should be *adapted* to new QoS demands through smooth changes in their communication and processing infrastructure. Open-source operating systems allow some adaptations either through modifications of the kernel code, or by the simple reconfiguration of some components. Of conrse, another way of reconfiguration is the kernel update software supplied by a vendor. Generally, these actions are followed by the recompilation of the kernel or, at least, by the system restart. This kind of *design time flexibility* may be satisfactory when the demand for new services is low. *Operation time adaptability*, on the other hand, empowers the system with a greater reconfiguration capability since the kernel becomes highly dynamic (this feature is very desirable in environments with QoS support). In *microkernel* architectures, for instance, the majority of the operating system functionalities can be implemented as servers that are available for operation-time adaptations [12]. Unfortunately, microkernel architectures are not very popular when considering their adoption on end-stations.

Most operating systems presents some obstacles for service adaptabilities and QoS provisioning. This comes mainly both due to the weakness of the popular time-sharing scheduling mechanisms and the usual monolithic kernel structure. Additionally, there are few mechanisms for resource reservation in the kernel and for introducing runtime adaptations. Moreover, packet transmission queues are usually shared between all application flows, leaving no room for the classification or prioritization of packets. Finally, process schedulers assign priorities in order to privilege the execution of some processes in detriment to others. QoSOS defines several functionalities that overcome these barriers.

3. ARCHITECTURE DESCRIPTION

Figure 2 illustrates how the *generic frameworks for QoS provisioning* [3] are specialized into the QoSOS frameworks through the fulfillment of some environment-specific hot spots. This section presents not only the resulting QoSOS but also shows how its resulting frameworks can be settled for different orchestration scenarios and adapted to new QoS demands through the fulfilling of other hot spots. We also provide an adaptation mechanism that helps designers in setting up new resource-sharing policies at run time. Thus, QoSOS defines meta-services for OSs that comprises not only the configuration of the resource-sharing mechanisms by the QoS negotiation mechanisms, but also their adaptation to new QoS requirements.

In what follows, the QoSOS frameworks will be presented in a brief textual format. A complete description can be found in [13].

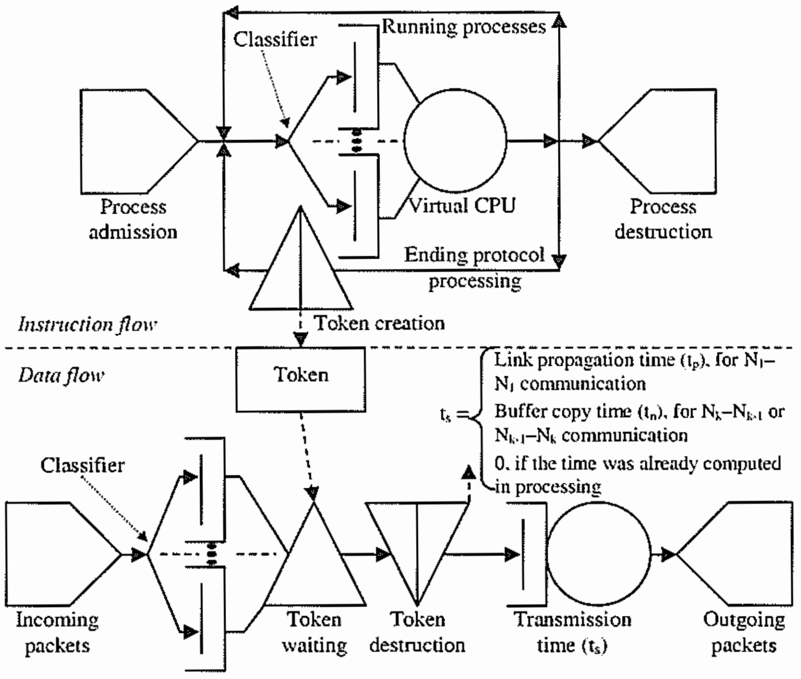
Classifier
Running processes
Process admission
Virtual CPU
Process destruction
Ending protocol processing
Instruction flow
Token creation
Data flow
Token
$t_s =$
Link propagation time (t_p), for N_l–N_l communication
Buffer copy time (t_n), for N_k–N_{k-1} or N_{k-1}–N_k communication
0, if the time was already computed in processing
Classifier
Incoming packets
Token waiting
Token destruction
Transmission time (t_s)
Outgoing packets

Figure 1. Relationship between data and instruction flows within a network operating system.

3.1 SERVICE PARAMETERIZATION FRAMEWORK

The Service Parameterization framework models the structure that defines the behavior of user flows (load characterization parameters), the QoS requested (QoS characterization parameters), and information about the internal state of each OS subsystem (like provider performance, availability, etc.).

When a service request is issued, the requesting user must provide the load and QoS characterization parameters of the highest abstraction level of the provided service. Different abstraction levels usually call for different kinds of descriptions and parameters. User parameters usually must be translated into lower abstraction level characterization parameters, that may be translated into even lower level parameters and so on. For example, a user load specification of a "video with TV quality", could be translated into a lower specification of 30 frames per second with SIF (standard interchange format) resolution, which could be translated into an even lower abstraction, like one million instructions per second for a CPU and a load of 242 Mbps for a communication channel. The *QoS negotiation mechanisms*, explained later, are responsible for parameter mappings from a higher abstraction level to lower ones, until those that describe resource behaviors.

Whenever a new flow admission request is performed or when the subsystem is a target of some kind of service adaptation, each subsystem's internal state parameters must change accordingly. The service provider must take care of the maintenance of these parameters.

The service parameterization framework allows the creation of abstract parameters that can be hierarchically specialized according to particular needs. Parameters can also be grouped in sets called Service Categories. The association of policies to *service categories* simplifies QoS provisioning mechanisms.

3.2 RESOURCE SHARING FRAMEWORKS

In order to facilitate the application of several scheduling and admission control algorithms on the same resource and thus offer a wide and flexible set of services in a single system, resources are arranged in a structure called *virtual resource tree*. Figure 3 illustrates a virtual resource tree example. The Resource Sharing frameworks allow the creation and management of virtual resources trees.

A virtual resource tree abstractly denotes the hierarchical usage division for one or more resources. *Virtual resources* represent usage portions of either a real resource, or another virtual resource, or even a set of resources. The root node of a tree is called *root resource scheduler*, where the root resource may correspond to: i) a virtual or real resource (e.g. bandwidth, CPU, memory, etc); ii) a set of real or virtual resources that can be viewed as a single resource; iii) a set of real or virtual resources that must be viewed as isolated resources. In any case, it is a root resource scheduler's responsibility to distribute usage portions of the root resource among its child nodes. Each node, by its turn, distributes its own resource usage portion among its child nodes, and so on, up to the leaf nodes, which are named *final virtual resources*. The intermediary nodes of the tree are called *virtual resource schedulers*.

Each node in the tree, except the leaves, is associated with a service category (with a set of parameters) and some corresponding QoS provisioning policies. The policies include strategies for scheduling, admission and user/network parameter control, and a virtual resource factory component. In the Resource Sharing frameworks, the *virtual resource factory* allows the inclusion of a virtual resource in the list of responsibilities of the scheduler and the configuration of the classifying and policing mechanisms. Classifying mechanisms are responsible for forwarding a data or instruction flow to the

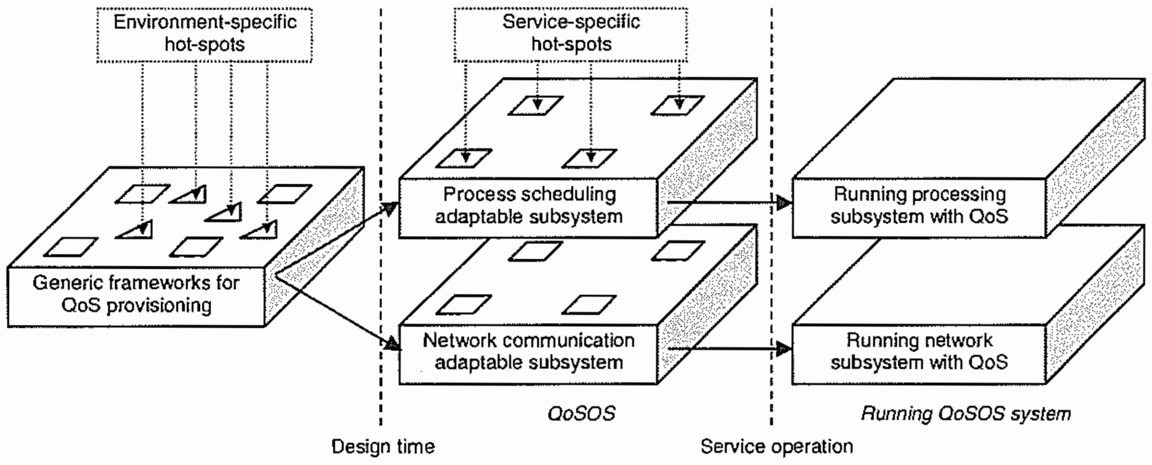
Environment-specific hot-spots
Service-specific hot-spots
Generic frameworks for QoS provisioning
Process scheduling adaptable subsystem
Network communication adaptable subsystem
Running processing subsystem with QoS
Running network subsystem with QoS
QoSOS
Running QoSOS system
Design time
Service operation

Figure 2. Hot spot types in the specialization of the generic frameworks.

appropriate child resource. Policing mechanisms must verify if user flows are in accordance with the characterized traffic and, based on the user/network parameter control strategies, take some actions to limit them (e.g. traffic shaping, packet discarding, retransmissions etc).

Flow admission is done based on information about resource usage of the scheduler associated with the requested service category. If the admission is possible, the virtual resource may be created through the virtual resource factories. When the root resource is a single resource or a set of resources viewed as a single one, the admission of new user flows is named primitive, because there is no need for negotiation among other mechanisms. In other words, in this case, admittance tests can be done directly over the virtual resource tree. Otherwise, when the set of resources cannot be viewed as a single one, the flow admission is recursively delegated to virtual resource trees of the lower abstraction levels. The process stops when a virtual resource with primitive admission is achieved. This point will be clarified in the next section.

3.3 RESOURCE ORCHESTRATION FRAMEWORKS

Orchestration is the process by which a provider, in this case the operating system, divides the QoS provisioning responsibility among its virtual resources. As the set of resources of an operating system cannot be viewed as a single one, the flow admission is recursively delegated to virtual resource trees of the lower abstraction levels, as afore mentioned. In QoSOS, resource orchestration is done by the specialization of two different frameworks: the *QoS Negotiation framework* and the *QoS Tuning framework*.

The QoS Negotiation framework runs before the service starts, during the service establishment phase. The QoS Tuning framework works during the service provisioning, in the service maintenance phase.

In the QoS Negotiation framework, upon receiving a new service request, the *admission controller* verifies the feasibility of its admittance, taking into account the current resource utilization and the proposed load associated with the new request. Then, as mentioned in Section 3.2, the admission controller starts the *OS negotiation agent*, which must identify all possible resources that would be involved in the service provisioning. The negotiation agent then establishes portions of the QoS responsibility for each identified resource.

After assigning portions of responsibilities to each resource, the *mapping mechanisms* are launched to translate the requested service category (and its associated parameters) to service categories (and parameters) directly related to the operation capacity of each assigned resource. Then, the admission control mechanisms linked to each resource are called. The admission process repeats recurrently in each resource until the primitive admission controllers are reached, when the test of admittance is directly executed over the resource (of course, primitive admission controller has no negotiation agent). In any abstraction level, if all involved admission controllers return an affirmative answer to the negotiation agent, it passes this answer to the higher-level admission controller. If all involved admission controllers return an affirmative answer, virtual resource factories are executed, creating new virtual resources, and the service request is accepted. In any other case, the request is immediately denied or a new negotiation process is started, with a new redistribution of portions of QoS responsibilities or with more relaxed QoS parameters.

During the service operation, however, some system adjustments can be necessary in order to honor the QoS specifications of admitted flows. The QoS Tuning framework works during this phase. The monitoring of resources drives the identification of operational malfunctions, which might be either user faults (traffic flows violations) or system faults (e.g.: resource hardware errors or inaccurate computations for reservation). Monitors must issue alerts to the tuning mechanism when disturbs are detected. The tuning actions may vary from small parameter adjustments in some schedulers to the request of a complete QoS renegotiation in a manner similar to the establishment phase.

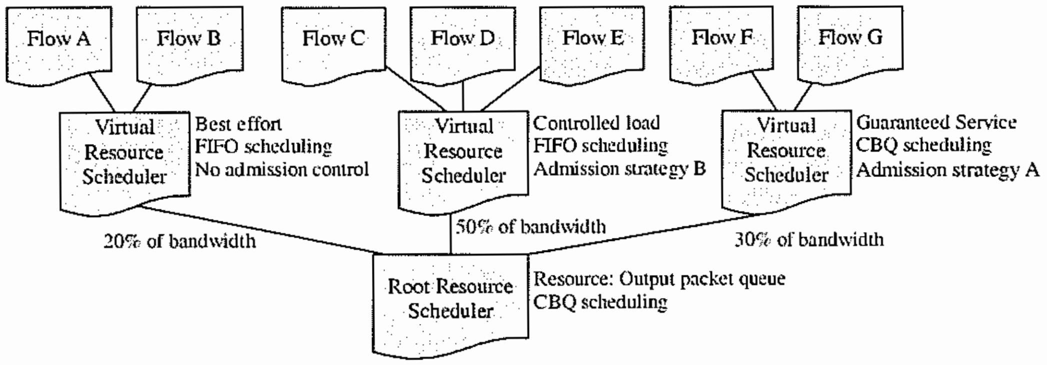

Figure 3. Virtual resource tree example, for a communication channel in the OS network subsystem.

3.3.1 MODELING THE RESOURCE ORCHESTRATION FRAMEWORKS WITH LINDAQOS

The specification of a system or a family of systems that follows the orchestration frameworks can become very hard and time-consuming as the number of resources and the nesting of involved abstraction levels grow. In order to smooth this task, we have developed a Domain-Specific Language (DSL), named LindaQoS [10]. The language is small, simple, concise and easy to learn.

LindaQoS specifications can be automatically translated into a software architecture description or to a programming language, using two different developed compilers. At the architectural level, styles are used to describe the Resource Orchestration frameworks, currently using Wright ADL [14]. This architectural description can be used as a formal tool to verify properties (using Wright analysis tools). At the implementation level, the compiler can be used to translate LindaQoS specifications to a specialization of the generic frameworks for QoS provisioning [3] (currently in JAVA). This specialization provides the basis for the Resource Orchestration frameworks instantiation.

A complete LindaQoS description of the QoS Negotiation framework is composed of some System sections and just one Hierarchy section. A System section describes the components instantiation of a particular level of abstraction (Instances subsection) and the bindings between components of that level (Attachments subsection). The hierarchy section, by its turn, describes all relations among the different systems. There are three basic types of components in LindaQoS, used in the Instances subsection: the *AdmCtrl* (admission controller); the *QoSNeg* (QoS negotiator) and the *QoSMap* (QoS mapper).

The compiler verifies when an attachment is invalid, exhibiting error messages. Each hierarchy clause must attach two components of different systems (a *QoSNeg* and an *AdmCtrl*). These attachments clauses must associate each *AdmCtrl* component to a unique *QoSNeg* component. Moreover these clauses must also associate *QoSNeg* components to one or more *QoSMap* components.

Figure 4 shows the LindaQoS specification for the QoSOS Negotiation framework. It describes the *QoSOS*, *CPU* and *LinkQueue* systems. The first one receives service requests from the user. The *AdmissionController* redirects these requests to the *Negotiator*, who centralizes the negotiation mechanism and distributes portions of responsibility among the *CPU* and *LinkQueue* subsystems. Negotiator uses the *CPUMapper* to translate higher-level QoS parameters to *CPU* subsystem related parameters. In a similar way, the *QueueMapper* translates those parameters to the *LinkQueue* subsystem related parameters. Both the *AdmissionController* in the *CPU* subsystem and the *AdmissionController* in the *LinkQueue* subsystem are primitive admission controllers. The Hierarchy section binds the *Negotiator* component to the *AdmissionController* component of the *CPU* subsystem and to the *AdmissionController* component of the *Queue* subsystem.

A complete LindaQoS specification of the QoS Tuning framework can be found in [10].

```
System QoSOS
  Instances
    AdmissionController : AdmCtrl();
    Negotiator : QoSNeg();
    CPUMapper : QoSMap();
    QueueMapper : QoSMap();
  Attachments
    AdmissionConotroller ⇆ Negotiator;
    Negotiator ⇆ CPUMapper;
    Negotiator ⇆ QueueMapper;
EndSystem

System CPU
  Instances
    AdmissionController : AdmCtrl();
  Attachments
EndSystem

System LinkQueue
  Instances
    AdmissionController : AdmCtrl();
  Attachments
EndSystem

Hierarchy QoSOSNegFramework
  QoSOS.Negotiator ⇆ CPU.AdmissionController;
  QoSOS.Negotiator ⇆ LinkQueue.AdmissionController;
Hierarchy
```

Figure 4. QoSOS negotiation framework in LindaQoS.

3.4 SERVICE ADAPTATION FRAMEWORKS

The generic frameworks for QoS provisioning provide service-specific hot spots that can be used by designers to model adaptable systems. However, these hot spots may present implicit-dependent relationships that can make the system consistency difficult to be maintained face to adaptation actions. In this context, the implementation of "meta-mechanisms", which automate the system adaptation to new services or new QoS provisioning policies, is highly desirable. In doing so, consistence and security are responsibilities of these meta-mechanisms. The *service adaptation framework* was built to fulfill this lack, in an operating systems specific approach.

Adaptation actions are requested by system administrators or external meta-mechanisms (e.g.: an open signaling protocol or management mechanism) and should be controlled by an *adaptation manager*. This manager is responsible for handling incoming requests, verifying adaptations feasibility and consistency, and inserting or replacing old components by new ones (which will perform the new functions or adapt the old ones). The new components are supplied by an adapting agent that is a member of the meta-service environment. In order to create a complete new service, all components that represent QoS provisioning policies must be supplied to the manager, together with the precise location of the new service category in the service category hierarchy.

Some of the referred tests fall into the security verification that is made at the component insertion delegated by the adaptation manager to a specific agent, named security manager. The security manager must analyze the following basic aspects in a new component:

- Authentication: the component supplier must be reliable;
- Context restriction: the instructions described within the component must be restricted to the context in which it will be applied;
- Isolation: the component actions, if logically wrong, cannot interfere in the quality offered by other service categories or in the operation of other subsystems.

If these verifications are successful, the adaptation manager submits the component implementation to the correspondent *adaptation port*. Adaptation ports are structures present in the operating system that make component implementations available to external clients. An example of adaptation port is the Linux kernel module subsystem (although it was not built for this purpose – see section 4). Finally, the adaptation manager updates all structures that maintain any reference to the original components, like a scheduler does to a component for resource creation or a scheduling strategy.

When removing a component, besides the security tests, *consistence verification* must be executed. The consistence test for removal will verify if the elimination of a component does not cause other component failure.

Note that the adaptation mechanisms can be applied not only on the QoS provisioning infrastructure, but also on other parts of the operating system, like the network subsystem (protocol stack), driver management, file system and others. Obviously, the kernel must support this functionality, defining adaptation ports in these subsystems. This paper presents in Section 4 adaptation mechanisms for introducing scheduling and admission strategies to the communication buffers in the Linux system.

4. QoS PROVISIONING IN THE LINUX NETWORK SUBSYSTEM

In order to demonstrate how some QoSOS mechanisms can be applied in a real QoS provisioning scenario (see Figure 2), this section describes the modeling and implementation of an adaptable service support to the Linux network output queues (link layer queues). The goal is to provide Linux with Intserv [15] service categories. In order to be able to introduce QoS support and also to allow runtime adaptations for admission and scheduling strategies, some modifications to the system kernel were needed. Details about kernel code changes can be found in [13]. From now on, we will name the framework instantiation as QoSOS Linux.

Intserv was chosen in order to complement our prior instantiation for QoS provisioning on the Internet [16]. This previous work also demonstrated how the Diffserv [17] architecture should be modeled using the generic frameworks for QoS. In the operating system context, the Diffserv service categories must have adequate treatment with different QoS policies from those of Intserv model. For this sake, the design-time QoSOS flexibility and the QoSOSLinux runtime adaptability allow the modeling and deploying of various QoS architectures, which can have totally different reservation needs.

As mentioned in Section 2, one of the main QoS characteristics in network operating systems is the interdependency between data and instruction flows. This fact implies in an OS architecture that implements QoS support in both processing and communication subsystems. QoSOS Linux is a work in progress, a project that aims the inclusion of QoS capabilities in the Linux operating system for a better support to distributed multimedia applications. In the present paper we will discuss only the simple QoS resource orchestration that just embraces the communication link queue management.

This section is organized as follows. Section 4.1 presents a brief discussion on the Linux packet queuing subsystem, called LinuxTC (our instantiation target), showing its mechanisms for packet scheduling management on network output interfaces. Section 4.2 describes two application program interfaces (APIs) built for QoS service request and for adaptation request. The sequence of tasks performed after API method invocations are also presented, in a brief textual format. Section 4.3 uses UML as a tool to model the instantiation of the frameworks that were discussed on Sections 3.1, 3.2, 3.3 and 3.4.

4.1 THE LINUX TRAFFIC CONTROL (LINUXTC)

Recent versions of the Linux kernel offer a large set of functions for network traffic control [18], handling mechanisms that support packet scheduling for the Intserv and Diffserv architectures. The following conceptual components are defined: queuing disciplines, classes and classification/policing filters. Each network interface has an associated queuing discipline, which governs the queuing policies for the device. A queuing discipline can be as simple as a single FIFO, or it can use complex structures such as filters that classify packets into different classes for further differentiated processing (Figure 5).

With LinuxTC, it is possible to configure the way packets will be queued and when they must be discarded (in a congestion situation, for example). LinuxTC allows users to define the order for packet dispatching through the assignment of priorities to the flows and to insert delay between packets of a flow (which can be used to limit the output data rate). With this toolbox, many types of policies for packet scheduling may be configured. In particular, a hierarchical structure can be used, in which each class of a queuing discipline can be linked to another discipline that will be responsible for the scheduling of packets belonging to that class. The virtual resource tree concept, introduced in section 3.2 is sufficiently generic to embrace this Linux traffic control in our proposed QoSOS instantiation.

LinuxTC provides a rich but non-dynamic method for the configuration of its components through a command line program called "tc". It is a non-dynamic method in the sense that there is no native programming interface that allows applications or protocols to configure the performance characteristics of network communications. Nevertheless, the "IBM developerWorks[4]" contributors have recently created an international open source project called TCAPI [19] in order to fulfill this lacuna. TCAPI version 1.2 was initially used to configure LinuxTC in this paper proposed scenario. However, it was noted that several LinuxTC functionalities were not supported and also some bugs were found. Thanks to the easy access to the TCAPI code, we could make various modifications until an acceptable version with the required and new functions was achieved. We have submitted these modifications to the TCAPI project coordinator and now, after approval, they became a part of the original software [5].

4.2 THE QoSOS LINUX API'S

Two distinct APIs for the QoSOS Linux environment were developed. One that allows users to make their requests to specific services with a specific QoS, and other that allows the dynamic configuration or adaptation of QoS mechanisms. The first API is the one to be used by service negotiation agents (whether they are located in routers or end-systems) or end-users (from now on, both will be referenced as users). Functions like admission control and virtual resource creation are included in the tasks provided by the API. The second API provides methods for service adaptations to be used by system administrators or any management mechanism (both will be called managers)[6]. Operations like component insertion, removal or replacement are included specifically for the admission and scheduling strategies related with the QoSOS Linux service categories.

[4]DeveloperWorks, the IBM's resource for developers, hosts a variety of open source projects. See <http://www.ibm.com/developerworks/oss/>.

[5]TCAPI is available at <http://www.ibm.com/developerworks/projects/tcapi/>.

[6]Users and managers are roles that can be assumed by the same agents.

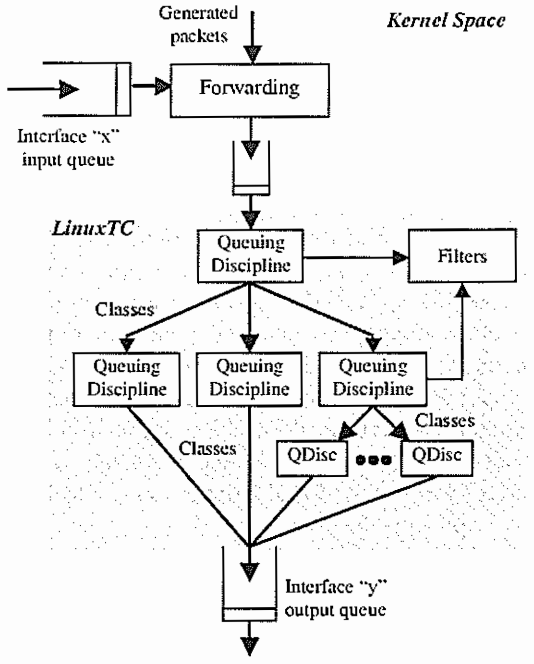

Figure 5. LinuxTC subsystem overview.

Figure 6 gives a general view of QoSOS Linux, illustrating which mechanisms will be triggered when requests are made to each of the APIs, and how these mechanisms interact to adapt the LinuxTC structure. As we can see, the *QoSOS Admission Controller* and the *QoSOS Adaptation Manager* are the front-end mechanisms that provide the service and adaptation request APIs, respectively.

4.2.1 QoSOS LINUX SERVICE REQUEST PROCESS

The *services request API* is the interface between users and the negotiation mechanisms that provides methods to accomplish the following tasks:

- Service admission (or request), which starts the QoS negotiation process, including the admission tests;
- Service commitment (or confirmation), which starts the effective resource reservation process, according to the portions of responsibility defined in the negotiation process;
- Service release, which starts the resource releasing process.

Figure 6 illustrates (in dashed lines) the interaction between the service request mechanisms. The QoS negotiation process begins when the QoSOS Admission Controller receives a service request that specifies the desired service category, the associated parameters and information about the data flow (source and target addresses, ports, protocol etc). The QoSOS Admission Controller delegates to the QoSOS Negotiator the responsibility of identifying the virtual resources that will participate in the service provisioning. Since we are only dealing with the QoS provision in output packet queues, the QoS negotiator just submits the same information to the *Link Queue Admission Controller*[7]. This component is able to verify if the requested QoS level (expressed by the informed service parameters) can be reserved in the LinuxTC resource tree. If the corresponding admission strategy concludes that there are sufficient available resources, the Link Queue Admission Controller performs a "fake reservation" (or pre-reservation) of the required resources and returns a positive answer to the negotiator. While a pre-reservation is valid, resources are not definitively reserved although they will be considered as so to other admission requests performed during this period. If the admission strategy concludes that the available resources are not sufficient, the pre-reservation is not performed and a negative answer is returned to the negotiator. In both cases, the negotiator will redirect returned answer to the QoSOS Admission Controller, which adequately informs the user.

The concrete resource reservation begins when the QoSOS Admission Controller receives from the user a *service commitment request* with a reference to a previous pre-reservation. This reference is passed to the QoSOS negotiator and then to the Link Queue Admission Controller that will verify the current pre-reservation status (if it has expired or not). If the pre-reservation is still valid, the Link Queue Admission Controller invokes the *Link Queue Factory*, which configures the LinuxTC resource tree based on the pre-reservation information. At this point, the service agreement is established.

The releasing process begins when the QoSOS Admission Controller receives a *service release request* from the user with a reference to a reservation. This reference follows the same path down to the Link Queue Admission Controller, which will ask the Link Queue Factory to release the corresponding resources.

4.2.2 THE QOSOS LINUX ADAPTATION REQUEST PROCESS

The *adaptation request API* is the interface between managers and the adaptation mechanisms, providing methods to accomplish the following tasks:

- Component insertion, which is useful when creating new service categories;
- Component removal, which is useful when removing service categories;
- Component replacement, which is useful for service adaptations.

The adaptation request mechanisms interactions are illustrated by dotted lines in Figure 6. In QoSOS Linux, component insertion, removal and replacement are done directly into Linux kernel modules subsystem in a rather simple fashion. The components that can be adapted are the admission and schedulers strategies. The only security test made is the one provided by the subsystem, which requires the manager to have sufficient rights to do kernel module operations.

[7]The need for the QoS negotiator is justified by the fact that the system is already prepared to orchestrate other resources like the CPU.

When the QoSOS Adaptation Manager receives a request for a component insertion, it attaches the component into the kernel, which in turn registers the new function names as kernel symbols making all necessary reference updates (as those maintained by the resource schedulers). When removing a component, the kernel deregisters the names and nullifies existing references. Finally, component replacement can be viewed as a component removal followed by a component insertion. The implementation of security and consistency issues discussed in Section 3.4 needs further investigation, so they are left as future work.

4.3 QoSOS ARCHITECTURE INSTANTIATION

As mentioned before, the QoSOS instantiation description will use UML diagrams. The classes with adornments like "<<X>>" indicate that they represent the specialization of a particular mechanism "X", in the QoS frameworks, by the fulfillment of some environment-specific hot spots. The classes in gray represent hot spots that can be fulfilled (through class derivations) to account for new QoS requirements being offered to users. In the textual description, the abstracts classes and methods are notated in *italic*.

4.3.1 SERVICE PARAMETERIZATION FRAMEWORK

Figure 7 illustrates the instantiation of the Service Parameterization framework, discussed in Section 3.1. The guaranteed and controlled load service categories, defined by the Intserv model, are represented by the classes GuarServiceCategory e CLServiceCategory, respectively.

The Rspec parameter (reservation specification) must be associated only to objects of class GuarServiceCategory to denote QoS requirements that will be granted by the system. The TSpec parameter (traffic specification) is used by both service categories and describes the user-generated traffic characterization. R, s, r, b, p, m e M have equivalent definitions from the same name parameters of Intserv model.

There is no need for the definition of any other parameters that may be directly related to the resource being allocated, as the LinuxTC elements used in the instantiation can be characterized by the same parameters described by (or, at least, by parameters that have the same meaning of) the Rspec and Tspec structures.

4.3.2 RESOURCE SHARING FRAMEWORKS

The Resource sharing frameworks introduced in Section 3.2 comprises the Resource Scheduling and Resource Allocation frameworks. The Resource Scheduling framework represents the scheduling mechanisms of the virtual resource tree. An initial tree for the proposed scenario may be similar to that presented in Figure 3.

Figure 8 illustrates the instantiation of the Resource Scheduling framework. It just shows how the LinuxTC mechanisms can be represented with QoSOS components. The framework elements and methods follow naming convention internally used in LinuxTC, exactly as described in [18].

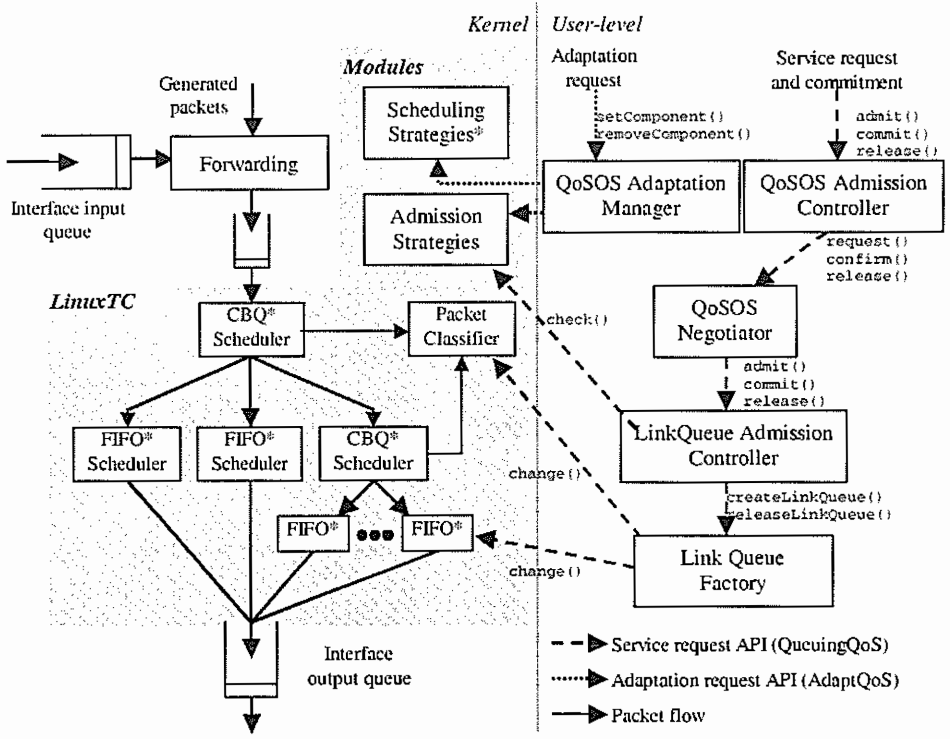

Figure 6. QoSOS Linux overview.

The class DeviceQueuingDiscipline represents the root scheduler of the network interface, or in our model, the root of the virtual resource tree. The method wakeup() starts the packet scheduling sequence, which is, in fact, a sequence of dequeue() function calls (in each involved scheduler). The other disciplines (virtual resource schedulers) of the same interface complete the virtual resource tree. These inner schedulers are modeled by the class InnerQueuingDiscipline, which also uses the dequeue() function to continue the scheduling process.

When the OS network protocol stack finishes the output processing of a packet, it signals the root scheduler associated with the output interface selected by the forwarding function. Then, this root scheduler must conduct the packet to the classifier filters, modeled by the class LinkQueueFilter, which identifies the service category that the packet belongs to. The root scheduler invokes the enqueue() method of the virtual resource scheduler that corresponds to the identified service category. The classes CBQSchedStrategy and FIFOSchedStrategy represent the scheduling strategies supplied by LinuxTC used in our Intserv scenario for guaranteed and controlled load packets, respectively.

Figure 9 shows the instantiation of the Resource Allocation framework. As LinuxTC does not provide virtual resource factories, their implementations were made in user space. Two factory components were instantiated, each one corresponding to a specific service category with different reservation requirements (classes GauranteedLQFactory and ControlledLoadLQFactory).

The method createGuarLink() of the GuaranteedLQFactory performs the resource allocation for guaranteed service category flows. This operation can be viewed as two steps: the creation of classification and policing filters; and the effective reservation of the requested bandwidth. A single LinuxTC filter may execute both packet classification (based on some header information) and flow policing (according to some traffic profile rules). Therefore, the factory component for the guaranteed service category must request the creation of a filter (using TCAPI), calling the method change() of a LinkQueueFilter object. The bandwidth reservation is done setting up a new LinuxTC class into the discipline that is handling the guaranteed service flows. This is done (also through TCAPI) by another change() function, now pertaining to the class *LinkQueueScheduler*.

The factory associated with the controlled load service category, by its turn, may just request the creation of the classifying/policing filter, since Intserv does not force the bandwidth reservation for flows belonging to this category.

4.3.3 RESOURCE ORCHESTRATION FRAMEWORKS

Figure 10 shows the instantiation of the QoS Negotiation framework (described in Section 3.3). The LindaQoS compiler, presented in Section 3, was used to generate (automatically) the QoSOS Resource Orchestration frameworks. Particularly, this compilation produced the abstract classes represented in Figure 10 written in Java.

The QoS negotiation process starts with a service request by the user through the admit() primitive. Then the operating system admission controller (class QoSOSLinuxAdmissionController) passes the supplied parameters to the QoSOS negotiator (class QoSOSLinuxNegotiator). The ne-

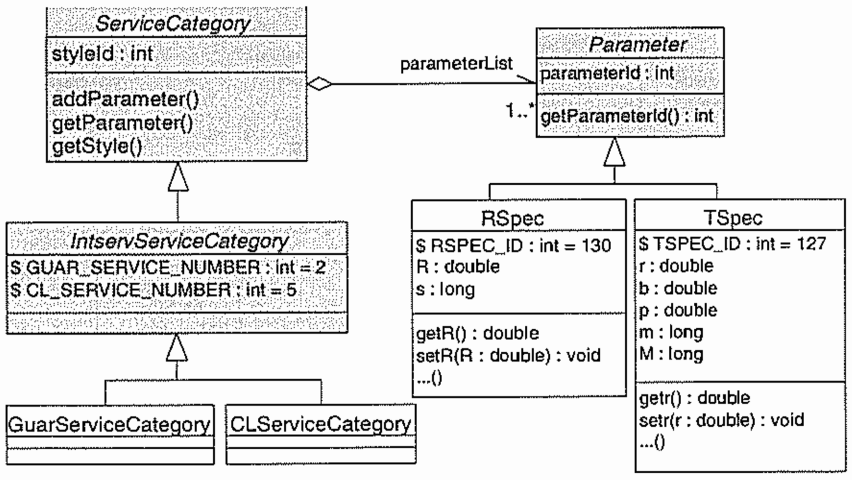

Figure 7. Instantiation of Service Parameterization framework.

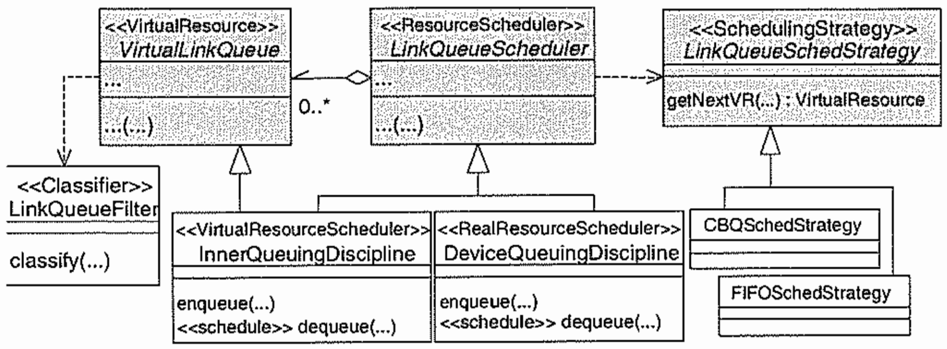

Figure 8. Instantiation of the Resource Scheduling framework.

gotiator will conclude that only the network queues will participate of the resource orchestration. Then, it invokes the admit() method of the link queue admission controller (class LinkQueueAdmissionController) with the same parameters from the request. There is no need for QoS mapping, since the virtual resources on LinuxTC can be created in terms of the Intserv QoS and traffic characterization parameters.

The link queue admission controller invokes the check() method of the correspondent admission strategy (class *LinkQueueAdmissionStrategy*), according to the requested service category. This method analyses the current performance parameters of the resources and compares them to the requested QoS. If the request is feasible, the admission controller is informed and generates a pre-reservation identifier, later used at service confirmation time. The link queue admission controller returns the identifier to the QoSOS negotiator, which then returns it to the QoSOS admission controller. If we had more than one admission controller in the service orchestration, the QoSOS negotiator should maintain a table for identifier mappings.

The initially available admission strategies correspond to the classes GuaranteedAdmStrategy and ControlledLoadAdmStrategy. To approve the admission of a guaranteed service flow, the data rate (R) must be available at the virtual resource scheduler. This strategy is called simple sum, and corresponds to the strategy A in the virtual resource tree on Figure 3. The controlled load admission strategy is the strategy B in the same figure. It consists of the following test: the sum of the r parameters of the current admitted controlled load flows with the r parameter informed in the traffic characterization of the solicitant cannot exceed the pre-allocated bandwidth for the controlled load service category. This strategy is equivalent to the simple sum, but takes in account the traffic characterization parameters.

When users want to confirm the service request, the commit() method of the QoSOS admission controller must be called, informing the pre-reservation identifier, which will be passed to the QoSOS negotiator and then to the link queue admission controller. If it is a valid identifier, the pre-reservation data are recovered and the appropriate virtual resource factory (a specialization of class *LinkQueueFactory*) is invoked, as mentioned in the resource allocation framework. This calling sequence is also illustrated in Figure 6.

4.3.4 SERVICE ADAPTATION FRAMEWORK

The Linux monolithic kernel has a subsystem for kernel modules management, including functions for module insertion, removal and probing. This subsystem was initially designed for adding device drivers. Nevertheless, several works

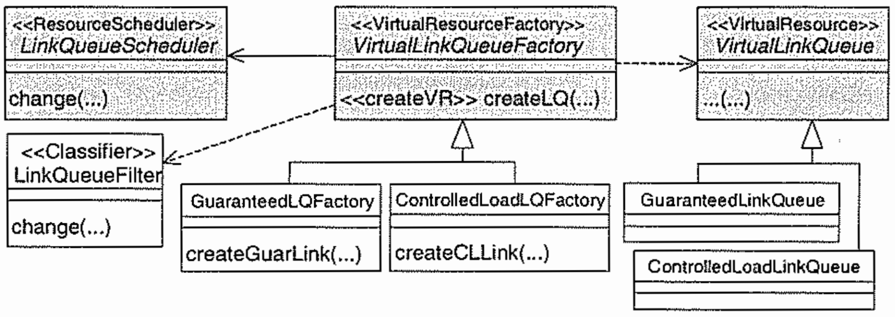

Figure 9. Instantiation of the Resource Allocation framework.

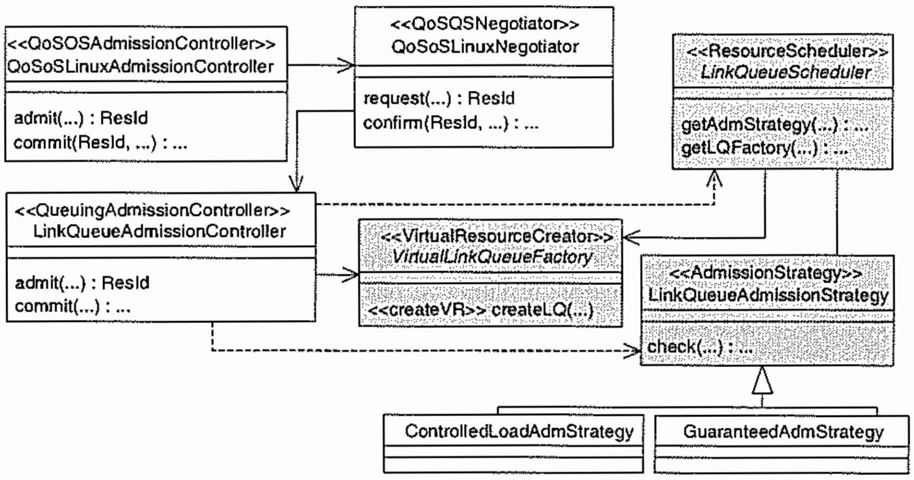

Figure 10. Instantiation of the QoS Negotiation framework.

use this functionality to provide configuration of kernel internal parts, like process scheduling [20]. Many modifications must be performed into the kernel, however, for runtime adaptability, since this was not previewed in the original Linux design philosophy. These modifications are listed in details in [13].

Figure 11 illustrates the instantiation of the Service Adaptation framework, discussed in Section 3.4. The class LinuxQoSAdaptationManager represents the adaptation manager, implemented in user-space. Its functionality is restricted to the insertion and replacement of modules that implement packet scheduling and admission strategies. The kernel modules (adaptable components) are represented by the classes AdmissionStrategyComponent, SchedulingStrategyComponent, KernelModulePort and ObjectFile. Since it becomes possible to adapt scheduling strategies, the system administrators are not limited to the LinuxTC supplied algorithms anymore.

When the system administrator (or any other authorized management mechanism) requests the insertion or replacement (setComponent()) of a component, the adaptation manager knows that the adaptation port is the kernel module subsystem. Upon receiving the component implementation (ins_mod()), the kernel module subsystem makes a simple security verification, modeled by the class LinuxAdaptationSecurityManager. This class checks (through the capable() function) if the solicitant capabilities grant module management rights.

5. RELATED WORK

Current research related to the QoS provisioning in operating systems vary from simple extensions to completely new operating system designs, both targeting the mechanisms for resource management. Extensions are focused on fixing some of the OSs drawbacks for QoS provisioning, like the unfair priority-based scheduling, the interruption-guided network subsystem (that may cause some scheduling anomalies), the impossibility to assign priorities to packets in shared buffers, scarce resource reservation mechanisms and poor adaptability support.

References [21], [22] and [23] propose new structures to CPU hierarchical partitioning in order to provide fair support to different application needs. This is mainly achieved through the utilization of different process schedulers. The proposed architectures also allow scheduler code adaptations during runtime. In our framework, the hierarchical scheduling structure is extended to handle several other operating system resources, besides the CPU. System resources are generalized through the virtual resource tree concept, discussed in Section 3.

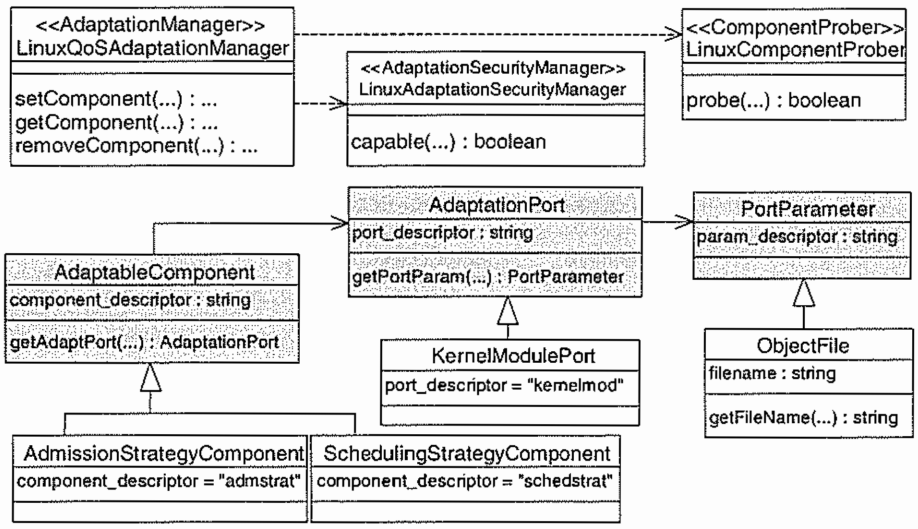
<<AdaptationManager>>
LinuxQoSAdaptationManager
setComponent(...) : ...
getComponent(...) : ...
removeComponent(...) : ...
<<AdaptationSecurityManager>>
LinuxAdaptationSecurityManager
capable(...) : boolean
<<ComponentProber>>
LinuxComponentProber
probe(...) : boolean
AdaptationPort
port_descriptor : string
getPortParam(...) : PortParameter
PortParameter
param_descriptor : string
AdaptableComponent
component_descriptor : string
getAdaptPort(...) : AdaptationPort
KernelModulePort
port_descriptor = "kernelmod"
ObjectFile
filename : string
getFileName(...) : string
AdmissionStrategyComponent
component_descriptor = "admstrat"
SchedulingStrategyComponent
component_descriptor = "schedstrat"

Figure 11. Instantiation of the Service Adaptation framework.

In the context of adaptation, the LDS (load dependent scheduler) [24] algorithm introduced a great contribution. It allows the emulation of several scheduling algorithms simply through an adequate filling of its operation parameters. Hence, the insertion of a new scheduler can be done without any further programming or any need for kernel recompilation. The drawback in the LDS algorithm is its incapacity to emulate real time scheduling algorithms based on deadlines. QoSOS can easily model the adaptation through parameter setting.

Operating systems like Sumo [12] and Nemesis [25] are microkernel-based and propose a better support for distributed multimedia applications. One of their common features is the network stack protocol definition in the user space. Each application must have its own instantiation of the stack, with dedicated buffers for its flows. On the other hand, the LRP architecture (lazy receiver processing) [26] is an extension to the BSD network standard, with the aim of purging software interruptions during the kernel stack processing. In this paper, we discussed the behavior of several communication architectures regarding to the orchestration of processing and communication resources. QoSOS design takes into account, and thus use, all these approaches.

6. CONCLUSIONS AND FUTURE WORK

This paper presents a family of frameworks for adaptable QoS provisioning in network operating systems, called QoSOS. The proposed architecture allows the definition of recurrent structures, the homogenization of the QoS mechanism representation and service interfaces, and the specification of a simple QoS orchestration among OS subsystems.

In order to illustrate how a general-purpose operating system can provide adaptable QoS mechanisms, despite all well-known deficiencies of these systems to provide this kind of service, a prototype using a Linux system was implemented based on QoSOS. The implementation effort resulted in important contribution to the TCAPI initiative [19], an open source implementation for the traffic control API in Linux systems. The extension of the Linux system to provide QoS is one of our goals in a project to build an infrastructure to support interactive TV applications.

This work opens many questions and leaves space for further investigation and research activities such as:

Inclusion of other relevant resources on the resource orchestration, like memory, memory paging, secondary memory, etc.

Implementation of QoS mechanisms in the Linux process scheduling. These mechanisms are already modeled for the next instantiation of the architecture, including the utilization of the LDS algorithm. With the CPU QoS management, a system performance evaluation will be interesting to validate our orchestration model.

Service adaptation framework refinement, including procedures to verify the security consistence issues in the component inclusion and removal.

Translation of the negotiation and tuning framework specification to Wright ADL using the LindaQoS Compiler [10]. As a consequence, consistency verification using Wright analysis tools will be able to be carried out (e.g. deadlock-free tests, style constraints).

Extension of LindaQoS to describe other QoS provisioning service, and not only the resource orchestration meta-services.

7. ACKNOWLEDGEMENTS

This work was granted by the Brazilian Telecommunications Technological Development Fund (FUNTTEL), through the contract n. 0594/02.

REFERENCES

[1] N. Kosmas and K. Turner "Requirements for service creation environments". *2nd International Workshop on Applied Formal Methods in System Design*, p. 133 - 137, 1997.

[2] A. Campbell et al, (1999) "A Survey of Programmable Networks". *ACM SIGCOMM Computer Communications Review*, vol. 29, no. 2, p. 7 - 23, 1999.

[3] A.T.A. Gomes, S. Colcher, L.F.G. Soares, L.F.G. "Modeling QoS Provision on Adaptable Communication Environments", *Proceedings of the IEEE International Communication Conference (ICC2001)*, Helsinki, Finland, 2001.

[4] W. Pree, "Design patterns for object-oriented software development". Addison Wesley, 1995.

[5] F. Costa, H. Duran-Limon, N. Parlavantzas, N.K. Saikoski, G. Blair, G. Coulson, "The role of reflective middleware in supporting the engineering of dynamic applications". *Reflection and Software Engineering*, 2000.

[6] M. Kolberg, R. Sinnott, E. Magill "Experiences modeling and using forma object-oriented telecommunication service frameworks". *Computer Networks Magazine*, no. 31, 1999.

[7] S. Colcher, "Um Meta Modelo para Aplicações e Serviços de Comunicação Adaptáveis e com Qualidade de Serviço". *D.Sc. Thesis, Pontifical Catholic University of Rio de Janeiro*, 1999.

[8] Rational Software Corporation. "Unified Modeling Language: Notation Guide", 1997.

[9] A. Deursen, P. Klint, J. Visser, "Domain-Specific Languages: An Annotated Bibliography". *ACM SIGPLAN Notices*, vol. 35, no. 6, p. 26 - 36, 2000.

[10] C.S. Soares Neto, "LindaQoS: Descrição Arquitetural da Provisão de QoS em Amhientes Genéricos de Processamento e Comunicação". *M.Sc. Thesis, Pontifical Catholic University of Rio de Janeiro*, 2003.

[11] W.A. Macnair, "Elements of Practical Performance Modelling", Prentice-Hall, 1985.

[12] G. Coulson, G. Blair, "Architectural principles and techniques for distributed multimedia application support in operating systems". *ACM Operating Systems Review*, vol. 29, no. 4, p. 17 - 24, 1995.

[13] M.F. Moreno, "Um framework para provisão de QoS em sistemas operacionais". *M.Sc. Thesis, Pontifical Catholic University of Rio de Janeiro*, 2002.

[14] R. J. Allen, "A Formal Approach to Software Architecture". *Ph.D. Thesis*, Carnegie Mellon University, 1997.

[15] R. Braden, D. Clark, S. Shenker, "Integrated services in the internet architecture: an overview". *IETF Request for Comments*, RFC1633, 1994.

[16] O. T. Mota, "Uma arquitetura adaptável para provisão de QoS na internet". *M.Sc. Thesis, Pontifical Catholic University of Rio de Janeiro*, 2001.

[17] S. Blake et al, "An architecture for differentiated services". *IETF Request for Comments*, RFC2475, 1998.

[18] W. Almesberger, "Linux network traffic control: Implementation overview". *Implementation Details*, 1999. <ftp://icaftp.epfl.ch/pub/people/almesber/pub/tcio-current.ps.gz>.

[19] D. Olshefski, "Notes on linux network QoS - TCAPI version 1.0". *Work in progress*, 2001. <ftp://www-126.ibm.com/pub/tcapi/tcapi.tar.gz>

[20] M. Barabanov, "A linux-based real-time operating system". *Master Degree dissertation, New Mexico Institute of Mining Technology*, 1997.

[21] P. Goyal, X. Guo, H. Vin, "A hierarchical CPU scheduler for multimedia operating systems" in *Proc. 2nd Symposium on Operating System Design and Implementation (OSDI'96)*, p. 107 - 122, 1996.

[22] B. Ford, S. Susarla, "CPU inheritance scheduling" in *Proc. 2nd Symposium on Operating Systems Design and Implementation (OSDI'96)*, 1996.

[23] J. Regher, "Using Hierarchical Scheduling to Support Soft Real-Time Applications in General-Purpose Operating Systems". *Ph.D. thesis, Univ. of Virginia*, 2001.

[24] M. Barria, R. Vallejos, L.F.G Soares, "LDS: A Load Dependent Scheduling Algorithm" in *Proc. IFIP ATM & IP 2000 Workshop, In Participants*, 2000.

[25] R. Black et al. "Protocol implementation in a vertically structured operating system" in *Proc. of 22nd Conference on Local Computer Networks (LCN'97)*, 1997.

[26] P. Druschel, G. Banga "Lazy receiver processing (LRP): a network subsystem architecture for server systems" in *Proc. 2nd Symposium on Operating System Design and Implementation (OSDI'96)*, p. 261 - 275, 1996.

Marcelo Ferreira Moreno is a Ph.D. candidate in computer science at the Pontifical Catholic University of Rio de Janeiro, Brazil, where he received the M.Sc. degree in computer science in 2002. He received the B.Sc. degree in computer science from the Federal University of Viçosa (UFV), Brazil, in 2000. He has been a researcher in the TeleMidia Laboratory in the department of informatics of the Pontifical Catholic University of Rio de Janeiro since 2001. He is currently a professor of the Computer Networks specialization course at the same university. His research interests include adaptable QoS provisioning in network operating systems for a better multimedia application support.

Carlos de Salles Soares Neto obtained his Bachelor degree in computer science from the Federal University of Maranhão - UFMA in 2000 and the MSc. in computer science from the Pontifical Catholic University of Rio de Janeiro - PUC-Rio - in 2003. Since 2001, he has been a researcher at TeleMidia Laboratory in the department of informatics of PUC-Rio, Brazil. He is currently a professor of the Computer Networks specialization course at the same university. His main interest is Quality of Service provisioning for Multimedia Application support.

Antônio Tadeu de Azevedo Gomes is a Ph.D. candidate in computer science at the Pontifical Catholic University of Rio de Janeiro, Brazil, where he received the M.Sc. degree in computer science in 1999. He received the B.Sc. degree in computer science from the Federal University of Rio de Janeiro (UFRJ), Brazil, in 1995. He is currently a visiting researcher in the Computer Science Department of Lancaster University. His research interests are in QoS modeling for communication and processing environments.

Sérgio Colcher received the B.Sc. degree in computer engineering from the Pontifical University of Rio de Janeiro, Brazil, in 1991, where he also received the M.Sc. and Ph.D. degrees in computer science in 1993 and 1999, respectively. He is currently a professor at the department of informatics of the Pontifical Catholic University of Rio de Janeiro. His research interests include high-speed networks and traffic engineering.

Luiz Fernando Gomes Soares received the B.Sc. degree in electrical engineering from the Pontifical Catholic university of Rio de Janeiro, PUC-Rio, Brazil. He received, from the same university, the M.Sc. degree in electrical engineering in 1979 and the Ph.D. Degree in computer science in 1983. He has worked as a researcher at COBRA (Brazilian Computers S/A) and at IBM Scientific Center in Rio de Janeiro. He is currently a professor at the department of informatics of the Pontifical Catholic University of Rio de Janeiro, PUC-Rio, Brazil. His research interests are in multimedia and hypermedia systems and in high-speed networks, in which areas he has several books and papers published.