The Utility Model Applied to Layer-coded Sources

by

Lei Chen
B.Sc., University of Victoria, 1996

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

We accept this thesis as conforming
to the required standard

[redacted]
Dr. Eric G. Manning, Supervisor (Department of Computer Science)

[redacted]
Dr. Gholamali C. Shoja, Department Member (Department of Computer Science)

[redacted]
Dr. Kin F. Li, Outside Member (Department of Electrical and Computer Engineering)

[redacted]
Dr. Gerald W. Neufeld, External Examiner
(Department of Computer Science, University of British Columbia)

Supervisor: Dr. Eric G. Manning

Abstract

In distributed multimedia systems, real-time media are streamed over a network from senders to receivers. To obtain good playback quality at the receiver end, sufficient network and system resources must be allocated to transmit and process the media streams within rigid real-time constraints. However, a major problem is: how should network bandwidth be shared among applications during real-time multimedia transport, so that contention is avoided?

This thesis presents a solution to the above problem, by combining two well established concepts: the *Utility Model* and *layered coding and transmission*. The Utility Model deals with *quality of service* (QoS) adaptation of multi-session multimedia systems. The QoS of individual multimedia sessions is dynamically adapted in order to achieve the system utility objective (such as maximizing server revenue), while obeying system resource constraints (total available memory, for example) and application constraints (the minimal set of resources necessary to achieve minimal acceptable QoS). With *layered coding and transmission*, a media source is encoded into several signal layers, each sent over a distinct network channel (such as an IP Multicast group); the receiver can pick a subset of layers to receive and decode, thus varying both the delivered media quality and the bandwidth requirement. Hence, when applied to layer-coded sources, the Utility Model should be able to decide which subset of layers to receive, based on available reception bandwidth.

To demonstrate our approach, we implemented the Utility Model and resolved issues such as QoS mapping and resource monitoring. We present a Quality Utility Extension (QUE) implementing the Utility Model in a prototype multimedia system. We describe the QUE architecture and its two activities: admission control and quality adaptation. In the QUE, QoS mappings are done by QoS Agents, per-application proxies which handle the communication with a Utility Model Engine. The QUE also deals with issues in resource monitoring and contention avoidance under the constraint of a traditional best-effort operating system. Finally, we present some implementation specifics of the QUE, along

with two demo applications: VoD and QFTP.

Our experiments and results show that:

1. in a mixed (best-effort and reversion-based) system, the QUE is able to reactively adapt the QoS of applications in its domain to avoid bandwidth contention;
2. in an entirely reservation based system, the QUE respects both application and system resource constraints, while optimally adapting the QoS of applications to maximize total system utility.

The performance measurements of our implementation show that it is well-suited for real-time multimedia applications.

Examiners:

[redacted]

Dr. Eric G. Manning, Supervisor (Department of Computer Science)

[redacted]

Dr. Gholamali C. Shoja, Department Member (Department of Computer Science)

[redacted]

Dr. Kin F. Li, Outside Member (Department of Electrical and Computer Engineering)

[redacted]

Dr. Gerald W. Neufeld, External Examiner
(Department of Computer Science, University of British Columbia)

Table of Contents

List of Tables

List of Figures

Acknowledgments

I would like to thank my supervisor, Dr. Eric Manning, for his guidance during this research work, and helping me with "seeing the forest". His enthusiasm and encouragement have turned this challenging research work into an fun-filled area. I am grateful to Dr. Ali Shoja and Dr. Micaela Serra for the help and advice that have made my graduate study a meaningful one. I am thankful to Dr. Kin Li for taking time from his busy schedule and joining in my thesis committee.

I would like to thank the members of the PANDA group for their support during the course of my graduate study. I am grateful to have the chance to participate in the Wave Café and work with John Foxgord, James C. Pang, and Shahadat Khan. Their insightful discussions and brain-storming sessions have helped me explore the ideas in this fruitful area of research. I would also like to give special thanks to Shahadat Khan, who provided me with much encouragement and thoughtful comments to the earlier drafts of this thesis.

I gratefully acknowledge the financial support of the University of Victoria, in the form of a University Graduate Fellowship.

Dedication

To Wave Café

1 Introduction

1.1 The Problem

In distributed multimedia systems, real-time media are streamed over a network from senders to receivers. To obtain good playback quality at the receiver end, sufficient network and system resources must be allocated to transmit and process the media streams within rigid real-time constraints. However, a major problem is: how should network bandwidth be shared among applications during real-time multimedia transport?

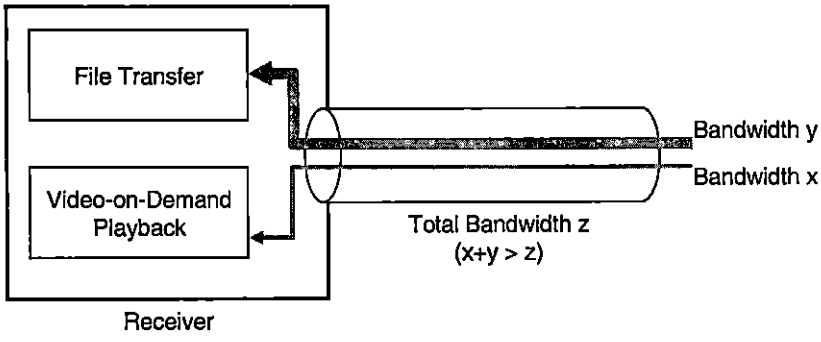

Figure 1.1 Network bandwidth sharing.

Consider a user engaged in a video-on-demand movie session (see Figure 1.1). Video is streamed from a server to the client system. It requires a minimum network bandwidth x to be allocated, such as 20Mbps for HDTV using MPEG, so that video frames can arrive in time for playback at the receiver. Now suppose that the user wishes to start a file

transfer, (perhaps fetching an article mentioned in the movie), but without degrading the movie quality. The file transfer consumes bandwidth y, but $x+y$ exceeds the total available bandwidth z. Such *bandwidth contention* causes network congestion, resulting in poor playback quality[1]. Clearly, either x or y must be reduced to solve the contention problem. However, with most current systems,

- reducing x means violating real-time video delivery constraints,
- there is no adequate means to reduce y, and
- the system provides no means for users to express preferences (e.g. more bandwidth should be allocated to the video stream).

In addressing the above problem, our research is motivated by two well established concepts: *Layered Coding and Transmission* (LCT)[27][18], and the *Utility Model* (UM)[17].

- With LCT, a media source is encoded into several signal layers, each sent over a distinct network channel (such as an IP Multicast group); the receiver can pick a subset of layers to receive and decode, thus varying both the delivered media quality and the bandwidth requirement. Therefore, the receiver can solve the bandwidth contention problem by choosing the right number of layers, based on available reception bandwidth[2].
- The Utility Model[17] deals with quality of service (QoS) adaptation of multi-session multimedia systems. The QoS of individual multimedia sessions is dynamically adapted in order to achieve an overall *system utility objective* (such as

1. Typically, congestion will cause the network to delay and ultimately discard packets at random, hence resulting unpredictable (and unacceptable) distortion for multimedia playback.
2. This has an effect of scaling multimedia playback quality (that is acceptable to the user), since we decide which packets can be discarded safely when congestion occurs.

maximizing server revenue), while obeying *system resource constraints* (total available memory, for example) and *application constraints* (the minimum set of resources necessary to achieve minimally acceptable QoS). User sessions are not *admitted* if the latter constraint cannot be satisfied (the notion of *admission control*). Furthermore, users can express their session quality preferences through some quality-to-utility mapping (such as paying more for a better level of quality). Therefore, when applied to LCT, the UM should be able to decide which subset of layers to receive, based on available reception bandwidth.

Khan[17] proposed the Utility Model, formulated it as a generalized Knapsack Problem, gave exact and approximate solution techniques, and demonstrated its on-line use in a simulated environment. However, no-one has demonstrated its use to optimally control a real, operational multimedia system. The goal of this thesis is to do that, in a variety of realistic situations. Specifically, this approach will allow us to do the following:

- in a system where the file transfer is not subject to the UM, (e.g., in best-effort mixed systems), and so there is no way to scale back the file transfer bandwidth requirement, the UM will scale back video playback quality to adapt to the remaining available bandwidth. This is typical of today's Internet, where applications needing QoS guarantees must coexist with best-effort applications.
- in an entirely reservation based system, where both video playback and file transfer are in the UM's domain, the QoS of both sessions are optimally adapted, while respecting users' preferences. This will be typical of future systems, where QoS guarantees will be provided to all applications in a QoS domain, segregated from those which do not.

Finally, by fully incorporating the UM into a real system, we can generalize our approach to deal with other shared resources such as CPU and memory.

1.2 Major Issues

To implement the Utility Model in a real system (such as the present Internet), and use it to control real applications, the following design issues must be considered:

- How do we design the interface between LCT and the UM, so as to accommodate different types of applications?
- What resources need to be monitored, and how to monitor them using an off-the-shelf operating system?
- How to derive quality-to-resource mappings?
- Since the UM was designed with a reservation based system in mind, how to use it in a best-effort system, such as the present Internet?

The last point is worthy of comment. Today's Internet is entirely based on best-effort services (as opposed to guaranteed services) and uses datagram services - late binding of resources - throughout. The same is true of traditional host operating systems, where resource reservation is often difficult or impossible. Multimedia applications, however, demand absolute, rigid guarantees of service which are best provided by *resource reservation* (early binding of resources). This in turn is most easily provided by the circuit switched primitives of ATM or the traditional voice network, rather than the late-binding datagram primitives of the current Internet. And so the wheel turns, from circuits to datagrams... and back again!

1.3 Implementation and Demonstration

To demonstrate our concepts and design, we need to implement the following:

- An adaptation framework involving the Utility Model, resource monitoring, and QoS scaling mechanisms like LCT.
- A video-on-demand (VoD) application, where live video is streamed over multiple IP Multicast groups using LCT.
- A client-server file transfer program (QFTP) capable of operating at multiple levels of quality.

In principle, the Utility Model should work reactively[1] in a best-effort system (such as the Internet) and pro-actively in a reservation based system (such as an ATM-based network). Therefore, we want to demonstrate two cases with our implementation:

- **Case 1: best-effort system** (see Figure 1.2). Suppose VoD is the only session supervised by the UM, and an unsupervised FTP session rudely seizes part of the reception bandwidth. We must demonstrate that the UM adapts by scaling back the playback quality of VoD. Again, this case arises in the Internet, where one attempts to indiscriminately mix applications needing QoS guarantees with traditional, best-effort Internet traffic such as email and FTP.

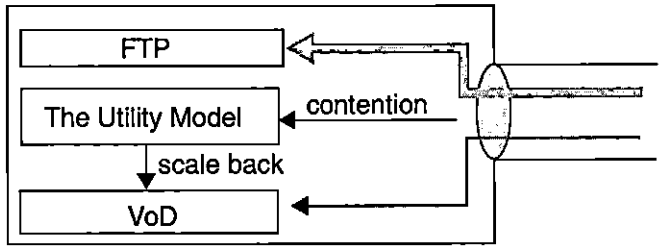

Figure 1.2 Reactive adaptation in a best-effort system.

1. Here, "reactive" means making a decision based on external stimuli; "proactive" is the opposite.

- **Case 2: reservation based system** (see Figure 1.3). Suppose all sessions (VoD and QFTP) are supervised by the UM. We show that the UM optimally adapts both session qualities, while respecting system resource constraints, application constraints, and user preferences. For example, if the user gives higher utility values (i.e. more dollars) to VoD, more bandwidth will be allocated to it. Again, the only way to provide absolute QoS guarantees to multimedia and other applications which require them, will be to provide a resource-sharing domain in which **all** applications are subject to admission and adaptation control. This tension between the growing need for QoS guarantees and the design of today's no-guarantees, best-effort Internet (and operating systems) lies at the root of the ATM versus datagram wars.

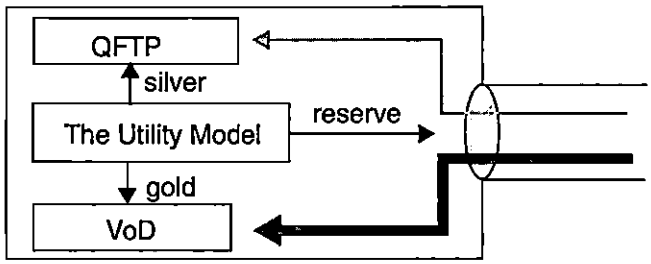

Figure 1.3 Pro-active adaptation in a reservation system.

The remainder of the thesis is organized as follows. Chapter 2 provides a brief description of the Utility Model and of LCT. Chapter 3 proposes how the Utility Model can be applied to LCT. Chapter 4 describes our Quality Utility Extension (QUE) implementing the Utility Model. Chapter 5 presents some implementation details of the QUE, along with two applications: VoD and QFTP. In Chapter 6, the experiments and results of our implementation are discussed. Finally, we present conclusions and future work in Chapter 7.

2 Related Work

2.1 The Utility Model

The Utility Model[17] is a mathematical model which captures the dynamics of adaptive multimedia systems. This model is based on the concepts of *quality profile*, *quality-resource mapping*, *session* and *system utility*, and *resource constraints*[1]. In this model, each session provides a *QoS profile*, which is a set of *operating qualities* arranged from the minimum acceptable quality to the maximum desired quality. Any operating quality may be mapped to the resources required to provide it using a *quality-resource mapping*, and to a session utility using a *quality-utility mapping*. The *Adaptive Multimedia Problem* (AMP) is to maximize some system utility function, subject to the system and application resource constraints. This model formulates the adaptive multimedia problem as the multiple-choice multi-dimension 0-1 knapsack problem (MMKP). It incorporates the dynamics of resource management and the interplay of media components of the existing sessions. Moreover, it provides a unified and computationally feasible way to solve the admission problem of new multimedia sessions, and the dynamic quality adaptation and integrated resource allocation problems of existing session.

1. Here, resource constraints include system resource constraints (e.g., total bandwidth) and application resource constraints (e.g. resources required for minimal acceptable quality).

The AMP can be expressed as a multiconstraint multiple-choice 0-1 knapsack problem (MMKP), a generalization of the classical 0-1 knapsack problem. Suppose there are n groups of items (or quality profiles), and m resources. Group i has l_i items (or operating qualities). When item j of group i is picked, it yields a utility value of u_{ij}, and consumes resource r_{kij} amount of resource k. The total consumption of resource k is constrained by R_k, amount of resource available. The MMKP is to pick one item from each group in order to maximize the total utility value of the pick, subject to the resource constraints. The following expresses the MMKP formally:

$$U = maximize \sum_{i=1}^{n} \sum_{j=1}^{l_i} x_{ij} u_{ij}$$

such that

$$\sum_{i=1}^{n} \sum_{j=1}^{l_i} x_{ij} r_{kij} \leq R_k \qquad (k = 1, \ldots, m)$$

$$\sum_{j=1}^{l_i} x_{ij} = 1 \qquad (i = 1, \ldots, n)$$

$$x_{ij} \in \{0, 1\} \qquad (i = 1, \ldots, n; j = 1, \ldots, l_i)$$

Figure 2.1 illustrates a simple example of the AMP as an MMKP. The simplest form of the Knapsack problem entailed packing a knapsack with rocks so as to maximize the total weight of the rocks chosen, without violating the single knapsack constraint of total volume. Here, in the MMKP, we have generalized rocks and knapsack. A knapsack (representing a multimedia system) has some constraints (total CPU and memory). Each multimedia session profile contains several quality choices (rocks). Each choice specifies a set of resource requirements (processor and memory) and a utility value (dollars). We

want to pick one rock from each session profile, so that the total utility value (revenue) of the knapsack is maximized, without violating the knapsack constraints (total system resources).

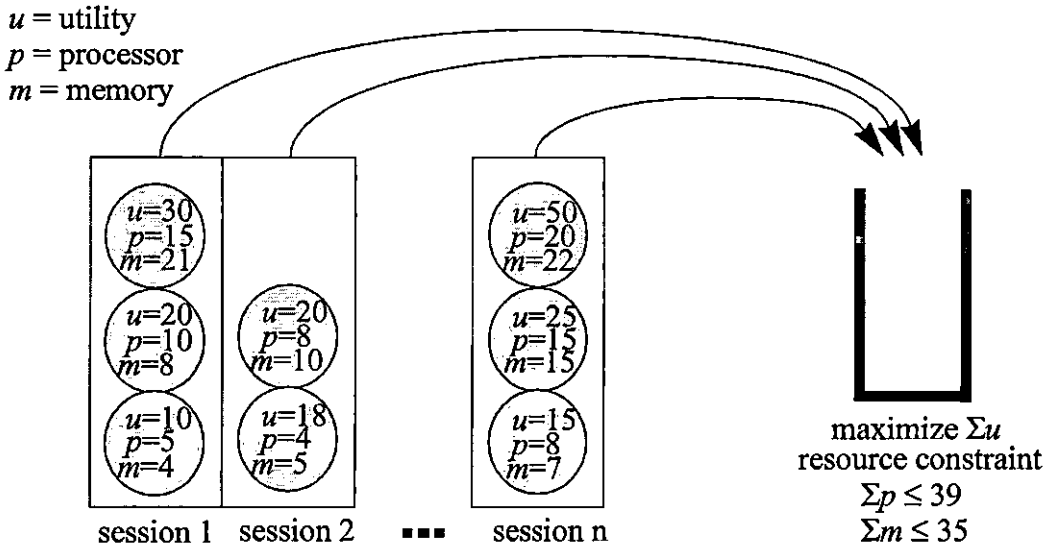

Figure 2.1 An example of the AMP as an MMKP in the Utility Model.

2.2 Layered coding and transmission

In the current Internet, real-time multimedia applications must adapt as best they can to fluctuating network conditions. A situation where adaptation is necessary is network congestion. By using layered coding and transmission (LCT), media at the source are encoded into hierarchical layers, each transmitted over a distinct network channel. A receiver can vary quality by selecting a subset of the layers to receive and decode. The best possible playback quality can be achieved by choosing the maximum number of layers such that network congestion is avoided.

2.2.1 Layered coding

Representations of multimedia are often *scalable*[27]. For example, a given image can have different resolutions, a given video stream can have different frame rates, and a given audio stream can have different bit rates. For real-time multimedia transmission, this scaling property translates to variable network bandwidth requirements. In video, two frequently exploited scalable elements are *Spatial scaling*, which reduces the number of pixels of each image in a video, and *Temporal scaling*, which reduces video resolution by decreasing the frame rate. The same techniques can also be applied to audio, although they may be unacceptable in practice[2].

Layered coding, a scaling method that exploits the scalable elements in multimedia, involves encoding media samples into hierarchical layers, different subsets of which can be decoded to obtain the media stream with different resolutions. Two of the most commonly used layering techniques are *Spatial layering* and *Temporal layering*.

- Typically used for images and video, spatial layering is used to generate multiple bit-streams or layers from a source, so that better pixel resolutions can be obtained by decoding incrementally more layers. Examples of Spatial layering include Progressive JPEG[9], MPEG[13], Pyramid coding[2], and Subband coding[11][29]. In particular, subband coding (the simplest form of which entails splitting a signal into two sub-signals and encoding them separately, in order to achieve better compression) is also the basis of many audio codec standards, such as MPEG audio, Dolby AC-2 and AC-3, the Sony MiniDisc system, and RealAudio[11][3].

---

2. This is because human hearing is very sensitive to quality variations in audio signals.
3. However, most audio coders interleave layers into a single bit stream because of tight synchronization requirements.

- Mainly used for video, temporal layering allows some of the frames to be dropped, resulting in different frame rates at the decoder. It can be easily applied to some popular video codecs, such as Motion-JPEG, H.261[30], and DVI[16]. There are also many other codecs which employ temporal layering techniques, such as MPEG-2[8], 3D Subband Video Coding[28], and Spatio-temporal Wavelet Pyramid[10].

Coders with hybrids of spatial and temporal layering also exist, such as PVH[18].

2.2.2 Layered transmission

Layered transmission can be implemented by sending the encoded signal *layers* over distinct network *channels*. In the Internet, this can be achieved by using IP multicast[6][4]. IP multicast is implemented using hardware multicast in a local area network, also called a *multicast island*. Between the *islands*, *tunnels* are built between multicast routers or *mrouters*. Mrouters use the Distance Vector Multicast Routing (DVMR) protocol[31] to calculate a multicast routing tree rooted at each source. This network of mrouters forms a multicast backbone, also known as the *MBone*[12]. In the MBone, hosts can receive multicast traffic by joining and leaving IP multicast groups using the Internet Group Management Protocol (IGMP). If a particular island has no members or receivers for a multicast group, the multicast branch (and hence the traffic) to the island is pruned for that group.

The ability to join and leave IP multicast groups without notifying the source is the key to solving multicast scalability and host heterogeneity problems. Work in layered transmission includes RLM[22][18], DSG[3], and LVMR[14][15]. Table 2.1 presents a

brief description of their approaches.

Protocol	Description
RLM	• The source transmits layered video streams over multiple IP multicast groups. The receivers attempt to obtain the best feasible quality video by joining all the groups incrementally. • Upon network congestion, indicated by packet losses, a receiver drops a layer. When spare network bandwidth is available, discovered by join experiments, a receiver will add a layer. • In a subnet, receivers use shared-learning to coordinate join-experiments.
DSG	• A video source maintains a small number of video streams, carrying the same video content but each targeted at receivers with different capabilities. Each video stream is feedback controlled within prescribed limits by its group of receivers. • The receivers use an intra-stream protocol to adjust the data rate of the stream, and an inter-stream protocol to change to a higher or lower quality stream as their needs (or capabilities) change.
LVMR	• Layered-video transmission from a sender to receivers using IP multicast. Receivers use join-experiments to obtain the best possible quality video. • Retransmission of loss packets given an upper bound on recovery time, and applying an adaptive playback point scheme to help achieve successful retransmission. • Adapt to network congestion and heterogeneity with a hierarchical rate control mechanism, which distributes join-experiments and congestion information among the sender, receivers, and some agents in the network in selecting the right number of video layers to transmit and receive.

Table 2.1 Approaches to layered transmission.

3 Applying the Utility Model

The key to solving the network bandwidth contention problem described previously is to select the right number of layers to receive and decode using Layered Coding and Transmission (LCT) at the receiver, based on available network bandwidth. We propose to use the Utility Model (UM) to provide adaptation decisions in such a receiver-driven system[1].

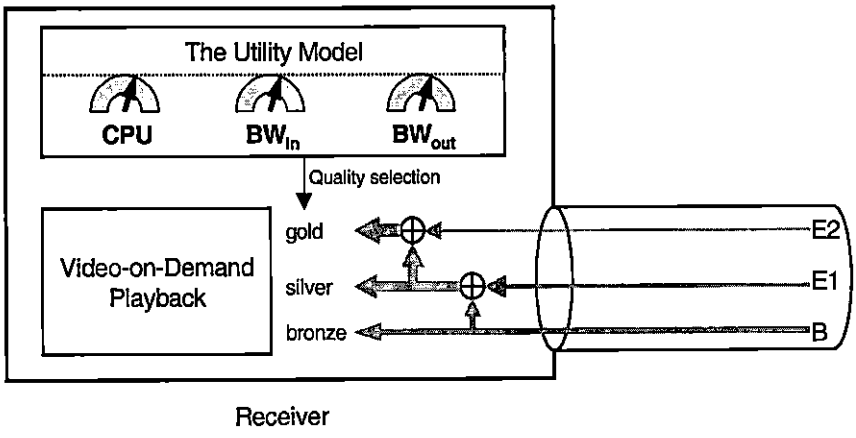

Figure 3.1 Network bandwidth sharing.

We propose to use LCT to provide QoS scaling (refer to Figure 3.1). During a video session, a live video source is hierarchically encoded at the sender into three layers: base (B), Enhancement One and Enhancement Two (E1 and E2). Each layer is sent over a

1. A system is receiver-driven if it makes adaptation decisions without involving the sender.

distinct IP multicast group, and the receiver receives it by joining the multicast group. The sender also defines three quality levels for the video stream: gold, silver and bronze. Each quality level is obtained at the receiver by receiving the base layer and, incrementally, the enhancement layers.

To apply the Utility Model to this real-world situation, the receiver first maps each reception quality level to a set of resource requirements[2]: inbound bandwidth, outbound bandwidth, and CPU. These requirements are necessary for satisfying the real-time constraints of video transmission and processing. For each quality level, a utility value (in dollars) is also given to express the user's willingness to pay. These mappings are presented to the UM as a session profile.

For example, suppose the bandwidth requirements of B, E1 and E2 in LCT are 30KByte/s, 30KByte/s and 40KByte/s, respectively. To process the layered signals, we may need 5% of the CPU cycles for decoding and 1KBtye/s outbound bandwidth for acknowledgment messages for each layer. Therefore, a session profile can be constructed as follows[3]:

	Bronze(B)	Silver(B+E1)	Gold(B+E1+E2)
inbound bandwidth (KByte/s)	30	60	100
outbound bandwidth (KByte/s)	1	2	3
CPU (% of cycles)	5	10	15
Utility value ($)	10	30	50

Table 3.1 Session profile.

2. Strictly speaking, only inbound bandwidth needs to be considered to deal with reception bandwidth contention among applications; however, we choose not to limit our design, so as to further explore other inter-resource and quality trade-offs among applications.
3. Note that, in this particular case, resource requirements increase for higher levels of qualities. However, this may not always be the case, as we expect trade-offs in resource requirements for different quality levels.

This session profile represents two mappings (see Figure 3.2): the quality-resource mapping and the quality-utility mapping. The quality-utility mapping is obtained from the session utility function $u(\cdot)$. The quality-resource mapping $r(\cdot)$ specifies the resource requirement for each level of operating quality Q_i. From these mappings, the UM is able to derive the total system utility $\sum_i u_i(Q_i)$, and total resource requirement $\sum_i r_i(Q_i)$.

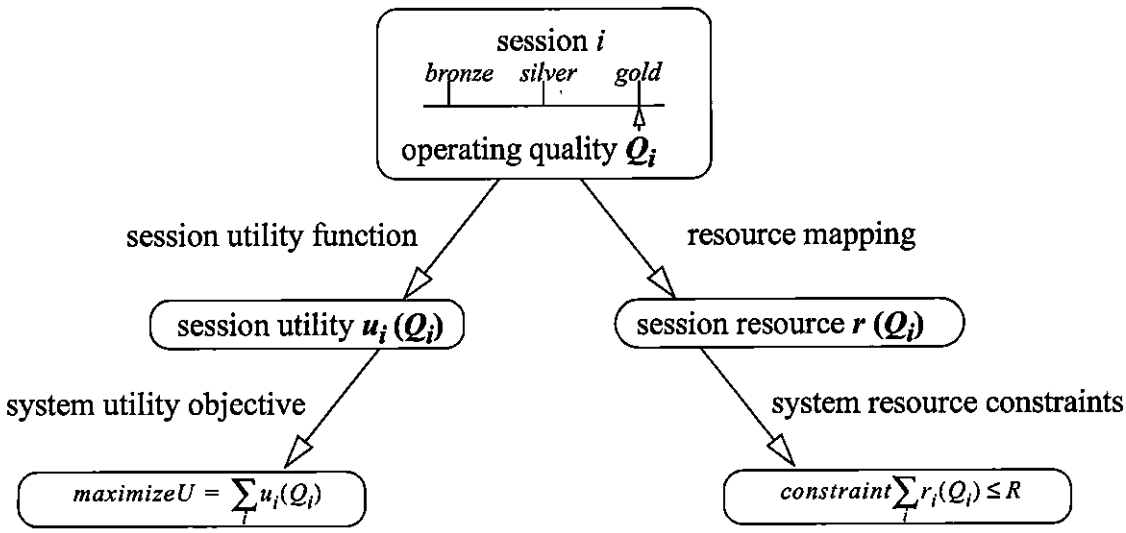

Figure 3.2 Quality-utility mapping and quality-resource mapping.

The goal of the UM is to maximize the revenue of the receiver system[4], subject to available system resources, while guaranteeing each application's minimal acceptable level of quality. Thus, the admission control of the UM will reject a new session if its minimal acceptable quality can not be provided, either because there are not enough available system resources, or because it is not profitable to admit the session. After the video session has been admitted, the UM guarantees its minimal quality (bronze) by

4. In this thesis, we only consider the receiver system. Our approach should work equally well in the sender system, where the UM may also be applied; however, the interaction among the UMs between both systems (if any) is beyond the scope of this thesis.

reserving the amount of bandwidth required for receiving the base (B) layer. The enhanced playback quality (silver or gold) is determined by the UM dynamically, after evaluating session profiles from all admitted applications and the available network bandwidth[5].

The user can, in effect, influence how the network bandwidth is allocated among concurrent applications by adjusting the utility values (dollars it will pay) in the quality profiles, such as paying more for the gold quality in the video session.

5. For the moment, we assume other resources (CPU and outbound bandwidth) are ample.

4 The Quality Utility Extension

In the previous chapter, we have described how the Utility Model (UM) can be applied to layer-coded sources. To implement it in a real system, however, we must consider the following design issues:

- How does the UM interact with different types of applications?
- What resources need to be monitored and how to monitor them, in an aging, classical operating system (e.g. UNIX™)?
- How to derive *QoS mappings*, and how accurate should they be?
- Since the UM was designed with a reservation based system in mind, how can we use it in a best-effort system, such as the present Internet?

No existing system addresses all of these issues. Hence, we had to develop a system implementation architecture for the Utility Model, called the *Quality Utility Extension* (QUE).

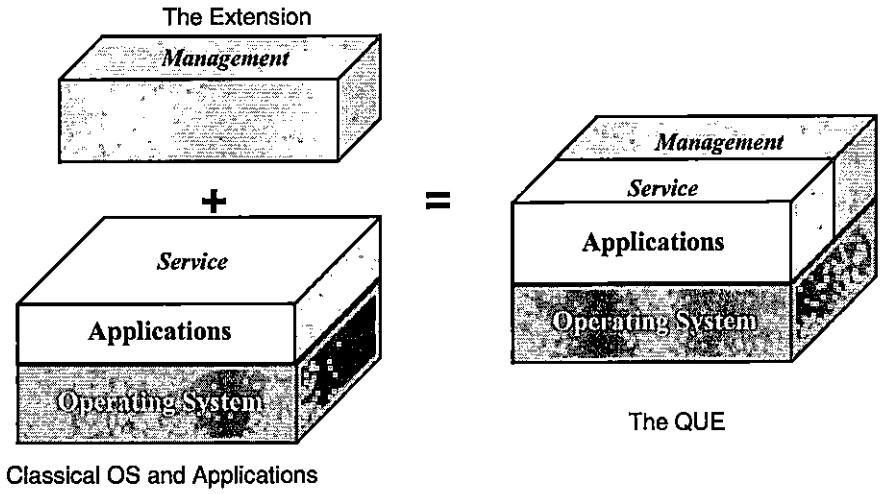

Figure 4.1 The Quality Utility Extension.

The QUE is a resource management extension implementing the Utility Model in a prototype system (see Figure 4.1). It allows multiple levels of service quality with resource management. In the QUE, *service* and *management* are two distinct activities. *Service* is provided by applications which consume resources provided by the operating system; *management* is based on the Utility Model, where system resources are monitored and levels of service quality are dynamically adapted. The QUE is reservation based[1] — application execution is subject to admission control and quality adaptation policies (such as maximizing total system revenue).

1. Here, "reservation" is used to describe the behavior (*policy*) *in our model.* This is not to be confused with the reservation *mechanisms* provided in operating systems and networks. We argue that if all applications conform to the reservation model, then we can implement a reservation-based system on top of a best-effort operating system and network.

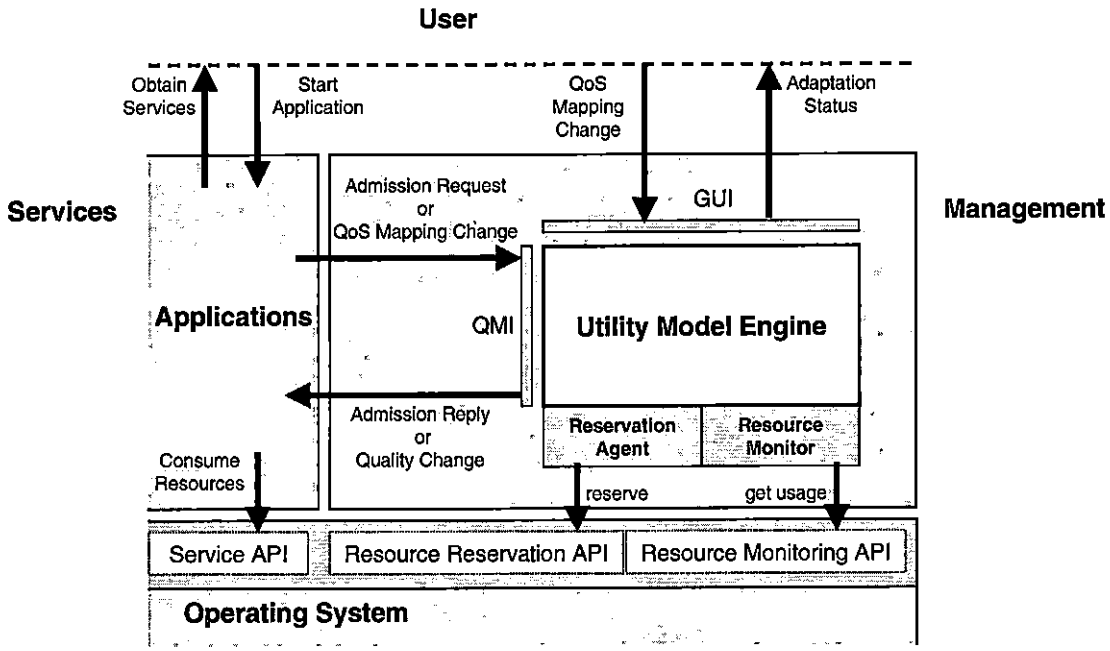

Figure 4.2 QUE components.

4.1 The QUE Architecture

The QUE architecture comprises a *Utility Model Engine*, the operating system, and applications (see Figure 4.2). A user obtains services by starting the applications. Built on top of the operating system's Service API, applications interact with the Utility Model Engine (UME) through a *Quality Management Interface* (QMI). The QMI supports messages such as admission request and reply, QoS mappings and adaptations. The UME provides the user with a *graphical user interface* (GUI) showing adaptation status, as well as accepting QoS and utility mapping changes from the user. We assume the operating system has a *Resource Reservation* API (such as in RT-Mach[24] and in ATM networks[23]) and a mandatory *Resource Monitoring* API[2]. From these two APIs, the UME can pro-actively reserve resources and reactively adapt to externally-imposed

changes in available resources. The UME carries out the core functionality of the QUE; it is based on the concepts defined in the Utility Model: *session profile*, *quality-resource mapping*, *session* and *system utility*, and *resource constraints*; it also implements the algorithm for solving the MMKP. There are two key activities in the QUE: *admission control* and *quality adaptation*.

4.1.1 Admission Control

In order to guarantee the resources necessary to provide it the minimal acceptable QoS, each application in the QUE must be first *admitted* before it can start execution. To be admitted, it must provide a *session profile* to the UME. A *session profile* contains a *quality-resource mapping* and a *quality-utility mapping*. The admission control is based on available system resources and session profiles from all other active applications. The admission contract requires that, **once admitted, the application must obey the adaptation decisions of the UME, and that the UME must guarantee the minimal quality requirements of the application by reserving the minimal resources required.** The total amounts of resources needed to support minimal quality requirements of all active applications are thus referred to as *reserved resources*. However, an application will be rejected if one of the two following conditions occur during admission control:

1. when all other active applications are already running at minimal qualities, leaving no room for adaptation, and the UME cannot find enough resources to support the application's minimal quality;

---

2. These assumptions are made so that the Utility Model, which is based on resource reservation, can be better captured by our system. The actual implementation of our system architecture may vary, however, due to the constraints imposed by many real world scenarios. These are further explored in the later sections of the thesis.

2. when there is room to support the application's minimal quality (e.g, by adapting other applications' qualities to free up resources), but doing so results in decreased total system utility.

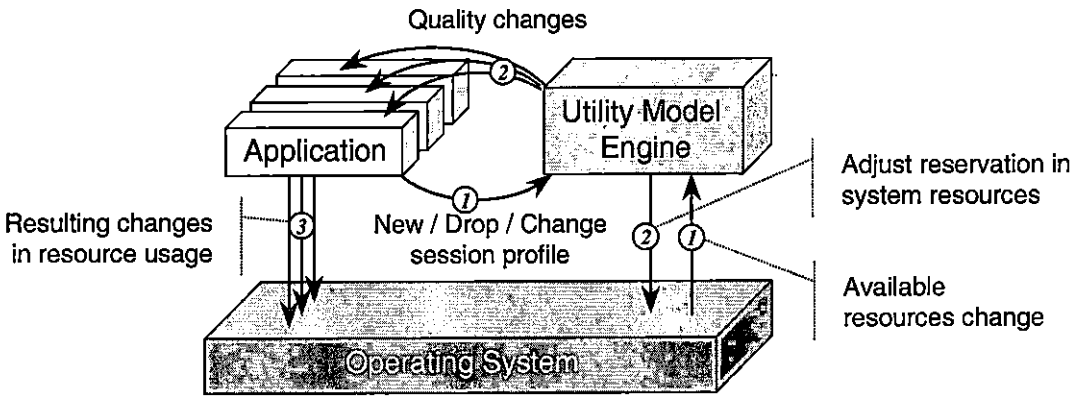

Figure 4.3 Quality Adaptation.

4.1.2 Quality Adaptation

In the QUE, the use of available system resources is managed by dynamically adapting active sessions' operating qualities. Hence, the UME performs quality adaptation to enforce the resource management policy: maximizing total system utility. Quality Adaptation (see Figure 4.3) occurs when

- a new application is admitted;
- an application changes the QoS mapping in its session profile;
- an application is dropped (i.e. its execution terminates); or
- there is a change in the total available system resources, due to failure, repair or reconfiguration.

Quality adaptation decisions are results of the BBLP[17] algorithm or the HEU[17]

heuristic implemented by the UME. After adapting each active session's operating quality, the UME sends "quality change" messages to the corresponding applications, while *allocating* enough resources to meet the quality requirements and making reservation adjustments at the OS level when necessary. The total resource requirements for all active sessions' operating qualities are thus referred to as the *allocated resources*. We note that the allocated resources are a superset of the reserved resources.

It is important to note that the contract between an application and the UME must always be upheld. Therefore when an application changes its QoS mapping, the resource requirement and the utility for its minimal quality are not allowed to change. Otherwise, Admission Control can be defeated, since a session may get more than its promised share of resources by increasing its minimal quality resource requirements after admission, but without paying more; or, in a similar fashion, an application could cheat by reducing the utility (or payment) for the minimal quality after being admitted.

Changes in the total available system resources are possible if the UME is used to manage only a portion of the total system resources. For example, an operating system may wish to allocate 5% of CPU bandwidth to kernel threads and leave the remaining 95% to the UME, but at run-time, it may reduce the CPU bandwidth managed by the UME to 80%. Furthermore, in a best-effort mixed system, where only some applications conform to the QUE, system resources may be consumed uncontrollably by other applications. Hence, in such a system, the total resources available to the UME are effectively the remaining free resources. Finally, hardware failure, repair or reconfiguration can change the total resources available to the UME.

4.2 QoS Mapping

In a generalized multimedia system, there are many levels of QoS. Quality at the *user level* is specified through *user-level* QoS or *perceived* QoS. For example, "good quality" may mean smooth video playback and surround sound. At the *application level*, on the other hand, smooth video playback could mean 30 frames per second. Finally, at the *system level*, QoS is specified in terms of 50% of the CPU cycles and 1.5 Mbps network bandwidth. Thus, for an application offering services of a particular level of quality, QoS mappings from user-level to application-level and system-level are essential.

For example (see Figure 4.4), a service provider may offer a TV-like multimedia program with three levels of quality (i.e. user-level QoSs) to its customers: gold, silver, and bronze. In a receiver, these quality levels are mapped to different QoS parameters at the application level, such as video frame rates, audio quality, image resolution, 3D-space consistency, and data (e.g. stock ticker) latency. At the system level, a certain amount of resources is required to support each of the application-level QoS settings.

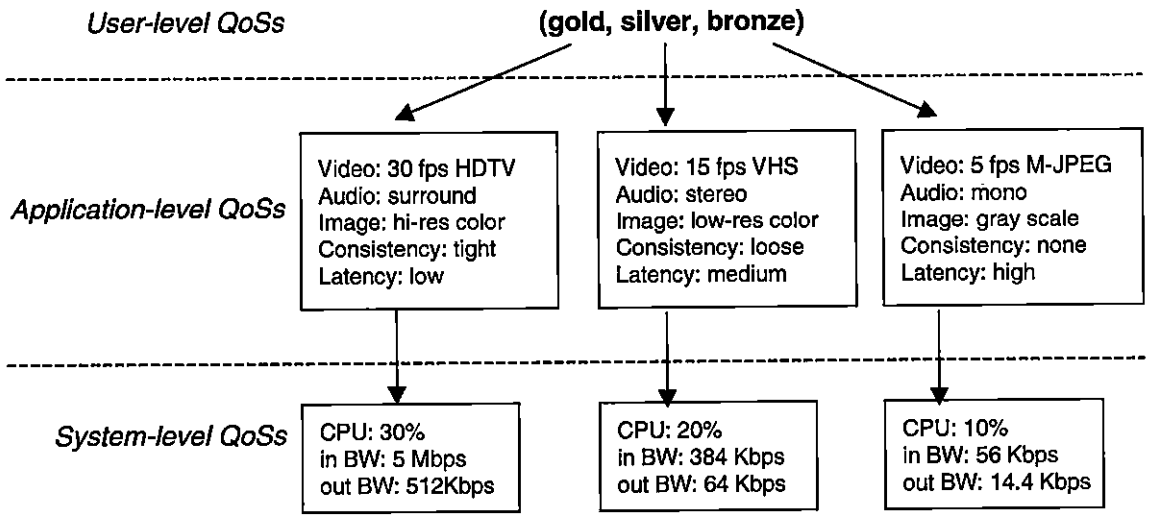

Figure 4.4 QoS mappings.

4.2.1 Quality-methods-resources Mapping

In the QUE, quality-resource mappings, $r(\cdot)$ (see Figure 4.5), required by the Utility Model are broken down into *quality-methods mappings* $m(\cdot)$ and *method-resources mappings* $r(m_i)$. The quality-methods mapping maps from user-level QoS to application-level QoS; The method-resources mapping maps from application-level QoS to system-level QoS.

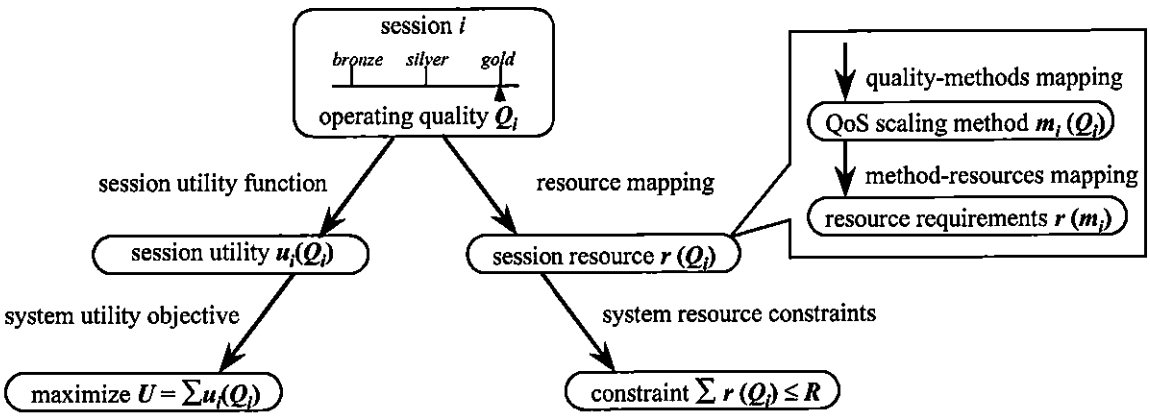

Figure 4.5 QoS mappings in the QUE vs. the Utility Model.

The user-level QoSs in the QUE are *gold*, *silver*, and *bronze*. Applications are responsible for implementing user-level QoSs and mapping them on to a set of application-level QoSs. Such implementations are called *QoS scaling methods*, and the mapping is thus called a quality-methods mapping. For example, when using a modem connection, a video-on-demand application may choose to implement the gold quality by a method

"`set_fps(15); set_resolution(VHS); set_sound(mono)`",

but when running in a fast ATM network, it may map the gold quality to the method

"`set_fps(30); set_resolution(HDTV); set_sound(surround)`".

Furthermore, since QoS scaling methods are application and implementation specific, an application programmer can analyze their resource requirements at the time of development. Therefore, applications are also responsible for providing the application-level to system-level QoS mappings by method-resources mappings

("`set_fps(15)`" maps to 30 Kbps, for example)

reflecting the peak resource requirement of QoS scaling methods.

Following the example given previously in Figure 4.4, an application receiving the multimedia TV program may use different sets of function calls to tune the application-level QoSs:

"`set_fps()`", "`set_sound()`", "`set_img_res()`", "`join_consistency_group()`", and so on (see Figure 4.6). Therefore, each user-level QoS is mapped to a method, a set of above function calls, which then maps to a set of aggregate system resource requirements.

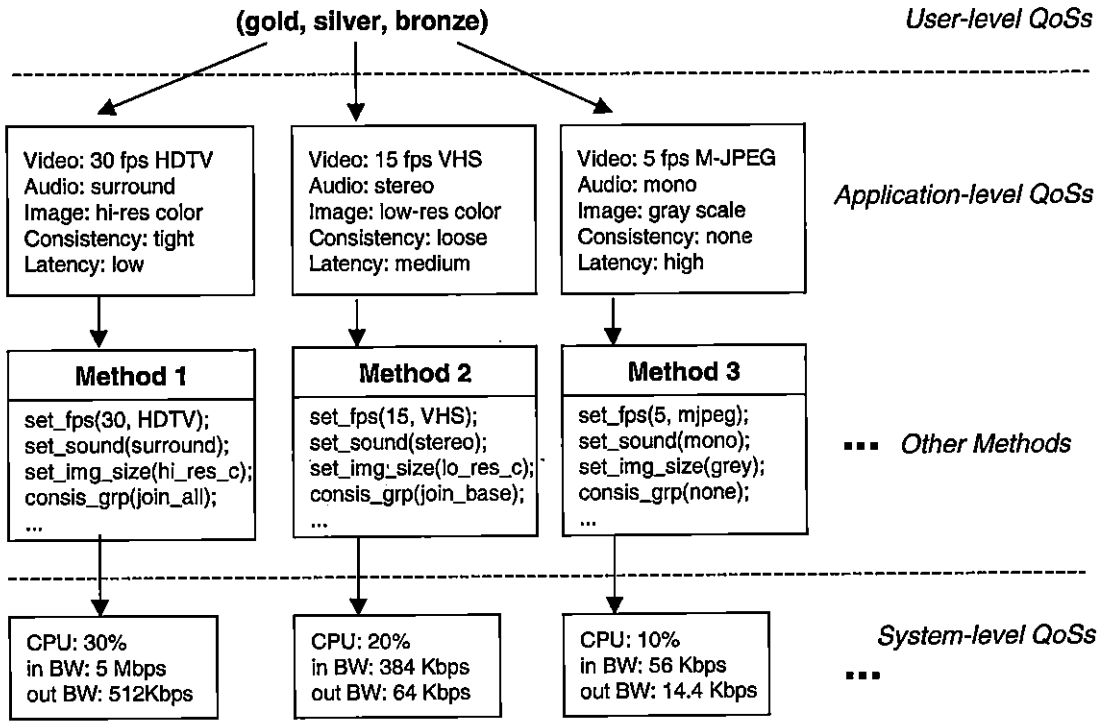

Figure 4.6 Quality-methods-resources mappings.

This structure of quality-methods and method-resources mappings allows flexible implementation of services with multiple qualities and integrated resource requirement specification. For example, an application may provide several QoS scaling methods (e.g. for ATM and Ethernet environments) at compile time, but adaptively maps to the gold quality at the run-time. A more intelligent application may even profile its QoS scaling methods dynamically at run-time to calculate more accurate method-resources mappings. Ultimately, this quality-methods-resources mapping implements the quality-resource mapping required by the Utility Model.

4.2.2 QoS Agents and Session Profiles

For the UME to manage applications' qualities and system resources in the QUE, each application must implement the above described quality-methods-resources mappings and quality-utility mappings. These mappings are handled by a per-application proxy called the *QoS Agent*, which facilitates communication with the UME through the Quality Management Interface (see Figure 4.7). Below, we present an implementation reference model for QoS Agents and the format of the session profiles they generate.

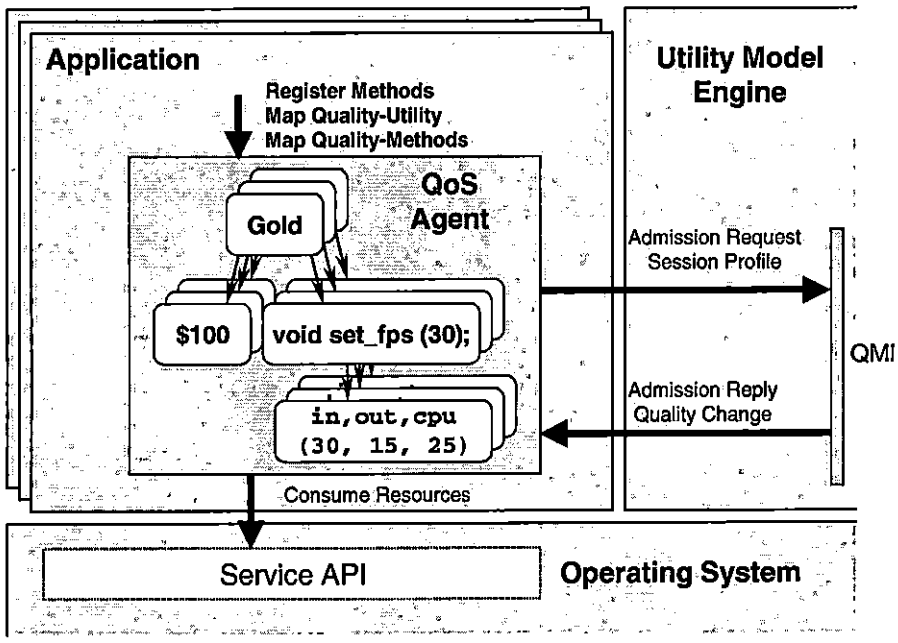

Figure 4.7 The implementation reference model of a QoS agent.

A QoS Agent (QA), provided for each application in the QUE, maintains the application's quality-utility mappings and QoS mappings, communicates with the UME, and adjusts the levels of service quality on behalf of the application. The quality-utility mappings are provided by users prior to application execution and may be changed dynamically at run-time. To obtain QoS mappings, a QA allows an application to register QoS scaling methods and provide quality-methods and method-resources mappings. Upon the start-up of an application, its QA sends an admission request to the UME, followed by a session-profile encapsulating the QoS mappings. If the UME rejects the admission request, the QA terminates the application. Otherwise, the UME will admit the application by replying to its QA with an operating quality. Once the application has been admitted, for the duration of its execution, its QA listens for quality change signals from the UME

(as results of quality adaptation) and activates the corresponding registered QoS scaling methods.

In addition to the above basic operations, a QA may also implement some advanced features. These may include

- a graphical user interface allowing run-time QoS and utility mapping changes from the user,
- *application-level QoS monitoring* that allows quality-utility and quality-methods mapping decisions to be made dynamically, (for instance, if a QA finds that 30 fps video required by the gold quality is not delivered[3], it may decrease the payment for the gold quality),
- *run-time profiling* for QoS scaling methods that allows more accurate methods-resource mapping, by obtaining fine-grained resource usage measurements for its application at run-time, and
- *auto-bidding* which adjusts the quality-utility mapping, so that a user specified quality can be persistently obtained, by paying whatever it takes to obtain that quality.

Although we leave the implementation of QoS agents to each application[4], the protocol governing exchanges between a QA and the UME needs to be clearly defined (see Figure 4.8). It consists of several types of messages exchanged through the UME's QMI. Each message contains a message type, a session ID, and a message body. The session ID is a unique name identifying the QA. The message type is one of "new",

3. If the sender reduces the video to 20 fps, for example.
4. The QUE only provides a QA with the basic functionality for each application. An application must extend the basic QA to implement above-specified advanced features.

"change", and "drop", augmented by the *session profile* encapsulated in the message body. In addition to the *quality-utility* and *quality-methods-resource* mappings, a *session profile* defines the number of quality levels (i.e. user-level QoSs), the number of system resource types, and the number of *QoS scaling methods*. Assuming quality levels are zero-based integer values, the UME replies to each successful admission request with a message containing an integer value, indicating the admitted operating quality; a rejection message, on the other hand, contains a negative integer value. The UME uses the same method to inform each QA of its operating quality, after each quality adaptation. For example, a QA may receive the following types of message from the UME:

- `-2` : normal termination by user request,
- `-1` : admission request rejected,
- `0` : bronze quality level
- `1` : silver quality level
- `2` : gold quality level
- `3` : platinum quality level.

Upon reception of such message, each QA invokes the corresponding QoS scaling method, by consulting the quality-method mappings in its session profile.

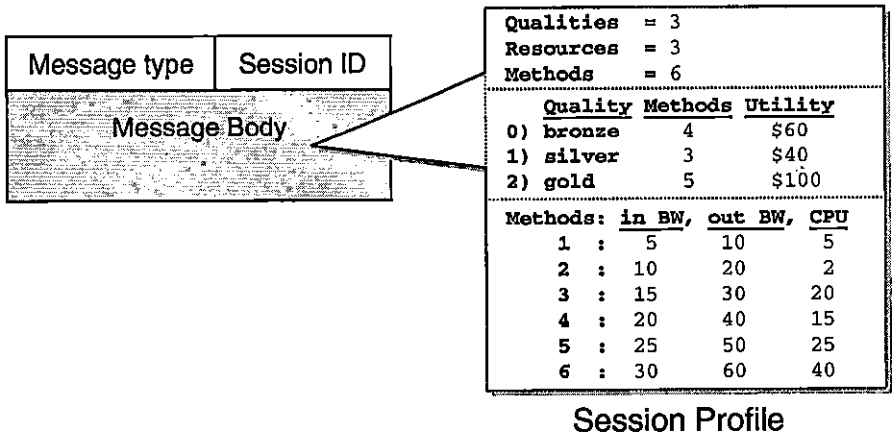

Figure 4.8 QMI interface message format.

4.3 Resource Monitoring in the Real World

In oder to make quality adaptation decisions, the UME must rely on system resource monitoring. In an ideal multimedia system, the operating system will provide accurate system resource usage information for each application, as well as resource availability and reservation status; the UME or the applications could access this information through a *Resource Monitoring* API and a *Resource Reservation* API.

However, in reality, there are no such well defined API and standards, and existing OSs fall far short of providing them. Most approaches to system resource monitoring follow conventions and are by-products of specific operating system implementations.

For example, in UNIX™, many system resource monitoring tools monitor the kernel data structures through the "proc" pseudo-file system available in the System V implementation. Similarly, network interface card (NIC) utilization can be obtained from a SNMP[5] implementation, which, however, is designed for network management.

Furthermore, resource reservation is not supported by current general purpose operating systems and networks (such as UNIX™ and the Internet), and the APIs for reservation mechanisms in reservation-capable systems and networks (such as RT-Mach and ATM based networks) are far from trivial. Therefore, it is a challenge to design the interface between the UME and available resource monitoring and reservation mechanisms in a general way.

Since, at the time of writing, the only research platform available to us is UNIX™ and the Internet, the study of reservation mechanisms in reservation capable systems and networks is out of the scope of this thesis. Therefore, our discussion of resource monitoring in the UME is based on what is available in an elderly but still popular general

purpose operating system (the Linux operating system, for example). In the following sections, we describe how resources (CPU, inbound and outbound bandwidth) are monitored in the Linux operating system, how resources managed by the UME are adapted, and finally how resource contention is avoided by the UME.

4.3.1 Requirements for Resource Monitoring

The QUE requires both *fine-grained* and *coarse-grained* resource monitoring. *Fine-grained* resource monitoring means that the use of each resource is monitored for each application. *Coarse-grained* resource monitoring means that the use of each resource is monitored for the system. The resources which need to be monitored are CPU, disk, memory, inbound bandwidth, and outbound bandwidth. For simplicity, we concentrate only on CPU, inbound and outbound bandwidth in this thesis.

In an ideal operating system, fine-grained resource monitoring enables the shaping or policing of the use of each resource, guaranteeing a reservation for each application. It also allows each application to find out the utilization of its reserved resources, so as to make intelligent decisions (such as increasing the reservation when a backlog persists for resource requests). Coarse-grained resource monitoring allows resource management at the system level (such as increasing network bandwidth reservation for the entire system). Therefore, in such an ideal system, the QUE will use fine-grained monitoring for QoS Agents (e.g. for dynamic profiling), and use coarse-grained monitoring for the UME (e.g. to obtain the total amount of a remaining free resource).

However, existing best-effort systems (such as UNIX™ and the Internet) fall far short of meeting this resource monitoring requirement. Therefore, we must use available

monitoring mechanisms in our implementation platform, the Linux operating system, for the QUE. This demands compromises, not beautiful, which are described in the next section.

4.3.2 System Level Resource Monitoring in Linux

The most common way to monitor the system state in Linux is to monitor the pseudo-file system mounted on */proc*, which interfaces to the OS kernel's data structures. */proc* is used primarily for debugging during kernel development, but in our case, it also shows how some of the system resources are being used. In particular, for system level resource monitoring, we can monitor */proc/stat* for CPU utilization and */proc/net/dev* for network utilization. Appendix A shows the file formats of the */proc* hierarchy.

During each time interval t (e.g. one second), we obtain the *monitored* resources comprising CPU (CPU_t) and outbound (Tx_t) and inbound (Rx_t) network bandwidth usage. In */proc/stat*, the scheduler in the OS kernel records the CPU time spent in *user* processes, *nice* processes, *idle* processes, and *system* (i.e. kernel) processes. Under normal circumstances, the CPU utilization is approximated by the CPU time spent in *user* processes[5]. In */proc/net/dev*, the kernel records cumulative total transmitted (*totalTx*) and received (*totalRx*) packets for each network interface card. Thus, the *monitored* resources can be obtained by the following:

---

5. *Nice* CPU time is consumed by user processes with explicit low priority which are only executed when all other higher priority processes are blocked (e.g. waiting for input); *system* CPU time is consumed by kernel mode operations and is usually negligible; *idle* CPU time is consumed by the idle task, which executes only when all other processes in the system is blocked.

Monitored resource at time t = $\left[CPU_t , Tx_t , Rx_t\right]$

where

$$CPU_t = \frac{user}{user + nice + idle + system} \quad \text{(\% of total CPU cycles per second)} \quad \textbf{(4.2.1)}$$

$$Tx_t = totalTx_t - totalTx_{t-1} \quad \text{(packets per second)}$$

$$Rx_t = totalRx_t - totalRx_{t-1} \quad \text{(packets per second)}$$

Note, however, at the system level, resource monitoring is coarse grained, since there is no well defined per-process resource monitoring mechanism[6].

4.3.3 Adapting Resources Managed by the UME

Traditional general purpose operating systems (such as Linux) cannot provide any hard guarantees for resource allocation. Thus, if an application out of the domain of the UM seizes a portion of resources previously managed by the UME, resource contention is inevitable. Therefore, for each resource, the UME must decide how much of it is still available or *manageable*. However, at the system level, the resource monitor described previously is unable to distinguish the resource used by the application in the QUE from the resource used by out-of-domain processes. So, how can the UME derive the total amount of remaining resource?

To solve this problem, we first define the total amount of a system hardware resource as the *raw* resource, (such as 100% of CPU cycles or 10Mbits/sec bandwidth), which comprises *unmanageable* (i.e. consumed by out-of-domain processes) and *manageable* resource. As a result of quality adaptation at time *t-1*, the UME will partition the $manageable_{t-1}$ resource into $allocated_{t-1}$ and $free_{t-1}$ resource. At time *t*, however, the only

6. In Linux, per-process monitoring is only available for some resources (e.g. CPU and memory).

known measurements are raw_t and $monitored_t$. The problem now becomes: given raw_t and $monitored_t$, how much resource is $manageable_t$? (see Figure 4.9.)

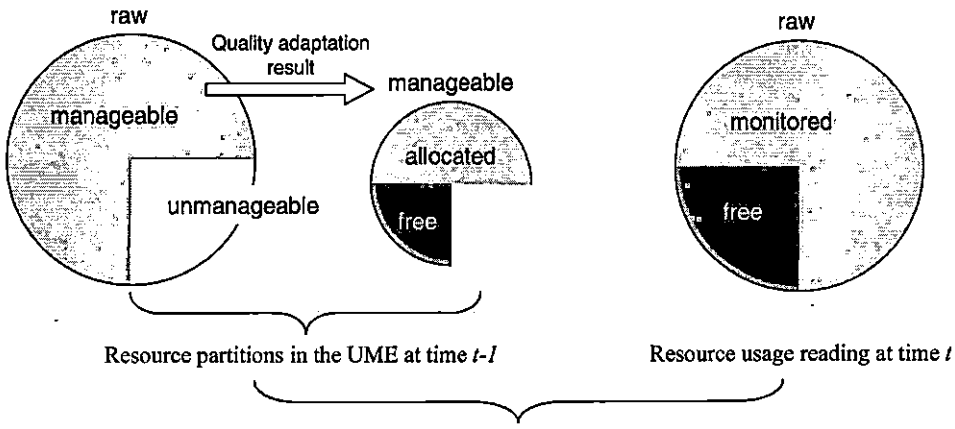

Figure 4.9 Adapting the amounts of manageable resources.

Suppose that we have an initial allocation of manageable resource at time $t=0$. This may be the result of a static allocation, such as giving 95% of the raw resource to the UME (see 4.3.1).

$$\begin{aligned} manageable_0 &= 0.95 \times raw_0 \\ unmanageable_0 &= 0.05 \times raw_0 \end{aligned} \qquad \textbf{(4.3.1)}$$

Now, at run-time, applications in the QUE consume their allocated resources, and out-of-domain processes consume the unmanageable resources. At time t, the monitored resource, $monitored_t$, is a measurement of resource usage during the interval $(t\text{-}1, t)$. Therefore, it should consist of the expected usage of $allocated_{t-1}$ and $unmanageable_{t-1}$. We assume that applications in the QUE do not exceed their allocated amounts of resource, and that if only a portion of an allocated resource is consumed, the remaining portion can

be borrowed by out-of-domain processes. Thus, there are two cases to consider at time t:

Case one: $Monitored_t$ is less than or equal to $allocated_{t-1}$. This means that there are no (significant) out-of-domain activities. Therefore, we should keep the manageable resource to the maximum (i.e. the initial condition).

Case two: $Monitored_t$ is greater than $allocated_{t-1}$. In this case, there are significant out-of-domain activities, and we need to decide whether the manageable resource is to be adjusted. If out-of-domain activities increase their resource consumption, and we do not scale back the manageable resource, then there is a danger of resource contention[7]. For example, when the manageable resource is 95%, the allocated resource is 0%, but the monitored resource is 15% (10% more than it should be), then we may encounter contention if the UME admits new sessions based on the value of 95% for manageable resource. To scale back the manageable resource, we first assume that the allocated resource is consumed entirely by applications in the QUE. Thus the actual consumption of the unmanageable resource, $unmanageable_{t-1}$ (15% here), is given by subtracting $allocated_{t-1}$ (0%) from $monitored_t$ (15%). Now, to avoid possible resource contention at time t, we must scale back the $manageable_t$ (to 85% here) based on the assumption that the resource usage pattern will continue during the time interval $(t, t+1)$. That is, we expect that $unmanageable_t$ will be the same as $unmanageable_{t-1}$. Thus, for time interval $(t, t+1)$, the amount of $manageable_t$ resource can be obtained by the subtracting $unmanageable_t$ (15%) from the raw_t (100%). Note that this also takes into consideration of changes in raw_t, due to hardware error or reconfiguration. Equation 4.3.2 summarizes the above

7. On the other hand, if out-of-domain activities decrease their resource consumption, then we want the manageable resource to be replenished.

calculation.

$$\left.\begin{array}{r} manageable_t + unmanageable_t = raw_t \\ allocated_{t-1} + unmanageable_{t-1} = monitored_t \end{array}\right\} \quad \text{(4.3.2)}$$
$$\Rightarrow manageable_t = raw_t - (monitored_t - allocated_{t-1})$$

After the value of the variable $manageable_t$ is determined at the beginning of a time interval, the UME runs the BBLP algorithm or the HEU heuristic for quality adaptation. The quality adaptation may cause changes in applications' operating qualities and their allocated resources, thus deriving the values of the variables $allocated_t$ and $free_t$.

If out-of-domain processes do not increase their resource consumption ($unmanageable_t$) from the last time interval, then our adjustment to the manageable resource guarantees that resource contention will not occur. If, on the other hand, $unmanageable_t$ increases, then resource contention could result, and we must detect and react to it. In the following section, we explore this scenario. (Of course, none of this would arise if all QoS-demanding processes and a fixed set of resources were in the domain of the UM, as they should be in a well-designed, modern OS.)

4.3.4 Avoidance of Resource Contention

Resource usage contention occurs only when the required amount of resource exceeds the total available resource (i.e. the raw resource). In the absence of operating system level reservation and policing mechanisms, such as in a traditional best-effort general purpose operating system, contention occurs and detection is very important. Even more so, contention is to be avoided at all cost, because when it is detected, it is often too late to react to it. However, with coarse grained system level resource monitoring, it is difficult to

determine resource usage contention (see Figure 4.10). We know that contention **could** occur if the amount of a monitored resource is equal to the raw resource (i.e. *free = 0*); but, it is also possible that the system is at a full utilization point without any contention. In other words, our resource monitoring shows how much resource is used, not how much is *overused.* This is also reflected in the previous calculation (Equation 4.3.2) of manageable resource: when the monitored resource equals the amount of raw resource, the manageable resource is equal to the allocated resource, showing full utilization of the system resource. Hence, a problem arises: since there is no way to detect contention at the system level, it is difficult for the UME to react to it with quality adaptation.

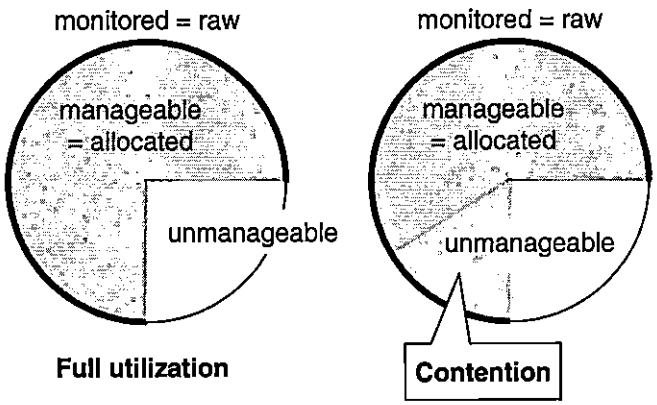

Figure 4.10 Resource contention detection problem.

The solution to the contention detection problem is to use *contention avoidance.* That is, we try to scale back the *manageable* resource **before** it reaches a possible contention state. Because the manageable resource is well regulated by the UME, the cause of contention can only be the increase of the unmanageable resource. Our previous adjustment to the manageable resource (see Equation 4.3.2) avoids possible contention caused by monitored increase of the unmanageable resource. If the monitored resource is

100% at present, and if the unmanageable resource increases for the next time interval, then we have contention. Thus, to avoid it, we place a *Cushioning Region* (CR) between the manageable resource and the unmanageable resource (see Equation 4.4.1).

$$\begin{aligned} & manageable + CR + unmanageable = raw \\ \Rightarrow & manageable = raw - unmanageable - CR \end{aligned} \tag{4.4.1}$$

Effectively, the value of the variable CR (such as 1% of the *raw* resource) indicates how much the manageable resource should be scaled back to avoid potential contention when the resource usage approaches a full utilization point. The larger the value of CR, the more aggressively the UME will scale back the manageable resource. Therefore, when the monitored resource equals the raw resource, we modify the manageable resource as follows:

$$\left.\begin{aligned} manageable_t + CR + unmanageable_t &= raw_t \\ allocated_{t-1} + unmanageable_{t-1} &= monitored_t \\ monitored_t &= raw_t \end{aligned}\right\} \tag{4.4.2}$$

$$\begin{aligned} \Rightarrow manageable_t &= raw_t - (monitored_t - allocated_{t-1}) - CR \\ \Rightarrow manageable_t &= allocated_{t-1} - CR \end{aligned}$$

As we can see in Equation 4.4.2, when there is contention danger (the monitored resource equals the raw resource), the value of the manageable resource is the value of the allocated resource, less the value of CR. As a result, the manageable resource is reduced below the previous amount of allocated resource, meaning that the system can no longer support the applications' previous operating qualities. Therefore, the quality adaptation in the UME will force active sessions in the QUE to obey the new system resource constraints by scaling back their operating qualities and reducing their resource

requirements. Consequently, in the new time interval, there should be at least value-of-CR amount of resource cushioning the effect of possible increases in the unmanageable resource.

Finally, Figure 4.11 shows the pseudo-code for dynamic adaptation of the manageable resource.

```
handle_resource_change (monitored) {
   if (monitored < allocated) {
      manageable = 0.95 * raw;
   } else if (monitored < raw) {
      manageable = raw - (monitored - allocated);
   } else if (monitored = raw) {
      manageable = allocated - CR;
   }
   if (manageable < reserved) {
      manageable = reserved;
   }
   solution = UME_adapt_quality (manageable);
   allocated = solution->allocated;
}
```

Figure 4.11 The pseudo-code for adapting a manageable resource.

With the use of CR in a general purpose operating system, the UME reacts to the full system resource utilization point; thus, at any given time, eliminating the risk of running into a contention state. However, **a CR can only enable contention avoidance to the point where the UME will no longer scale back the manageable resources** (i.e. it is not allowed to scale back beyond the reserved resources, in order to obey admission contracts in force). In such a case, we have reached the limit of the reactive capability of the UME.

4.4 Summary

In this chapter, we described the architecture of the Quality Utility Extension, which implements the Utility Model. We described its two key activities: admission control and quality adaptation, as well as some design issues, such as QoS mapping with QoS Agents and session profiles. We also outlined some challenges in resource monitoring and presented solutions in the context of a best-effort general purpose operating system (GPOS), such as UNIX™ or Windows™, which was not designed to support QoS guarantees and thus is fundamentally unsuitable for applications which demand them. It is important to note, however, that implementing the QUE in a GPOS can nevertheless be beneficial, although it lacks the resource reservation and policing mechanisms which the Utility Model demands. This is because a best-effort GPOS, such as UNIX™ and Windows™ plus the Internet,

1. is currently the most widely available and the most widely used operating system,
2. allows us to access the most multimedia research tools and applications, and
3. allows us to examine the effectiveness of reactive adaptation in the QUE.

In an operating system that supports resource reservation (such as RT-Mach plus ATM based networks), we can use the UME to manage the entirety of system resources and we can enforce its adaptation decisions.

Finally, in the case of hardware error or reconfiguration, adaptation of manageable resource enables a certain degree of fault tolerance, regardless of the operating system and network used.

5 Implementation

In the previous chapter, we described the Quality Utility Extension (QUE) architecture based on the Utility Model, as well as ideas such as QoS mappings, resource monitoring, and contention avoidance. In this chapter, we present some implementation details of the Utility Model Engine (UME), a Video-on-Demand (VoD) application, and a rate-controllable file-transfer application (QFTP).

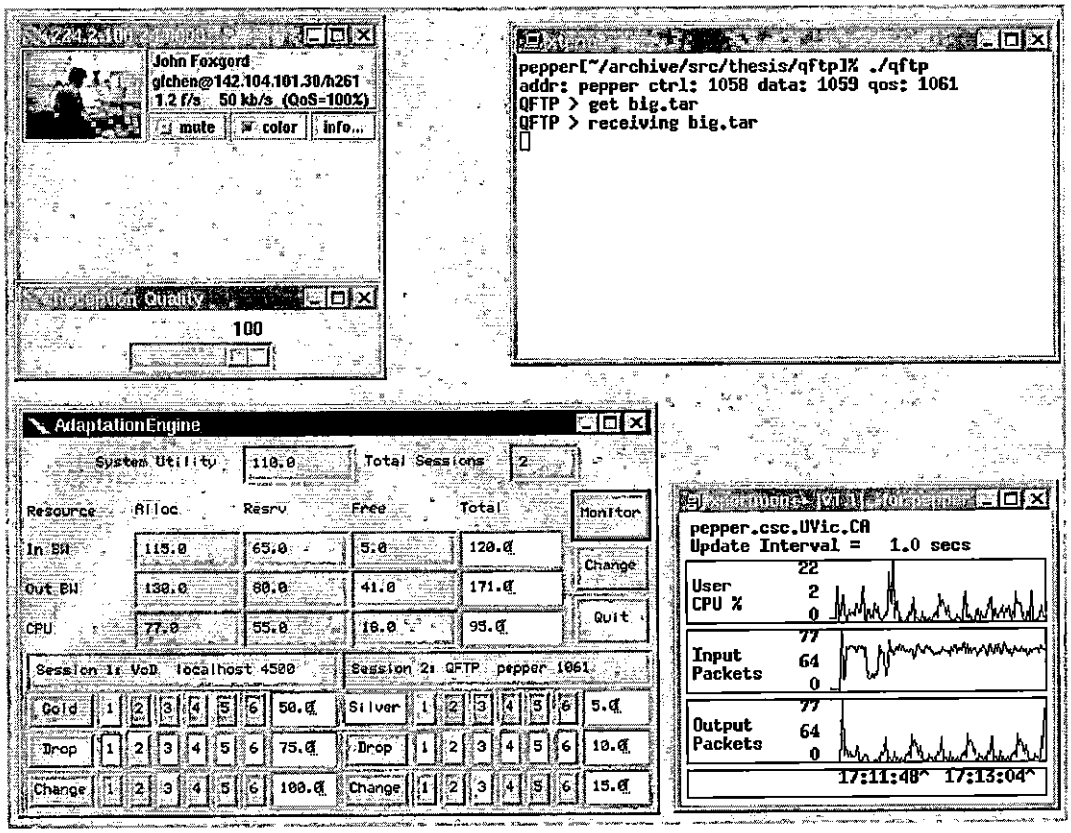

Figure 5.1 A Snapshot of the QUE implementation.

In our prototype system, the QUE is to be used in a receiver (running Linux) to supervise a VoD client and a QFTP client. Figure 5.1 shows a snapshot of the prototype.

5.1 The Utility Model Engine

The Utility Model Engine (UME) consists of two major components: a *Session Directory* (SDR) and an *Adaptation Engine*. The SDR stores session profiles, whereas the Adaptation Engine implements the HEU[1] heuristic. Since our implementation of the QUE is done on top of a Linux operating system, which lacks the *Resource Reservation API*, we were not able to implement the *Reservation Agent*, as specified in the QUE architecture. Furthermore, we were constrained to use the only resource monitoring mechanisms available - the proc pseudo file system. Thus, as a compromise, we choose to implement the *Resource Monitor* as a separate process, which monitors kernel data structures through the proc pseudo file system and sends resource usage measurements to the UME[2].

For the communication with QoS Agents, the Resource Monitor, and the user, the UME implements three interfaces (see Figure 5.2): a *Quality Management Interface* (QMI), a *System Interface* (SI), and a *Graphical User Interface* (GUI). For simplicity, both the QMI and the SI are implemented using sockets with fixed addressees. Communication through the interfaces is implemented by sending and receiving text messages.

---

1. We choose HEU over BBLP because HEU runs faster and has more predictable performance.
2. In this way, our UME implementation can be easily ported to other general purpose operating systems (such as Solaris™ and Windows™, both of which have tools for collecting resource usage statistics, and thus can be reused).

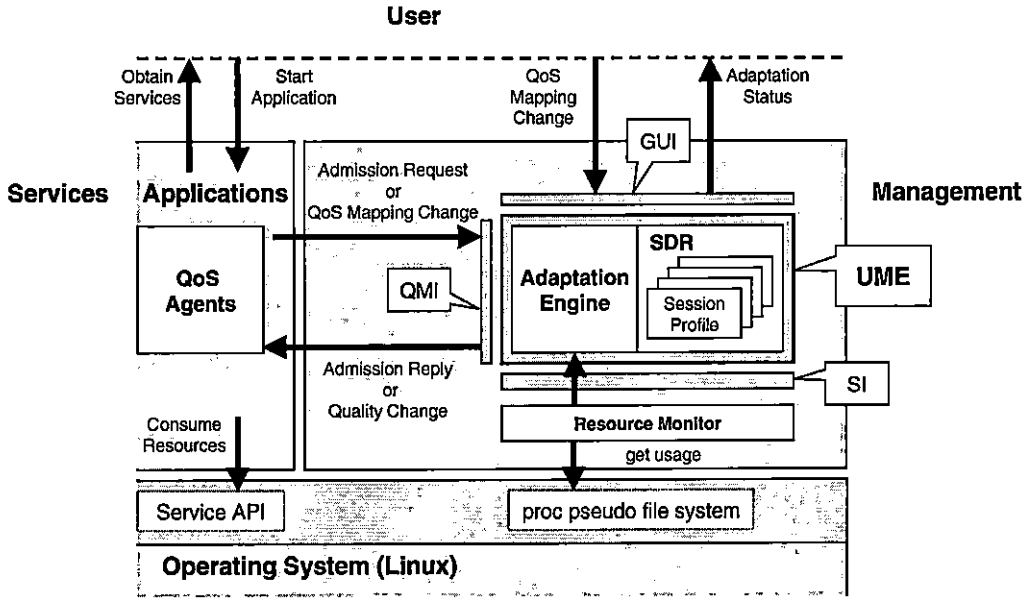

Figure 5.2 The Utility Model Engine and its interfaces.

The *Quality Management Interface* accepts messages such as *new*, *drop*, and *change*. Each message contains a request type, a *Session ID* (SID), and a message body. A SID consists of the name of the application's QoS Agent and the socket address on which it is listening. A session profile must be sent in the message body with each *new* and *change* message.

As described in the previous chapter, a QoS Agent for each application maintains the *quality-method-resource mapping* and handles the communication with the UME on behalf of the application, which include *admission control* and *quality adaptation* (see Figure 5.3). After an application has registered its QoS scaling Methods and their associated resource requirements, such as

"`register (set_video_fps, 30, (inBW) 20, (outBW) 10, (CPU) 50)`",

its QoS Agent establishes a quality-utility mapping, a quality-method mapping, and a

method-resource mapping[3]. For simplicity, we assume that quality levels are zero-based integers, and that a negative integer value indicates a rejection (or termination) by the UME. For the use of *new* and *change* messages, a session profile is constructed by each QoS Agent, which contains the quality-utility mapping and quality-methods-resources mappings. Figure 5.4 illustrates an example of a *new* message. After admission control or quality adaptation action by the UME, a message (containing an integer value) is sent back to each QoS Agent to either reject a new request or inform it of the operating quality of the application.

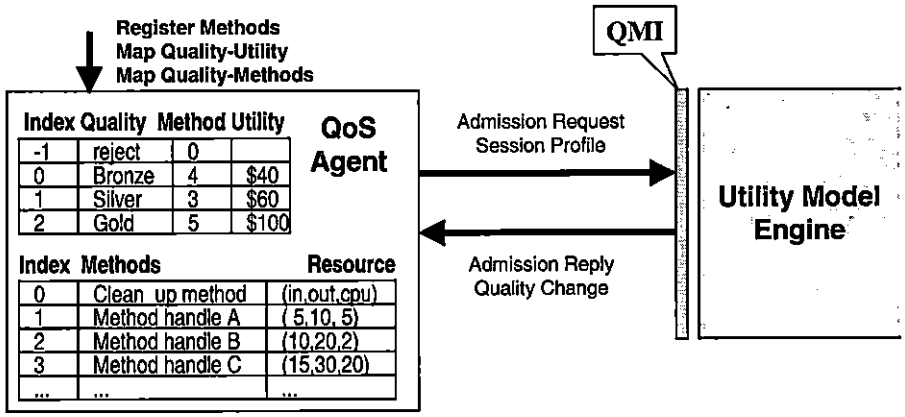

Figure 5.3 QoS mapping, QoS Agent, and QMI.

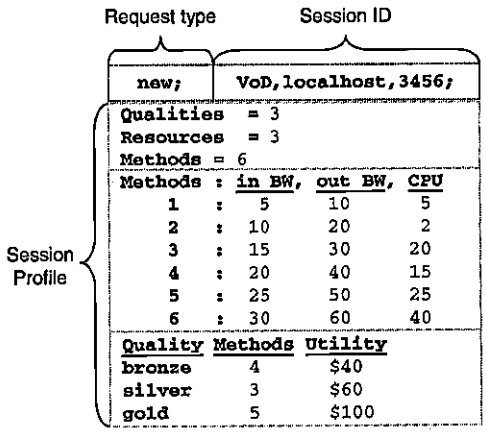

Figure 5.4 An example of a Quality Management Interface message.

3. This is done either by default mappings or by explicit mapping calls from the application.

Through the *System Interface*, the Adaptation Engine receives system resource utilization measurements. The system resource types include CPU (in percentage of total cycles per second), inbound bandwidth (in packets per second or pps), and outbound bandwidth (pps). For the purpose of resource monitoring, we modified a publicly available system performance monitor, xperfmon++[4]. It samples resource usage every second and forwards the reading to the system interface[5] (see Figure 5.5).

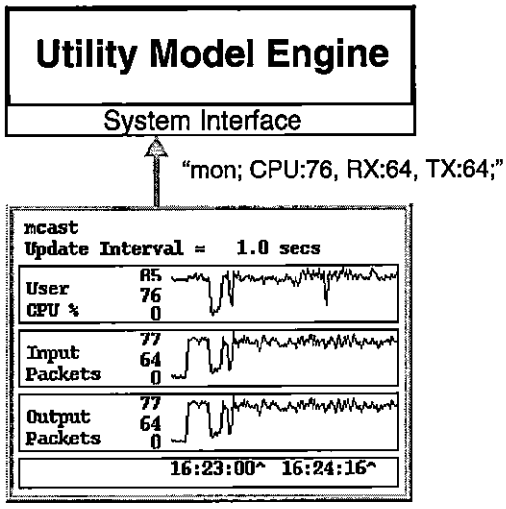

Figure 5.5 System resource monitor.

The *Graphical User Interface* drives a graphical display showing the status of the Utility Model Engine, such as total system utility, total available resource, reserved resource, allocated resource, and free resource. Applications' session profiles are also displayed, allowing a user to change quality-utility mappings and observe quality

4. Xperfmon++ is a program written by Roger Smith at Sterling Software, NASA-Ames Research Center, Moffett Field, California. It is made publicly available as ftp:// sunsite.unc.edu/pub/ Linux/X11/xutils/status/xperfmon++.tar.gz
5. We choose to do so for two reasons: xperfmon++ readily provides the code to read (and parse) the files in */proc*; and the QUE can be easily ported to another platform (e.g. Windows™) by replacing xperfmon++ with another resource monitoring tool (e.g. the *Performance Monitor* in Windows™).

adaptation. Here, as illustrated in Figure 5.6, the UME is supervising two sessions. Each session provides six QoS scaling methods, of which three are used in QoS mappings. The "quality" button in each session interface is color coded to show the current operating quality: gold, silver, or bronze. When an application starts, it sends an admission request through the QMI, along with a quality-utility mapping and QoS mappings in its session profile. After it is admitted, its session profile is displayed in this interface, where a user is able to change, at run-time:

1. how each quality is mapped to a QoS specification which specifies the corresponding resource requirements, and
2. how each quality is assigned a utility value (dollars) which conveys the user's preference for that quality.

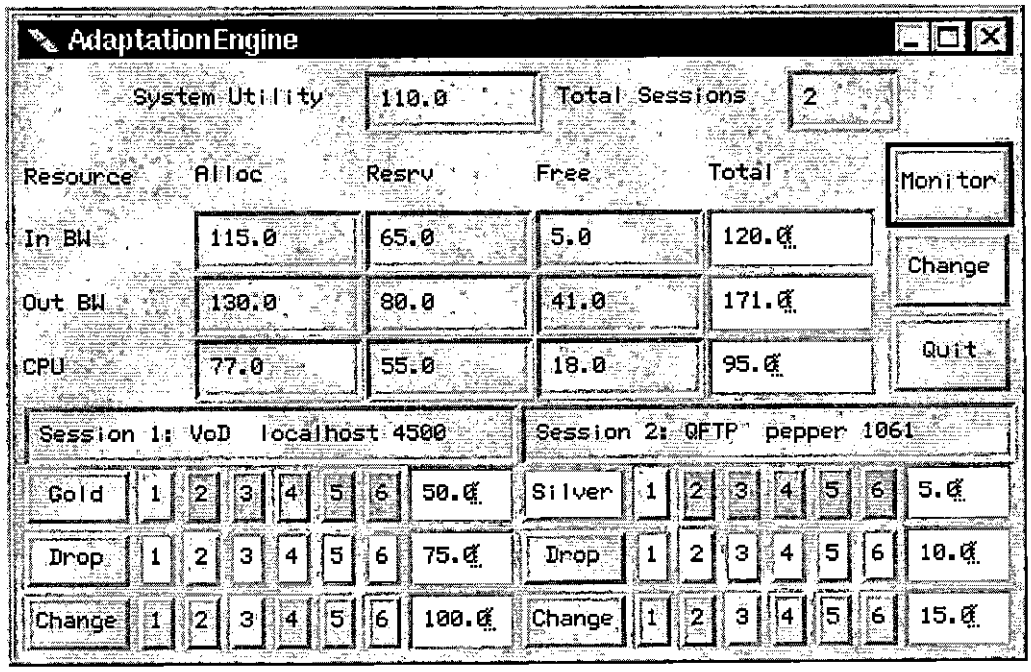

Figure 5.6 The graphical user interface of the Utility Model Engine.

5.2 VoD: Video Transport with LCT

VoD transports live video over the Internet from a server to a client (see Figure 5.7). It uses *layered coding and transmission* (LCT) in order to deliver multi-quality video streams. At the receiver system, where the QUE is implemented, a VoD client uses a QoS Agent to select an appropriate video reception quality, adhering to the adaptation decisions of the UME.

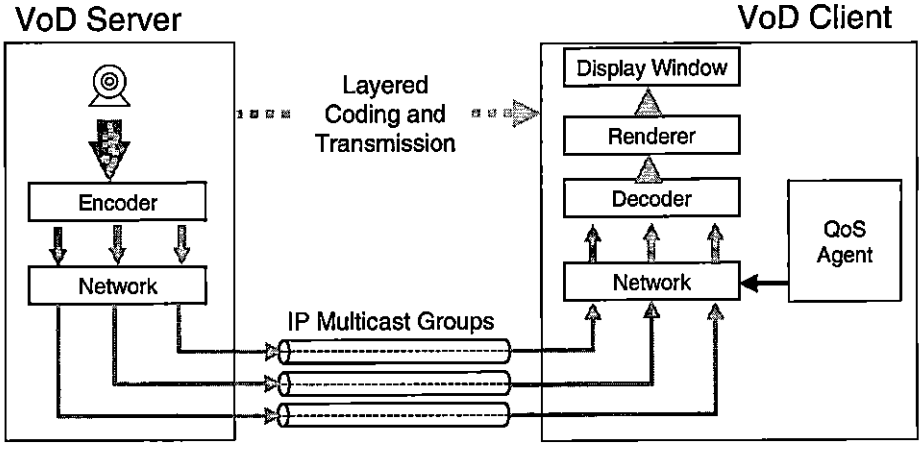

Figure 5.7 VoD: Layered video coding and transport.

A VoD server receives image frames from a Color Quickcam and encodes them with an H.261 codec. As illustrated in Figure 5.8, a simple temporal layering is used to vary the frame rates of the video transmission, which includes a base layer (B), enhancement layer one (E1), and enhancement layer two (E2). To achieve layered transmission, each layer is transmitted over a distinct IP multicast group; the client receives the video layers by joining the multicast groups. To simulate the effect of multicast branch pruning (which reduces unwanted network traffic), the layer dropped by the client is suppressed by the server.

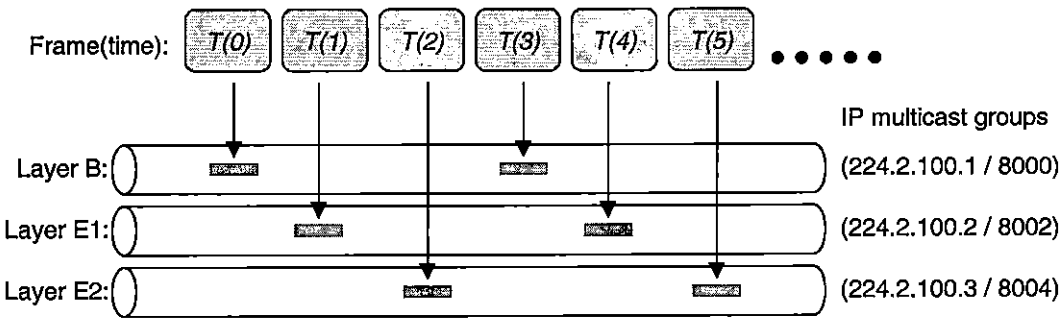

Figure 5.8 Temporal layering in VoD.

A VoD client provides multi-quality video reception by QoS scaling methods which add and drop network layers. These methods are registered with its QoS agent, which maintains its session profile and handles its quality changes according to the UME's adaptation decisions. In Figure 5.9, we show an example of the construction of a session profile, assuming three layers. In the session profile, the quality-utility mapping of video reception is configured statically (by some default); the method-resources mapping is obtained by profiling[6] the client on the target platform.

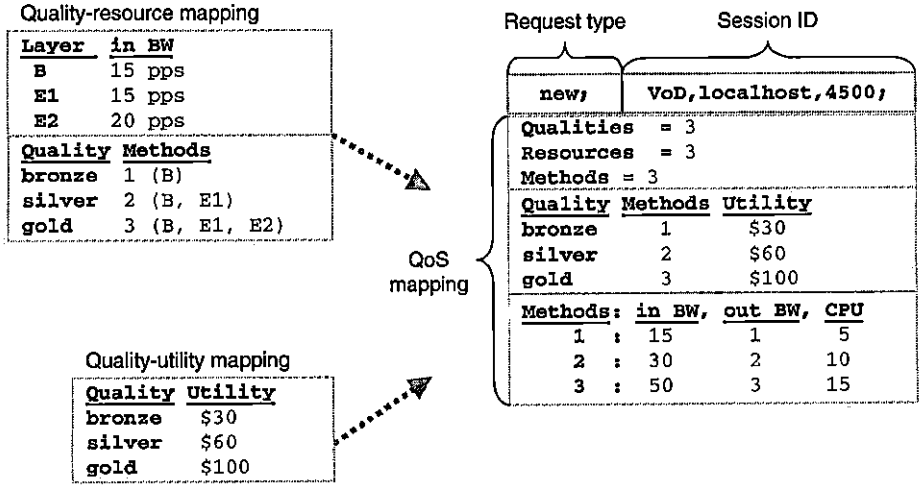

Figure 5.9 The session profile of VoD.

6. This is done by enabling each individual QoS scaling method while observing the resource utilization from a resource monitor (e.g. xperfmon++) in an unloaded system.

5.3 QFTP: Rate-Controllable File Transfer

In the current Internet, file transfer is mostly implemented with TCP. However, it is difficult to control bandwidth consumption under TCP[7]. Therefore, we developed a simple *rate-controllable file transfer program* (QFTP). The QFTP client is to be used in a receiver where the QUE is used to mange system resources and applications' qualities[8].

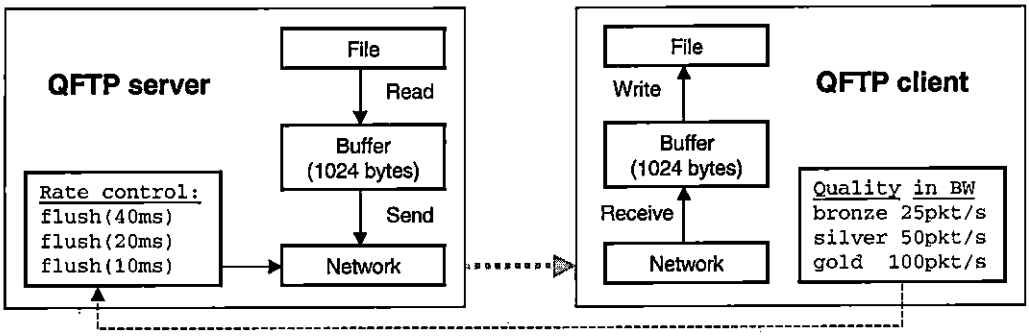

Figure 5.10 QFTP: rate-controllable file transfer.

In QFTP, the server controls the sending rate, but the client controls the rate at which the server sends. As seen in Figure 5.10, the server may flush and send the contents of the send buffer every 10, 20, and 40 milliseconds. Each flush causes a packet to be sent to the receiver. Therefore, the inbound bandwidth requirements at the receiver are one of 100, 50, or 25 packets per second. Figure 5.11 shows an example of the construction of a session profile for a QFTP client.

---

7. TCP's slow-start congestion avoidance algorithm[5] is very aggressive in maximizing the transmission rate, and it adapts only to packet loss. Therefore, it is difficult to control its behavior without substantial modification to its implementation.
8. The remote machine, which runs the QFTP server, can also implement the QUE architecture if desired. In this case, the QFTP server could implement a QoS Agent which controls the sending rate according to its available local system resources.

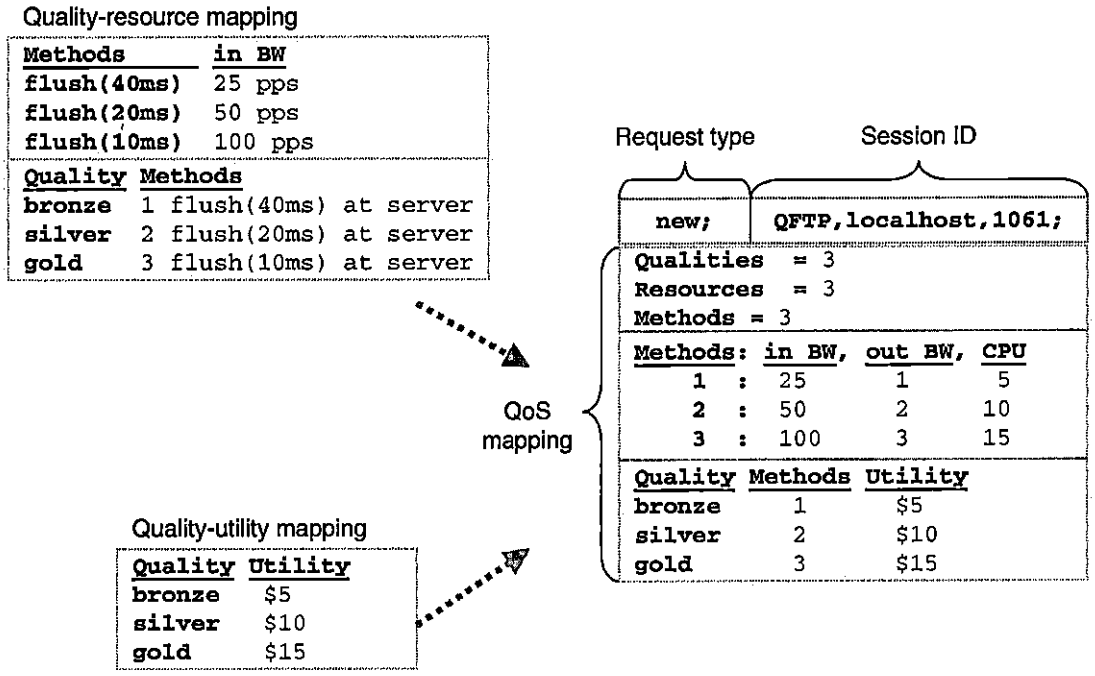

Figure 5.11 The session profile of QFTP.

6 Experiments and Results

The experiments and results given in this chapter validate our implementation of the Quality Utility Extension (QUE) and show the effectiveness of the Utility Model applied to the layer-encoded sources.

The experiments involved two server machines (A and B) and a client machine (C) on a 10 Mbps Ethernet-based LAN (see Figure 6.1). The QUE was implemented in the client machine (C). Machine A runs the Solaris 2.4 operating system; Machine B and C both run the Redhat Linux 2.1 operating system. We assume each machine has (at least) 1.6 Mbps inbound and outbound bandwidth[1], and each ethernet packet is 1024 bytes in size. We further assume 10% bandwidth overhead in transmitting each ethernet packet. Thus, the inbound and outbound bandwidths are each approximately 180 packets/second (pps). Hence, the raw resource configuration for the QUE in machine C is 180 pps inbound, 180 pps outbound, and 100% CPU.

1. These assumptions are necessary for us to simulate a generic Internet topology, and were validated by carefully configuring the experimental equipment.

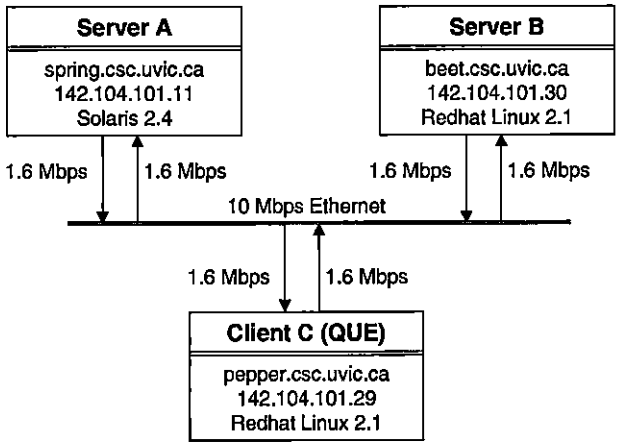

Figure 6.1 Experiment topology.

Our experiments are designed to demonstrate two separate cases: a best-effort mixed system and an entirely reservation based system. In a best-effort mixed system (the Internet is a well known example) not all applications require QoS guarantees and are in the Utility Model (UM) domain. Applications which do not conform to the QUE may cause resource contention. When this happens, we want the Utility Model to scale back the quality of the applications in its domain. In an entirely reservation based system, on the other hand, all applications must conform to the QUE. Therefore, in this case, the quality adaptation is optimal, as we will show.

To illustrate adaptation in the QUE, we will show how the inbound bandwidth is used over time, according to the classification of resources defined in Chapter 4:

- *manageable* (what is allocatable by the UME),
- *unmanageable* (what is consumed by out-of-domain applications),
- *allocated* (what is required to support the UM-domain applications' current operating qualities),
- *reserved* (what is required by the UM-domain applications' minimally acceptable QoSs), and
- *monitored* (what is actually consumed by all applications in the system).

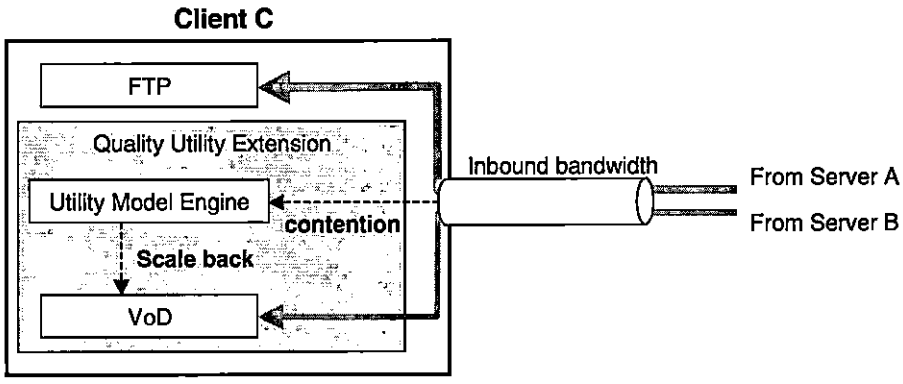

Figure 6.2 A best-effort mixed system.

6.1 Best-effort Mixed System

In the experiment for the best-effort mixed system (see Figure 6.2), we placed an FTP server on machine A, and a VoD server on machine B. At the client C, we first started the VoD client, receiving video from machine B; we then started an FTP connection to machine A, fetching eight files. The events labelled in Figure 6.3 are as follows:

- event A: VoD starts;
- event B: FTP starts;
- events 1-8: FTP fetches eight files;
- event C: FTP terminates.

In VoD, the peak inbound bandwidth requirements for bronze, silver and gold quality are respectively 25, 35 and 60 pps. The inbound bandwidth requirement for the FTP traffic is unknown. We do know, however, that FTP will cause network bandwidth contention because it uses TCP, whose slow-start algorithm exponentially increases its bandwidth consumption until packet loss occurs. Since there are no resource guarantees in the OS, video delivery will be deprived of bandwidth as a result of such contention.

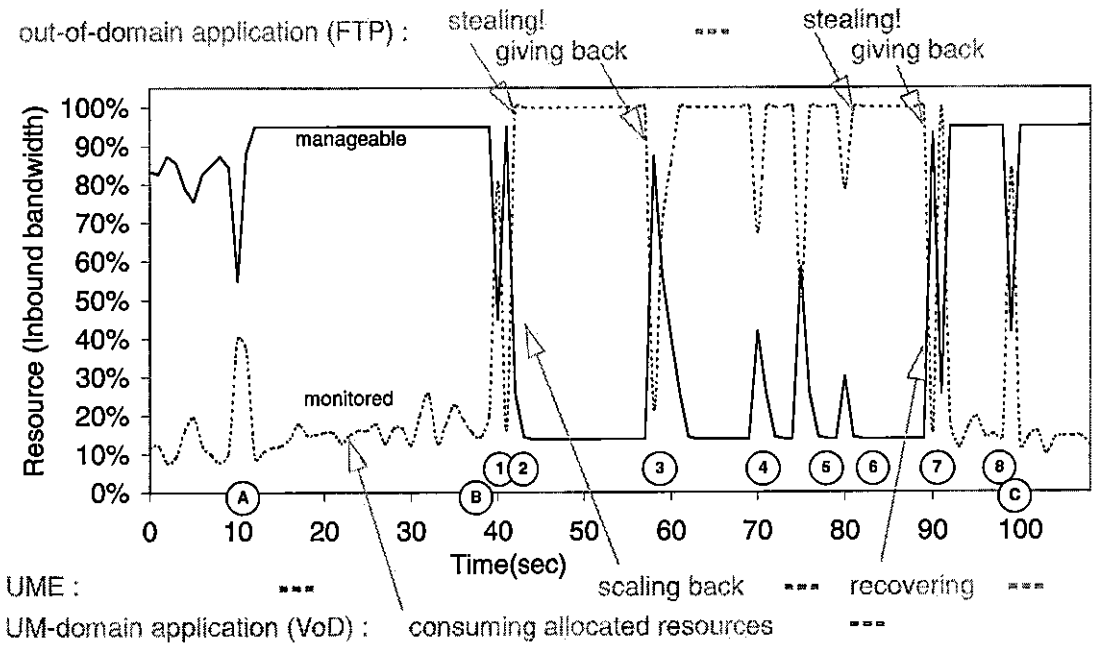

Figure 6.3 Scaling the manageable resource.

Figure 6.3 shows the adaptation events in our experiment and the resulting scaling of manageable resource based on the monitored resource. Initially, 5% of the raw resource is allocated for system use, and the UME manages the remaining 95%. In the figure, the area above the manageable line represents the portion of the resource unmanageable by the UME. We observe that, before event A, the QUE monitored some background network activities. It adjusted the amount of manageable resource, so that resource would not be over-subscribed when admitting new sessions.

At event A[2], when VoD was admitted (not started!), the bandwidth required for its minimal quality (bronze) was reserved, and the bandwidth for its desired quality (gold) was allocated. After event A, as VoD started to receive the video stream, the monitored resource shows the resource consumption of VoD; since no resource contention was

2. Readers may note the surge of monitored resource at event A. This is caused by the loading of the VoD application by the operating system via the Network File System (NFS). Hence, it was treated as out-of-domain activities, forcing the adaptation of the manageable resource.

detected, the amount of manageable resource remained steady.

Now, as soon as FTP started at event B, our resource monitor shows that large amounts of bandwidth were seized by the transferring of the eight files. To avoid contention, the amount of manageable resource was scaled back accordingly by the QUE. Once the FTP terminated at event C, the manageable resource returned to the initial 95%.

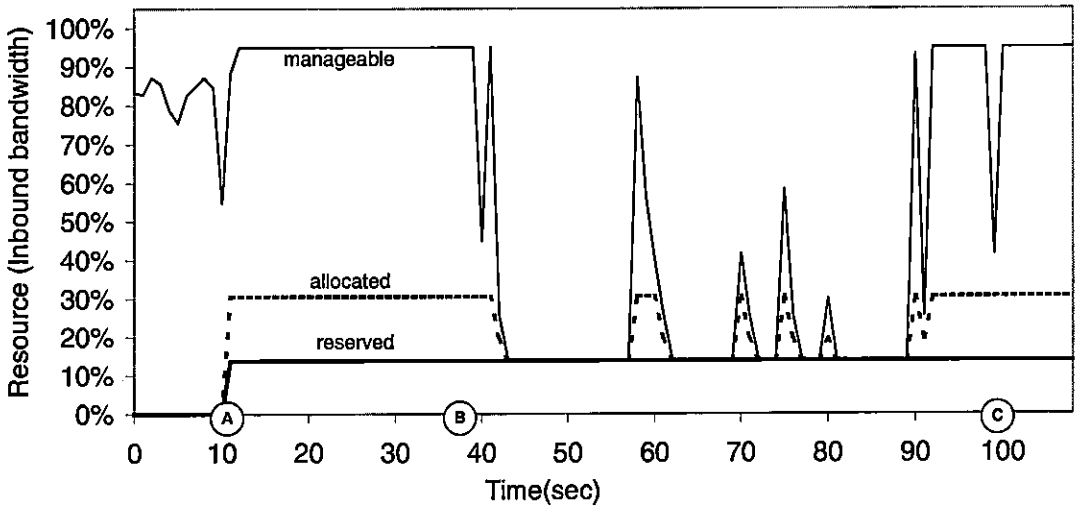

Figure 6.4 Quality adaptations and the scaling of *manageable* resource.

Although the manageable resource was scaled frequently, based on monitored resource usage contention, it did not always trigger quality adaptation. When the amount of manageable resource was reduced beyond the allocated resource, quality adaptation took place. This is illustrated in Figure 6.4. We see that the manageable resource was reduced ten times, but the allocated resource for VoD was scaled back six times during bandwidth contention avoidance. When the danger of contention disappears, the resource management policy (maximizing system revenue) in the QUE restored the resource allocation for VoD. It is also evident that the amount of reserved resource is fixed since the admission of VoD, and that the quality adaptation did not decrease the manageable resource beyond the reserved resource. Therefore, the admission contract was upheld by

the QUE, and application QoS constraints were respected.

Overall, as seen in Figure 6.5, the QUE obeys both the system resource constraints and the application resource constraints in a mixed best-effort environment: it dynamically adjusts the amount of manageable resource and adapts the application's operating quality to avoid resource usage contention, and it reserves the resources required to support the application's minimal quality.

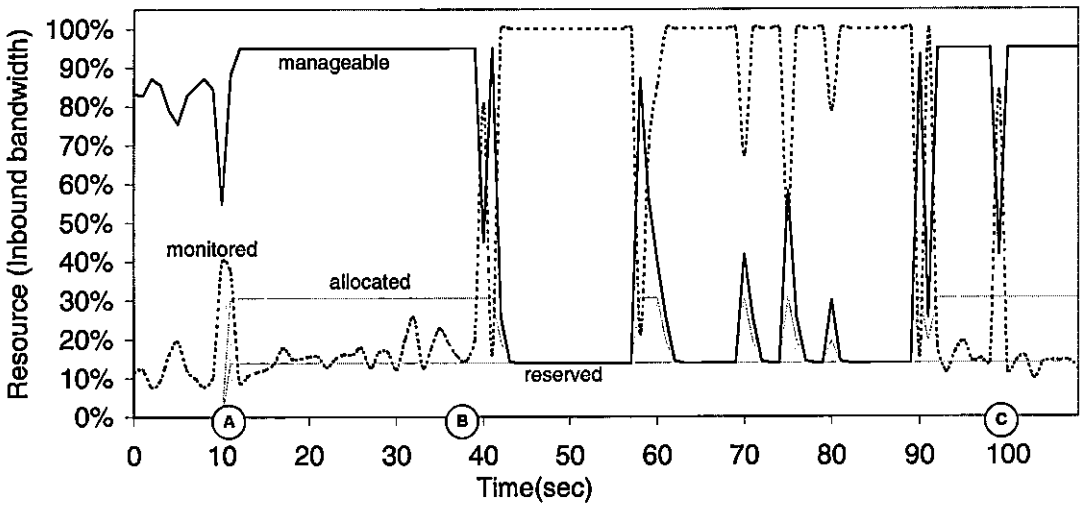

Figure 6.5 Quality adaptation for a best-effort mixed system.

Between events A and B, we see the admission control of the QUE and the resulting resource reservation and allocation. The monitored resource usage illustrates that such reserved and allocated resources are both necessary and sufficient. Between events B and C, we see the interplay of two forces: the contention avoidance mechanism to scale back the manageable resource and the revenue maximization policy to drive up applications' operating qualities. At points where the QUE no longer scales back the manageable resource (in order to respect application constraints), resource contention is possible if the underlying OS violates any of the application constraints.

6.2 Reservation Based System

In an entirely reservation based system, the QUE is to control the use of system resources (95% of the raw resources). In our experiment, we placed a QFTP server at machine A and a VoD server at Machine B. Client C implements the QUE, which supervises a QFTP client and a VoD client. As depicted in Figure 6.6, we allocated more money to the QoS levels of VoD to indicate our quality preference between the two client applications. During the experiment, we changed our quality preference by giving $150 to QFTP's gold quality.

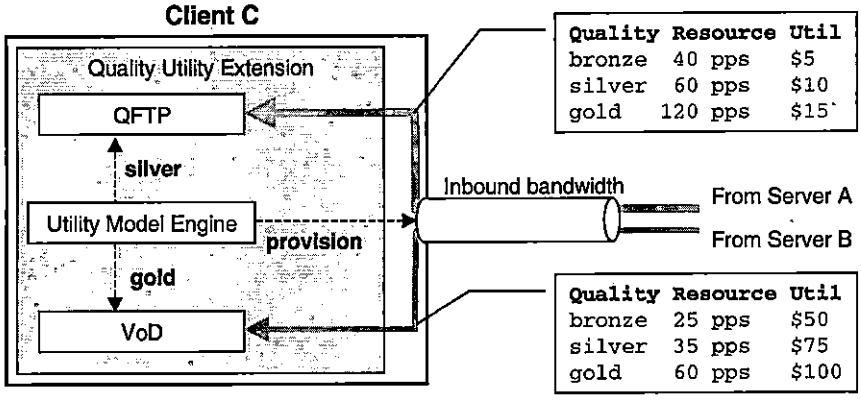

Figure 6.6 An entirely reservation based system.

In the subsequent figures, we label the following events:

- event A: VoD starts;
- event B: QFTP starts;
- event C: the utility of the gold quality is raised to $150 for QFTP;
- event D: QFTP finishes.

As shown in Figure 6.7, resource usage is well regulated, since all applications conform to the QUE. There is no need to scale back the manageable resource[3]. At events

A and B, where VoD and QFTP are admitted, the resource required by each application's minimal quality (bronze) is reserved, and the resource required by each application's actual quality is allocated.

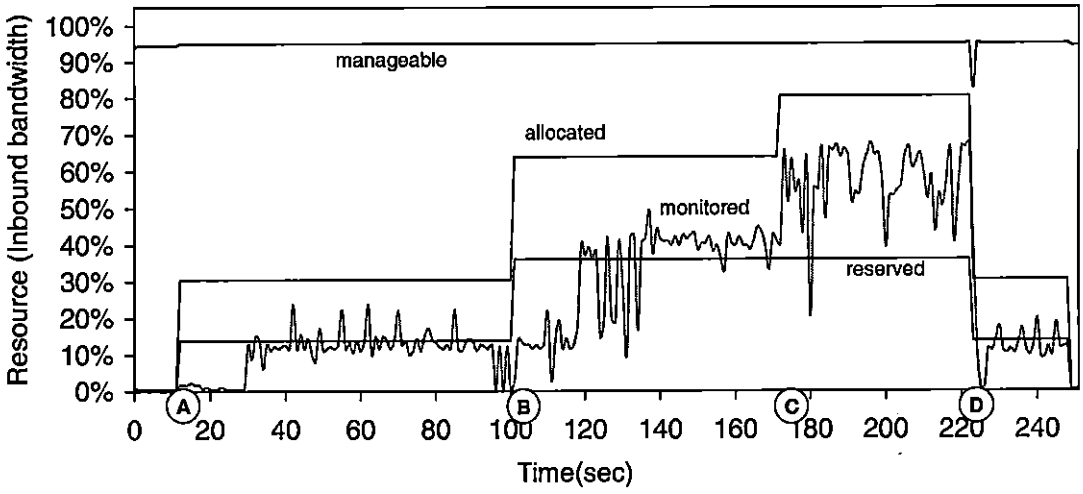

Figure 6.7 Quality adaptation for an entirely reservation based system.

VoD starts at the gold quality. The QUE reserves 25pps for its minimal quality, and allocates 60pps for its desired (gold) quality. The monitored resource between events A and B shows that the reserved resources are entirely consumed by VoD, and that the allocated resource value is large enough to accommodate its the peak bandwidth requirement.

QFTP runs at the silver quality, since there is not enough bandwidth to support both applications' gold qualities at the same time. The QUE reserved 40pps for QFTP's bronze, and allocated 60pps for its silver. From the monitored resource, we see that the aggregate of the reserved resources is entirely consumed, and that the aggregate of peak bandwidth

3. Note that at event D, the *manageable* resource was reduced for a moment. This is transient error which occurred when QFTP terminated, but its last resource consumption reading was still in the resource monitor and was thus treated as being consumed by out-of-domain activities.

requirements of both applications is accommodated.

At event C, the utility value for the gold quality in QFTP is raised from $15 to $150, resulting in a quality adaptation, which increases the quality of QFTP to gold, but decreases the quality of VoD to silver. Finally, when QFTP finishes at event D, VoD's quality is restored to gold. It is also evident that the above quality adaptation is reflected in the actual resource usage, as illustrated by the monitored resource line in the figure.

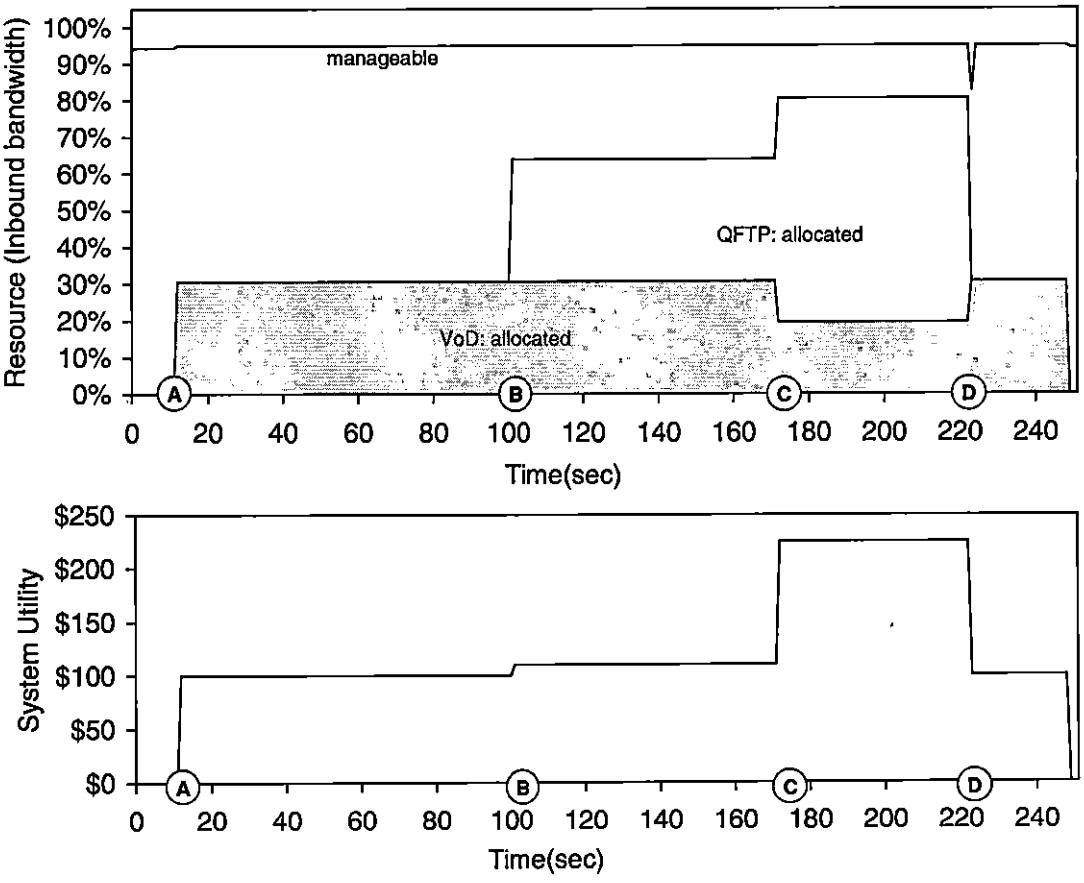

Figure 6.8 Optimal quality adaptation in the QUE.

More important, since the QUE is an implementation of the Utility Model, each quality adaptation is optimal. To illustrate this point, Figure 6.8 shows the bandwidth allocation and the total system utility resulting from each adaptation. We observe that

there is not enough bandwidth to support both applications' gold quality simultaneously, but VoD offered more money for the gold quality than QFTP did. So when QFTP starts at event B, it is only allowed to run at silver quality, and the allocated resource for VoD is unchanged. However, as soon as the QFTP offered $150 for its gold quality (at event C), more bandwidth is allocated for it, and its quality upgraded to gold. The operating quality for VoD, on the other hand, is degraded to silver. All the while, the total system utility is increasing. When the QFTP finished execution at event D, the operating quality of VoD is restored to gold.

6.3 Performance

Our implementation of the QUE achieves good response time during admission control and quality adaptation. To process a message or event, the UME first parses the message contents, it then performs quality adaptation using the HEU heuristic; the result is sent to the applications under its control, and the adaptation details are displayed in the GUI. For the timing of the UME's event processing, we measure the total time (in milliseconds) required (from message arrival to the display of adaptation results in the GUI). We obtained some measurements using a Pentium 166MMX with 40MB memory. Assuming that a typical multimedia system will engage in 3 to 12 multimedia sessions, our measurements showed that, on average, it takes about 300 milliseconds to admit a new session and less than 2 milliseconds to perform a quality adaptation (see Table 6.1)[4]. The cost of admission control in the UME is mainly the creation of a session profile and its

4. Readers may notice that the average adaptation time for 12 sessions is less than all other cases. This is because there is less room for adaptation (hence less computation, due to the specific heuristic used), as the system becomes heavily loaded (i.e. approaching full utilization of system resources).

associated GUI objects. In particular, the GUI in the UME is the main bottleneck, which could be corrected in future implementations.

	quality adaptation			admission control		
	min	max	avg	min	max	avg
3 sessions	0	18	1.09	288	292	290.33
5 sessions	0	9	1.21	187	292	218.20
8 sessions	0	45	1.54	292	319	301.75
12 sessions	0	11	1.07	293	327	307.50

Table 6.1 Quality adaptation and admission control (in milliseconds).

6.4 Summary

In this chapter, we described two experiments demonstrating the QUE: a best-effort mixed system and an entirely reservation based system. The results of our experiments showed that,

- in the best-effort mixed system, where applications outside of the Utility Model domain contend for resources, the QUE is able to avoid resource contention by adapting the amounts of manageable resources and scaling back the qualities of the applications in the Utility Model domain;
- in the entirely reservation based system, where all applications conform to the Utility Model, their qualities are optimally adapted, so that total system utility is maximized.

In both cases, application resource constraints are respected, and system resource constraints are not violated.

Although we have demonstrated only two very simple cases of the use of the QUE, the effectiveness shown by their results suggests that the Utility Model will perform as well in

practice as in theory. Given well-defined system resource constraints and reasonably accurate session profiles from applications, the QUE is able to perform according to the Utility Model.

7 Conclusion

In this thesis, we outlined how the Utility Model can be applied to layered coding and transmission for solving the local network bandwidth contention problem during real-time multimedia transport. This is the first time that the Utility Model has been taken from a purely theoretical model to a real system implementation. Furthermore, our approach bridges two important research topics in distributed multimedia systems - adaptive multimedia transport and multimedia system modeling. Below, we summarize our contributions and suggest future work.

7.1 Contributions

In addition to showing how the Utility Model can be applied to layer-coded sources, our contributions in this thesis include a Quality Utility Extension (QUE) implementing the Utility Model, QoS mapping with QoS Agents, system resource monitoring, and contention avoidance.

7.1.1 The application of Utility Model to layer-coded sources

We based our research on well-established concepts of scalable media coding and layered transmission. The Utility Model is then used at a receiver system to choose the right

number of layers to receive, depending on available system resources. This gives us the capability of varying the resource requirements for real-time media reception, so as to avoid contention in the local system. More important, as an inherent benefit of the Utility Model, we are able to **optimally** adapt operating qualities for all applications in the Utility Model domain, maximizing total system utility.

7.1.2 The Quality Utility Extension

The Quality Utility Extension (QUE), implementing the Utility Model, is presented as a management extension to a classical (not to say obsolescent) operating system. In the QUE, a Utility Model Engine (UME) dynamically manages both applications' operating qualities and available system resources. In order to respect both applications' QoS constraints and system resource constraints, the QUE uses admission control to guarantee applications' minimal acceptable QoSs and quality adaptation to dynamically adjust applications' desirable QoSs.

7.1.3 QoS Mapping and QoS Agents

In the QUE, mappings from user-level QoSs to application-level QoSs and system-level QoSs are done by each application using quality-methods mappings and method-resources mappings. These mappings are encapsulated in a session profile, which is used by the UME during admission control and quality adaptation. To simplify the design of applications in the QUE, we presented an implementation reference model of an application-level proxy, the QoS Agent, to handle QoS mappings and the interactions with the UME. QoS Agents maintain applications' quality-utility mappings and QoS mappings (quality-methods mappings and method-resources mappings), generate session profiles

and admission requests, and execute the quality adaptation decisions made by the UME. Thus, none of this needs to be implemented as part of the application.

7.1.4 System resource monitoring

Observing that existing, obsolescent operating systems (like UNIX™ and Windows™) fall far short of meeting the requirements for resource monitoring in the QUE, we showed how system resources (CPU, inbound and outbound bandwidth) can be monitored using available mechanisms in our implementation platform - a Linux operating system.

7.1.5 Contention avoidance

We demonstrated how resource contention can be avoided by the UME, using resource monitoring and quality adaptation. First, the usage of a resource is broken down into *manageable* and *unmanageable* parts. Within the manageable resource, the UME optimally allocates resources among applications in its domain, free of any danger of contention. Second, the UME adapts the manageable resource based on the monitored usage of a resource. Therefore, when there is an increase of unmanageable resource, the UME eliminates the danger of contention by scaling back the manageable resource accordingly. Finally, as the usage of a resource approaches the full capacity of the hardware, the UME avoids the imminent resource contention by using a Cushioning Region. This causes the manageable resource to be reduced beyond the allocated resource, so that the quality adaptation in the QUE will scale back the qualities (and thus the resource requirements) of the applications in its domain.

We also pointed out that such contention avoidance is not without limits - the manageable values of resources must not be scaled back beyond their reserved values.

7.1.6 Implementation and results

To demonstrate our approach, we implemented the QUE above a Linux operating system. We also demonstrated two scenarios: reactive adaptation in a best-effort mixed system and proactive adaptation in a entirely reservation-based system.

In a best-effort mixed system, only some of the applications conform to the QUE. Our demonstration included a video-on-demand application, VoD, (supervised by the QUE), and a competing FTP session. We showed that the QUE avoided resource contention by scaling back the qualities of application(s) in its domain (i.e. VoD).

In an entirely reservation-based system, every application is in the UM domain. Our demonstration included the same video-on-demand application, VoD, and a rate-scalable file transfer application, QFTP. We showed that the QUE optimally adapts both session qualities, free of any resource contention, while respecting system resource constraints, application resource constraints, and user preferences.

We also showed that the performance of our prototype is reasonable for real-time multimedia adaptation, and mentioned that it can be further improved.

7.2 Future Work

As mentioned, this thesis presented the first attempt at implementing the Utility Model in a real multimedia system, and we have just scratched the surface of the richness of interactions allowed by the Utility Model. Below, we list some areas where further implementation and experiments are needed.

7.2.1 The QUE and reservation-capable operating systems

Reservation-capable operating systems, such as RT-Mach, are better in meeting the requirements of implementing the QUE. However, how to interface the QUE with the reservation mechanisms in such systems is a challenge.

7.2.2 Improvements in a best-effort operating system

It has been shown in this thesis that reactive adaptation to available system resources is possible with the QUE in a traditional best-effort general purpose operating system, such as UNIX™. However, this requires accurate resource monitoring at the system level. Therefore, defining and implementing a resource monitoring API for a general purpose operating system will be worthwhile.

Recent work (such as RSVP[32], CBQ[7], and so on) has potentially given reservation capabilities to UNIX™ systems and the Internet, resulting in a mixed (reservation and best-effort).Internet. Studying how the QUE will work in such environment can be extremely beneficial.

7.2.3 Exploring QoS Agents

The QoS Agents presented in this thesis contain only static QoS mappings. As indicated in the reference implementation model, a QoS Agent can posses much more functionality, such as application-level QoS monitoring, run-time profiling (i.e. per-application resource usage monitoring to give more accurate method-resources mapping at run-time), auto-bidding (i.e. intelligent quality-utility mapping), and so on. The QoS Agent reference model also imposes a degree of discipline in designing and programming applications with multiple levels of QoS. Thus exploring how QoS Agents can be used with different

types of applications, and what levels of intelligence they should possess, will be very interesting.

7.2.4 The deployment of the QUE

Finally, we envision the QUE in use with every system where the Utility Model can be applied. For instance, a server and a client can each use the QUE to manage their own local resources. In this case, understanding how local policies and adaptation decisions may affect the end-to-end QoS is very important. Furthermore, the quality-utility mapping could provide service providers with a metric that can influence the actual billing of services provided.

References

[1] Brown, T. B.; Cantrell, P. E. and Gibson, J. D. "Multicast Layered Video Teleconferencing: Overcoming Bandwidth Heterogeneity," Department of Electrical Engineering, Texas A&M University, http://www-mcnl.tamu.edu/telcomaustin.ps.

[2] Burt, P. J. and Adelson, E. H. The laplacian pyramid as a compact image code. *IEEE Transactions on Communications*, COM-31(4):532-540, April 1983.

[3] Cheung, S. Y.; Ammar, M. H.and Li, X. "Using Destination Set Grouping to Improve Fairness in Multicast Video Distribution," in *Proceedings of INFOCOM '96, IEEE*, March 1996.

[4] Comer, D. E. *Internetworking with TCP/IP Volume I: Principles, Protocols, and Architecture*, 2nd Ed., Prentice-Hall, 1991.

[5] Comer, D. E and Stevens D. L. *Internetworking with TCP/IP Volume II: Design, Implementation, and Internals*, 2nd Ed., Prentice-Hall, 1991.

[6] Deering, S., "Host Extensions for IP Multicasting", STD 5, RFC 1112, Stanford University, August 1989.

[7] Floyd, S. and Jacobson, V. "Link-sharing and Resource Management Models for Packet Networks." *IEEE/ACM Transactions on Networking*, Vol. 3 No. 4, pp.365-386, August 1995.

[8] Gonzales, C. A.; Viscito, E. and McCarthy, T. "Scalable Motion-Compensated Transform Coding of Motion Video: A Proposal for the ISO/MPEG-2 Standard." Research Report, IBM T.J. Watson Research Center, 1991.

[9] ISO/IEC IS 10918-1 "The JPEG Still Picture Compression Standard".

[10] Karlsson, G. "Asynchronous transfer of video." *IEEE Communications Magazine*, 34(8), August 1996.

[11] Kientzle, T. *A Programmer's Guide to Sound.* Addison-Wesley Developers Press, 1997.

[12] Kumar, Vinay, *MBone: Interactive Multimedia On The Internet*, Macmillan Publishing, November 1995.

[13] LeGall, D. "MPEG: A Video Compression Standard for Multimedia Applications." *Communications of the ACM*, Vol. 34, No 4, p47-58, April 1991.

[14] Li, X.and Ammar, M. H. "Bandwidth Control for Replicated-Stream Multicast Video Distribution." *Proceeding of the FIFTH IEEE INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE DISTRIBUTED COMPUTING (HPDC-5)*, Syracuse, NY, August 1996.

[15] Li, X.; Sanjoy, P,; Pancha, P. and Ammar, M. "Layered Video Multicast with Retransmission (LVMR): Evaluation of Error Recovery Schemes." *Proceeding of NOSSDAV 97*, St Louis, May 1997.

[16] Luther, A. C. *Digital Video in the PC Environment.* McGraw-Hill, 1991.

[17] Khan, S. *"Quality Adaptation in a Multisession Multimedia System: Model, Algorithms and Architecture.*" Ph.D thesis, University of Victoria, 1998.

[18] McCanne, S. "Scalable Compression and Transmission of Internet Multicast Video." Ph.D thesis, University of California, Berkley, 1996.

[19] McCanne, S., Jacobson, V., and Vetterli, M. "Receiver-driven Layered Multicast" *ACM SIGCOMM '96*, Stanford, CA. August 1996.

[20] McCanne, S. and Jacobson, V. "vic: A Flexible Framework Framework for Packet Video." *ACM Multimedia '95*, November 1995, San Francisco, CA, pp. 511-522.

[21] McCanne, S. et al., "Toward a Common Infrastructure for Multimedia-Networking Middleware," In *Proceedings of the 7th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV 97)*, St. Louis, Missouri (May 1997).

[22] McCanne, S; Vetterli, M. and Jacobson, V. "Low-complexity Video Coding for Receiver-driven Layered Multicast." *IEEE Journal on Selected Areas in Communications*, vol. 16, no. 6, pp. 983-1001, August 1997.

[23] McDysan, D.E. and Spohn, D.L. *ATM: Theory and Application,* McGraw-Hill, 1995.

[24] Mercer, C. W. and Rajkumar, Ragunathan. "An Interactive Interface and RT-Mach Support for Monitoring and Controlling Resource Management." In *Proceedings*

of the Real-Time Technology and Applications Symposium. May 1995.

[25] "RTP usage with Layered Multimedia Streams", Internet Draft, ftp://ftp.ietf.org/ internet-drafts/draft-speer-avt-layered-video-02.txt.

[26] Schulzrinne, H.; Casner, S.; Frederick, R. and Jacobson, V. "RTP: A Transport Protocol for Real-Time Applications", STD 1, RFC 1889, GMD Fokus/ Precept Software, Inc./ Xerox PARC/LBNL, January 1996.

[27] Steinmetz, R. and Nahrstedt, K. *Multimedia: Computing, Communications, and Applications.* Prentice Hall, 1995.

[28] Taubman, D. and Zakhor, A. Multi-rate 3-D subband coding of Video. *IEEE Transactions on Image Processing*, 3(5):572-588, September 1994.

[29] Vetterli, M. "Multidimensional subband coding: Some theory and algorithms." *Signal Processing*, 6(2):97-112, Februrary 1984.

[30] Video Codec for Audiovisual Services at px64 kbit/s. Recommendation H.261," CCITT Geneva, 1990.

[31] Waitzman, D.; Partridge, C. and Deering, S. Distance Vector Multicast Routing Protocol. ARPANET Working Group Requests for Comment, DDN Network Information Center, SRI International, Menlo Park, CA, November 1988. RFC-1075.

[32] Zhang, L.; Deering, S.; Estrin, D.; Shenker, S. and Zappala, D. "RSVP: A New Resource ReSerVation Protocol", *IEEE Network*, September 1993.

Appendices

A Linux Programmer's Manual for /proc

NAME

/proc - process information pseudo-filesystem

DESCRIPTION

/proc is a pseudo-filesystem which is used as an interface to kernel data structures rather than reading and interpreting /dev/kmem. Most of it is read-only, but some files allow kernel variables to be changed.

The following outline gives a quick tour through the /proc hierarchy.

[number]

There is a numerical subdirectory for each running process; the subdirectory is named by the process ID. Each contains the following pseudo-files and directories.

cmdline

This holds the complete command line for the process, unless the whole process has been swapped out, or unless the process is a zombie. In either of these later cases, there is nothing in this file: i.e. a read on this file will return as having read 0 characters. This file is null-terminated, but not newline-terminated.

cwd This is a link current working directory of the process. To find out the cwd of process 20, for instance, you can do this:

```
cd /proc/20/cwd; /bin/pwd
```

Note that the pwd command is often a shell builtin, and might not work properly in this context.

environ
: This file contains the environment for the process. The entries are separated by null characters, and there may be a null character at the end. Thus, to print out the environment of process 1, you would do: (cat /proc/1/environ; echo) | tr "\000" "\n"

(For a reason why one should want to do this, see lilo(8).)

exe
: a pointer to the binary which was executed, and appears as a symbolic link. readlink(2) on the exe special file returns a string in the format:

  [device]:inode
  : For example, [0301]:1502 would be inode 1502 on device major 03 (IDE, MFM, etc. drives) minor 01 (first partition on the first drive). Also, the symbolic link can be dereferenced normally - attempting to open "exe" will open the executable. You can even type /proc/[number]/exe to run another copy of the same process as [number]. find(1) with the -inum option can be used to locate the file.

fd
: This is a subdirectory containing one entry for each file which the process has open, named by its file descriptor, and which is a symbolic link to the actual file (as the exe entry does). Thus, 0 is standard input, 1 standard output, 2 standard error, etc.

  Programs that will take a filename, but will not take the standard input, and which write to a file, but will not send their output to standard output, can be effectively foiled this way, assuming that -i is the flag designating an input file and -o is the flag designating an output file:

  ```
  foobar -i /proc/self/fd/0 -o /proc/self/fd/1 ...
  ```

  and you have a working filter. Note that this will not work for programs that seek on their files, as the files in the fd directory are not seekable.

  /proc/self/fd/N is approximately the same as /dev/fd/N in some UNIX and UNIX-like systems. Most Linux MAKEDEV scripts symbolically link /dev/fd to /proc/self/fd, in fact.

maps
: A file containing the currently mapped memory regions and their access permissions.

The format is:

```
address           perms offset   dev   inode
00000000-0002f000 r-x-- 00000400 03:03 1401
0002f000-00032000 rwx-p 0002f400 03:03 1401
00032000-0005b000 rwx-p 00000000 00:00 0
60000000-60098000 rwx-p 00000400 03:03 215
60098000-600c7000 rwx-p 00000000 00:00 0
bfffa000-c0000000 rwx-p 00000000 00:00 0
```

where address is the address space in the process that it occupies, perms is a set of permissions:

r = read w = write
x = execute s = shared
p = private (copy on write)

offset is the offset into the file/whatever, dev is the device (major:minor), and inode is the inode on that device. 0 indicates that no inode is associated with the memory region, as the case would be with bss.

mem This is not the same as the mem (1,1) device, despite the fact that it has the same device numbers. The /dev/mem device is the physical memory before any address translation is done, but the mem file here is the memory of the process that accesses it. This cannot be mmap(2)'ed currently, and will not be until a general mmap(2) is added to the kernel. (This might have happened by the time you read this.)

mmap Directory of maps by mmap(2) which are symbolic links like exe, fd/*, etc. Note that maps includes a superset of this information, so /proc/*/mmap should be considered obsolete.

"0" is usually libc.so.4.

/proc/*/mmap was removed in Linux kernel version 1.1.40. (It really **was** obsolete!)

root Unix and linux support the idea of a perprocess root of the filesystem, set by the chroot(2) system call. Root points to the file system root, and behaves as exe, fd/*, etc. do.

stat Status information about the process. This is used by ps(1).

The fields, in order, with their proper scanf(3) format specifiers, are:

pid %d The process id.

comm %s

The filename of the executable, in parentheses. This is visible whether or not the executable is swapped out.

state %c
One character from the string "RSDZT" where R is running, S is sleeping in an interruptible wait, D is sleeping in an uninterruptible wait or swapping, Z is zombie, and T is traced or stopped (on a signal).

ppid %d
The PID of the parent.

pgrp %d
The process group ID of the process.

session %d
The session ID of the process.

tty %d

The tty the process uses.

tpgid %d
The process group ID of the process which currently owns the tty that the process is connected to.

flags %u
The flags of the process.
Currently, every flag has the math bit set, because crt0.s checks for math emulation, so this is not included in the output. This is probably a bug, as not every process is a compiled C program. The math bit should be a decimal 4, and the traced bit is decimal 10.

minflt %u
The number of minor faults the process has made, those which have not required loading a memory page from disk.

cminflt %u
The number of minor faults that the process and its children have made.

majflt %u
The number of major faults the process has made, those which have required

loading a memory page from disk.

cmajflt %u
The number of major faults that the process and its children have made.

utime %d
The number of jiffies that this process has been scheduled in user mode.

stime %d
The number of jiffies that this process has been scheduled in kernel mode.

cutime %d
The number of jiffies that this process and its children have been scheduled in user mode.

cstime %d
The number of jiffies that this process and its children have been scheduled in kernel mode.

counter %d
The current maximum size in jiffies of the process's next timeslice, of what is currently left of its current timeslice, if it is the currently running process.

priority %d
The standard nice value, plus fifteen. The value is never negative in the kernel.

timeout %u
The time in jiffies of the process's next timeout.

itrealvalue %u
The time (in jiffies) before the next SIGALRM is sent to the process due to an interval timer.

starttime %d Time the process started in jiffies after system
boot.

vsize %u
Virtual memory size

rss %u

Resident Set Size: number of pages the process has in real memory,

minus 3 for administrative purposes. This is just the pages which count towards text, data, or stack space. This does not include pages which have not been demand-loaded in, or which are swapped out.

rlim %u
Current limit in bytes on the rss of the process (usually 2,147,483,647).

startcode %u
The address above which program text can run.

endcode %u
The address below which program text can run.

startstack %u
The address of the start of the stack.

kstkesp %u
The current value of esp (32-bit stack pointer), as found in the kernel stack page for the process.

kstkeip %u
The current EIP (32-bit instruction pointer).

signal %d
The bitmap of pending signals (usually 0).

blocked %d
The bitmap of blocked signals (usually 0, 2 for shells).

sigignore %d
The bitmap of ignored signals.

sigcatch %d
The bitmap of catched signals.

wchan %u
This is the "channel" in which the process is waiting. This is the address of a system call, and can be looked up in a namelist if you need a textual name. (If you have an upto-

date /etc/psdatabase, then try ps -l to see the WCHAN field in action)

cpuinfo
This is a collection of CPU and system architecture dependent items, for each supported architecture a different list. The only two common entries are cpu which is (guess what) the CPU currently in use and BogoMIPS a system constant which is calculated during kernel initialization.

devices
Text listing of major numbers and device groups. This can be used by MAKEDEV scripts for consistency with the kernel.

dma This is a list of the registered ISA DMA (direct memory access) channels in use.

filesystems
A text listing of the filesystems which were compiled into the kernel. Incidentally, this is used by mount(1) to cycle through different filesystems when none is specified.

interrupts
This is used to record the number of interrupts per each IRQ on (at least) the i386 architechure. Very easy to read formatting, done in ASCII.

ioports
This is a list of currently registered Input-Output port regions that are in use.

kcore This file represents the physical memory of the system and is stored in the core file format. With this pseudo-file, and an unstripped kernel (/usr/src/linux/tools/zSystem) binary, GDB can be used to examine the current state of any kernel data structures.

The total length of the file is the size of physical memory (RAM) plus 4KB.

kmsg This file can be used instead of the syslog(2) system call to log kernel messages. A process must have superuser privileges to read this file, and only one process should read this file. This file should not be read if a syslog process is running which uses the syslog(2) system call facility to log kernel messages.

Information in this file is retrieved with the dmesg(8) program).

ksyms This holds the kernel exported symbol definitions used by the modules(X) tools to dynamically link and bind loadable modules.

loadavg

The load average numbers give the number of jobs in the run queue averaged over 1, 5 and 15 minutes. They are the same as the load average numbers given by uptime(1) and other programs.

malloc This file is only present if CONFIGDEBUGMALLOC was defined during compilation.

meminfo

This is used by free(1) to report the amount of free and used memory (both physical and swap) on the system as well as the shared memory and buffers used by the kernel.

It is in the same format as free(1), except in bytes rather than KB.

modules

A text list of the modules that have been loaded by the system.

net various net pseudo-files, all of which give the status of some part of the networking layer. These files contain ASCII structures, and are therefore readable with cat. However, the standard netstat(8) suite provides much cleaner access to these files.

arp This holds an ASCII readable dump of the kernel ARP table used for address resolutions. It will show both dynamically learned and preprogrammed ARP entries. The format is:

```
IP address       HW type  Flags   HW address
10.11.100.129    0x1      0x6    00:20:8A:00:0C:5A
10.11.100.5      0x1      0x2    00:C0:EA:00:00:4E
44.131.10.6       0x3       0x2    GW4PTS
```

Where 'IP address' is the IPv4 address of the machine, the 'HW type' is the hardware type of the address from RFC 826. The flags are the internal flags of the ARP structure (as defined in /usr/include/linux/if_arp.h) and the 'HW address' is the physical layer mapping for that IP address if it is known.

dev The dev pseudo-file contains network device status information. This gives the number of received and sent packets, the number of errors and collisions and other basic statistics. These

are used by the ifconfig(8) program to report device status. The format is:

```
Inter-|   Receive                                                |  Transmit
 face |packets errs drop fifo frame|packets errs drop fifo colls carrier
    lo:       0    0    0    0     0     2353    0    0    0     0       0
  eth0:  644324    1    0    0     1   563770    0    0    0   581       0
```

ipx No information.

ipx route
No information.

rarp This file uses the same format as the arp file and contains the current reverse mapping database used to provide rarp(8) reverse address lookup services. If RARP is not configured into the kernel this file will not be present.

raw Holds a dump of the RAW socket table. Much of the information is not of use apart from debugging. The 'sl' value is the kernel hash slot for the socket, the 'local address' is the local address and protocol number pair."St" is the internal status of the socket. The "tx_queue" and "rx_queue" are the outgoing and incoming data queue in terms of kernel memory usage. The "tr", "tm->when" and "rexmits" fields are not used by RAW. The uid field holds the creator euid of the socket.

route No information, but looks similar to route(8)

snmp This file holds the ASCII data needed for the IP, ICMP, TCP and UDP management information bases for an snmp agent. As of writing the TCP mib is incomplete. It is hoped to have it completed by 1.2.0.

tcp Holds a dump of the TCP socket table. Much of the information is not of use apart from debugging. The "sl" value is the kernel hash slot for the socket, the "local address" is the local address and port number pair. The "remote address" is the remote address and port number pair (if connected). 'St' is the internal status of the socket. The 'tx_queue' and 'rx_queue' are the outgoing and incoming data queue in terms of kernel memory usage. The "tr", "tm->when" and "rexmits" fields hold internal information of the kernel socket state and are only useful for debugging. The uid field holds the creator euid of the socket.

udp Holds a dump of the UDP socket table. Much of the information is not of use apart from debugging. The "sl" value is the kernel hash slot for the socket, the "local address" is the local address and port number pair. The "remote address" is the remote address and port number pair (if connected). "St" is the internal status of the socket. The "tx_queue" and "rx_queue" are the outgoing and incoming data queue in terms of kernel memory usage. The "tr", "tm->when" and "rexmits" fields are not used by UDP. The uid field holds the creator euid of the socket. The format is:

```
sl local_address rem_address   st tx_queue rx_queue tr rexmits  tm->when
uid
 1:01642C89:0201 0C642C89:03FF 01 00000000:00000001 01:000071BA 00000000
0
 1:00000000:0801 00000000:0000 0A 00000000:00000000 00:00000000 6F000100
0
 1:00000000:0201 00000000:0000 0A 00000000:00000000 00:00000000 00000000
0
```

unix Lists the UNIX domain sockets present within the system and their status. The format is:

```
Num RefCount Protocol Flags    Type St Path
 0: 00000002 00000000 00000000 0001 03
 1: 00000001 00000000 00010000 0001 01 /dev/
printer
```

Where 'Num' is the kernel table slot number, 'RefCount' is the number of users of the socket, 'Protocol' is currently always 0, 'Flags' represent the internal kernel flags holding the status of the socket. Type is always '1' currently (Unix domain datagram sockets are not yet supported in the kernel). 'St' is the internal state of the socket and Path is the bound path (if any) of the socket.

pci This is a listing of all PCI devices found during kernel initialization and their configuration.

scsi A directory with the scsi midlevel pseudo-file and various SCSI lowlevel driver directories, which contain a file for each SCSI host in this system, all of which give the status of some part of the SCSI IO subsystem. These files contain ASCII structures, and are therefore readable with cat.

You can also write to some of the files to reconfigure the subsystem or switch certain features on or off.

scsi This is a listing of all SCSI devices known to the kernel. The listing is similar to the one seen during bootup. scsi currently

supports only the singledevice command which allows root to add a hotplugged device to the list of known devices.

An **echo 'scsi singledevice 1 0 5 0' > /proc/scsi/scsi** will cause host scsi1 to scan on SCSI channel 0 for a device on ID 5 LUN 0. If there is already a device known on this address or the address is invalid an error will be returned.

drivername

drivername can currently be: NCR53c7xx, aha152x, aha1542, aha1740, aic7xxx, buslogic, eata_dma, eata_pio, fdomain, in2000, pas16, qlogic, scsi_debug, seagate, t128, u15-24f, ultrastore or wd7000. These directories show up for all drivers which registered at least one SCSI HBA. Every directory contains one file per registered host. Every host-file is named after the number the host got assigned during initilization.

Reading these files will usually show driver and host configuration, statistics etc.

Writing to these files allows different things on different hosts. For example with the latency and nolatency commands root can switch on and off command latency measurement code in the eata_dma driver. With the lockup and unlock commands root can control bus lockups simulated by the scsi_debug driver.

self This directory refers to the process accessing the /proc filesystem, and is identical to the /proc directory named by the process ID of the same process.

stat kernel/system statistics

cpu 3357 0 4313 1362393

The number of jiffies (1/100ths of a second) that the system spent in user mode, user mode with low priority (nice), system mode, and the idle task, respectively. The last value should be 100 times the second entry in the uptime pseudo-file.

disk 0 0 0 0

The four disk entries are not implemented at this time. I'm not even sure what this should be, since kernel statistics on other machines usually track both transfer rate and I/Os per second and this only allows for one field per drive.

page 5741 1808
The number of pages the system paged in and the number that were paged out (from disk).

swap 1 0
The number of swap pages that have been brought in and out.

intr 1462898
The number of interrupts received from the system boot.

ctxt 115315
The number of context switches that the system underwent.

btime 769041601
boot time, in seconds since the epoch (January 1, 1970).

sys This directory (present since 1.3.57) contains a number of files and subdirectories corresponding to kernel variables. These variables can be read and sometimes modified using the proc file system, and using the sysctl(2) system call. Presently, there are subdirectories kernel, net, vm that each contain more files and subdirectories.

kernel

This contains files domainname, file-max, file-nr, hostname, inode-max, inode-nr, osrelease, ostype, panic, real-root-dev, securelevel, version, with function fairly clear from the name.

The (read-only) file file-nr gives the number of files presently opened.

The file file-max gives the maximum number of open files the kernel is willing to handle. If 1024 is not enough for you, try
echo 4096 > /proc/sys/kernel/file-max

Similarly, the files inode-nr and inode-max indicate the present and the maximum number of inodes.

The files ostype, osrelease, version give substrings of /proc/version.

The file panic gives r/w access to the kernel variable panic_timeout. If this is zero, the kernel will loop on a panic; if nonzero it indicates that the kernel should autoreboot after this number of seconds.

The file securelevel seems rather meaningless at present - root is just too powerful.

uptime

This file contains two numbers: the uptime of the system (seconds), and the amount of time spent in idle process (seconds).

version

This strings identifies the kernel version that is currently running. For instance:

Linux version 1.0.9 (quinlan@phaze) #1 Sat May 14 01:51:54 EDT 1994

SEE ALSO

cat(1), find(1), free(1), mount(1), ps(1), tr(1), uptime(1), readlink(2), mmap(2), chroot(2), syslog(2), hier(7), arp(8), dmesg(8), netstat(8), route(8), ifconfig(8), procinfo(8) and much more

CONFORMS TO

This roughly conforms to a Linux 1.3.11 kernel. Please update this as necessary!

Last updated for Linux 1.3.11.

CAVEATS

Note that many strings (i.e., the environment and command line) are in the internal format, with sub-fields terminated by NUL bytes, so you may find that things are more readable if you use od -c or tr "\000" "\n" to read them.

This manual page is incomplete, possibly inaccurate, and is the kind of thing that needs to be updated very often.

BUGS

The /proc file system may introduce security holes into processes running with **chroot**(2). For example, if /proc is mounted in the **chroot** hierarchy, a **chdir**(2) to /proc/1/root will return to the original root of the file system. This may be considered a feature instead of a bug, since Linux does not yet support the **fchroot**(2) call.

Vita

Surname: Chen Given Names: Lei

Place of Birth: Shanghai, People's Republic of China.

Education Institutions Attended:

University of Victoria	1992 —1998
University of Lethbridge	1991 —1992

Degrees Awarded:

B.Sc. (Co-op)	University of Victoria	1996

Honours and Awards:

University of Victoria Graduate Fellowship	1996 —1998
University of Victoria International Student Scholarship	1995

Publications:

Serra, M. and Chen, G. L. "Pseudo-Random Pattern Generation and Fault Coverage of Delay Faults with Non Linear Finite State Machines with High Entropy," Proc. IEEE On-Line Testing Workshop, Crete, Greece, 1997.

Partial Copyright Licence

I hereby grant the right to lend my thesis to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library from any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis: The Utility Model Applied to Layer-coded Sources

Author 

Lei Chen

Date October 26, 1998