Q-Fabric: System Support for Continuous Online Quality Management

A Thesis
Presented to
The Academic Faculty

by

Christian Poellabauer

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

College of Computing
Georgia Institute of Technology
April 2004

Q-Fabric: System Support for Continuous Online Quality Management

To my wife Rumana

and my son Adam.

ACKNOWLEDGEMENTS

This work could have not been finished without the support and sacrifice of many people I had the pleasure to meet over the last couple of years.

I would like to thank deeply my adviser Karsten Schwan. Karsten has advised and mentored me right from my start at Georgia Tech and he has given me the freedom to work on problems and approaches of most interest to me. He has been a great source of ideas, invaluable feedback, and endless support and encouragement throughout all stages of my Ph.D. program. I would also like to acknowledge the other members of my thesis committee. Calton Pu and Santosh Pande have supported me and provided me with invaluable feedback. Greg Eisenhauer has been a great collaborator and I enjoyed our many discussions. Finally, Hubertus Franke, my mentor from IBM Research, has gracefully accepted to be part of my committee and he provided me with many insightful comments on this work.

I had the immense pleasure to work with a large number of the brightest students at the College of Computing, who made the process enjoyable and successful. I am especially indebted to Richard West and Ivan Ganev, who accompanied me on this journey right from the beginning.

I would like to thank my wife Rumana, whose love, support, sacrifice, and encouragement made the last few years an enjoyable and adventurous journey. And I would like to thank my little son Adam, who teaches me to enjoy every moment of life and to put everything in perspective. This work is dedicated to both of them.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

The explosive growth in networked systems and applications and the increase in device capabilities (as evidenced by the availability of inexpensive multimedia devices) enable novel complex distributed applications, including video conferencing, on-demand computing services, and virtual environments. These applications' need for high performance, real-time, or reliability requires the provision of Quality of Service (QoS) guarantees along the path of information exchange between two or more communicating systems. Execution environments that are prone to dynamic variability and uncertainty make QoS provision a challenging task, e.g., changes in user behavior, resource requirements, resource availabilities, or system failures are difficult or even impossible to predict. Further, with the coexistence of multiple adaptation techniques and resource management mechanisms, it becomes increasingly important to provide an integrated or cooperative approach to distributed QoS management.

This work's goals are the provision of system-level tools needed for the efficient integration of multiple adaptation approaches available at different layers of a system (e.g., application-level, operating system, or network) and the use of these tools such that distributed QoS management is performed efficiently with predictable results. These goals are addressed constructively and experimentally with the Q-Fabric architecture, which provides the required system-level mechanisms to efficiently integrate multiple adaptation techniques. The foundation of this integration is the event-based communication implemented by it, realizing a loosely-coupled group communication approach frequently found in multi-peer applications. Experimental evaluations are performed in the context of a mobile multimedia application, where the focus is directed toward efficient energy consumption on battery-operated devices. Here, integration is particularly important to prevent the multiple energy management techniques found on modern mobile devices to negate the energy savings of each other.

CHAPTER 1

INTRODUCTION

1.1 Motivation

In recent years, we have witnessed an explosive growth in networked systems and applications, ranging from on-demand access to high-end computational services offered by grid computing to the proliferation of multi-peer multimedia applications over wired and wireless links. This is accompanied by a dramatic increase in device capabilities, as evidenced by the availability of inexpensive multimedia devices (high-resolution color displays, cameras), more powerful processors, larger and faster storage devices, and multiple choices for wireless communication links such as WiFi, Bluetooth, or GSM. These trends enable novel complex distributed applications, including video conferencing [14], tele-teaching [5], distributed multi-player games [74], on-demand computing services [59], and virtual environments [62]. The need for high performance, real-time, or reliable service provision requires Quality of Service (QoS) guarantees along the path of information exchange between two or more end systems. For example, a network-based video player requires sufficient CPU, network, memory, and bus resources to receive, decompress, and display video frames at real-time rates, and Internet data centers that lease out processing resources, storage, or applications to paying customers must provide resource availability and performance guarantees. However, these execution environments are prone to dynamic variability and uncertainty caused by changes in user behavior, resource requirements, resource availabilities, or by system anomalies (e.g., failures). Further, application workloads are difficult to characterize a priori [143]. Therefore, in order to meet a user's requirements for Quality of Service, careful management of resources and applications is required. In the past, over-provisioning of resources has been used to address the heterogeneity, large scale, and dynamics of these systems. However, these approaches are inefficient or even infeasible as in the case of resource-scarce mobile and embedded devices. QoS management capabilities deployed into these systems can address the variations in resource requirements and availabilities by dynamically and autonomically adapting and responding to variations in user requirements, workloads, or resource availabilities, without the need for human intervention. The efficient provision of QoS is further complicated by the trend toward the use of general-purpose systems, which are not well equipped for high performance or real-time applications. Thus, applications will increasingly have to develop workload characterizations and resource requirements at runtime – instead of relying on accurate offline characterizations – and they will have to rely on the dynamic adaptation of resource allocations and applications to match resource requirements with resource capacities.

Previous work has introduced approaches at different layers of a system, e.g., at the application-, system-, or hardware-level, that address the shortcomings of general-purpose systems by extending them with novel quality management support [100, 116, 134, 35, 57, 44, 93]. More recently, there has been increasing effort on the *coordinated* management of multiple resources and applications within layers and across layer boundaries [156, 136, 147, 155, 49]. However, there are few comprehensive approaches to integrate multiple adaptation techniques across multiple layers of a system and across multiple hosts. Further, much remains to be done for the study of the effects of one adaptation technique on another and to understand the complex relationships between resource management and application-level adaptations.

The goal of this research is twofold: (1) to provide the system-level tools needed for the efficient integration of multiple adaptation techniques available at different layers of a system, and (2) to use these tools to integrate multiple adaptation techniques such that QoS management is performed efficiently with predictable results. This thesis addresses these goals constructively and experimentally with the *Q-Fabric* architecture, which provides the required system-level mechanisms to efficiently integrate multiple adaptation approaches. The foundation of this integration is the *event-based* communication implemented by Q-Fabric, realizing a loosely-coupled group communication approach much like those increasingly chosen in distributed applications such as video-conferencing or multi-player gaming. Experimental evaluations of the Q-Fabric approach are performed in the context of a mobile wireless multimedia application. They key advantage derived from using Q-Fabric is the ability to better understand the complex effects of using and managing multiple system resources, like CPU, network links, and memory, on both single machines and on distributed platforms. A particularly complex quality metric is energy consumption. Closely tied to the utilization of all system resources, energy is either non-replenishable for the duration of a mission (limited battery life time) or associated with costs in data centers (energy bill, cooling). The second part of this thesis uses Q-Fabric to study the integration of multiple energy management techniques [33, 32, 3, 124, 121, 97, 131] in a distributed system, with the goal to both reduce energy consumption and provide acceptable Quality of Service to end user applications. Q-Fabric affords us the ability to perform *end-to-end* resource and application management, which to date has received little attention with energy as the driving resource. Specifically, this work utilizes *local* energy management techniques (residing in one or more layers of a device) to achieve some *global* energy consumption goal, e.g., to maximize the operational time of a distributed application or to minimize the cooling costs of a cluster server.

1.2 Terminology

Quality of Service (QoS) is an umbrella term for a number of methods and approaches to match the needs of applications to the resources available in a distributed system. The definition of Quality of Service given by ITU-T Rec. E.800 reads as follows:

> *The collective effect of service performance which determines the degree of satisfaction of a user of the service.*

This implies that the ultimate decision of good versus bad quality is subjective to the user. Satisfaction is usually associated with non-functional requirements such as dependability, reliability, timeliness, throughput, or robustness. In order to satisfy these requirements, applications require the management of Quality of Service, which is fundamentally an *end-to-end* issue, i.e., from the producer of data to the consumer. An *end system* is the end point of a communication, i.e., the producer or consumer of data. Traditionally, QoS support was considered a problem of the network layer in a distributed system, however, this approach had to be extended in order to take all other services that contribute to user-perceived qualities into consideration. Therefore, quality is determined by the efficient integration of management across multiple layers, including the application layer, the operating system, and the network. The work presented in this dissertation focuses on end system QoS management, i.e., the management of applications and resources found on an end device, rather than on intermediate network nodes (e.g., routers and switches).

Multimedia is a key application domain in the research of quality management. New classes of distributed multimedia applications have emerged, including tele-teaching, video conferencing, video-on-demand, or multi-player gaming. These applications are characterized by their highly interactive nature, large and continuous data transfers, and their requirements for timeliness and small jitter. The following definition is taken from [149]:

> *Multimedia communication deals with the transfer, the protocols, services and mechanisms of discrete media data (such as text and graphics)* and *continuous media data (like audio and video) in/over digital networks. Such a communication requires all involved components to be capable of handling a well-defined quality of service. The most important QoS parameters are used to request (1) the required capacities of the involved resources, (2) compliance to end-to-end delay and jitter as timing restrictions, (3) restriction of the loss characteristics.*

QoS management is then the supervision and control of applications and resources from the producer to the consumers of multimedia. This management ensures that *user-perceived* qualities are attained and sustained, even when resource availabilities fluctuate, when user requirements change, or when multiple applications simultaneously compete for the same set of resources.

The user of a QoS-aware application has to be given the opportunity to express the requirements or desired qualities via a *QoS specification*. The task of *QoS translation* is to convert these user-perceived requirements (e.g., image quality, frame rate) into system-level resource requirements (e.g., CPU or network bandwidths). Further, since QoS management is an end-to-end issue, resources and applications residing on remote devices have to be supervised and controlled cooperatively, often with very different resource availabilities (e.g., devices with different amounts of memory and disk space, network link bandwidths, or processor speeds).

The task of a *QoS manager* is to perform high-level QoS management activities, e.g., the setup and teardown of control paths between distributed applications, the allocation of resources, the communication between applications and system-level services, and the high-level long-term planning of QoS management. In contrast, *resource managers* are responsible for the low-level adaptation, i.e., they control one or more attributes of system-level resources such as processor and network bandwidths or memory utilization. Both QoS and resource managers base adaptations on information retrieved from distributed *resource monitors*. This *feedback-based* approach to QoS and resource management has its roots in control theory and allows systems to react to observed mismatches between desired and achieved QoS, bottlenecks, or failures.

One contribution of this thesis is Q-Fabric's provision of a uniform mechanism for communication and cooperation between resource managers, QoS managers, and monitors, based on *events* and *event channels*. Resource monitors, resource managers, QoS managers, and even applications can subscribe to these event channels as event producers and consumers. Event communication is anonymous and asynchronous, thereby supporting decoupled communication that has been found useful in a number of distributed applications using group communications.

Finally, multimedia applications belong to the domain of *soft real-time* applications, i.e., besides performance and quality requirements, they also have timing requirements (e.g., in regard to latencies and jitter). However, these applications can also tolerate degraded performance (late or dropped frames or reduced image size), i.e., even if application or data quality is reduced, it is still of use to the user.

The unique resource of importance in many systems and applications is energy. Energy is unique in that it is closely linked to all resources previously investigated and managed in QoS management approaches, such as CPU, network, or disks. That is, increased utilization of any of those resources also increases the consumption of energy. It is therefore a suitable vehicle for demonstrating the advantages of Q-Fabric in the management and control of multiple resources and multiple systems. In the remainder of this thesis, energy is expressed in Joules (J) or Watt-seconds (Ws) and power is expressed in Watts (W), e.g., 1J expresses the power of 1W expended over a period of 1 second.

1.3 Quality of Service of Multimedia Applications

The driving application for this dissertation is multimedia in mobile environments, where QoS management has to consider the limited battery life times (and therefore the limited energy resources) as an additional constraint. This section, therefore, describes the QoS management challenges for mobile multimedia applications and discusses the importance of energy as a constraining resource in mobile and wireless computing environments.

ISO 91 defines multimedia as the property of handling several types of representation media, which is the type of data that defines the nature of the information as described in its coded format (e.g., for audio: CCITT G711, MIDI, MPEG/audio) [54]. Some media are characterized by sequences of finite sized samples with strict temporal dependencies and are categorized as continuous or streaming media. There is a growing need for the support of Quality of Service for multimedia applications, partly due to the increase in real-time applications such as video and audio streaming and the anticipated growth in wireless access to the Internet. However, the large-scale deployment of distributed multimedia applications will result in high demands on the resources of these systems. In the mobile computing domain, additional problems are caused by resource scarcity, the mobility of users, associated issues with connectivity and transmission errors, and limited battery life times. Multimedia applications can gracefully adapt to scarcity of resources, i.e., image qualities can change, as can frame rates or compression methods. Figure 1 shows a 60s snapshot of a video streaming application between two devices. The left graph shows the achievable frame replay rates (measured at the receiver) for two different video streams. After 30s, a sender-side CPU-intensive task (compiler) is started which affects the achievable frame rate and the maximum and average jitter (right graph). Further, while the video stream is not disturbed, the frame rates vary by 7% in the worst case, however, when the system is disturbed, the

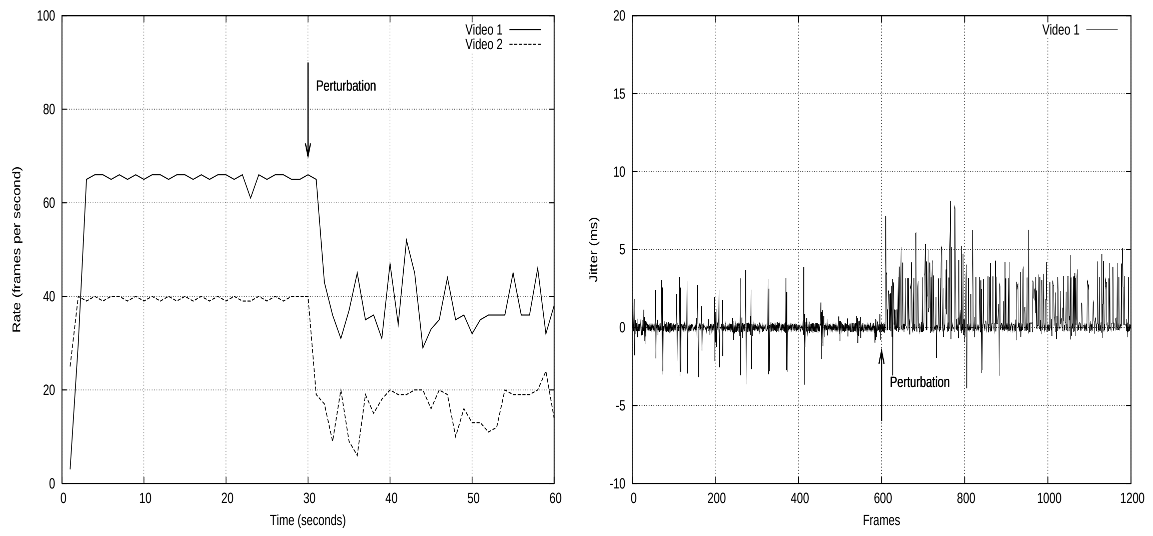


Figure 1: Frame rate (left) and jitter (right) for a video streaming example.

achieved rates vary by 80% and more. Another insight from these graphs is the difference in achievable frame rates for two different streams (left graph), which indicates that resource requirements also depend on the content of the data. Although multimedia applications require a continuous transfer of data, the actual frame rates, end-to-end latencies, and jitter can vary without degrading the overall quality below an unacceptable level. This can be exploited by trading resource utilization (and therefore quality) with energy consumption, as will be shown in the remainder of this thesis.

Energy as a First Class Resource. The resources of interest in this thesis are CPU, network, and energy. Energy is of particular interest because of its close ties to all other resources, i.e., utilization of a resource such as CPU and network always translate into energy consumption. Energy management can be considered as a type of quality management; the goal is to reduce energy consumption, while providing good or sufficient quality of service. Typically, devices were designed to deliver peak performance when requested, but workloads are variable and the full resources are only needed sometimes. Most resources are not fully utilized or do not have to be active all the time, therefore, *dynamic power management* (DPM) mechanisms detect idle times and switch unused components into low-power sleep or off modes. Other system-level approaches include energy-aware scheduling of tasks, re-arranging disk accesses (to reduce seek times), and buffering network transmissions (to increase traffic burstiness). Power management can also be supported at the application-level, i.e., applications can trade the use of 'expensive' (in terms of energy) resources for 'cheap' resources. However, for both system-level and application-level approaches, determining or predicting the energy costs associated with resource utilization is a difficult challenge. Moreover, when multiple energy management techniques are deployed,

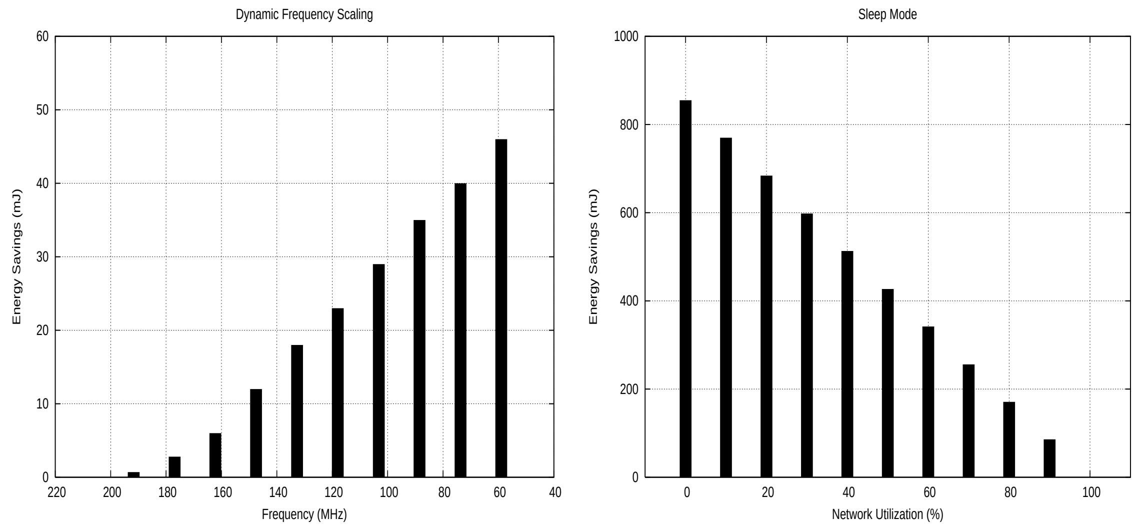


Figure 2: Energy savings when using dynamic frequency scaling (left) and the sleep mode of a wireless network card (right).

the uncoordinated combination can have unintended adverse effects, i.e., the combined energy savings are suboptimal. The left graph in Figure 2 shows the energy savings attainable if *dynamic frequency scaling (DFS)* [84] is used on a sample mobile processor for a snapshot of 1s and a CPU utilization of 25%. With DFS, the clock speed can be reduced, decreasing the device performance, but also decreasing the energy consumption. The frequency is scaled from 206.4MHz to 59MHz; at the lowest clock frequency the device saves 46mJ (compared to running at the default frequency). In comparison, the right graph shows the energy savings achievable by exploiting the *sleep mode* [87] of a wireless network card, i.e., when no communication occurs, the device is switched into a low-power mode. Depending on the network utilization, the savings can reach up to 850mJ. Compared to the savings of the frequency scaling, the network card's savings are much more significant. However, the achievable savings depend on the architecture, e.g., more recent mobile processors also offer a similar technique called *dynamic voltage scaling (DVS)* [122], which results in higher energy savings at the CPU-level. Figure 3 shows the expected and actual energy savings

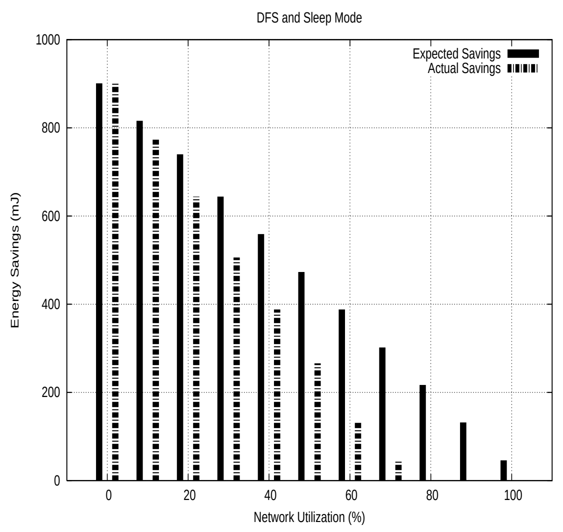


Figure 3: Expected energy savings (left bars) and actual energy savings (right bars) for the combined use of dynamic frequency scaling and a network card's sleep mode.

when both DFS and the network card's sleep mode are deployed. The CPU is operated at the lowest possible clock frequency, resulting in 46mJ derived from the use of DFS. The left bars show the *expected* energy savings, i.e., the sum of the energy savings for deploying DFS and the sleep mode of the network card, dependent on the network utilization. However, the actual energy savings are significantly lower than the expected savings (right bars), e.g., at 50% network utilization, the expected savings are 473mJ, the actual (measured) savings are only 266mJ (44% less). This is due to the effect frequency scaling has on network transmission, i.e., with DFS, all components of network communication that involve the CPU are slowed down, e.g., protocol processing and packet scheduling. Particularly if *fragmentation* is used (e.g., in the 802.11b standard), the MAC protocol layer and packet scheduler are invoked for each fragment (which can be as small as 256 bytes). This underlines that if multiple adaptation and management techniques are available, the integration of these techniques has to be performed carefully in order to obtain optimal results and to prevent one technique from negating the advantages of another. Therefore, Chapters 6 and 7 of this thesis will describe in detail the integrated management of multiple energy management techniques based on Q-Fabric.

1.4 The Thesis

Distributed complex applications deployed in uncertain environments require end-to-end QoS management. Moreover, with the availability of multiple adaptation 'knobs' at all layers of a system, an *integrated* approach to QoS management has to be taken.

This dissertation's thesis is that:

> **The integration of multiple QoS management mechanisms – possibly residing at different system layers or multiple hosts – is essential for achieving efficient adaptations and for preventing adverse effects stemming from the uncoordinated use of multiple management approaches. System support for this integration is important in order to obtain acceptable overheads, fine-grained adaptations, and unrestricted access to a system's resources. The approach is also shown to be useful for global energy management, which is key to the effective reduction of energy consumption in a distributed system.**

The main contribution of this dissertation is the extension of a general-purpose operating system to efficiently integrate quality management approaches at different layers of a system. Based on this integration, efficient adaptations of both applications and system resources can be performed in concert. The integration is based on system-level, event-based communication mechanisms, which ensure the low-overhead coordination necessary for fine-grained adaptations. The integrative approach introduced in this work is shown useful for multimedia applications in mobile wireless systems. For example, techniques such as energy-efficient media transcoding, energy-aware real-time video decoding, and the exploitation of resource idleness (processor, network card, etc.) are used collaboratively to ensure that energy consumption is reduced wherever it is necessary (e.g., as expressed in a global energy management goal). Integrated quality management is supported by the

Q-Fabric (Quality-Fabric) infrastructure, which provides the tools necessary to efficiently integrate application-level and system-level QoS management techniques in a distributed system.

1.5 Organization of the Dissertation

The rest of this document validates the thesis. Chapter 2 sets the context for this work. It explains the quality management model used, introduces the concept of integrated quality management, and discusses the key components of Q-Fabric.

The integration between applications and resources ('vertical' integration) is addressed in Chapter 3. The ECalls mechanism is a collection of communication tools, giving applications the flexibility to choose the most appropriate tool, e.g., depending on their real-time or performance requirements. In Chapter 4, the integration in the 'horizontal' direction (between devices) is addressed, which is implemented by the KECho event service.

Chapter 5 continues with a case study of the use of Q-Fabric for a distributed multimedia application. The results underline the importance of integration in both the vertical and the horizontal directions to ensure effective QoS management.

Chapter 6 begins the second part of the thesis, where the focus is moved toward energy as the driving constraint for the management of Quality of Service in distributed battery-operated systems. The chapter introduces several techniques for energy management and how they are linked into the Q-Fabric approach. This is continued in Chapter 7, where the previously introduced energy management techniques are combined to ensure efficiency in energy preservation, to prevent adverse effects of uncoordinated integration, and to achieve a global application-specific energy management goal.

Chapter 8 compares the work introduced in this dissertation with previous and ongoing related work in the areas of QoS management, event-based communications, resource monitoring and management, and energy-awareness. Finally, Chapter 9 concludes the dissertation with a summary of its contributions and an outlook on possible future research directions.

CHAPTER 2

THE Q-FABRIC ARCHITECTURE

In order to compensate for varying resource requirements and availabilities, applications rely on the adaptation of (a) application behavior, (b) the data streams these applications produce, and (c) the allocation of system-level resources. Over-provisioning of resources is inefficient or even infeasible in resource-scarce environments and limits the number of applications that can use these resources simultaneously. This dissertation, therefore, introduces a dynamic online Quality of Service management approach that addresses the real-time and performance needs of QoS-aware applications. This chapter introduces the key elements of the Q-Fabric architecture and the rationale behind the design decisions underlying this architecture.

2.1 A QoS Management Model

QoS management consists of multiple steps, including QoS specification, QoS translation, negotiation, setup, resource reservation, resource scheduling, and resource adaptation. A possible categorization of these tasks is presented in Figure 4. Q-Fabric's concentration is

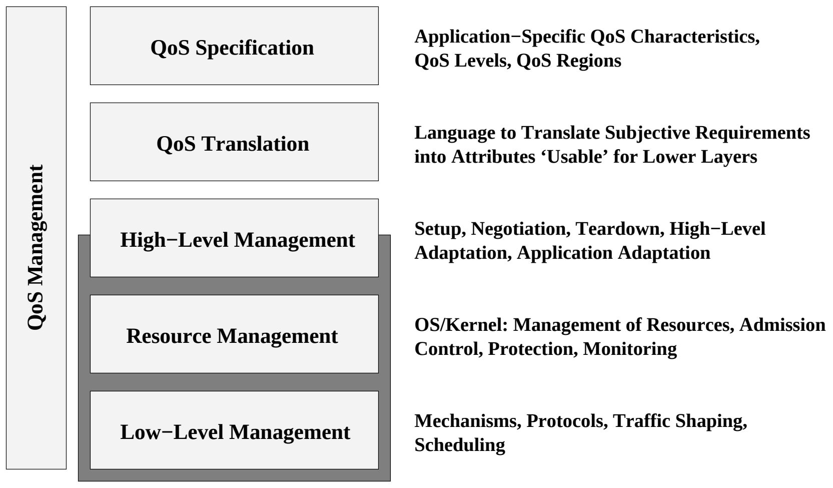

QoS Management
QoS Specification
Application−Specific QoS Characteristics, QoS Levels, QoS Regions
QoS Translation
Language to Translate Subjective Requirements into Attributes 'Usable' for Lower Layers
High−Level Management
Setup, Negotiation, Teardown, High−Level Adaptation, Application Adaptation
Resource Management
OS/Kernel: Management of Resources, Admission Control, Protection, Monitoring
Low−Level Management
Mechanisms, Protocols, Traffic Shaping, Scheduling

Figure 4: QoS management layers.

on the management issues at system level, including the low-level mechanisms such as CPU and packet schedulers or network protocols, and the functionality required to monitor and control resource availability and allocations. Although not the focus of this work, Q-Fabric includes simple solutions to QoS specification and translation. More specifically, with Q-Fabric, users can specify their requirements with *utility functions* [112], *QoS ranges* [52], and *weights* or *priorities*. A utility function is a non-decreasing relationship between the allocation of a resource or an application-specific metric and the gain derived from it by an application or user (see Figure 5). In other words, the more resources can be made

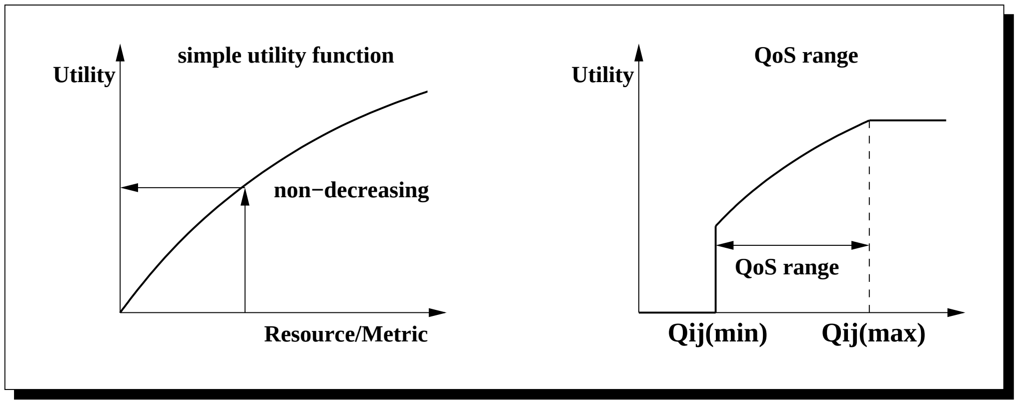


Figure 5: Utility functions and QoS ranges.

available to an application, the higher the user-perceived quality, e.g., expressed as more throughput for servers or better image quality for video streaming applications. Multiple application-specific utility functions can be provided, where the collection of these functions expresses a user's preferences. For example, for a video streaming application, a user could provide separate utility functions for the frame rate, the image size, and the color depth. The sum of the utilities for each metric j returns the total utility U_i for an application i:

$$U_i = \sum U_{ij}.$$

The total system utility U is expressed by the sum of all application utilities U_i, each multiplied by its weight w_i:

$$U = \sum (w_i * U_i).$$

Finally, a QoS range is expressed in the form $\{Q_{ij}(min), Q_{ij}(max)\}$ and indicates the portion of each utility function that is acceptable to the user (Figure 5). That is, the task of QoS management is to ensure that each utility function returns a non-zero utility (i.e., maintains a QoS metric above the lower boundary $Q_{ij}(min)$) and that no resources are wasted (i.e., maintains a QoS metric below $Q_{ij}(max)$).

Feedback control or closed-loop control has been key to the development of systems in uncertain and changing environments. Originating from work in electrical engineering, in recent years, researchers have adopted this theory for scheduling in QoS-aware and real-time systems [31, 133, 158]. Q-Fabric is based on the same theory, i.e., a monitor collects information that can be used to derive the quality of application performance and to detect possible bottlenecks. A controller utilizes this information, together with the QoS requirements specified by users, to decide *when*, *where*, and *how* to adapt [147]. The timing requirements of a feedback loop are typically more stringent than the timing requirements of the controlled application, i.e., if adaptation is too slow, it can exacerbate the detected problems, instead of solving them. With Q-Fabric, resources and applications at each host

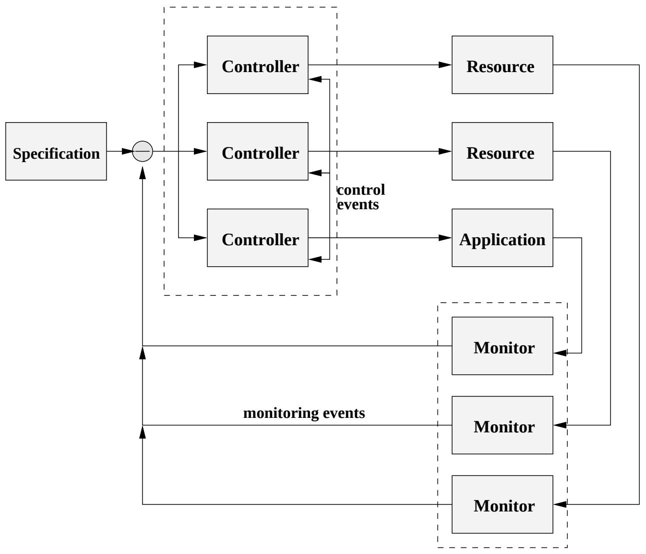


Figure 6: Feedback QoS control.

are ‘guarded’ by separate monitors, whose collected information is shared with controllers via *monitoring events* (Figure 6). The *error* between desired results (as expressed with the utility functions and QoS ranges) and the actual results is used by the controllers to decide whether adaptation is required. The controllers communicate with each other via *control events* in order to coordinate the adaptation of multiple resources or applications at multiple hosts.

2.2 Organization of Q-Fabric

The Q-Fabric architecture consists of several components (depicted in Figure 7), which cooperatively support the deployment of distributed QoS management policies. This section briefly describes the individual components.

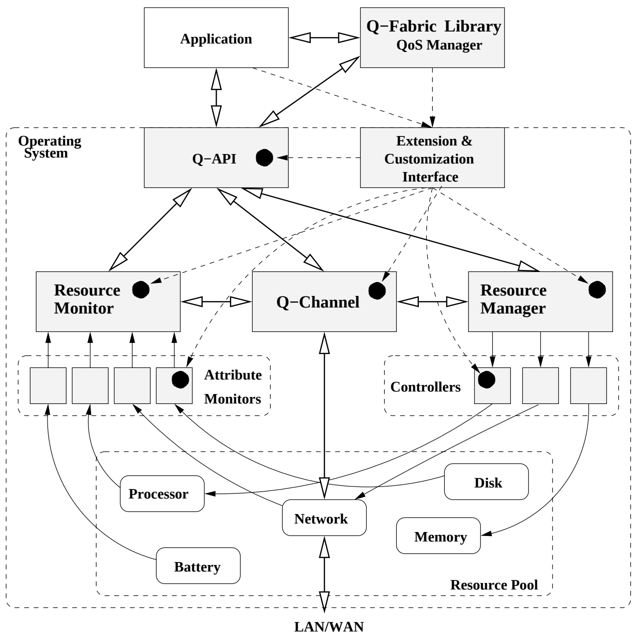


Figure 7: Q-Fabric overview.

QoS Manager and Q-Fabric Library. The QoS manager's task is to provide an application with the basic tools to specify its desired Quality of Service, to setup, control, and teardown QoS management connections with remote devices, and to coordinate the adaptation of applications in response to user requests or events from the system-level components of Q-Fabric. The QoS manager is part of Q-Fabric's user-level library, which has to be linked to any application that wishes to utilize Q-Fabric-based QoS management. This library also implements the data structures and function calls required by an application to setup QoS management and to specify its QoS requirements.

Resource Monitoring. Resource monitoring in Q-Fabric is a two-level process: each *attribute monitor* (Figure 7) has the responsibility to monitor one or more attributes of a resource, e.g., the run queue length of a CPU scheduler, the consumed bandwidths of a network connection, or the available memory or disk space (first level). These attribute monitors are called periodically by a *resource monitor* (second level), whose task it is to collect information from all attribute monitors and distribute it via monitoring events to other components of the Q-Fabric architecture. *Passive* attribute monitors are implemented as functions that, at the time of invocation by the resource monitor, inspect the attribute of the resource they are supposed to monitor, and return the current value. *Active* monitors, in contrast, are implemented as kernel threads that continuously monitor a resource attribute and return the collected information to the resource manager when called (e.g., as sums or averages). For example, an active attribute monitor for a network card might be activated whenever data arrives at the card, whereupon the monitor updates a data structure to reflect the amount of bytes received since the last invocation by the resource manager. The *poll period* is the time between successive invocations of attribute monitors, which is customizable by an application to any desired value and can be different for all attribute monitors. That is, an application can let Q-Fabric poll resources that are of importance to the application more frequently than others, or an application can provide approaches to dynamically determine (e.g., through feedback control) optimal values for poll periods. Finally, *thresholds* can be specified by applications for each active attribute monitor, i.e., when the monitored attribute exceeds or falls below a threshold, the attribute monitor calls back to the resource manager, instead of waiting for the current poll period to expire, thereby increasing the responsiveness of QoS management.

Resource Management. Similar to resource monitoring, resource management consists of two parts: (a) *resource controllers*, which are application-specific QoS policies, and (b) a *resource manager*, which is responsible for invoking the resource controllers whenever a monitoring or control event arrives. More specifically, each resource controller has a list of resources *of interest* associated with it, i.e., whenever the resource manager observes a change of a resource (through a received monitoring event), all resource controllers that have expressed interest in this resource are invoked. Further, the resource manager also has the responsibility of admission control and 'global' resource management. For example, if a new application is admitted, but as a consequence, the resource allocations of other applications have to be adjusted, the resource manager invokes the corresponding resource controllers to perform the required adaptations.

Q-Channel. Key to Q-Fabric's operation is the efficient *integration* of mechanisms and tools for quality management residing at different *layers* of a system. Figure 8 shows a

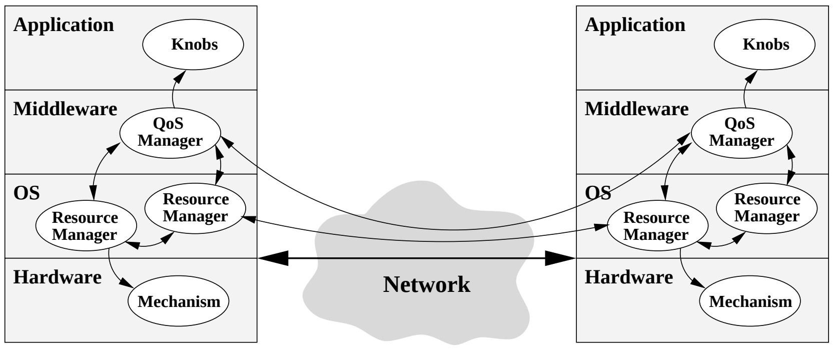


Figure 8: Layered system view.

scenario of multi-layer quality management, i.e., QoS management techniques are deployed at different layers of a system, and these techniques can cooperate within layers (*intra-layer integration*), across layers (*cross-layer integration*), and across device boundaries (*cross-device integration*). This document also refers to integration between different layers of a single system as *vertical* integration, whereas the cooperation within layers and among devices is called *horizontal* integration.

Event services have received increased attention as scalable tools for the composition of large-scale, distributed systems, as evidenced by their successful deployment in interactive multimedia applications and scientific collaborative tools. Past work on system monitoring [83, 139, 46] has routinely used event-based paradigms to represent and manage monitoring data, and this approach has also been extended to the domain of adaptive systems [148]. Further, for wide-area and web applications, event-based communication has received increased attention in the past, in part because of its support for decoupled communication: event producers are unaware of number or identities of event consumers (anonymous communication), and events can be raised anytime without the producer waiting for a response from the consumer and without the consumer having control over when events are raised (asynchronous communication).

With Q-Fabric, each distributed application has its own event channel, called a Q-Channel, which takes the function of a control path between distributed resource monitors and managers (i.e., integration in the horizontal plane). Many multimedia applications are *multi-peer* applications in nature, such as video conferencing, remote teaching, or multi-player gaming. These multi-peer applications are increasingly supported by group communication approaches, including event services [119, 27, 48, 77]. It is only natural to map these group communication between distributed applications onto the underlying system-level QoS management mechanisms. QoS managers, resource managers, and applications

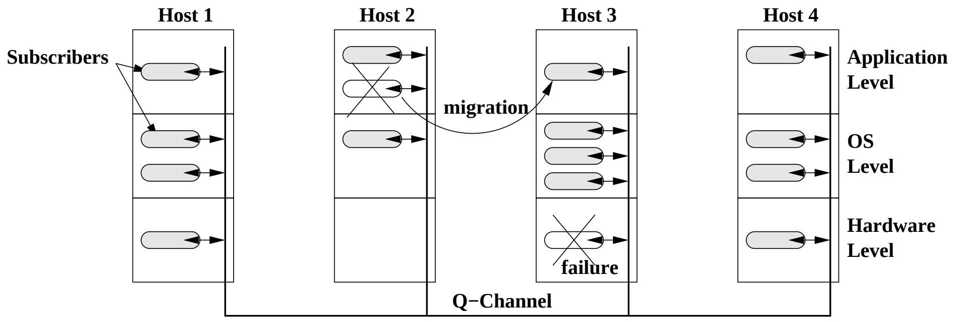


Figure 9: Q-Channel overview.

at different layers of a system can therefore subscribe to a Q-Channel as event producers and consumers (Figure 9). Event-based communication supports lightly coupled exchange of data, i.e., if a subscriber fails, leaves, or migrates, the communication between all other subscribers is unaffected. Event publication and receipt imply an action triggered by the event, like the execution of a handler function or the transfer of certain data. Such an action is defined at the time an application subscribes to the event channel. Furthermore, the specific publish/subscribe implementation of Q-Channels in Q-Fabric describes events' data content with well-defined *formats* known to producers and consumers. Using formats, event-based interactions can be enriched with application- or service-specific, dynamically created event *handlers*, able to manipulate event content. One result is that Q-Fabric's event communication need not prescribe a specific synchronization strategy for access to event data, thereby permitting consumers to receive any number of different events and use them as they see fit, subject only to restrictions in the total memory available for storing event representations.

Q-API. Q-Fabric supports a variety of well-defined, per-channel ways in which control is passed between application and system domain or between services within the system domain upon event production and receipt. By separating data and control, multiple specializations of each may be used to implement efficient system/user event sharing, thereby addressing different applications and usage scenarios. For instance, a 'real-time' event channel implies that upon event generation by the kernel, a real-time signal will be generated to the address space subscribed to this channel.

While a Q-Channel is responsible for the horizontal integration between remote devices, the Q-API is the interface linking application-level and system-level, i.e., in the vertical direction. The focus here is on flexibility and performance, e.g., by offering multiple ways of interaction between applications and Q-Fabric (including shared memories, system calls, signals, or the /proc interface). While Q-API was created with flexibility in mind, it was also a goal to keep the application interaction simple if desired by the user. For example, an application 'registers' with Q-Fabric by calling a function with the following prototype:

```
int qfabric_config (char *application_id,
                    char *group_id,
                    struct qos_node *qos_params);
```

The first two attributes are unique strings, identifying the application. A central *group* *server* is contacted by Q-Fabric, transparent to the application, to see if an application with the corresponding strings has already registered on a remote host. If so, Q-Fabric, again fully transparent to the application, sets up a Q-Channel between all subscribed monitors and managers. If no entry is found, Q-Fabric registers the application with the group server, thereby announcing its existence to future – remote – instances of the application contacting the group server. The third and last attribute, `qos_params`, is a data structure containing application-relevant QoS information, most importantly the desired QoS ranges. An example for such a data structure is given here for a video streaming application:

```
struct qos_node {
    int min_frame_rate;
    int max_frame_rate;
    int min_color_depth;
    int max_color_depth;
    int min_image_width;
    int max_image_width;
    int min_image_height;
    int max_image_height;
};
```

Utility functions are expressed as simple C code, e.g., the following two utility functions show a linear relationship between (a) the frame rate and its utility to an application and (b) the color-depth and its utility to an application:

```
int utility1 (int rate) {
    return rate*4;
}

int utility2 (int color_depth) {
    return color_depth;
}
```

In this example, these functions implicitly express that higher frame rates are preferable to higher color depths (in bits per pixel), e.g., a frame rate of 20 will return a utility of 80, while a color depth of 24 bits per pixel will return a utility of only 24.

Extension and Customization Interface. The extension interface allows applications to interact more closely with the system-level Q-Fabric components than Q-API would allow. It gives users the opportunity to add new functionality such as attribute monitors or resource controllers. The customization interface allows users to modify poll periods and thresholds for attribute monitors or to 'download' filters into Q-Channels to customize the event traffic to the needs of a particular application.

2.3 Arguments for a System-level Approach

Q-Fabric adopts a system-level approach to QoS management, i.e., besides a small user-level library that is linked with QoS-aware applications, its components are implemented as extensions to an operating system kernel. Past work has shown that fine-grain, kernel-level resource management can provide applications with benefits not derived from coarser-grain, user-level QoS management [51]. Specifically, with user-level QoS management, excessive delays or overheads experienced by dynamic adaptations can negatively affect or even negate the advantages of run-time adaptation for real-time applications [117]. The issue is that delays and overheads may be caused by an application-level QoS manager's interactions with the system-level mechanism they must use to monitor and steer resource allocations. OS kernels or network services may present interfaces to application-level resource managers that necessitate repeated kernel calls in order to determine the resources available for allocation to certain application tasks. Inappropriate interfaces may require managers to poll for changes in resource state or make unnecessary resource reservations (as also noted in [57] and [105]). In comparison, within OS kernels or in network services, it is straightforward to inspect and manipulate the data structures involved in resource allocation. In addition, OS kernels can enforce constraints on the delays experienced by applications when they are informed about changes in their allocations, or when they must be adapted to conform to new QoS requirements or resource availabilities, whereas application-level resource managers may be at the mercy of CPU schedulers. Finally, a system-level approach allows Q-Fabric to perform certain functions at *interrupt* time, i.e., outside of the context of any process, removing the need for context switches. The result is that certain operations (e.g., resource monitoring) can be performed at the expense of a function call, giving Q-Fabric even higher levels of predictability and accuracy.

2.4 Periodic Processing and Communication Model

In this thesis, the applications managed by Q-Fabric rely on a periodic processing and communication model, as can be found in many multimedia applications, e.g., video and audio streaming. The two key resources considered are CPU and network. This section describes how both of these resources can be controlled, enabling the QoS management required by multimedia applications.

2.4.1 DWCS Scheduling

The traditional UNIX scheduler has been shown to have unacceptable performance for multimedia applications [92]. For example, an application with a fixed real-time priority could have precedence over all other applications at all times, and therefore, starve best-effort applications. This has led to the development of new scheduling approaches, including those based on reservations and on proportional share resource allocations [100]. To efficiently support real-time applications, this thesis uses a hard real-time CPU scheduler, called DWCS (Dynamic Window-Constrained Scheduler) [146, 145]. DWCS assigns each process the following attributes: a period T, a service time C, and a window-constraint x/y. Using these attributes, DWCS **attempts** to service a process for at least C time units in a period of T time units, and it **guarantees** that it will service a process in $y - x$ periods in a window of y periods if the *CPU utilization* is less than or equal to 100%. Thus, the minimum CPU utilization consumed by a process i is determined by

$$U_i(min) = (1 - x_i/y_i) * C_i/T_i.$$

The period T_i of a process i is used to set a deadline until the scheduler has to service process i for at least C_i time units. If the process misses its deadline more than x_i times in a window of $T_i * y_i$, the scheduler *violated* the real-time guarantees to this process. Each process can be scheduled once in its period, unless it is marked as *work-conserving*; in that case it is possible to schedule this process several times within its period as long as CPU utilization allows.

Scheduling attributes are adjusted dynamically to reflect the progress of a process. DWCS distinguishes between the *original window-constraint* x/y and the *current window-constraint* x'/y', where the latter is modified dynamically according to the following rules:

Rule a: If the scheduler allocates C_i time units to process i within a period T_i, the window-constraint is relaxed by decrementing the window denominator. If the denominator and the numerator of the window-constraint are equal ($y'_i = x'_i$), both are decremented until they reach zero, at which they are reset to their original values:

```
if ($y'_i > x'_i$) then $y'_i = y'_i - 1$;
else if ($y'_i = x'_i$) and ($x'_i > 0$) then
    $x'_i = x'_i - 1$; $y'_i = y'_i - 1$;
if ($y'_i = x'_i = 0$) then $y'_i = y_i$; $x'_i = x_i$;
```

This ensures that a process that has been serviced already within its period, will relax its window-constraint.

Rule b: If a process misses to be scheduled within its current period, the window-constraint is adjusted to reflect an increased urgency:

```
if ($x'_i > 0$) then $x'_i = x'_i - 1$; $y'_i = y'_i - 1$;
    if ($y'_i = x'_i = 0$) then $x'_i = x_i$; $y'_i = y_i$;
else if ($x'_i = 0$) then $y'_i = y'_i + 1$;
```

This gives the process a tighter window-constraint and therefore an increased probability of being scheduled in the near future. Note, that if the window numerator is zero and a process misses to be scheduled within its period, a *violation* has occurred.

The precedence rules used by DWCS among processes are shown in Table 1.

The simplified pseudo-code for DWCS is as follows:

Table 1: Precedence rules.

Earliest deadline first (EDF)
Equal deadlines, then order tightest window-constraint first
Equal deadlines and zero window-constraints, then order highest window-denominator first
Equal deadlines and equal non-zero window-constraints, then order lowest window-numerator first
All other cases: first-come first-serve

```
while (TRUE) {
  find process i according to the precedence rules in Table 1;
  adjust window-constraints for process i (Rule a);
  for (each process j<>i missing its deadline) {
    adjust window-constraint for j (Rule b);
    adjust deadline for j;
  }
  schedule i;
  adjust deadline for i;
}
```

In [145], the following real-time guarantees of DWCS are demonstrated:

(a) DWCS is able to give firm bounds for the maximum delay of service to a given process on the run queue in both under-load and over-load situations.

(b) The least upper bound on the system utilization is 100% if $C_i = k$ and $T_i = nk\ \forall i$ with k, n being integers ≥ 1.

2.4.2 Coordinated CPU and Network Management

In this dissertation, the same scheduler, DWCS, is used for both CPU and packet scheduling. Here, the end of a period T specifies a deadline by which the transmission of a packet has to be initiated. The periods of the CPU scheduler and the packet scheduler are identical, in order to ensure that packet generation and packet transmission are synchronized (e.g., for video streaming, the period corresponds to the inverse of the frame rate). However, the periods of the packet scheduler are *phase-delayed* with the periods of the CPU scheduler,

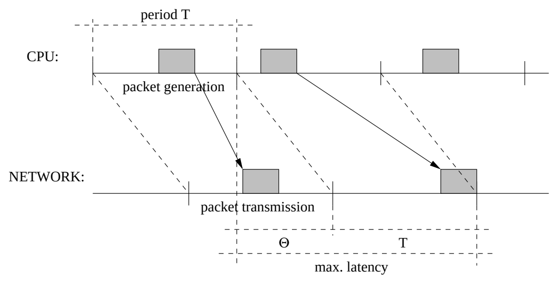


Figure 10: Coordinated CPU and packet scheduling.

as shown in Figure 10. This phase Θ determines the latency of a packet – from packet generation to transmission – acceptable to the user: $l_{max} = \Theta + T$. For example, a period of 30ms and a phase of 15ms cause a maximum latency of 45ms for a packet. The actual end-to-end latency is further determined by the transmission delays and the reception and processing delays at the client.

2.5 Summary

QoS management with Q-Fabric is based on the exchange of events and event channels, called Q-Channels. Once a Q-Channel has been created (transparent to an application), the QoS and resource managers and monitors of a system are free to join, leave, and communicate with other managers and monitors at remote hosts. Besides Q-Channels, the key components of Q-Fabric are the resource monitoring and controller parts, the Q-API interface, and an extension and customization interface. The coordinated management of distributed resources can be particularly complicated in large-scale applications, where (1) multiple data sources and sinks communicate, (2) data streams have relationships such as synchronization, (3) data streams reserve resources together (resource sharing), (4) resources or applications can migrate, or (5) the system is highly dynamic, i.e., applications and their Quality of Service requirements can change at any time. Q-Fabric addresses this complexity by tying the creation of event channels for resource management with the creation of data channels, i.e., the setup of control paths between distributed monitors and managers is transparent to the application and occurs at the same time applications set up their communication paths (e.g., the creation of a video streaming connection between a client and a server initiates the creation of a Q-Channel control path between the underlying resource managers). QoS ‘engineers’ can therefore rely on Q-Channels to automatically interconnect the resource managers along the path of a data stream, allowing them to focus on the development of efficient QoS policies instead of the actual linkage between distributed resource managers. Further, event communication in Q-Fabric is asynchronous, i.e., event sources publish events on Q-Channels without waiting for a response. This loose coupling of event publishers and subscribers supports the dynamic behavior of multimedia systems (e.g., resource migration). Also, subscribers to a Q-Channel are unaware of the identities or even number of other subscribers, which further facilitates the dynamic joining, leaving, or migrating of subscribers. Finally, the target application domain is mobile multimedia; therefore, the model used for resource allocation is based on the periodic processing and communication commonly found in these applications. Specifically, in this dissertation, the resources CPU and network are controlled by a real-time scheduler (DWCS), which provides applications with the timeliness they require.

CHAPTER 3

INTEGRATION ACROSS PROTECTION DOMAINS

System-level resource management and user-level application adaptation have to be performed cooperatively to attain efficient QoS management for multimedia applications. Efficient integration across protection boundaries, i.e., between kernel- and user-level, is essential to the successful coordination of QoS managers, applications, and resource managers. This chapter introduces ECalls, a novel mechanism to provide cooperation of application-level and system-level QoS management. Without ECalls, applications and user-level QoS managers would be restricted to use existing interfaces – such as system calls – that are known to be expensive and they would not have full access to system-level functionality. This chapter addresses the integration between user-level and system-level QoS management provided by ECalls as part of the Q-Fabric infrastructure and describes ECalls' components and approaches to cross-domain communication in detail. Using ECalls, user-level QoS managers and applications can leverage system-level mechanisms while limiting the penalties of frequent crossings of protection domain boundaries. It thereby provides the tools necessary for the efficient *integration* of user-level and system-level QoS management mechanisms in the 'vertical' direction (i.e., across protection domains within a system).

3.1 Introduction

Lack of QoS support from operating systems can be a detriment to the efficient implementation of applications like distributed virtual environments, multi-player games, telepresence, remote sensing, or remote collaboration. This is because these applications' multiple and continuous data streams require bounds on latency, data loss, and jitter, e.g., to prevent audible gaps in audio or choppy replay of video. Previous research has addressed these needs by enhancing operating systems with multimedia and real-time CPU schedulers [57] or fair share packet schedulers [44], or with distributed resource managers operating across multiple machines and resources [90]. Such work can take advantage of the extensibility of operating systems to add new functionality without having to modify and re-compile current system images. In commercial systems, such as Microsoft Windows products or Linux, OS extensibility is routinely used to support new hardware like disk drives or network cards with dynamically loadable device drivers. Extensibility is also used to realize more complex and demanding services, like load balancing mechanisms in parallel computing environments, facilities that support distributed resource management and Quality of Service for real-time applications, or kernel ports of certain application components that used to reside in user-level, but have shown performance gains if implemented in the kernel, such as the Linux in-kernel HTTP servers and accelerators *tux*[1] and *khttpd*[2].

For purposes of protection/safety and interoperability, applications and system services typically run in different *protection domains*, using well-defined interfaces between these domains. For instance, device drivers operating in the system domain use socket interfaces to isolate applications from the different types of network cards being used. There are two well-known problems with this approach:

- Cross-domain calls, such as system calls and signals, can be *expensive*. The resulting high delays in the transfer of control and data between caller and callee are detrimental to service responsiveness, and it limits the scalability of applications with high call-frequencies (e.g., busy web servers). In response, researchers have introduced ways to control the costs associated with cross-domain communication [26, 73] and ways to reduce the frequency of system calls by extending kernels with appropriate application-specific functionality [8, 29, 39].

- Cross-domain calls can be *restrictive*, in that they may not offer the flexibility needed by QoS-aware or real-time applications. Past work has addressed this issue by providing domain-specific interfaces (e.g., quality sockets [34]) or by better integrating kernel- with corresponding user-level actions via efficient upcall primitives [23, 53],

[1]http://www.redhat.com/products/software/tux
[2]http://www.fenrus.demon.nl

including variants addressing multimedia and real-time applications [43].

In comparison to previous work focused on specific applications or services, this chapter introduces a single, uniform cross-domain transfer facility for control and data that is (1) sufficiently flexible to support a wide range of QoS-aware applications or services and (2) customizable to individual needs. This facility, called *ECalls*, permits QoS-aware applications to communicate with OS-based resource management services (such as Q-Fabric) to monitor resource availability, to re-negotiate QoS specifications, or to be notified when an application must adapt its own behavior (e.g., adaptation of image quality to reduce processing and networking requirements). Moreover, ECalls implements multiple, alternative notification methods concerning system-level events. For instance, consider the notification of single-threaded web servers about the arrival of new service requests, which is commonly done in a pull-based fashion using *select()* or *poll()* system calls. Here, ECalls facilitates the implementation of alternate pull methods, since both *select()* and *poll()* calls have been shown to have poor scalability with high request frequencies [17].

The interaction model supported by ECalls offers the flexibility and efficiency needed for intra-machine cross-domain calls, but its design also permits its extension to cross-machine interactions. This is because ECalls uses the notion of events and event channels, where parties interested in certain events subscribe to shared channels to which events are produced and from which they are received. Event publication and receipt imply an action triggered by the event, like the execution of a handler function or the transfer of certain data. Such an action is defined at the time an application subscribes to the event channel. Furthermore, ECalls' publish/subscribe implementation describes events' data content with well-defined *formats* known to producers and consumers. Using formats, event-based interactions can be enriched with application- or service-specific, dynamically created event *handlers*, able to manipulate event content. One result is that ECalls need not prescribe a specific synchronization strategy for access to event data, thereby permitting an application to receive any number of different events and use them as it sees fit, subject only to restrictions in the total memory available for storing event representations. For example, the implementation of ECalls requires the memory areas shared between kernel and user to be pinned in memory, thereby defining certain limits on the amounts of data shared in this fashion.

ECalls uses the event channel paradigm to support the application- or service-customized transfer of control and data across protection domains:

- Event delivery means that an interested object (e.g., a multimedia application) receives a notification of a system-level event (e.g., the arrival of data at a network connection). The delivery of an event may include a *cross-domain control transfer*, where the event-related action (e.g., the execution of a handler routine) is performed in the context of the consumer process. This can require a *processor switch* to the consumer if it is not a currently active process. Alternatively, the event-related action can be performed by the event producer (e.g., an operating system service), where the action is performed by the producer on behalf of the consumer without the need for a control transfer.

- Events can be accompanied by data, as part of the event itself (e.g., passed as attributes to the event handler), or as separated data items (described by data formats). In the latter case, a *cross-domain data transfer* takes place, which consists of copying the data associated with the event to a memory area accessible by both the event producer and consumer.

Extending prior work, the event-based communication supported by the ECalls mechanism has the following, novel functionality:

- By its ability to link cross-domain with intra-domain calls, ECalls can provide new functionality. An example is the cooperation between its event dispatcher and the operating system's process scheduler to attain *bounded delays* on event delivery, which is important for real-time applications.

- *Event filters* can initiate low-overhead cross-domain control transfers and attach arbitrary, necessary constraints to such transfers [79], before other, more heavyweight control transfer facilities are utilized. This can be used to pre-process data associated with events or to provide behavior analogous to that of optimistic active messages [142].

- *Custom event handlers* can be deployed – or even dynamically generated – by applications, therefore specializing the system's behavior to the needs of the application using them.

- *Remote event notification* can be achieved by installing kernel-level event handlers that redirect event notification to remote applications without explicit application involvement. Further, applications can install event handlers and event filters on remote devices on behalf of other applications, e.g., to customize the type of events being delivered.

3.2 Design and Implementation

The key components of ECalls are:

- shared memory between applications and kernel services for both event notification and data sharing,

- multiple event notification approaches, e.g., signals and kernel event handlers,

- dynamic generation of kernel event handlers,

- coordination of event dispatch and CPU scheduling, and

- event notification across networks.

ECalls has been implemented as a dynamically loadable kernel module as part of the Q-Fabric approach. It provides event channels between event producers and event consumers, e.g., resource managers and applications. As an example, kernel services such as device drivers, load balancers, or resource managers can raise events which are passed on to one or more applications by the ECalls event notification. If multiple applications are consumers of an event, the order of event notification is determined by the applications' CPU scheduling priorities if a processor switch is involved. Otherwise, the event handlers are invoked in the same order the applications registered with the kernel service. ECalls offers several registration interfaces for both applications and kernel services:

- *Service Registration Interface* - kernel services announce the availability of their services by registering with this interface, specifying a unique name which will be used to identify the service. This unique name is exported to applications via the /proc virtual file system, i.e., applications can obtain a list of all currently available services by accessing the /proc directory.
- *Event Channel Subscription Interface* - applications express their interest in events published by a service by registering through this interface, where an event channel, and therefore a kernel service, is identified by the name of the service found in /proc.
- *Kernel Handler Registration Interface* - kernel services and kernel modules can use this interface to register kernel-level functions, that can be executed by ECalls on behalf of applications (as kernel event handlers).

The registration of a new kernel service has the following syntax:

```
kernel_service {
    ...
    service_id = register_service(NAME_OF_SERVICE);
    while (service_needed) {
        /* perform kernel service (e.g., resource management) */
        ...
        /* raise event */
        raise_event(service_id, pid, data_pointer, deadline, cpu);
        ...
    }
    unregister_service(NAME_OF_SERVICE);
}
```

The actual event notification is performed with the invocation of the *raise_event* function with the following attributes:

- **service_id**: this attribute contains the unique identifier returned by the service registration interface and is used by ECalls to associate an event producer (kernel service) with event consumers (applications).

- **pid**: even if multiple applications subscribe to the same event channel, a service is able to direct an event to only one event subscriber (identified by the process ID). If all event subscriber are to receive the event, pid is -1 and ECalls notifies all processes registered for this event. A *wake-one* policy as supported with the pid attribute is desirable if, for example, multiple server threads wait for requests on a socket. The kernel can then notify only one of these servers instead of all of them (e.g., in a round-robin fashion).

- **data_pointer**: this attribute of type *unsigned long* can be used to pass data along with an event, either as a simple unsigned long value or it can be pointer to a kernel- or user-level memory location holding the data to be shared.

- **deadline**: events can have a deadline (expressed in microseconds) associated and events are dispatched to applications according to the EDF scheduling policy; events that miss their deadline are discarded from the event queue.

- **cpu**: the final attribute, cpu, indicates if the kernel service wishes to take advantage of ECalls' ability to cooperate with the CPU scheduler ('1' yes, '0' no).

An application registers for events in the following way:

```
main {
  struct ecalls my_ecalls;
  struct sh_memory my_memory_up;
  struct sh_memory my_memory_down;
  ...
  my_ecalls.signal = SIGRTMIN + 0;
  my_ecalls.process_id = getpid();
  ECalls_subscribe(NAME_OF_SERVICE, &my_ecalls, &my_memory_up, &my_memory_down);
  ...
  while (running) {
    ...
  }
  ECalls_unsubscribe(NAME_OF_SERVICE);
}
```

An application initializes a data structure (*struct ecalls*) with information how it wishes to be notified of events. The data structure contains the following entries:

- **signal**: specifies the real-time signal number (between 32 and 63). It is the responsibility of the application to subscribe a signal handler with the chosen signal number.
- **process_id**: the application can specify if the event should be received by itself (*process_id = getpid();*) or by some other process, identified by its process ID.
- **k_handler**: this string identifies the name of a kernel event handler that has been previously registered with ECalls and will be called by ECalls on behalf of the application when events are raised.
- **k_code**: this string identifies C-like code that will be 'downloaded' into the kernel by ECalls, compiled, and added to the list of available kernel event handlers.
- **k_thread**: this flag indicates if a kernel event handler is to be executed in the context of a kernel thread provided by ECalls.

With ECalls, an application can register two shared memory segments, one for data transfer from user to kernel level, and one for the opposite direction, simplifying the synchronization between application and kernel service. Besides the methods described above, shared memory can be used for event notification, e.g., a kernel service can toggle a flag in the memory, which is being polled periodically by the application. Further information passed in *struct ecalls* includes the order of event notification (e.g., first 'k_handler', then 'signal') if more than one method is desired. After the data structure is initialized, the application registers interest in a kernel service and the associated event channel by calling *ECalls_subscribe*, specifying the name of the kernel service, the above mentioned data structure, and the memory locations for the data transfer between kernel service and application.

3.2.1 Application Events

Fast User-ECalls. Fast User-ECalls are a low-overhead version of system calls, with the restriction that the invoked handler function is not allowed to block. On return of a regular system call function, the kernel first checks for pending *bottom halves* (the *slow* part of interrupts). Next, the kernel checks if it is necessary to invoke the scheduler, and finally, the kernel looks for pending signals and invokes signal handlers if necessary. In the case of Fast User-ECalls, the default situation is to avoid these steps altogether and directly return to the user application. The return value of the short non-blocking function executed by a Fast User-ECall decides if any or all of the steps described above are required to be executed. In addition, the return value can indicate that it is necessary to turn the non-blocking Fast User-ECall into a regular (and possibly blocking) system call. In that case, the function executed upon a Fast User-ECall acts as an *optimistic* handler function, which returns immediately if the optimistic assumption that the handler function is not required to block, holds true. If the assumption fails, a regular system call is invoked. Fast User-ECalls are useful for short and simple actions such as toggling flags in the kernel or updating QoS attributes for resource managers, i.e., where ordinary system calls would be too expensive.

Deferred User-ECalls. Deferred User-ECalls are similar to Fast User-ECalls in that they invoke a non-blocking handler function. However, in contrast to Fast User-ECalls, they never handle bottom halves or signals, and the invocation of the function is deferred until a later point in time. More specifically, the application can decide *when* the handler function is invoked, on a per-call basis. Possible invocation times are: (a) at return from the next system call, (b) after a certain time delay (in multiples of jiffies, i.e., 10ms on Intel computers), or (c) before the next invocation of the CPU scheduler. The advantage of Deferred User-ECalls is the reduced number of crossings of the user/kernel boundary, particularly when several events cause only one invocation of the handler function. This is useful when kernel extensions want to handle several events at once (*batched events*), or when only the most recent event is of interest to the kernel service, while in both cases a delay in handler execution does not result in a significant performance loss. As an example, an application might want to update QoS attributes in the kernel, but the updated values will not be required until the next invocation of the resource manager.

System Calls. ECalls also offers a generic system call, which takes the unique character string identifying the service as a parameter. ECalls then redirects the system call to the corresponding kernel extension, which executes a system call function (useful when it is not desired to implement new system calls for each new kernel extension).

3.2.2 Kernel Events

A kernel service (e.g., QoS manager, load balancer) raises an event and associates a *deadline* with the event. The event is added to an event queue which is ordered 'earliest deadline first'. Events are taken from the head of the queue and delivered to all subscriber processes

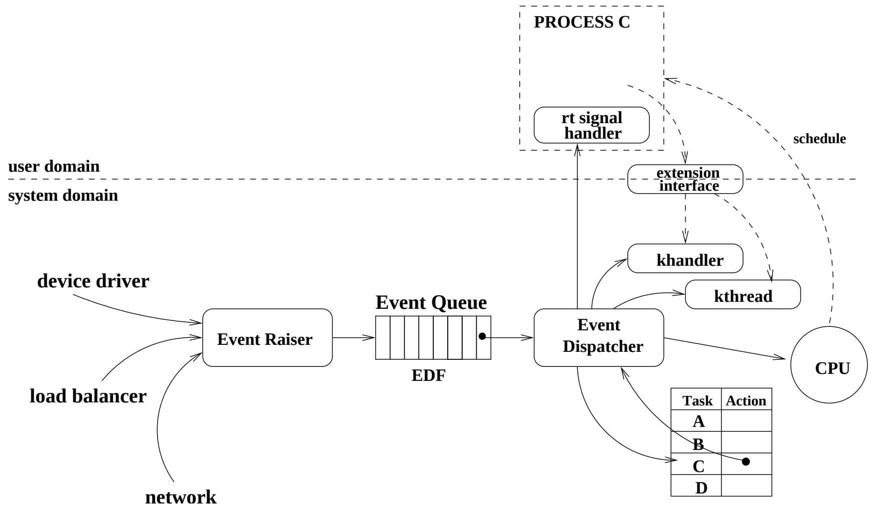


Figure 11: ECalls event delivery architecture.

(see Figure 11). Each process has one or more handler actions associated with an event, e.g., the execution of a kernel event handler, or the raising of a real-time signal. If more than one action has been registered, the actions are performed one-by-one and return values of actions can determine if subsequent steps are skipped. Consider, for example, a kernel service that notifies processes of activity on sockets (i.e., as replacement for *select()* system calls). The first activity could be a kernel event handler that reads the incoming data and copies it into a pre-allocated user-level memory area belonging to the process receiving the event. The second activity is the raising of a real-time signal; upon signal receipt, the process is able to immediately use the received data. In a similar scenario, a kernel event handler could aggregate several events and submit summaries by raising signals as a second step if the number of received events has reached a certain number or after certain time limits have expired (e.g., for periodic collection of monitoring information).

Real-Time Signals. The POSIX.4 standard extends the signal interface with real-time signals. Unlike regular signals, real-time signals are queued and can carry a small amount of data with them. In [17], the authors show that real-time signals are a highly efficient mechanism, providing good throughput compared to *select()* or *poll()* system calls. To ensure predictability and high responsiveness in the dispatching of real-time system calls, the signal handling sequence implemented in Linux 2.4.19 has been modified. In particular, Linux supports 32 regular and 32 real-time signals. The signals are identified by a 64 bit variable, each bit indicating if a signal has been raised or not. In the original implementation, the lower 32 bit are checked first, and if a bit is set, the corresponding signal handler is invoked. However, the lower 32 bit correspond to the regular signals, therefore, regular signals are handled before real-time signals. Keeping in mind that regular signals can be 'caught' and handled by user-specified signal handler code (except SIGKILL and SIGSTOP), this can lead to delays to the invocation of – potentially more important or time-constrained – real-time signals. Therefore, the sequence has been changed to the following order: (a) SIGKILL and SIGSTOP are checked first (the only two signals which can not be caught by the user and which always result in termination or stopping of the process), (b) all real-time signals are checked, with SIGRTMIN having the highest priority and SIGRTMAX having the lowest priority of all real-time signals, and (c) all remaining regular signals are checked and handled if required.

Kernel Event Handlers. Kernel event handlers can be either provided by kernel-loadable modules or dynamically inserted into the kernel by applications themselves. The main advantage of using kernel event handlers on behalf of applications is the minimal overhead of handler invocation, i.e., no cross-domain calls are needed. An example of the use of such an event handler is the recent trend of implementing HTTP accelerators in kernel space. A kernel event handler can be invoked through activity on a socket and as a consequence, the handler reads the request from the socket, analyzes it, and decides whether to service the request directly from within the kernel (e.g., static web requests) or whether to pass the request on to an application-level web server (e.g., dynamic web requests).

Kernel Threads. Kernel event handlers are a powerful and efficient way to handle events. However, if an event is dispatched outside of a process context (e.g., during a timer interrupt) and the event handler would run too long if executed at interrupt context, the event handler can be invoked within the context of a kernel thread provided by ECalls. ECalls maintains a pool of pre-forked threads, where the pool size is dynamically modified as needed.

Shared Memory. The memory areas shared between an application and a kernel service can be used to notify an application of kernel-level events. Here, instead of executing costly system calls to obtain information about kernel-level events, an application can simply scan entries in the shared memory which are modified by the kernel service whenever an event is raised.

Execution Context. If an application is notified of an event through real-time signals or via the pinned shared memory, the event is handled in the context of the application. However, when a kernel thread is executed, the event is handled in the context of that kernel thread. If this kernel thread has to access resources of the application, e.g., file and socket descriptors, certain provisions may be necessary to overcome access restrictions. As example, as part of the ECalls mechanism, the kernel source was modified such that a kernel service is able to access the file descriptors of the applications that registered with this kernel service via ECalls. Further, if kernel event handlers are used, the handler function may be executed in interrupt context. Again, provisions have to be made to ensure that an application's resources can be accessed.

3.2.3 Cross-Domain Data Transfer

During registration of an application with a kernel extension, two memory areas are created and locked into memory, which are used for event notification and the exchange of data between the application and the extension. The reason for using two separate memory areas is to minimize the need for synchronization between application and kernel service. The memory's structure is described in a C header file, and is organized in one of two possible ways. In either way, the first entry is an integer value (called *flag*), which can be modified each time an event is generated or handled. The following code shows the two possible structures of the memory:

```
struct sh_memory {                      struct sh_memory {
  int flag;                                int flag;
  unsigned long bit_pattern[MAX];          int front;
  [data part]                              int back;
};                                         [data part]
                                         };
```

In the first case, an array called *bit_pattern* holds a bit per data entry in the following data part. When an event is generated, *flag* is incremented, the event data is written into the corresponding position in the data part, and the corresponding bit in *bit_pattern* is set. The bit pattern facilitates the search for new events in the memory segment. There are three possibilities to use the memory: (1) each entry in the data part is of the same data type and new event data is put into the next available slot; (2) each entry is of the same data type as in (1), but the position of the new event data denotes its priority and therefore the sequence in which new data is read; and (3) entries in the data part have different data types and new event data is put into the corresponding entry according to its type. In the second approach, the memory segment can be structured as a ring buffer, in that case the *flag* entry is followed by a *front* and a *back* entry, pointing to the beginning and the end of the momentarily used part of the memory segment, respectively. In the case of the ring buffer, new event data is put at the the end of the written part of the memory segment (indicated by *back*) as long as there is sufficient space. In both cases, the size of the data part is determined offline by the developer of the particular kernel service and can not be changed during runtime.

3.2.4 Dynamic Handler Generation

Another functionality of ECalls is its ability to support *dynamic instrumentation* of kernel functionality by allowing applications to insert dynamically generated code into a running kernel. Unlike kernel modules, this feature supports simple functions that can be shipped between systems as strings and translated into native machine code by a low-overhead in-kernel compilation component. These functions are expressed in *E-code*, a C-like language that has been developed as part of the ECho event service [28]. E-code itself is based on Icode, an internal interface developed at MIT as part of the 'C project [106]. For ECalls, the E-code code generator has been ported to the Linux kernel and consists of two loadable modules. Currently, E-code supports the C operators, `for` loops, `if` statements, and `return` statements. While these limitations restrict the capabilities of E-code, they also facilitate the protection from malicious code. However, future work will extend E-code's capabilities (e.g., adding dynamic memory management and pointers), while at the same time adding protection mechanisms. In addition, other efforts [144, 38] have contributed protection mechanisms that can be used in conjunction with ECalls.

In order to generate and install new code in the kernel, an application passes a string carrying the code (e.g., with a system call or via /proc) to the ECalls mechanism, where the code is translated into machine code. When the corresponding kernel service raises an event, ECalls invokes the newly generated kernel event handler on behalf of the application. Figure 12 shows how the code is deployed in ECalls: an application passes the code as string to ECalls, where it is passed to its 'dynamic code generation' component. The string

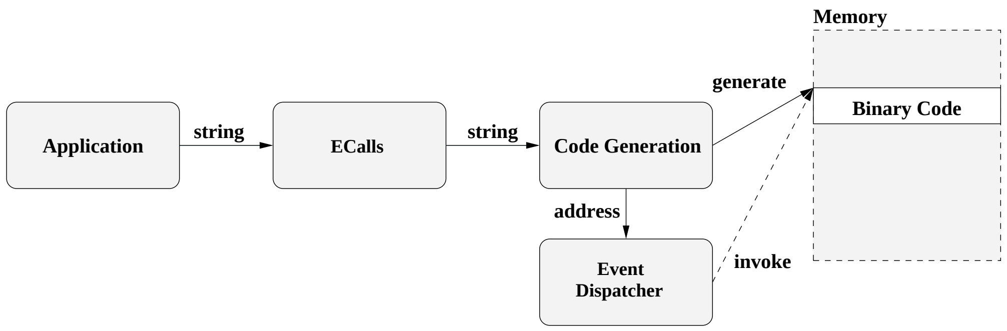


Figure 12: Dynamic code generation.

is parsed and translated into binary code and placed into memory. The memory location of the newly generated binary code is shared with the event dispatcher, which will invoke the new code whenever an event for the corresponding application occurs. The following code shows a simple example of an E-code function, where resource attributes – possibly collected by a resource monitor – are inspected. With E-code, parameters can be passed as basic types (e.g., integer, char) or as structures as shown with *input* in the sample code. Similarly, return results can be basic types or structures (e.g., *output* in the sample code).

```
char * my_code = ‘‘{
    int i;
    int j;
    for (i = 0; i < NUM_RESOURCES; i++) {
        for (j = 0; j < input.resource[i].num_attributes; j++) {
            if (input.resource[i].attribute[j] > input.resource[i].threshold[j])
                output.resource[i].exceeded[j] = 1;
            else
                output.resource[i].exceeded[j] = 0;
        }
    }
}’’
```

In Q-Fabric, ECalls’ code generation functionality is used to ‘download’ application-specific code into Q-Fabric, e.g., to customize the event traffic on Q-Channels, to deploy novel resource or attribute monitors, and to deploy application-specific QoS policies.

3.2.5 Combinations of Event Notifications

Event notification can consist of more than one action, where the first action can modify or even block events and the associated data. Two scenarios are considered: event preprocessing and event filtering.

Event Preprocessing. The different methods of event notification offered by ECalls can be combined, e.g., where the first handler acts as an *event preprocessor*. Here, a kernel event handler can inspect certain data and modify it before a second action is invoked. Consider the following examples:

- A kernel event handler can read service requests from a socket and put them into the shared memory. As a second step, an application is notified of the newly arrived data, either through the shared memory or a real-time signal. However, the application can access the request immediately in the shared memory and service the request.
- A kernel event handler can collect information from different resources in the system and compute averages, which are passed to applications via a second event notification approach (e.g., signals).

Event Filtering. In contrast to event preprocessing, the goal of event filtering is to reduce the overheads for an application by blocking unnecessary information from being passed to user space. Consider the following examples:

- A kernel event handler inspects data received from the network or from an in-kernel resource monitor and decides if the data is of interest to an application. If so, the information is passed to the application, e.g., through the shared memory, otherwise the data is discarded in the kernel.
- An kernel event handler serving as HTTP accelerator can inspect service requests and handles them directly if the requests are for static web pages, otherwise they are passed, as a second step, to a user-level web server (e.g., for dynamic web requests).

3.3 Event-Aware CPU Scheduling

The timely delivery and processing of events is particularly important for time-constrained applications such as multimedia streaming, virtual environments, or interactive distributed simulations. Consider, for instance, a distributed game for which (1) jitter in the replay of media streams should be minimized, and (2) game events like position updates and certain actions of avatars must be delivered in a timely fashion. Techniques like proportional share scheduling of tasks and communications can reduce jitter for continuous media streams. However, the coordination of task scheduling with important game events can further reduce variations in inter-frame times and increase responsiveness to player actions.

To illustrate the performance advantages derived from event-awareness realized with ECalls, consider a distributed video player, which uses *timed waits* to achieve the *inter-frame times* necessary for its desired frame rates. In other words, this application *sleeps* for a certain amount of time, and when it *wakes up*, it is placed back into the run queue of the CPU scheduler. However, the delay between the point when this application becomes schedulable (i.e., wakes up) and when it begins to run (i.e., enters the 'running state') (see Figure 13) varies depending on the scheduling policy implemented, the scheduling attributes assigned to this and other schedulable applications, and the current CPU load. These delays

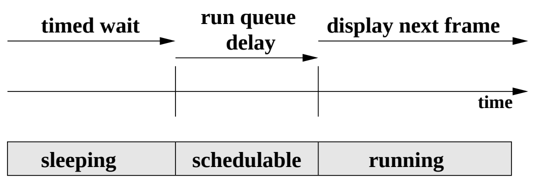


Figure 13: Run queue delays depend on the CPU scheduler, the scheduling attributes of all schedulable applications, and the current CPU load.

– termed *run queue delays* – can increase latencies and jitters for continuous media streams, and they can reduce the responsiveness of real-time applications like distributed games. For instance, when running on a general-purpose operating system like Linux, a single video player can experience significant run queue delays when it has to compete with a second real-time process due to the coarse granularity of the system's time base, which is 10ms on Intel-based Linux systems. When it has to compete with other video players for the same CPU, run queue delays increase substantially, resulting in significant variations in inter-frame times even for a small number of video players, as shown in Figure 14.

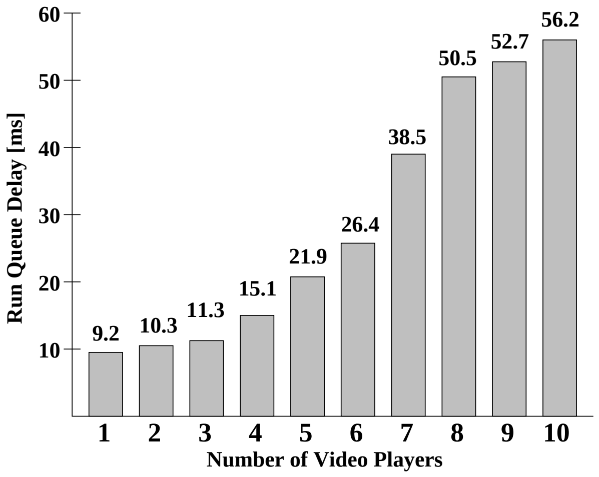


Figure 14: Average run queue delays for a number of video players that have to compete for the CPU with each other and with another real-time process (running in an endless loop).

In the ECalls approach, the arrival of a message at a network connection triggers an ECalls event, where the ECalls mechanism not only notifies the application of the arrival of the event, but also cooperates with the CPU scheduler to ensure the timely delivery of the event.

Most systems deploy CPU schedulers that ignore important application-level events like message arrivals. ECalls offers the basic functionality needed for creating event-aware systems by linking the scheduling of processes with the delivery of events for these processes. The effect is that processes acting as sinks of events are favored over other processes whenever they receive events. Thus, ECalls may be used to implement policies by which applications cooperate with system services like CPU scheduling.

Coordinated scheduling support for processes and events can be implemented for any ECalls-enabled CPU scheduler. The current implementation supports two such schedulers: the traditional UNIX scheduler and the DWCS hard real-time scheduler. Note that event and CPU schedulers are separated, thus permitting the event scheduler to utilize any appropriate CPU scheduler. This is achieved by building ECalls' event scheduler 'on top' of the

CPU scheduler, i.e., ECalls possibly revises the CPU scheduler's decision, without modifications to the actual CPU scheduler implementation. In this chapter, the event-aware task scheduling for the real-time DWCS CPU scheduler is described.

The remainder of this section describes the cooperation between ECalls' event scheduler and DWCS, where the goal is to maximize event responsiveness without compromising the hard real-time guarantees of DWCS.

3.3.1 Event scheduling with DWCS

If ECalls' event queue is non-empty, the event scheduler is invoked each time the CPU scheduler runs. After the CPU scheduler finished the selection of the next process, the event scheduler compares the scheduling attributes of this process with the attributes of the sink process for the first event in the event queue.

Assume that process i is the process selected by DWCS and process j is the sink of the first event on the event queue. The event scheduler applies the following five rules to processes i and j:

Rule 1: If $j = i$ (i.e., DWCS already selected the sink process), the only action the event scheduler has to perform is to remove the event from the event queue.

Rule 2: If task i is a best-effort task, ECalls replaces i by j and removes the event for process j from the event queue. DWCS schedules best-effort processes only if all runnable real-time processes have been serviced within their respective periods and none of them is a work-conserving process. That means further that process j receives an additional time unit in its current period, so that it is able to react to an event immediately. No real-time guarantees are compromised since all real-time processes have been serviced in their corresponding periods.

Rule 3: If process i is a work-conserving process that received at least C_i time units of CPU time in its current period T_i, the event scheduler replaces i with j and removes the event for process j from the event queue. The real-time guarantees of i are not compromised in this case, since process i received C_i time units in its current period already.

Rule 4: Assume that both processes i and j have not been serviced in their current periods yet, and both have the same deadline. Further assume, that DWCS selected process i as the next running process due to its tighter window-constraint compared to process j. ECalls' event scheduler gives process j preference over process i, if this does not lead to a missed deadline for i (i.e., $\Delta t - C_j - C_i > 0$, where Δt is the remaining time in period T_i). In other words, process i will be delayed by C_j, but since its deadline will not expire, DWCS will select this process after process j has exhausted its service time C_j.

Rule 5: In addition to the rules above, the notion of a *task server* is introduced, which is a pseudo process with scheduling attributes determined as follows:

$$x_{ts}/y_{ts} = 0/y_{max}, y_{max} = max\{y_i\} + 1.$$

This assigns the task server the tightest window constraint possible. The service time C_{ts} is the same as the service time of the sink process of the first event in the event queue, or 1 otherwise. The *rest utilization Ur* of the system, which is the *maximum utilization* minus the *current utilization,* is used to determine the value of the period T_{ts}:

$$T_{ts} = C_{ts}/U_r.$$

The attributes for the task server have to be re-calculated when the service time of the first event in the event queue changes (e.g., when the first event has been delivered and the new event at the front of the queue has a different service time). Each time the task server is selected by DWCS, the event scheduler replaces it with the sink of the first event in the event queue. If there are no events pending, a best-effort task can be scheduled instead. The purpose of the task server is to *reserve* the remaining CPU time for processes that have events pending.

Examples for these rules are shown in Figure 15, where 'BE' is a best-effort task and 'TS' is the task server. In each graph, the top part shows the scheduling output of DWCS,

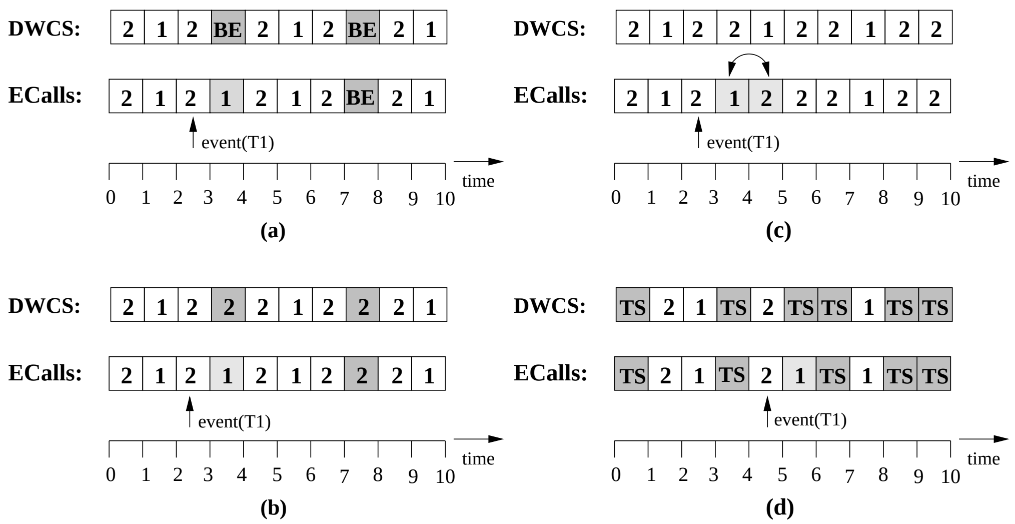


Figure 15: Examples for Rule 2 (a), Rule 3 (b), Rule 4 (c), and Rule 5 (d).

while the bottom part shows the scheduling output revised by ECalls. In graphs (a) and (b), task T_1 has the following attributes: $T = 4$, $C = 1$, $x/y = 1/2$ and task T_2 has the following attributes: $T = 2$, $C = 1$, $x/y = 1/4$. In both cases, T_1 is being notified of an event at time 2.5, however, in (a) both tasks are non-work-conserving, while in (b) they are both work-conserving. In graphs (c) and (d), both tasks have a period $T = 3$ and a service time $C = 1$. T_1's value for x/y is 1/2, while T_2's value for x/y is 1/4. Again, both tasks are work-conserving. In (c), the event is raised at time 2.5, while in (d) the event is raised at time 4.5. Further, in (d), the task server's period is computed as follows: $T = C/U_r = 1/0.58 = 1.7 \Rightarrow T = 2$.

The event scheduler is presented in the following pseudo-code, where i is the process selected by DWCS and j is the sink process for the first event on the event queue:

```
while (TRUE) {
  if (i == j) schedule i;
  else if (i is best-effort task) schedule j;
  else if (i is work-conserving and has been serviced
    in its current period) schedule j;
  else if (deadline(i) = deadline (j) and a delay of i
    does not cause a violation for i) schedule j;
  else if (i is task server) schedule j;
  else schedule i;
}
```

3.4 *Remote Event Notification*

ECalls event notification can operate across multiple machines, i.e., cross-domain event communication is mapped to cross-machine event communication. ECalls offers the ability to (a) deploy event notification mechanisms remotely and (b) redirect event notifications to remote locations. This section addresses these two scenarios.

Remote Handler Deployment. Figure 16 presents the use of the remote handler deployment: an application A (on node A) passes an E-code-based function as string to its local ECalls module. ECalls passes the string to a 'forwarding service', which then forwards the string to one or more remote ECalls modules. An application has the opportunity to

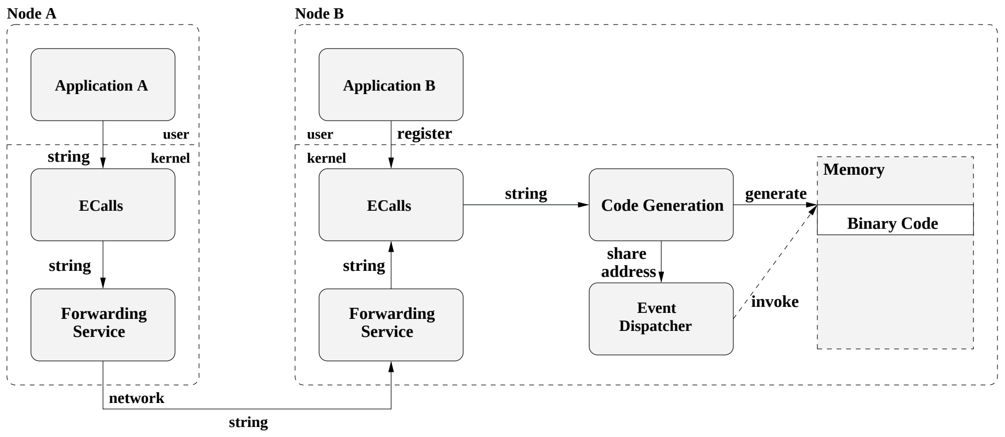


Figure 16: Remote deployment of a kernel event handler.

specify the remote target application (which will receive the events) either directly by its process ID and host name or by a unique name. As an example, applications such as video conferencing or remote teaching typically involve a large number of participants, where these participants obtain information about other participants via a centralized group service. Similarly, ECalls can obtain the host names of all hosts involved in a distributed application from a group service, and the string containing these names is forwarded to all participants. When a string is received by the event service, it is passed to ECalls which then generates binary code as described before. This allows ECalls to distribute new kernel functionality and to deploy event handlers without concern about differences in system architecture. Scenarios where this approach is of use include remote maintenance of systems, remote system optimizations, or remote debugging.

Event Redirection. Similarly, an application can deploy a handler function whose task it is to redirect some or all occurring events of interests to one or more remote sites, via the distributed event service. Figure 17 shows the scenario for event redirection. A kernel

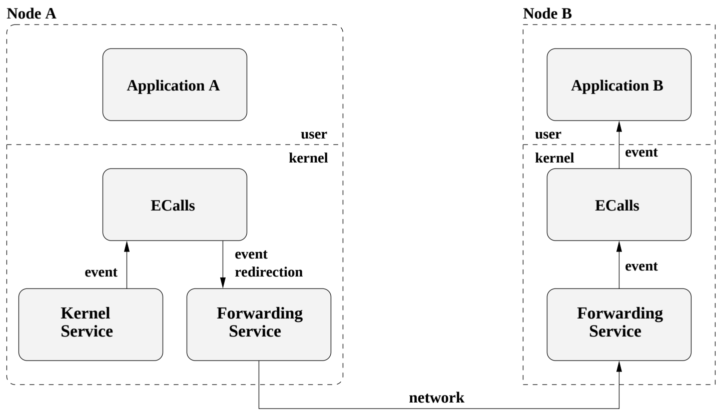


Figure 17: Event redirection.

service on node A issues an event, which is submitted to a kernel event handler by ECalls. The kernel event handler then returns the event to ECalls for transmission to one or more remote systems, e.g., node B. At node B, the event is again passed to ECalls, which then notifies application B of the event. Possible scenarios for the use of this approach include:

- *Remote monitoring:* consider a cluster server, where certain system events, e.g., exceptions or exceeded thresholds for buffer queue fill levels or a processor's heat emission, etc., are redirected to a centralized load balancer, which – based on the collected information – decides to relieve an overloaded server.

- *Remote debugging:* exceptions, failures, and error messages can be collected remotely, where a user or an application can analyze this information, act upon this information (e.g., by migrating essential tasks and data to other hosts), and plan and execute measures to predict and prevent future failures.

- *Intrusion detection:* imagine a kernel-level mechanism that watches file and memory accesses, detects system modifications, or unaccounted use of resources. Events raised by such a kernel-level mechanism can be redirected to a remote – trusted – system, where these events are analyzed and possible strategies for defense are devised.

3.5 Case Studies and Experimental Evaluation

3.5.1 Implementation of an I/O Event Delivery Module

Unix systems provide *select()* and *poll()* system calls, which query a set of file descriptors passed in an array for activity. The system call returns when there is activity in at least one of these descriptors or when the system call times out. The application then has to scan a returned array to find the descriptors that are actually active. Web servers such as Zeus, Flash [96], or thttpd use the 'select' approach, which for thousands of file descriptors does not scale very well. The problem here is that the kernel has to scan the entire array each time a system call is executed.

ECalls is used to implement a scalable I/O event delivery module (called *I/O module* in the following) using the Linux 2.4.19 kernel. Applications register their interest in sockets via the ECalls mechanism. If data arrives at one of these sockets, the registered application will be notified using one or more of the methods described in the sections above. A similar example has been presented in [6], which introduces a scalable event notification mechanism to replace the expensive *select()* system call. To be able to support this notification mechanism, approximately 20 lines of code to the networking code inside the kernel and one additional entry into the *sock* structure have been added, the latter being a flag that can be used by the socket owner to express interest in event notification if socket activity is monitored.

When the I/O module detects activity on one of the monitored sockets (Figure 26), it generates an event for the application owning this socket. Each application has two data structures, which are both locked into memory and shared between the application and the event delivery mechanism. The first data structure has the following entries:

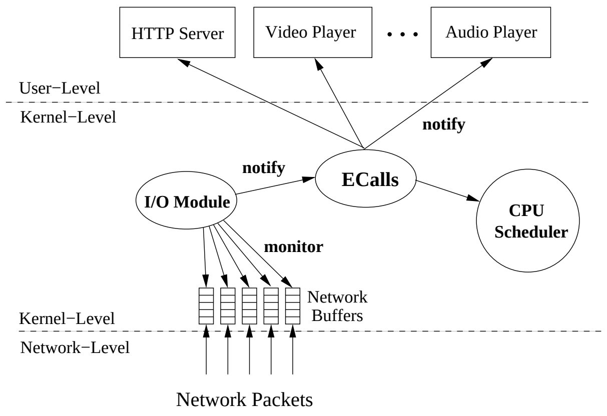


Figure 18: Notification of socket activity using the I/O event delivery module and the ECalls interface.

```
struct socket_interest {
  int flag;
  unsigned long fd_list[MAX];
  unsigned long updated_fd_list[MAX];
};
```

The first entry, called *flag*, is incremented each time the application submits a change of interest. The next entry, called *fd_list*, is an array used to indicate which file descriptors the application has registered for, each bit in *fd_list* corresponds to a file descriptor. The second array, called *updated_fd_list*, is used to indicate the changes in *fd_list* since the last time the registration module read from this data structure. Its purpose is to accelerate the registration process.

The second data structure looks as follows:

```
struct socket_ready {
  int flag;
  unsigned long fd_active[MAX];
};
```

The value of *flag* indicates how many sockets are active, i.e., how many sockets have data in the receive buffer. The actual file descriptors for the active sockets can be found in the next entry, called *fd_active*.

3.5.2 Experiments with a Distributed Video Player

ECalls provides a flexible mechanism for coordination and information sharing for real-time and multimedia applications that use certain kernel services or that extend kernels with application-specific functions. A distributed MPEG video player has been modified such that it uses ECalls to communicate with the I/O module described above. In this experiment, a number of video players (running on a Pentium II with 450MHz and 512MB RAM) request video streams from several video servers (running on five Ultra 30 with 248MHz and 128MB RAM each).

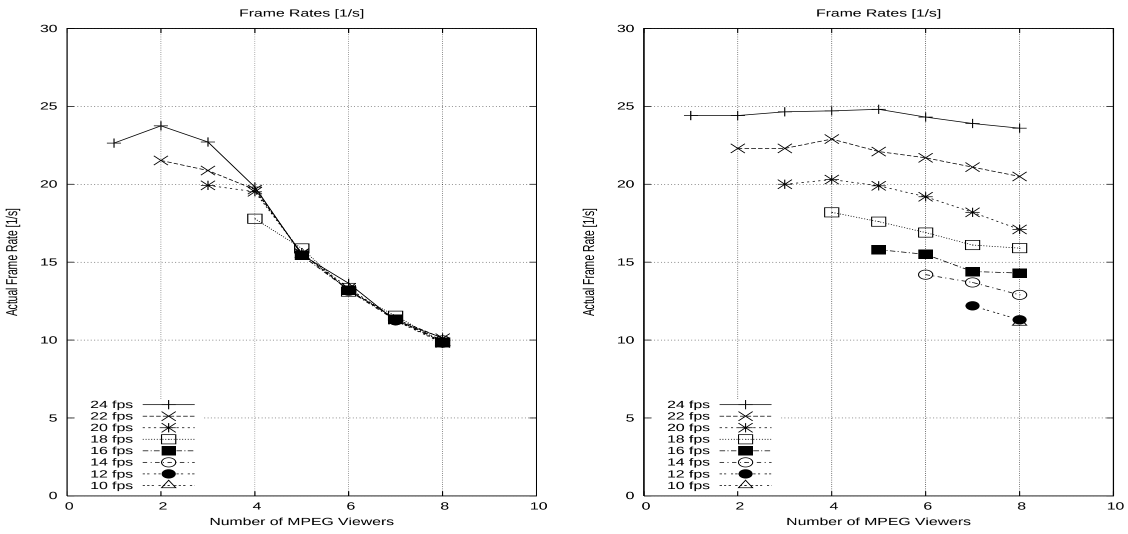


Figure 19: Achieved frame rates without ECalls (left) and with ECalls (right).

Each video player writes the *desired frame rate*, a *frame counter*, and the *time stamp* of the last displayed frame into the pinned memory supplied by ECalls. The frame rate can be changed dynamically if desired (e.g., as image resolution or compression changes). The I/O module uses this information to compute the display time of the next frame. Incoming frames are monitored by the I/O module, and ECalls places the notification events into the event queue ordered by the display time of the next frame. The DWCS CPU scheduler uses this information to modify the scheduling priority of the video players. In this experiment all players have the same attributes of $x/y = 1/5$ and service time $C = 10ms$, and a priority T corresponding to the desired frame rate, e.g., for a frame rate of 10 fps, $T = 100ms$. This experiment shows that the interaction between an application and a kernel-level service (using ECalls) allows the application to achieve its desired QoS, even when the host is perturbed by several CPU-intensive tasks. Figure 19 (left) shows that the achieved frame rates drop rapidly when the number of players increases. Using ECalls, one is able to maintain frame rates close to the desired frame rates (Figure 19 (right)). ECalls achieves that by delaying event notification for a period of time determined by the frame rate and by influencing the scheduling decisions such that the scheduler reorders the run queue to favor applications receiving these events.

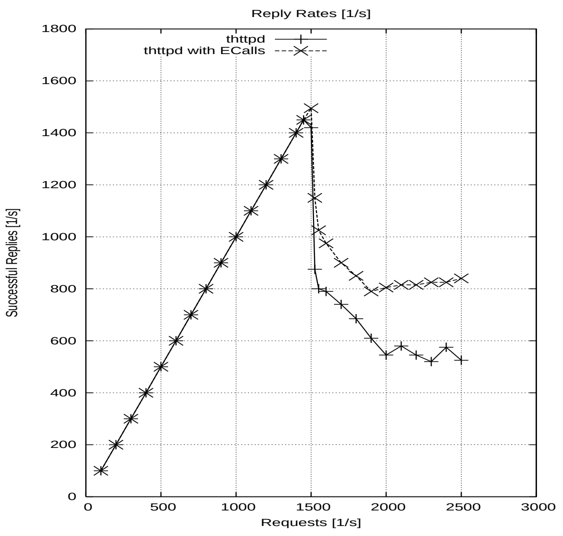
3.5.3 Experiments with a Web Server

The next experiments investigate the performance changes in a web server running on top of ECalls. Here, thttpd[3], a small and fast single-process event-driven web server, has been modified such that it uses the I/O event delivery module described above. The thttpd web server uses the *select()* system call for all HTTP requests. Further, the API has been modified such that thttpd subscribes every new incoming request with the I/O event delivery module and instead of a *select()* call, thttpd continues to service requests and selects the next connection to service via the *fd_active* array.

The client requests are generated using the httperf Version 0.8 [89] performance tool. The HTTP server is a Pentium II with 450 MHz and 512MB RAM, running the modified thttpd application. The client machines are five Sun Ultra 30 with a 248MHz processor and 128MB RAM each. The machines are connected via a switched 100Mbps Ethernet.

In this experiment, the clients request a small static web page for a duration of 180s, each request with a timeout value of 1s. Figure 20 (left) shows the achieved reply rates with both the original thttpd server and the modified server using ECalls (thttpd-ECalls). The web server thttpd is highly optimized, so that the difference in performance for small loads is irrelevant, as evident from the graph. On the other hand, in the case of overload, thttpd displays poor behavior: its reply rate drops to 25% at request rates of 2000 per second. The reply rate for thttpd with ECalls also decreases in the case of overload, but less dramatically (e.g., to 40% at a request rate of 2000 per second).

[3]http://www.acme.com/software/thttpd



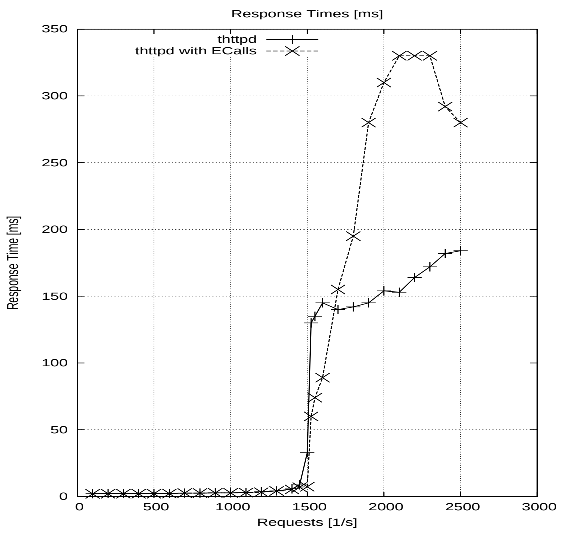


Figure 20: Reply rates (left) and response times (right) for both the thttpd web server and the modified thttpd web server using ECalls and the I/O event delivery module.

Figure 20 (right) shows the average response times for thttpd and thttpd-ECalls. Here, the response times of thttpd settle at approximately 150ms when the server is overloaded, whereas in the case of thttpd-ECalls the response time increases until it settles at approximately 300ms. The reason for this behavior is the higher reply rate of thttpd-ECalls, i.e., thttpd-ECalls is able to respond to more requests than thttpd, resulting in higher response times.

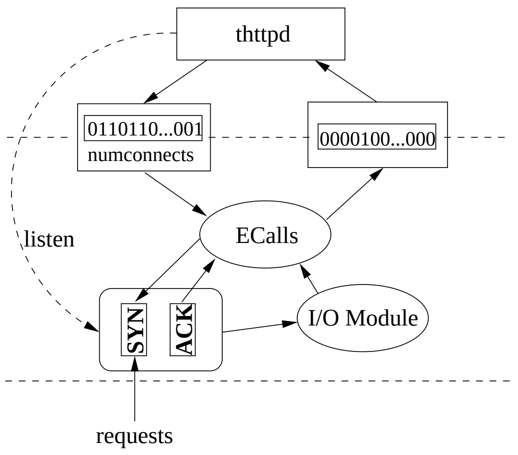
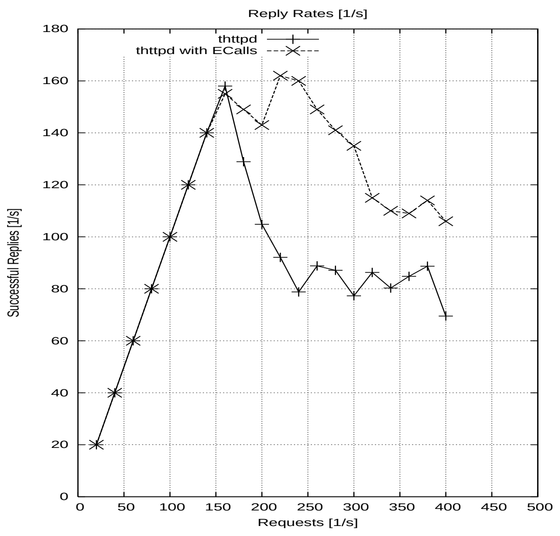


Figure 21: Modified thttpd web server using ECalls and the I/O module: ECalls monitors both the number of open connections (*numconnects*) and the buffer fill level of the listen queue with completed requests (ACK-queue) to determine the size of the listen queue with incomplete requests (SYN-queue).

The next experiment investigates whether the use of ECalls can improve the overload behavior shown above. In [108, 109], the authors analyzed the overload behavior for the thttpd and phhttpd web servers. In [108], real-time signals are used to notify the web server of new requests, and the number of signals and therefore, the number of pending requests is used as indicator for server overload. The overload behavior shown in Figure 20 is referred to as *receive livelock*; in [85], the authors suggest to drop requests as early as possible to achieve more request completions. While in [108] requests are dropped by the server if overload is detected, ECalls drops requests early in the kernel. That is, the server is flooded with requests, this time with a more complex web page, and monitor overload behavior, but then the overload behavior is improved by using the ability of ECalls to cheaply exchange information between user- and kernel-level. Specifically, the web server continuously updates a new variable, *numconnects*, in the memory segment, telling ECalls the current number of open connections being serviced by the server. In addition, ECalls monitors the buffer fill level of the *completed connection queue* of the listening socket (ACK-queue in Figure 21). If both values (number of connections and buffer fill level) are above a certain threshold, ECalls reduces the buffer length of the *incomplete connection queue* (SYN-queue in Figure 21), until either the number of connections or the number of accepted requests drops under their respective thresholds. The reason why two criteria are used is to prevent ECalls from decreasing the queue size in case of transient overloads. As an example, if the ACK-queue is above the threshold but the number of open connections is under its threshold, it is assumed that the server will soon be able to service this burst of requests. On the other hand, if the number of connections is over the threshold, but the buffer fill level is under its threshold, it is assumed that the small number of pending requests will reduce the server load soon such that it is not necessary to drop requests.

Both graphs in Figure 22 show the results of this experiment. ECalls is able to improve the overload case such that more replies are generated and at the same time average response times are reduced. As an example, Figure 22 (left) shows that thttpd is able to service 80 requests per second at a request rate of 300 per second, while thttpd using ECalls is able to service 140 requests per second. Figure 22 (right) indicates a 250% better



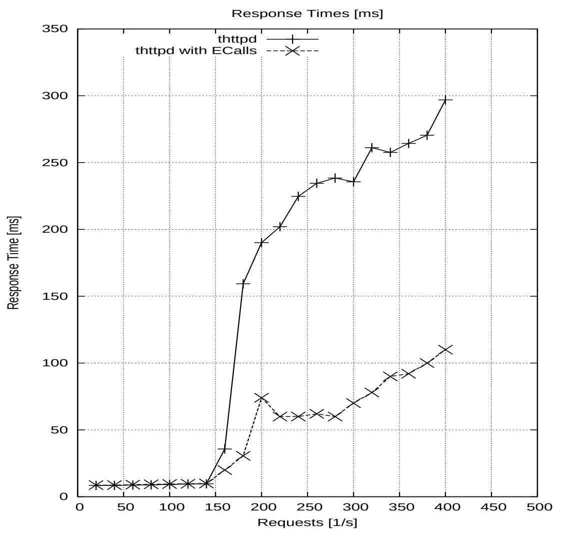


Figure 22: This graph shows the improved overload behavior of the web server when ECalls modifies the size of the *incomplete connection queue* of the listening socket according to the load (left) and how this adaptive approach also improves the response time of requests (right).

response times than the original thttpd web server. The values for the thresholds have been determined via experimentation, and could also be made adaptive instead of fixed as done in this experiment. Also, more knowledge from both kernel-level and user-level can help improve the overload case even more (such as the average response time measured in the web server or the type of a request).

3.5.4 Resource Monitoring and Management

Monitoring of system resources is essential to resource and QoS management techniques; the information collected is analyzed and possible resource re-allocations are planned and executed. In distributed systems, such as cluster servers, parallel systems, multimedia streaming applications, etc., the monitored information has to be transmitted to either a centralized resource manager or to all other nodes in the system if a decentralized approach is chosen (as in Q-Fabric). In either case, the frequency of transmission and the amount of data transmitted determines both the overheads introduced by the monitoring and the quality of resource management possible. Insufficient information, for example, can prevent intelligent resource adaptations; on the other hand, information that is not needed only increases the overhead. Infrequent transmissions force the resource manager to use stale information or prevent it from reacting to resource shortages quickly, while too frequent transmissions may be undesired because of the overheads associated with them. In other words, the type of information transmitted and the frequency of transmissions depends on what applications are running, which resources are used and controlled, and what is the desirable granularity of resource adaptations.

In this experiment, a distributed monitoring system that periodically exchanges relevant resource information is used. A resource monitor resides in each node, inspecting resource usage with a poll period. After this information is collected, an event is raised. A remote centralized resource manager previously installed a kernel event handler in each node, where the event handler has two tasks: (a) to filter and aggregate the collected information according to the needs of the resource manager and (b) to redirect the event to the centralized resource manager. The monitored information in this example consists of the CPU scheduler's run queue length and the number of open and pending service requests (obtained from the activity at the sockets). A centralized load balancer uses the monitored informa-

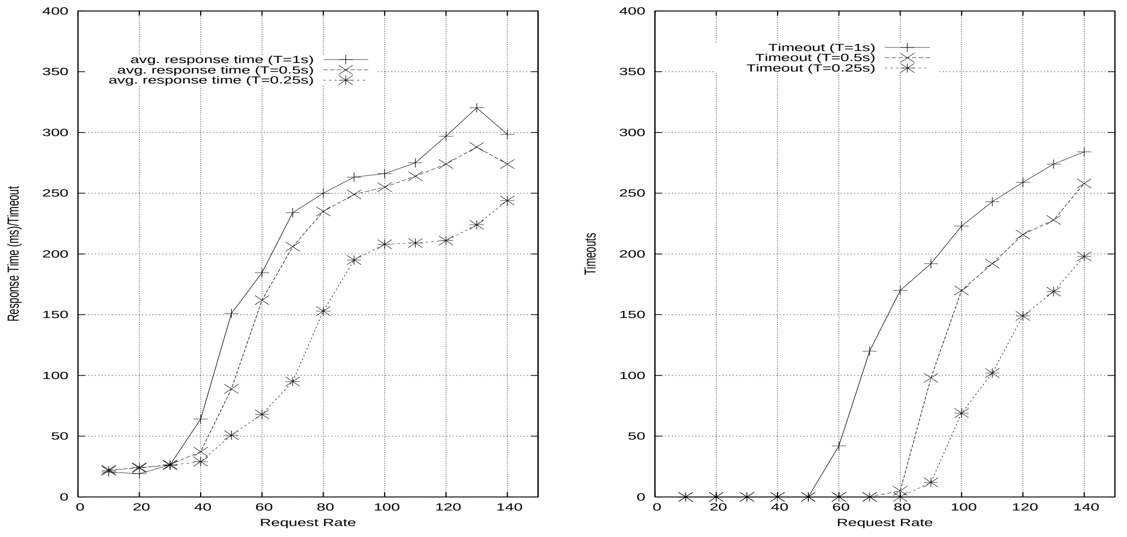
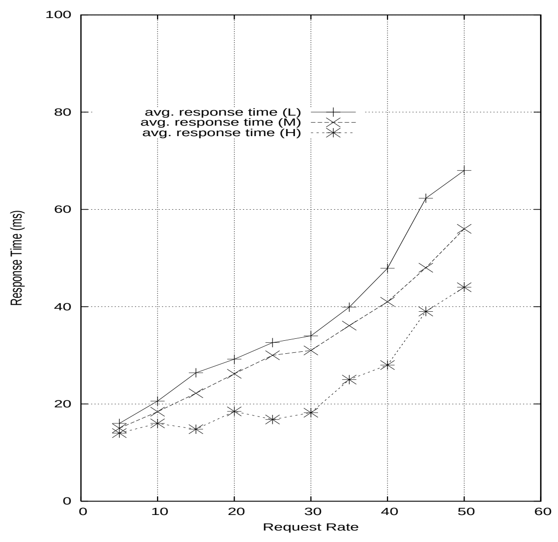


Figure 23: Average response times (left) and number of timeouts (right) for different poll periods.

tion to distribute service requests among a number of back-end servers. This experiment is performed on a cluster of 8 Pentium Pros with 200MHz processing speed and 512MB



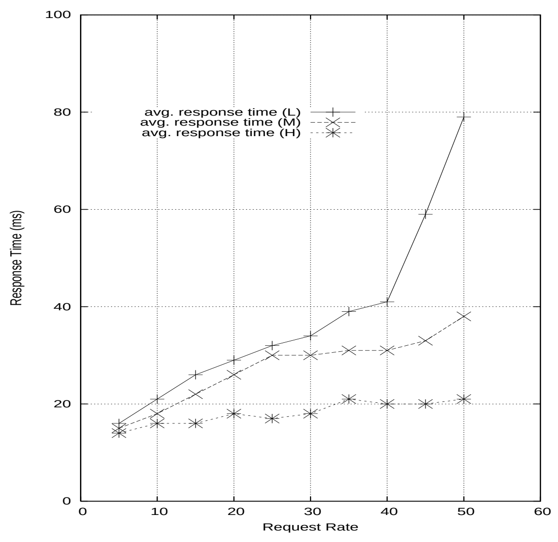


Figure 24: Resource management with static CPU priorities (left) and dynamic CPU priorities (right).

RAM each, connected via a switched 100MBit Ethernet. As before, service requests are generated using the httperf benchmarking tool and all requests have a timeout value of 1s. Figure 23 compares the response times and the number of timeouts for a back-end server when the monitoring frequency is varied between 1s, 0.5s, and 0.25s. Both response times and timeouts improve significantly with increasing monitoring frequencies, and therefore fresher information. However, increasing the frequency also increases the overheads caused by resource monitoring, e.g., in the given example, the cost of redirecting an event to the distributed event service is $80\mu s$ and the cost of handling an event is $250\mu s$. Next, the resource information is used to provide differentiated service to service requests. Requests are categorized into three classes: low priority (L), medium priority (M), and high priority (H). Figure 24 (left) shows the average response times for each individual class when server threads are given static CPU scheduler priorities appropriate to the class of service handled. This approach is then extended such that a kernel-level resource monitor watches the socket activity and determines the rate at which requests are being handled for different classes. This is being achieved by using separate listening sockets for each request class. The resource monitor periodically raises an event which is then caught by a E-code-based kernel event handler. The kernel event handler increases the priority of all threads in a class if the rate of responses is lower than the rate of requests for this class. However, it also ensures that high priority requests are given higher priority than low priority requests. This ensures that dynamic variations, e.g., bursts of requests within a class, etc., are considered and acted upon, resulting in the improved response times shown in Figure 24 (right).

3.6 Summary

The efficient implementation of QoS-aware applications requires that the underlying operating system supports these applications' needs for timely delivery of information. In particular, the existing interfaces between applications and system-level services (such as resource managers) are restrictive and expensive. The ECalls mechanism supports the efficient communication across protection boundaries with a variety of features: (a) communication is based on event channels, where deadlines can be associated with events, (b) besides standard call mechanisms such as signals, ECalls uses shared memory between applications and kernel services for low-overhead event notification and data sharing, (c) ECalls supports the linking of event dispatching with CPU scheduling, therefore maximizing the responsiveness of applications to critical events, (d) multiple event handling approaches can be combined in order to support 'filters', or 'optimistic' event handlers, and finally (e) applications can be notified of 'remote' events by extending the cross-domain approach of event notification to a cross-machine approach. Particularly, kernel event handlers are a powerful mechanism, because (a) they offer minimal overheads for event handling and (b) they can be dynamically generated and inserted into a running kernel by an application, both in local and remote systems. The experiments shown in this chapter with a video player application and a web server show the utility and performance improvements achievable with ECalls. In the context of the Q-Fabric architecture, ECalls is used to link user-level applications and QoS managers with system-level resource managers and monitors. The use of ECalls ensures that the cooperative or integrated management of system- and user-level entities (such as CPU schedulers, network protocols, or applications) is performed efficiently and that relevant events (e.g., exceeded thresholds) are shared in a timely manner.

CHAPTER 4

CROSS-DEVICE INTEGRATION

Q-Fabric relies on *Q-Channels* for the integration in the 'horizontal' direction, i.e., between multiple hosts. Similar to the integration in the vertical direction provided by ECalls (Chapter 3), Q-Channels rely on events and event channels for the communication and coordination among multiple resource managers and monitors. Q-Channels themselves rely on *KECho*, a kernel-level publish/subscribe mechanism for the run-time coordination among distributed kernel services, such as resource monitors and resource controllers. Based on this event communication, QoS management in Q-Fabric can be extended to any number of hosts, i.e., monitoring and control events can be shared efficiently among a group of distributed system-level and user-level QoS management mechanisms.

4.1 Introduction

The need to offer high or predictable levels of performance, especially in distributed and embedded systems, has resulted in the kernel-level implementation of certain applications and services. Examples include the in-kernel web servers khttpd and tux on Linux, kernel-level QoS management and resource management mechanisms [100], and load balancing algorithms [9]. To attain desired gains in predictable performance, distributed kernel-level extensions must coordinate their operation. For example, for load balancing, multiple machines in a web server cluster must not only exchange information about their respective CPU and device loads (e.g., disks), but must also be able to forward requests to each other without undue involvement of clients and forwarding engines [4]. Similarly, to ensure the timely execution of pipelined sensor or display processing applications in embedded systems, hosts must not only share detailed information on their respective CPU schedules and the operation of the communication links they share [116, 134], but they must also coordinate the ways in which they allocate resources to pipelined tasks. Finally, the run-time coordination among kernel-level services illustrated above is highly dynamic, involving only those kernel services and machines that currently conduct a shared application-level task. In addition, the extent of such cooperation strongly depends on the application-level quality criteria being sought, ranging from simply 'better performance' to strong properties like 'deadline guarantees.'

This chapter presents KECho, a kernel-level publish/subscribe mechanism for the run-time coordination among distributed kernel services, such as resource monitors and resource controllers. KECho is the tool used to implement Q-Fabric's Q-Channels, using which any number of kernel-level services, such as resource managers, residing on multiple hosts can dynamically join and leave a group of information-sharing, cooperating hosts. Using KECho, services can exchange resource information, share resources (e.g., via request forwarding), and coordinate their operation to meet desired QoS guarantees. KECho uses anonymous event-based notification and data exchange, thereby contrasting it to lower-level mechanisms like kernel-to-kernel socket communications, RPC [11], or the RPC-like active messaging developed in previous work [140]. Furthermore, compared to object-based kernel interactions [47] or to the way in which distributed CORBA, DCOM, or Java objects interact at the user level [94, 12, 150], KECho's model of communication provides improved flexibility, since its use of anonymous event notification permits services to interact without explicit knowledge of each others identities.

The KECho kernel-level publish/subscribe mechanism shares several important attributes with its user-level counterparts. First, KECho events may be used to notify interested subscribers of internal changes of system state or of external changes captured by the system [77]. Second, it may be used to implement kernel-level coordination among distributed services, perhaps even to complement the application-level coordination implemented with user-level event notification architectures [119, 27, 48, 77]. Applications constructed with event-based architectures include peer-to-peer applications like distributed virtual environments, collaborative tools, multi-player games, and certain real-time control systems.

Third, KECho's functionality is in part identical to that of known user-level event systems, which means that in this document it is described using interchangeable terms like *event notification mechanism*, *event service*, and *publish/subscribe mechanism*. Further, KECho's event services faithfully implement the publish/subscribe paradigm, where events are sent by publishers (or *sources*) directly to all subscribers (or *sinks*). Channel members are anonymous, which implies that members are freed from the necessity to *learn* about dynamically joining and leaving members. Application-level counterparts to such functionality typically require additional kernel calls and inter-machine communications, and they may even require the implementation of extensions to existing user/kernel interfaces, so that applications can gather the resource information they need from their respective operating system kernels. In contrast, the kernel-level solutions to distributed resource management enabled by KECho can access any kernel or network service and any kernel data structures without restrictions, which is particularly important for fine-grained resource monitoring or control. Finally, KECho can also be used directly by applications, thereby permitting them to directly interact with their distributed components.

4.2 Kernel Event Channels

Event notification systems have been used in applications including virtual environments, scientific computing, and real-time control. Compared to user-level implementations of event services, the advantages of a kernel-level implementation include:

- *Performance:* each call to a user-level function of the event system (e.g., residing in statically or dynamically linked libraries associated with the application) can internally result in a high number of system calls. These calls can block, thereby delaying an application and causing unpredictable application behavior. By using a kernel-based service, one can significantly reduce both the number of system calls used in its implementation and the effects on predictability of its execution. Furthermore, if the application components using event services are implemented entirely within the kernel, then no system calls are required at all, and performance is improved further by minimized blocking delays within the kernel. Specifically, a kernel-thread waiting for an event can be invoked immediately after the event occurs, while a user-level application may suffer further delays by waiting in the CPU scheduler's run queue for a time period dependent on its scheduling priority and the current system load.

- *Functionality:* an increasing number of services is being implemented inside of an operating system's kernel, mainly for performance reasons. Only a direct, kernel-to-kernel connection of such services without the additional overheads of user/kernel crossings allows for fine-grained and direct communication and coordination among remote kernel services.

- *Accessibility of resources:* typical user/kernel interfaces restrict the number and type of kernel resources that can be accessed. Kernel-based implementations have no restrictions regarding the access to such resources, that is, resources and kernel data structures (e.g., task structures, file structures) can be accessed and used directly (however, a QoS-developer still has to consider the careful use of atomic locks to access the resources, if required). Using a system-level approach allows for 'smarter' decisions compared to user-level solutions.

4.2.1 Architecture of KECho

The goal of a kernel-based event service is to support the coordination and communication among distributed operating system services. Figure 25 shows the use of KECho, where both kernel- and user-level OS services and user-level applications can dynamically create and open event channels, subscribe to these channels as publishers and subscribers, and then submit and receive events. Although the event channel in Figure 25 is depicted as a logically centralized element, it is a distributed entity in practice, where channel members are connected via direct communication links. The channel creator has a prominent role in these communications only in that it serves as the contact point for anyone wishing to join or leave a group. Any number of kernel services can subscribe to an event channel, and events can be typed, the latter meaning that only events that fit a certain description will be forwarded to subscribers.

The implementation of KECho is based on its user-level counterpart, called *ECho* [27],

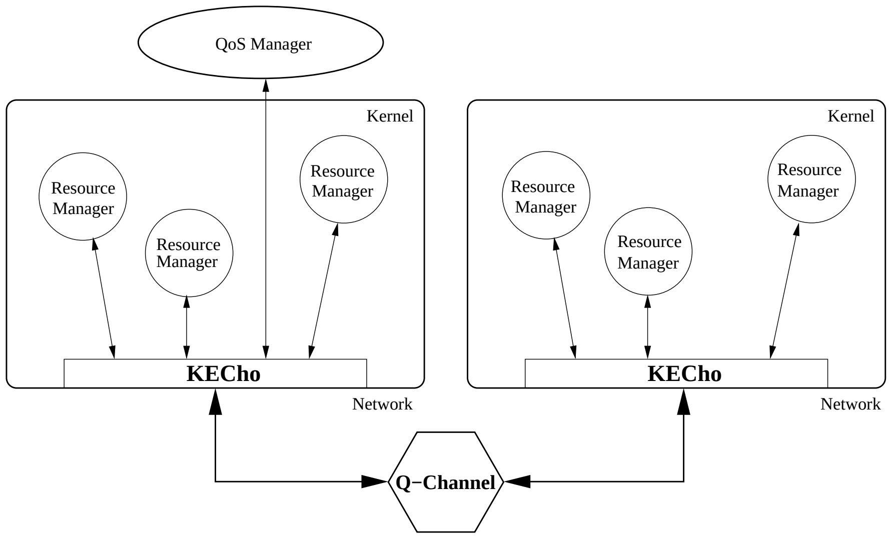


Figure 25: A KECho-based Q-Channel.

the libraries of which have been ported to a number of kernel-loadable modules, each with a certain task:

- *KECho Module:* the main interface to kernel services for channel management and event submission/handling.
- *Group Manager Module:* a user-level *group server*, running on a publicized host, serves as channel registry, where channel creators store their contact information and channel subscribers can retrieve this information. This module supports the communication among subscribers and the group server.
- *Communication Manager Module (CM):* this module is responsible for the connection management, including creating and operating the connections between remote and local channel members.
- *Attribute List Module:* this module implements attributes, which are name-value pairs with which performance or QoS information may be piggybacked onto events.
- *Network Monitoring Module (NW-MON):* this module monitors socket activity and notifies the CM module of newly arrived data at any of the sockets associated with an event channel.

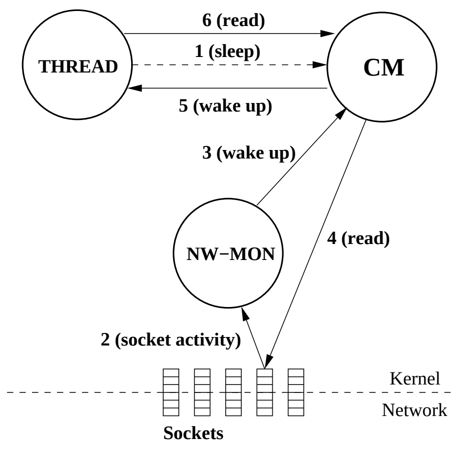


Figure 26: Event delivery in KECho.

4.2.2 Event Delivery

The lowest module in the KECho module stack, the network monitoring module (NW-MON), allows KECho to register *interest* in certain sockets. Specifically, KECho registers interest in all sockets associated to event channels. NW-MON then will be notified by the network interrupt handler once data arrives at one of these sockets. In return, this module then notifies the CM module of this event.

As an example, a subscriber waits for a new event (step 1 in Figure 26) by sleeping or blocking. Activity of a socket related to an event channel (step 2) prompts the network monitoring module to send a *wake-up* call to the CM module (step 3). CM then reads the data from the socket (step 4) and identifies and notifies (step 5) the thread owning this socket. Finally, the thread can now copy the received data from the CM module (step 6) and act upon this event.

While CM awakens and notifies waiting threads about the arrival of events, it can also *accelerate* event responsiveness by increasing the CPU scheduling priority of the process receiving an event. This is part of the ECalls module and is described in more detail in Chapter 3.

4.2.3 Filtering

Most event systems offer the possibility to limit the number of events received through *event filters*. Filters can be placed at either the event sink or the event source and can significantly reduce event processing and network overheads. The most basic filters ensure that events are delivered only if they are of certain *types*. Typical event systems allow those filters to base their decisions only on a per-connection basis, where a filter makes its decision without considering the overall channel condition. In addition to event filters, KECho offers so-called *channel filters*, which (1) can be dynamically inserted by the event source and (2) can decide on a per-channel basis which sinks will receive an event, that is, filtering decisions are based on information collected from the publishers and subscribers via separate event channels or via attributes piggybacked onto events. As an example consider the task of load balancing. Here, a service request from a machine in a web server cluster is forwarded to an event channel if the local server is not able to service this request. A filtering function can collect load information from all other servers and then decide which other server will receive the event carrying the forwarded request. Alternatively, if load information is outdated and requests are idempotent, then the quality of load balancing can be improved by simultaneously forwarding the request to n servers, where n is chosen by the event source. Upon delivery of the event to the n best servers (e.g, the servers with the lightest loads) and completed event handling, duplicate responses can be discarded by the load balancing mechanism. In this example, the event source supplies the number of desired recipients of a forwarded request and all event sinks supply their current load information.

A filter can also be applied to incoming events, in which case it is simply invoked each time an event arrives at the channel. For example, such a filter can decide – based on information from the event source and from all sinks – to which sinks the event will be dispatched. In the load balancing example mentioned above, this kind of filter could make sure that the response to a request is being returned to only the one sink that issued the original request, or it could block multiple responses to the same request. A kernel

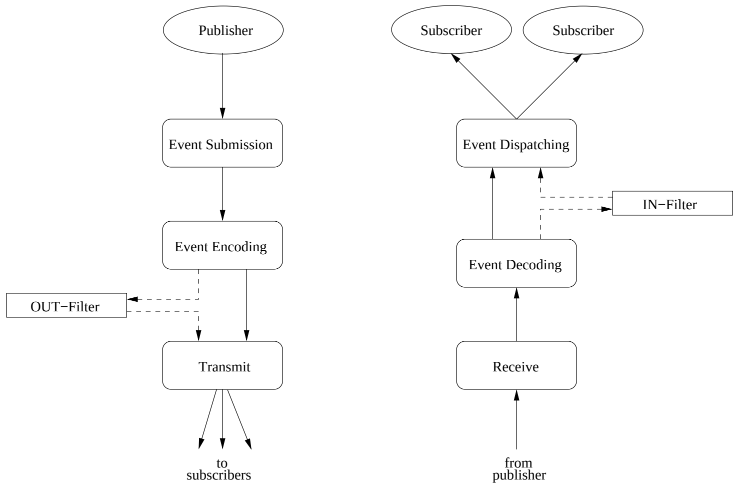


Figure 27: Channel filters in KECho.

service can register two filter functions with an event channel, an IN-filter and an OUT-filter (Figure 27). An IN-filter is invoked each time an event is being received by KECho. The IN-filter is able to investigate the event before it is being dispatched to the event sinks. On the other hand, an OUT-filter is being invoked each time an event is being submitted by a local event source. Again, the filter inspects the event and can decide which remote sinks will ultimately receive the event.

4.3 Resource Management with KECho

Applications rely on the availability of certain *system resources* in order to perform their tasks successfully. System resources can include processing power, network bandwidth, disk bandwidth, RAM, and input/output devices such as cameras or printers. Resource management systems [61, 35] have the task to allow applications to discover, allocate, and monitor such distributed resources. This task is made difficult by (i) the dynamic behavior of resources (i.e., resources can join and leave at any time), (ii) the dynamic arrival and departure of application components requiring resources (e.g., through process migration), and (iii) run-time variations in the current resources required by an application.

Figure 28 shows how KECho connects resource managers to facilitate the task of locating and acquiring resources for applications. Kernels I and II have 3 resp. 2 resources that are shared with other hosts, e.g., CPU, disk, and network resources. As an alternative, a kernel

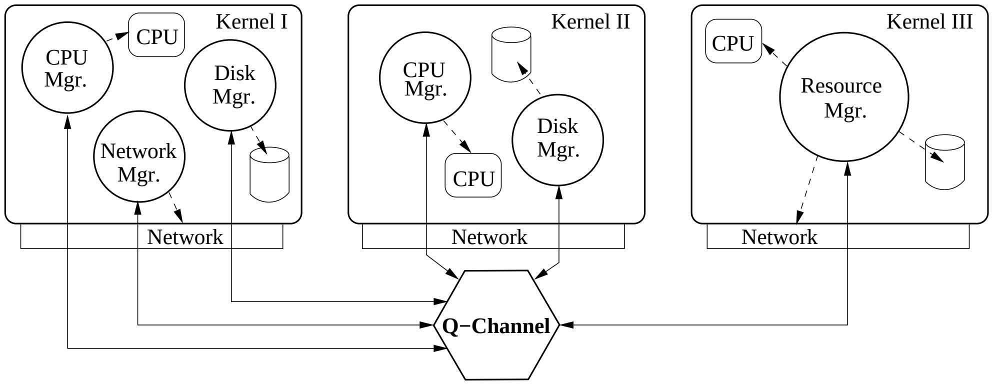


Figure 28: Resource management with KECho.

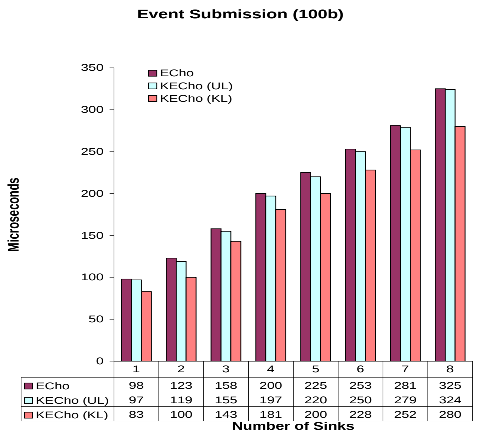
could have only one resource manager, which assumes the task of managing all available resources at a host, as shown in kernel III. In both cases, resource managers can forward requests for resource allocations from applications to other, remote resource managers by submitting an event. If a remote resource manager can fulfill the request, it responds accordingly to the manager that forwarded the original request. If there are several positive responses, a resource manager can use certain criteria (e.g., response times, location of the resource) to decide which response to accept or discard. Resource managers can dynamically join or leave resource-sharing groups, by joining a group it makes its resources publicly available to all other members in the group. However, all managers are unaware of the number or the location of other group members and resource requests are submitted and accepted/denied via events.

4.4 Microbenchmarks

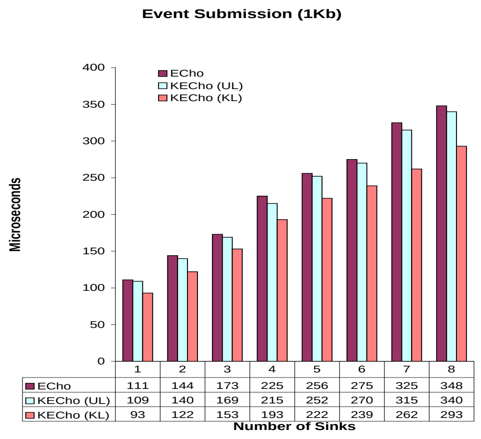
The following microbenchmarks have been performed on a dual-Pentium III with 2x800MHz, and 1GB RAM. The intent is to investigate the overheads associated with event submission and delivery, channel management, and filtering.

4.4.1 Event submission

The first measurement compares the event submission overheads of the user-level implementation of event channels (ECho), the kernel-level event channels used by a user-level application (KECho-UL), and the kernel-level event channels used by a kernel-thread (KECho-KL).



	1	2	3	4	5	6	7	8
▨ ECho	98	123	158	200	225	253	281	325
▢ KECho (UL)	97	119	155	197	220	250	279	324
▨ KECho (KL)	83	100	143	181	200	228	252	280



	1	2	3	4	5	6	7	8
▨ ECho	111	144	173	225	256	275	325	348
▢ KECho (UL)	109	140	169	215	252	270	315	340
▨ KECho (KL)	93	122	153	193	222	239	262	293

Figure 29: Event submission overheads for data sizes of 100 bytes (left) and 1 Kbyte (right).

The graphs in Figure 29 compare the event submission overheads of these three scenarios for 100b and 1Kbyte, where the overheads of ECho and KECho-UL differ only minimally. This can be explained by the fact that ECho uses only two system calls per sink for the submission of an event, where KECho requires also two system calls, but that number is independent from the number of sinks. Event submissions with KECho-KL show up to 15% (for 100b) and up to 20% (for 1Kb) less overhead compared to ECho.

Table 2: Overheads and number of system calls.

	ECho	KECho-UL	KECho-KL
Channel Creation	850μs (56)	182μs (5)	170μs (-)
Channel Opening	approx. 1.5s (117)	approx. 1.5s (5)	approx. 1.5s (-)
Event Submission	100μs (2 per sink)	95μs (2)	85μs (-)
Event Polling	32μs (4)	40μs (2)	5μs (-)

Table 2 compares the performance of some of the functionality of KECho (KECho-UL/KECho-KL) with the performance of the user-level implementation ECho. Channel creation requires 850μs in ECho, compared to 182μs in KECho-UL and 170μs in KECho-KL. The large difference between kernel-level and user-level approach can be explained by the number of system calls required for the creation of a channel in ECho, which is 56, compared to 5 in KECho-UL. The opening of a channel depends on the current number of subscribers, the network transmission delays and other factors, however, typical values for this operation are approximately 1.5s in all three cases. Event submission takes about 100μs per event subscriber for ECho, compared to 95μs and 85μs for KECho-UL and KECho-UL, respectively. In ECho, the overhead for polling for new events is 32μs (4 system calls) compared to 40μs (2 system calls) in KECho-UL. The reason for this increase are some inefficiencies in the implementation which will be addressed in the future work. However, the overhead for event polling in KECho-KL decreases to only 5μs. Note that while typical applications using ECho have to periodically poll for new events, KECho is able to notify kernel threads almost immediately of the arrival of a new event. This ability is investigated in the following section.

4.4.2 Event Delivery

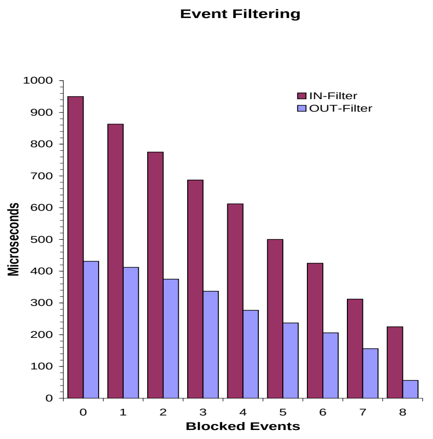
Events in KECho are pushed from event sources to event sinks. The network monitoring module of KECho is able to immediately notify a waiting thread of the arrival of such an event. Typical latencies measured from the arrival of an event at a socket to the invocation of a handler function are in the range of 250-300μs. In the case of ECho and KECho-UL, these latencies depend heavily on the polling frequency, the systems load, and the scheduling priority of the application receiving the event. However, ECalls' ability to *boost* the scheduling priority of an application that receives a newly arrived event can significantly reduce these latencies (as described in the previous chapter).

4.4.3 Filtering Overhead

The following measurements have been performed on a cluster of 4x200MHz Pentium Pros, with 512MB RAM, connected via 100Mbps Ethernet, running Q-Fabric.

The filtering functions (IN- and OUT-filter) serve to reduce processing and network overhead depending on application-specific attributes, supplied by the event producer and the event subscribers.

The left graph in Figure 30 compares the advantages of event filtering with IN- and OUT-filters. The left bars show the event handling overhead for a host with 8 sinks, i.e., an incoming event is dispatched to all 8 sinks and the overhead is approximately 950μs (event



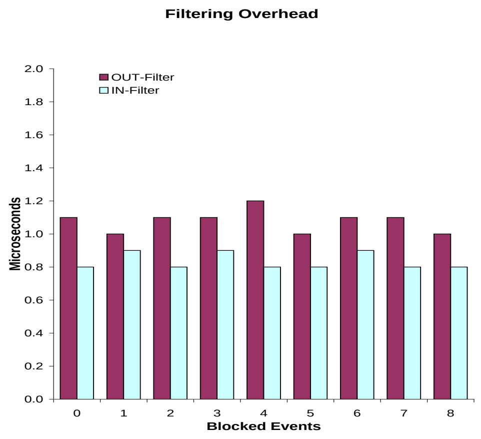


Figure 30: Filtering of events can reduce event submission and event handling overheads (left), while the filtering overhead is only in the microsecond range (right).

handling in this example means copying of the incoming event into a buffer and printing a time-stamp into a file). This overhead can be reduced significantly when an IN-filter is used to block the event from being dispatched to all 8 sinks, e.g., if only one sink receives the event, the overhead is reduced to 312μs. If the filter blocks the event completely (i.e., the event is discarded), the overhead is a little more than 200μs. The right bars in the same graph compare a similar scenario, however, the overhead shown in the graph is the overhead associated with event submission, when the number of remote sinks is 8. The overhead in this example is 430μs. However, when an OUT-filter is being used to block the submission of the event to some servers, this overhead can be reduced, e.g., if the event is submitted to only one sink, the overhead is 156μs. If the event is discarded (i.e., no sink will receive the event), the overhead is 56μs. The right graph in Figure 30 compares the overhead of the IN- and OUT-filters that have been used for the results in the left graph. Both the IN-filter and the OUT-filter use a number of simple if-else statements to decide if an event has to be submitted/dispatched to a certain sink or not. The overheads are independent of the number of events submitted or blocked and are very low in the example shown here, e.g., approximately 1μs for the OUT-filter and 0.9μs for the IN-filter.

4.5 Simulated Web Server Results

For this section, measurements have been performed on a cluster of 8 nodes, acting as a web server cluster. The simulated web servers receive requests at rates ranging from 20 to 50 requests per second. Each request requires a simulated web server to perform processing for approximately 38ms. The left graph in Figure 31 shows the response times (in milliseconds) without any load balancing compared to the scenario where load balancing is being used. Requests in this experiment have a time-out of 5s, leading to the leveling off at 5s of the first line in the graph, i.e., requests are either being handled within 5s after request receipt or discarded otherwise. Next, the server is modified such that requests that have been waiting for more than 2.5s are being forwarded to other servers in the cluster. In these experiments, it is assumed that there is at least one server in the cluster with utilization less than 10%.

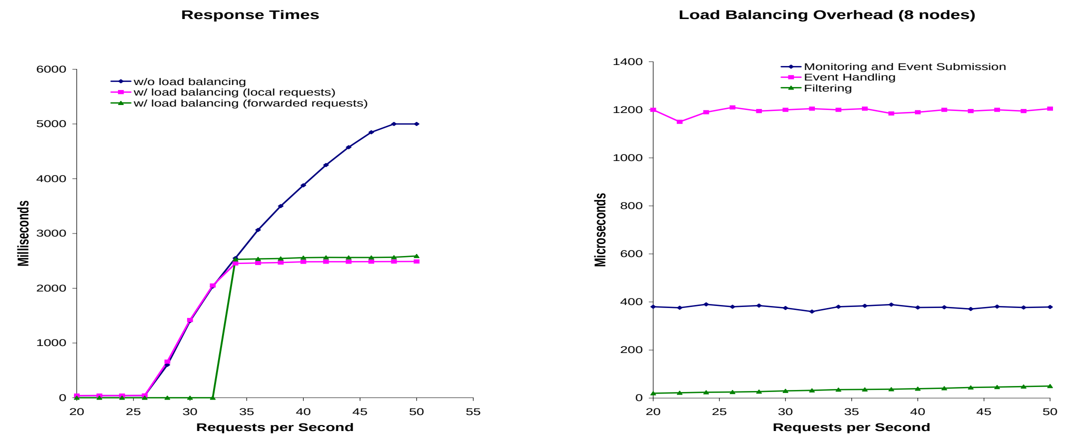


Figure 31: Response times for a simulated web server cluster (left) and overheads of the load balancing mechanism used in this experiment (right).

The second line in the graph (with load balancing - local requests) shows the response times of all requests which are handled on the local node. This time, the response times level off at 2.5 at request rates of approximately 33 per second. The third line shows the response time of the requests being handled on remote servers, which is slightly higher than the times measured at the local server due to the overhead of two events being submitted and received (forwarded request and request response). The right graph in Figure 31 analyzes the overhead for the load balancing mechanism, which makes sure that only one other server (dependent on load information collected from these servers) will receive the forwarded request. The graph compares the overhead of three actions performed by the load balancing mechanism: (i) the monitoring of CPU utilization and the submission of events carrying this information, (ii) the handling of incoming CPU information from other servers in the cluster, and (iii) the filtering necessary to ensure the delivery of the forwarded request to the server with the lowest utilization. The graph shows that all these overheads vary only minimally with the number of requests, where the task of event handling is the most expensive (approximately 70% of the total load balancing overhead).

The final experiment investigates the advantage of event filtering in more detail. The OUT-filter introduced above forwards requests to the servers with low load to ensure small response times. However, the frequency of load information exchange among the nodes in a server cluster has an obvious influence on the load balancing quality, i.e., if load information is not exchanged frequently enough, the forwarding decision can be based on outdated information, which reduces the effectiveness of load balancing.

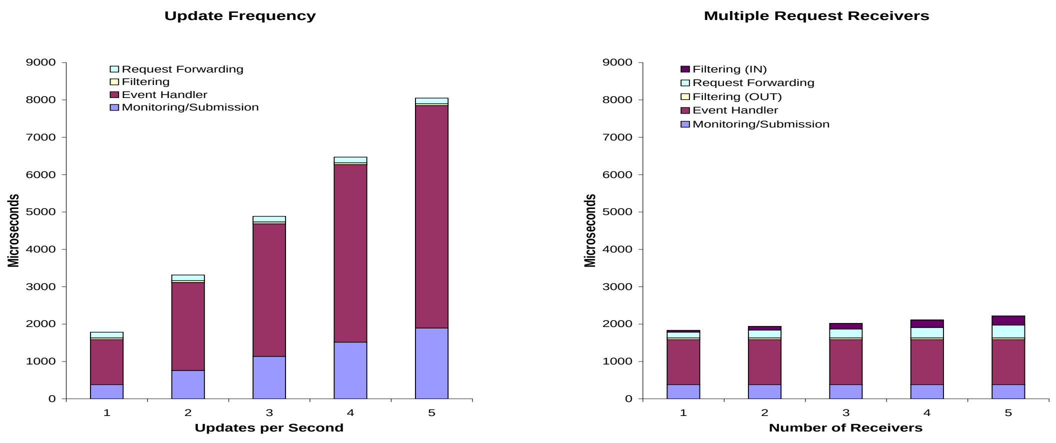


Figure 32: Comparison of update frequency of load information (left) and forwarding of events to more than one server in the cluster (right).

The left graph in Figure 32 compares the overhead of load balancing dependent on the frequency of load information events. The overhead is mainly due to the event handling process, followed by the load monitoring and event submission process. Smaller overheads are caused by the actual forwarding of the requests and the filtering functions. The overhead increases rapidly with the number of events exchanged per second, e.g., more than 8ms with a frequency of 5 events per second. The right graph in Figure 32 compares the approach, where the frequency of load events is kept constantly at 1 per second, however, the filter forwards the request to up to 5 different servers. In other words, multiple servers in the cluster respond to the event and only the first response is being used by the server that issued the event carrying the forwarded request. Again, the event handling and the load monitoring and event submission contribute most to the overheads, but the overhead increases only minimally with the number of event sinks. The biggest increase in overhead is caused by the IN-filter, which has the task of discarding duplicate responses. This experiment ignores the increased total utilization in the whole cluster due to the request handling by multiple servers. As an alternative to the solution suggested above, a server could issue a *cancel event* to all other servers, that makes sure that only one server handles a request. If several servers issue a cancel event, a time-stamp or some other criterion can decide which server wins. This approach reduces the unnecessary processing on the servers, however it increases the event communication by up to n cancel events per forwarded request.

4.6 Summary

Event services have received increased attention as scalable tools for the composition of large-scale, distributed systems, as evidenced by their successful deployment in interactive multimedia applications and scientific collaborative tools. This chapter introduced KECho, a kernel-based event service aimed at supporting the coordination among multiple kernel services in distributed systems, typically to provide applications using these services with certain levels of Quality of Service. The publish/subscribe communication supported by KECho permits components of remote kernels as well as applications to coordinate their operation. The target group of such a kernel-based event service is the rapidly increasing number of extensions that are being added to existing operating systems and are intended to support the Quality of Service and real-time requirements of distributed and embedded applications. For example, in Q-Fabric, KECho is used to establish ‘Q-Channels’ between distributed resource and QoS managers, supporting the integration of distributed system-level resource managers and monitors in the ‘horizontal’ direction, thereby providing the backbone required for system-level end-to-end QoS management.

CHAPTER 5

END-TO-END QOS MANAGEMENT

Distributed multimedia applications require the dynamic management of the underlying system resources, including CPUs, networks, disks, and sensor/display devices. For instance, remote sensing applications need sufficient network bandwidth to receive images with the latencies they require, and they need sufficient memory and CPU cycles to process and display these images when needed by end users. In all such cases, the resource managers located at the hosts involved in a distributed multimedia application have to dynamically allocate the resources required, monitor the QoS received, alter resource allocations when necessary, and perform run-time adaptations of applications, middleware, and operating or communication systems [1, 90, 115]. Specifically, *local resource management* on a host ensures that resources are distributed across applications to help them achieve their desired Quality of Service. However, since achieving and maintaining QoS for distributed applications is an end-to-end issue [91], multiple local resource managers must cooperate – i.e., perform *global resource management* – so that QoS guarantees can be applied to the entire flow of data. Such end-to-end QoS management addresses the delivery and processing of data from the server to the client and the management of associated resources, including CPU, memory, disk, and network bandwidth, along the path of the application data flow (e.g., from a server to its clients). In addition and to meet the specific needs of individual applications, QoS-awareness of applications [70] has been shown important.

This chapter introduces a concrete example of end-to-end QoS management, based on the Q-Fabric architecture, with the goal of demonstrating the importance of efficient and low-overhead cooperation of distributed adaptations of applications and system resources. The used scenario is that of a video conferencing application, where video streams are shared between multiple hosts and Q-Fabric is used link the resource managers across these hosts and across protection boundaries.

5.1 Multipoint Feedback Adaptation

5.1.1 Experimental Setup

Adaptations in Q-Fabric are based on feedback provided by resource monitors that notice changes in resource availabilities or insufficient Quality of Service provided to an application. A resource monitor uses a Q-Channel to communicate these observations with other resource monitors and with resource managers. This simple scenario becomes a complex problem if a stream is received by a large number of clients, i.e., a large number of resource monitors issue monitoring events back to a server-based resource manager. A common problem of large-scale feedback-based systems has been described as *reply implosions* [154] in the literature, where a server solicits information from clients and all clients reply almost simultaneously to the server.

To evaluate the performance and functionality of Q-Fabric, the *vic* video conferencing tool has been modified such that it operates on top of the KECho event service instead of standard sockets. Using Q-Fabric, a resource manager and a resource monitor have been implemented. Whenever user A starts an instance of vic, a data channel is set up. Transparently to vic (and the user), the resource monitor and the resource manager subscribe to a newly created Q-Channel. Once a second user B requests to receive a video stream from A, the resource managers of B subscribe to the Q-Channel. If both A and B decide to submit and receive each others streams, their resource managers subscribe to two different Q-Channels, one created by A and managing A's data stream and one created by B and managing B's data stream.

The task of the resource monitor is to monitor the rate of incoming packets over the data channel. This is done by installing a filter into the data channel, which keeps track of the received images from each participants. The resource monitor periodically submits a monitoring event. Another filter in the Q-Channel makes sure that only the resource manager at the data source will receive such monitoring events. On the other hand, upon receipt of a monitoring event, the QoS controller at the data source reconsiders its own resource allocations for this stream or issues a control event, which is submitted to either (a) one specific resource manager, e.g., the same that issued the monitoring event, or (b) all resource managers. The resource managers on the sinks then reconsider their resource allocations upon receipt of a control event.

5.1.2 Event Submission and Handling

As argued in previous sections of this dissertation, kernel-level implementations of resource management can profit from limited calls across protection boundaries, e.g., with system calls. Kernel services and data structures are directly accessible to the resource management mechanism, permitting fine-grain adaptations. A Q-Channel achieves further performance gains by ensuring that new events arriving on a channel are dispatched to the corresponding threads with minimal delays. This is achieved by the cooperation of KECho with the CPU scheduler (via ECalls) such that kernel threads with pending events are given preference over other (user-level) processes [104].

The following experiments have been performed on a cluster of 8 Quad-Pentium Pros with 200MHz each, 512 MB RAM, running Q-Fabric. The left graph in Figure 33 compares the cost associated with event submission in the kernel-level event service, KECho, with the event submission of a similar user-level implementation, called ECho [27] (both rely on TCP for event communication). In these measurements a source transmits an event

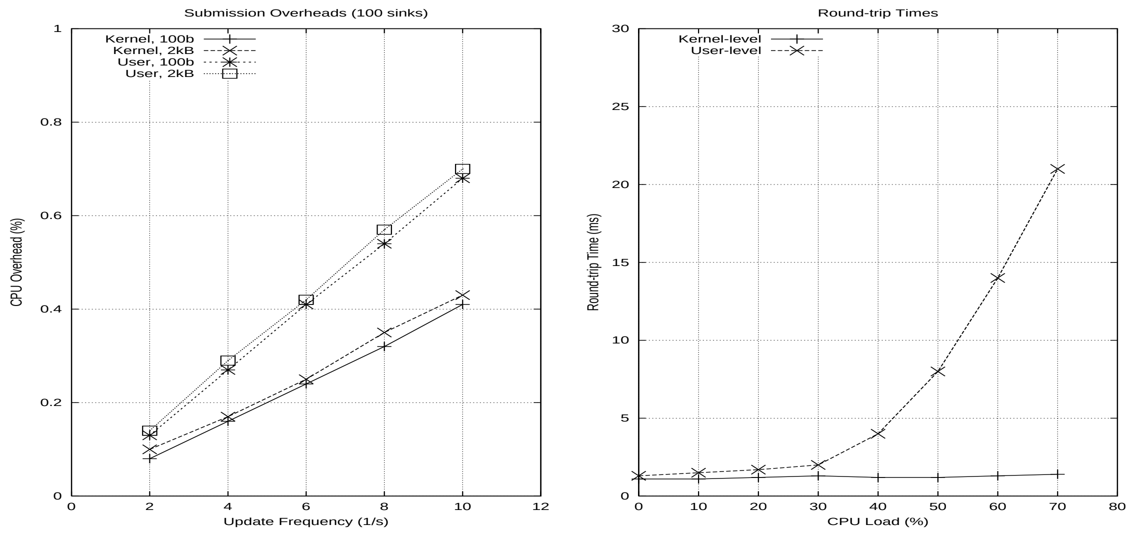


Figure 33: Event submission overheads (left) and round-trip delays (right).

of size (i) 100 bytes and (ii) 2 kBytes to 100 sinks with a variable **update frequency** (number of transmissions of events per second) in the range of 2-10. The chosen data sizes reflect the sizes of monitoring and control events used in these implementations, which are typically only a few hundred bytes large. With an update frequency of 10 events per second, 1000 TCP packets have to be sent out each second. In this situation, the CPU overhead of event submission in KECho is slightly above 0.4%, compared to 0.7% for the user-level equivalent of KECho (excluding protocol processing). The graph shows further that this overhead increases minimally with larger data sizes. The right graph in Figure 33 shows the measured **round-trip times**: here the time beginning from the submission of an event at the server until receipt of a reply event from the client is measured. The event size is 100 bytes and a **disturber process** is run such that it consumes CPU bandwidth from 0 to 70%. It can be seen that the kernel-level implementation shows constant round-trip times independent from the CPU load, whereas the user-level implementation suffers significant increases in round-trip-times above CPU loads of 30%. This is due to ECalls' ability to coordinate event notification with the CPU processor such that handler functions are invoked as soon as possible after event arrival to increase the responsiveness of the resource management system, while maintaining the real-time requirements of all running tasks [104].

5.1.3 Multipoint Feedback Control

Consider a point-multipoint scenario where a source streams data to several sinks and each sink declares a QoS range. The first, high-priority client declares a QoS range of $\{20, 25\}$, a medium-priority client declares a QoS range of $\{15, 20\}$, and a low-priority client declares a QoS range of $\{10, 15\}$. The resource management mechanism tries to supply all clients with the best possible quality within their ranges. If this quality cannot be sustained, it is desirable to reduce the qualities of the low-priority sink first, then of the medium-priority sink, and finally of the high-priority sink.

In this particular example (1 source, many sinks), monitoring events are only issued by sink-based resource monitors and directed to the source-based resource manager. On the other hand, control events are only issued by the source-based resource manager to all sink-based resource managers. Q-Channel filters make sure that sink-issued monitoring events

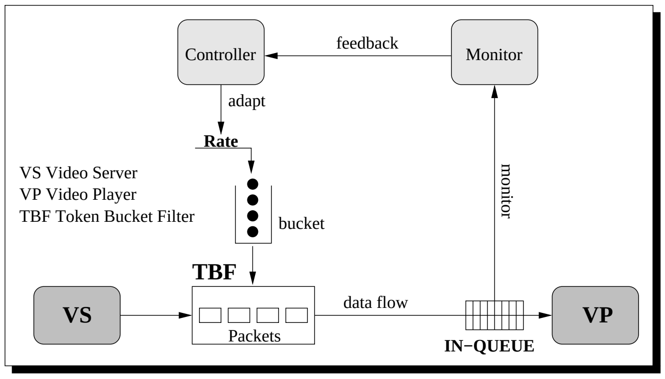


Figure 34: Feedback mechanism.

are propagated to the source only, and that source-issued quality events are propagated to all sinks or to a subgroup of them.

The resource controlled for the investigated application is network bandwidth. The vic video conferencing tool is executed as a real-time process in the SCHED_RR (round-robin) queue with priority 1. To limit and adapt communication bandwidth, Class-Based Queuing (CBQ) is used to define classes with a bandwidth of 200 Kbit each, and the Token Bucket Filter (TBF) algorithm is used to transmit packets. This algorithm has been modified such that the QoS controller is able to influence the rate of token generation and therefore, influence the rate of packet transmission.

Stream Management. Figure 34 shows the mechanism for one stream: a video server streams packets through a token bucket filter to the video player, where a QoS monitor watches the packet arrival at the video player and feeds this information back to a QoS controller at the server. The controller is then able to adjust the rate at which tokens are added to the bucket, and therefore the rate at which packets are sent over the event channel. Figure 35 shows the achieved frame rates of 3 streams, one with high priority, one with medium priority, and the last one with low priority. First, all 3 streams are started simultaneously, and the rate control achieved with the token bucket filter ensures that all

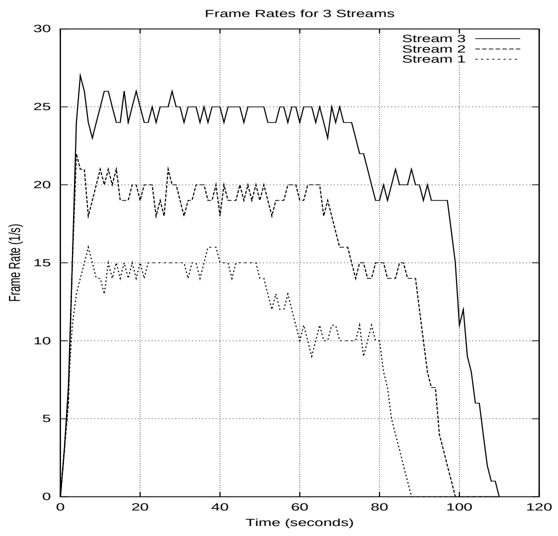


Figure 35: Frame rates for 3 streams.

three streams achieve the highest frame rate in their desired QoS ranges. After 40 seconds, a *disturber process* is started, which transfers files at increasing rates from the server to the clients, thereby causing the network to get saturated. After 50 seconds, the resource manager is not able to further sustain the frame rates and it starts reducing the frame rate of the low-priority stream, while sustaining the rates of the two other streams. After the low-priority stream reaches its minimum (10 fps), the medium-priority stream suffers a drop in achieved frame rate. Finally, after both the low- and medium-priority streams have dropped to their minima, the high-priority stream is reduced to its minimum value of 20 fps. After 80 seconds, the controller is not able to sustain all of the desired frame rates and starts reducing the rate of the low-priority stream until it reaches 0. It continues this process until, after 110s, all three streams have stopped.

Reply Implosion. The problem of receiving a large number of requests almost simultaneously is referred to as reply implosion. Solutions to this problem include *probabilistic replies*, *statistical probing*, and *randomly delayed replies* [154]. The advantage of the Q-Fabric-based approach is that a push-based mechanism is used, which largely avoids this problem. This is because resource monitors submit their monitoring events to a server-based resource manager independently at certain intervals (e.g., every 500ms). However, it is still possible, particularly in large-scale applications, that many monitoring events are issued almost simultaneously. In Table 1 we analyze the **event distribution** for 1 second in the same setup as described in Section 5.1.2 (i.e., 1 server and 100 clients). The first

Table 3: Event distribution.

	w/o adapt.		w/ adapt.	
Time	t=10	t=100	t=10	t=100
0-100ms	9	8	7	9
100-200ms	14	16	9	10
200-300ms	14	12	13	11
300-400ms	7	7	16	10
400-500ms	11	12	7	11
500-600ms	14	13	5	10
600-700ms	2	6	11	9
700-800ms	11	8	14	10
800-900ms	5	7	7	10
900ms-1s	13	11	11	10

column shows the event distribution measured after 10 seconds of running the experiment and displays the number of received monitoring events at the server per 100ms. With 100 clients and a update frequency of 1 (i.e., each client sends exactly 1 monitoring event every second), the ideal distribution in the simplified scenario would show 10 events per 100ms. The event distribution is measured again 90 seconds later, showing a similar distribution with minor deviations due to timing and measurement errors. However, it can be seen that the distribution ranges from 2 to 16 events. In a second experiment, an adaptation algorithm is used. The algorithm determines the number of clients N and the number of *expected events* per 100ms: $n = N/10$. For every time interval i of 100ms, the resource manager counts the number of actually received events (r_i). For all time intervals i, the number of *excess events* is computed ($e_i = r_i - n$ if $r_i > n$), and then e_i randomly chosen clients from interval i receive a *delay request* with the next quality event. This delay request forces the client (i.e., the QoS monitor) to submit the next monitoring event 100ms later. Over time, this succeeds in distributing the issuance of monitoring events more evenly as can be seen in the last column of Table 1.

Polling versus ECalls. The frequency of resource monitor and resource manager invocations influences both the granularity of adaptations achievable as well as the overhead of adaptations. The left graph in Figure 36 shows the behavior of a video stream with a

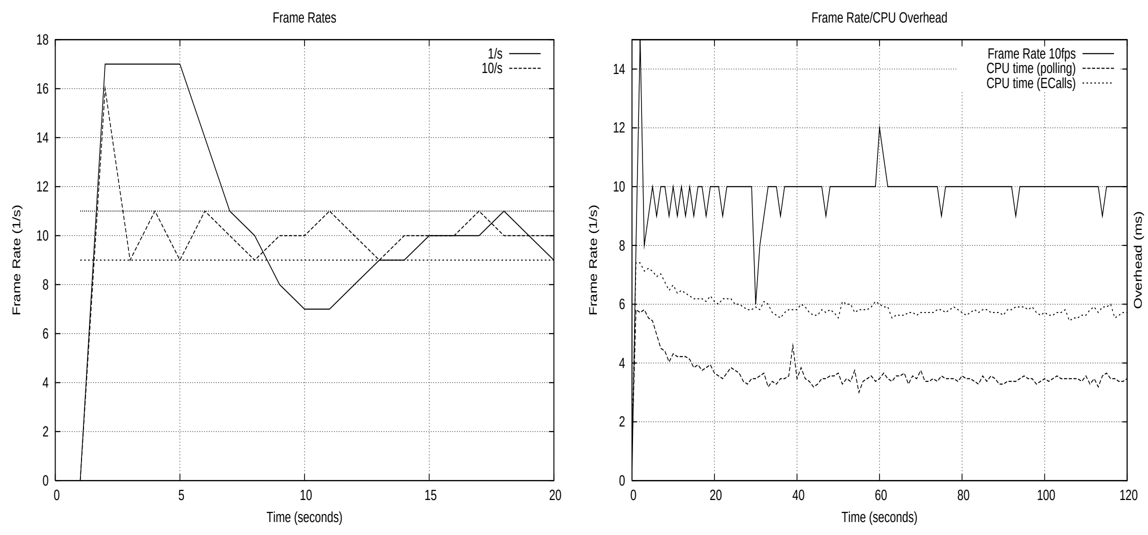


Figure 36: Effect of handler invocation frequency on the dynamics of the system (left) and event receipt with ECalls (right).

target frame rate of 10 fps±1. In the left graph, the resource manager is invoked (a) once per second and (b) ten times per second to poll for possibly pending monitoring events. In the case of a frequency of only 1/s, the video stream needs more than 12s to reach its target frame rate. When the resource manager is run ten times as often, the stream needs approximately 3s. However, KECho relies on the ECalls interface, which makes polling unnecessary by invoking the QoS controller immediately at event arrival. The right graph in Figure 36 shows a video stream operated at 10 fps, this time ECalls ensures the timely handling of incoming monitoring events. The graph also shows the CPU consumption of the QoS controller function when one (a) polls for events ten times per second and (b) uses ECalls instead. Without ECalls, the CPU consumption is approximately twice as much as with the support of ECalls because the resource manager is run *only* when there are monitoring events pending.

Q-Filters. Q-filters are application-specific event channel filters, parameterizable through the QoS management system built on top of Q-Channels. Such filters can perform tasks such as down-sampling, color depth reduction, or even dropping of images to reduce network load. In other words, additional computation at the server is introduced to reduce the required network bandwidth between server and client or to reduce the required computation at the client (e.g., visualization of images with fewer colors or smaller size). However, a resource manager can activate or affect such filters without the active involvement of the application, further improving the granularity of adaptations. This is particularly useful for transient overload situations, where frequent application adaptations can be counterproductive, however, the resource manager can simply change the parameters of the Q-filter with less overhead. Consider a Q-filter that simply drops certain frames (e.g., B- and P-frames of an MPEG stream). If a resource manager considers it necessary to change the number of frames transmitted to ease the network load temporarily, it can simply change a parameter of the Q-filter, which takes effect immediately. However, if the resource manager decides instead to notify the application to let itself perform this adaptation, the additional delay can be significant, particularly when the system is highly loaded. As an example, a filter that drops frames if the network is overloaded has been implemented. When the client-side QoS monitor detects a network overload, it issues a monitoring event directed to the server-side QoS controller. The controller then adjusts a Q-filter parameter that decides if and when a frame is being dropped. The overhead from the receipt of the monitoring event until the parameter is adjusted is in the range of 10μs. However, if a signal is sent to the application notifying it about the overload, the overhead from the receipt of the monitoring event until the application changes the rate of frame creation ranges from 50ms in a system with about 80% CPU load to several hundred milliseconds with CPU load $> 100\%$.

5.2 Experimental Setup

All experiments are performed on a dual-Pentium II with 400MHz, 512MB RAM, 512KB cache and a dual-Pentium II with 300MHz, 256MB RAM, 512KB cache, both connected via a switched 100Mbps Fast Ethernet and running Redhat Linux 7.1. The experiments in this section analyze the behavior of Q-Fabric in a specific application setting. The application used is that of a video conferencing tool, called *vic*, which is part of the OpenMASH toolkit[1].

Resource 1: CPU. The CPU scheduler used in this example is the Linux real-time round-robin scheduler. The resource manager adjusts an application's priority class to react to its varying computational needs. The CPU resource manager has no monitoring component, only an adaptation component, that is, only quality events and no monitoring events are being issued. Applications are assigned a default priority class (e.g., 50 in the following experiments) and can be modified in the range of 1 and 99.

Resource 2: Network. The attributes of the DWCS scheduler (period, window-constraint) translate easily to streaming multimedia applications that require the generation and transmission of data (such as video or audio) with a certain rate. However, such applications can often tolerate infrequent losses or misses of data generation or transmission. The adjustable parameters of a DWCS stream are the period and the window-constraint or *loss-rate.* The following experiments use a default period of 50ms (to achieve a frame rate of 20fps) and a loss rate of $x/y = 1/10$.

Application Adaptation. Vic can react to the receipt of control events by adjusting a variety of parameters:

- **Encoding Method.** For the experiments in this sections, H.261 and JPEG images are used. Vic can dynamically switch between different methods, where the choice of encoding can significantly affect the resulting CPU and network overheads.
- **Image Quality.** In vic, the used image qualities are in the range of 1 to 95 for JPEG images and 1 to 30 for H.261.
- **Image Size.** Both JPEG and H.261 images can have the two following sizes: *small* (176*144) and *medium* (352*288). In addition, JPEG images can also have the image size *large* (640*480).

The primary goal of the following experiments is to maintain a frame rate of 20fps or the best possible frame rate if resource contention is too high. The secondary goal is to minimize jitter at the client.

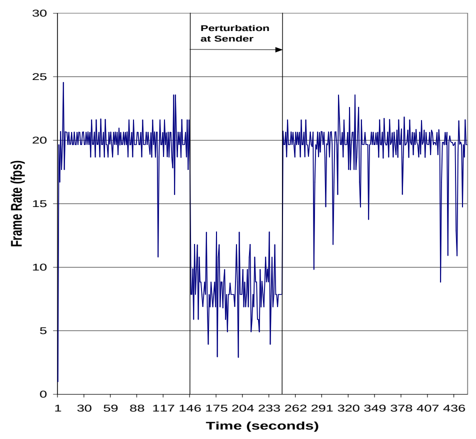
[1]www.openmash.org

5.3 *Application-level Adaptation*

Policy. The client-vic feeds achieved frame rates back to the server-vic (once per second), which then adjusts the image quality to maintain the desired frame rate. The images are H.261 encoded and have a default quality of 15. The quality is modified in steps of 1.

Perturbation. A CPU-intensive task (endless for-loop) is run first at the server, then at the client.

Details. Application-level adaptation is a useful tool to adjust the application behavior to changing environments. Application adaptation can significantly improve the overall quality of a media stream, however, the possibilities are limited without the support of a global resource management mechanism. The left graph in Figure 51 shows the frame generation rate at the server side of a vic conference. The desired frame rate is 20fps, however, the actual generation rate can vary significantly at times, for example, to 11fps at 113s. After



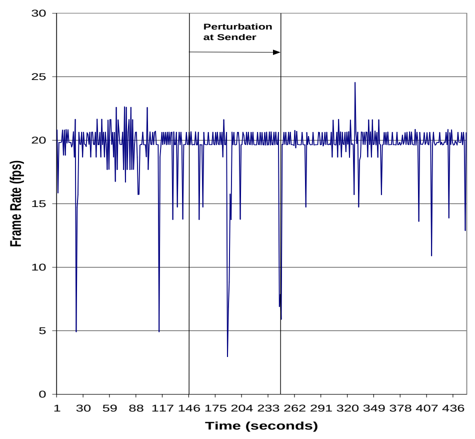
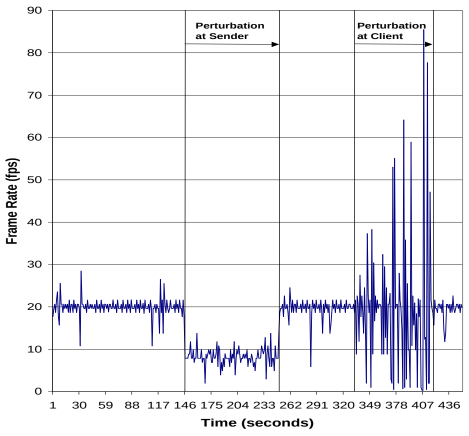


Figure 37: Frame generation rate (server) without resource management and adaptations (left) and frame generation rate (server) with application adaptation (right).

146s, a CPU-intensive task starts at the server, causing the frame generation rate of the vic server to drop to a range of 4fps to 13fps. After 250s, this perturbation is stopped. The right graph in Figure 51 repeats the same experiment, however this time, the client-vic uses Q-Fabric's event channel to inform the sender-vic about the achieved frame rate. After 146s, the server-side CPU perturbation is started and this time, the server manages to maintain the frame rate of 20fps by reducing the image quality. Note, however, that application adaptation is still not able to handle the strong deviations from the desired frame rates, which are caused by resource contention on both the server and the sink. The left graph



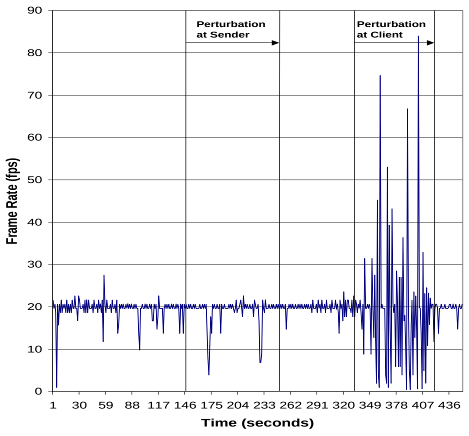
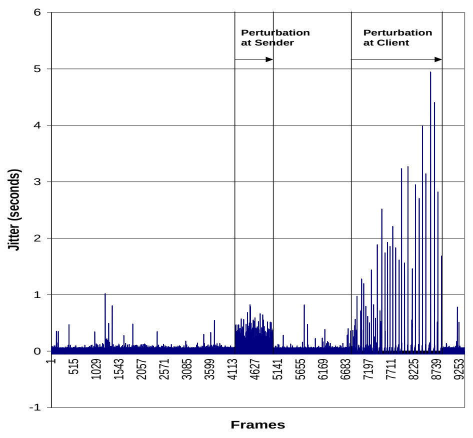


Figure 38: Frame replay rate (client) without resource management and adaptation (left) and frame replay rate (client) with server-side application adaptation (right).

in Figure 38 shows the behavior of the client-vic when (a) the server and (b) the client experiences CPU contention. After 146s, notice the drop in frame replay rate, which is due to the server-side CPU contention described in the previous measurements. After 330s, a newly started client-side CPU-intensive perturbation task creates strong variations of frame replay rates between 0fps and 86fps. The right graph in Figure 38 repeats this experiment with application-level adaptation. Though the adaptation manages to successfully handle the server-side contention, it fails to manage client-side contention properly even though image quality is reduced, resulting in smaller image sizes and less computational needs at the client-vic. This is also underlined by the next two graphs in Figure 39, which show the jitter at the client. Without adaptation, the jitter for the server-side contention increases from an average of $0.05s$ to $0.5s$ for the server-side contention and to more than $1s$ for the client-side contention. Peak values for the jitter for the client-side contention are close to $5s$. With adaptation it is possible to eliminate the increase in jitter for the server-side contention, however not for the client-side contention.



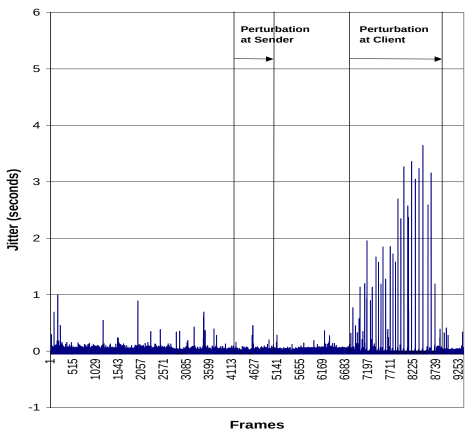


Figure 39: Jitter in frame replay (client) without adaptation (left) and jitter in frame replay (client) with adaptation (right).

Results. Summarizing, one can observe that in this example, application adaptation (e.g., changing the image quality) can significantly limit the negative effects of server-side CPU contention. However, application-level adaptation fails to efficiently support the application at client-side contention. Further, note that frame rates can vary significantly even without relevant CPU or network contention, due to a variety of reasons such as network latencies, contention with OS daemons, sending of frames in bursts, etc. In the following measurements it is further underlined that only management of all involved resources in conjunction with application adaptation leads to an acceptable application quality of service.

5.4 Distributed Resource Management

Policy. The network resource manager at the client-vic issues monitoring events containing the rate of received images at the client once per second. The CPU resource manager at the server modifies the CPU allocation to the server-vic if this rate differs from the desired frame rate. The default real-time priority is 50, which is being modified in steps of 1 (between 1 and 99).

Perturbation. A CPU-intensive task (endless for-loop) is run at the server. The CPU-intensive task has a real-time priority of 50.

Details. The problem with application-level adaptation is that it succeeds in reacting to changing environments in some cases only, and that competing applications have no way of preventing each other from stealing resources. Distributed or global resource reservation and management is required to distribute resources between applications. Figure 40 displays

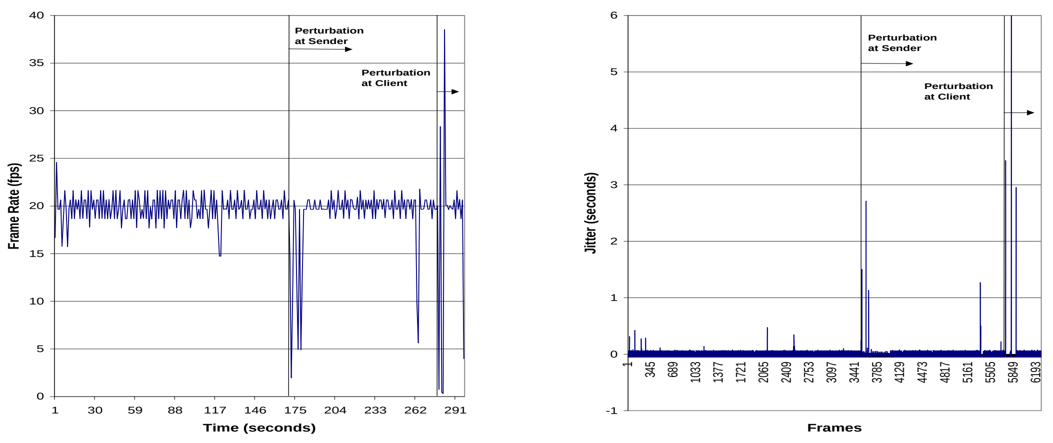


Figure 40: Frame replay rate (left) and jitter in frame replay (right) at the client with global resource management.

the client-side frame replay rate and the jitter for the same experiment, this time, however, with global resource management. The server CPU resource manager uses feedback from the client-based CPU resource manager to adjust the CPU allocations to the server-vic. In addition, the client CPU resource manager adjusts the CPU allocation for the client-vic if CPU contention exists.

Results. Global resource management succeeds in adjusting resource allocations such that the desired frame rate of 20fps can be maintained. The jitter can also be maintained at its average value of 0.05*s*. Note, however, the strong deviations in frame replay rate (which can be up to 100%) and jitter (which can reach several seconds) when contention starts. Although global resource management succeeds in maintaining a desirable replay rate and jitter eventually, it takes about 10s to stabilize both replay rate and jitter in these measurements.

5.5 Integrated Approach

The following experiments display the successful interaction of multiple resources (CPU, network) and applications to achieve optimal QoS management.

1st Approach: Client-feedback.

Policy. The client-vic issues monitoring events (once per second) to the resource managers and to the server-vic, containing the achieved frame rate. The server-vic adjusts the image quality of the H.261 encoded images (between 1 and 30) if the frame rate differs from the desired frame rate. Both the CPU resource managers at the server and the client adjust the CPU resource allocations as described in the previous experiment.

Perturbation. A CPU-intensive task (endless for-loop) is executed at the client.

Details. In the integrated approach of QoS management, applications and distributed resource managers cooperatively adjust their allocations and behaviors to react to changing system environments or load situations. Figure 41 shows the frame replay rate and the

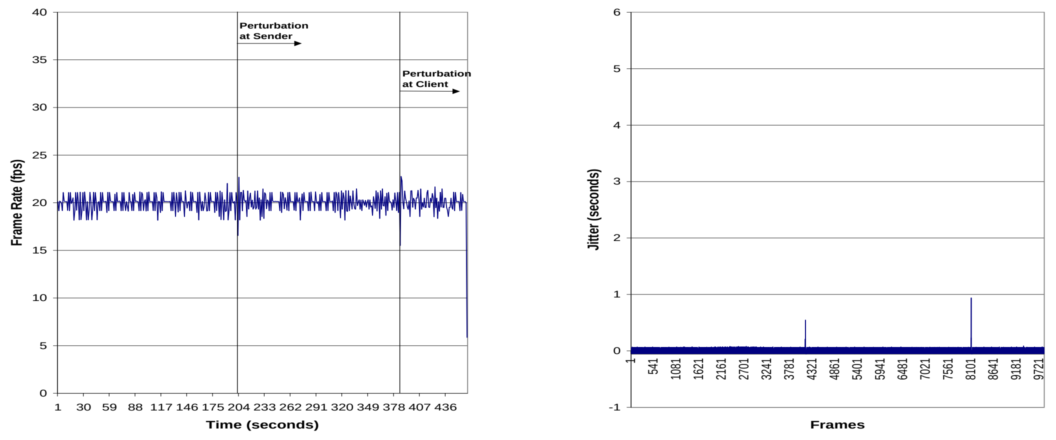


Figure 41: Frame replay rate (left) and jitter (right) at the client with global resource management and application adaptation.

jitter when global resource management and application adaptation are performed in conjunction.

Results. The left graph in Figure 41 shows that the integrated approach succeeds in maintaining the desired frame rate of 20fps at all times. The jitter graph (right graph in Figure 41), however, indicates where the contentions start with a large deviation from the desired value for both server-side and client-side contention. In a feedback-based approach as used in these experiments, these short deviations are unavoidable because they are required to trigger adaptations.

2nd Approach: Network-feedback.

Policy. The network resource manager uses information from the DWCS scheduler about the number of missed deadlines and issues this information as monitoring events once per second.

Perturbation. Network perturbation is simulated by limiting the network traffic between sender and client to 14kBps.

Details and Results. The left graph in Figure 42 shows the frame replay rate at the

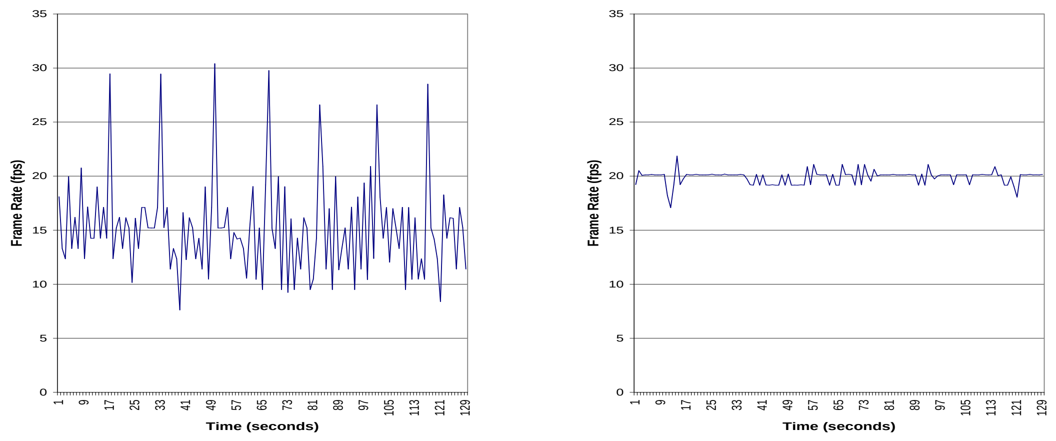


Figure 42: Frame replay rate at the client with kernel resource management (left) and kernel resource management and application adaptation (right).

client. While the desired frame rate is 20fps, the achieved frame rate fluctuates around 15fps. The right graph of Figure 42 displays the same scenario, this time with integrated adaptation and resource management.

3rd Approach: Client- and Server-feedback.

Policy. Feedback about the generated frame rate (server) and the achieved frame rate (client) are used to adjust image quality and size.

Perturbation. Instead of perturbation, JPEG compression is used, which causes CPU overheads too high to maintain a frame rate of 20fps.

Details. In this final set of experiments, JPEG images are used instead of H.261 due to their increased CPU requirements. In fact, when generating a video stream using JPEG compression, vic is not able to generate the frames at the desired rate of 20fps (see the left graph of Figure 43), even with the support of kernel resource management. The CPU

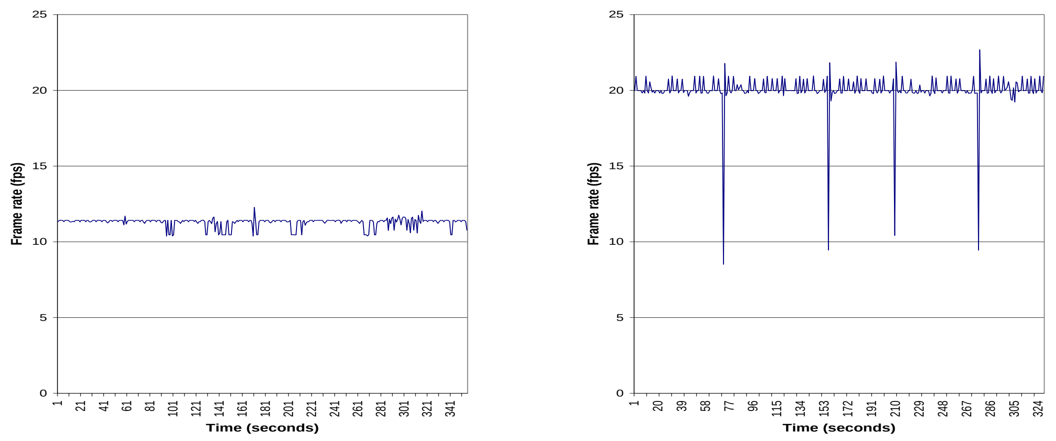


Figure 43: Frame generation rate (server) with kernel resource management but without application adaptation (left) and both kernel resource management and application adaptation (right).

resource manager tries to allocate more and more CPU bandwidth to the application, until it receives almost 100% of the CPU. In the second measurement shown in the right graph of Figure 43, we use feedback from the client-vic (frame replay rate) and from the server-vic (frame generation rate) to adjust the CPU resource managers (particularly CPU) and to adapt the application (reduce quality and image size). This time the desired frame generation rate of 20fps is achieved.

Results. This shows that when resource limits are hit, application adaptation can ensure that the quality of the stream degrades gracefully while achieving the desired frame rate. The conclusion is that the global management of resources complemented with adaptations of applications can benefit applications in their goal to achieve a desired quality of service, particularly when the application has to compete with others. In this chapter, an event-based cooperation scheme between multiple distributed resources and distributed applications has been introduced, allowing to cooperate their management decisions by exchanging **events** directly between each other. Receipt of an event triggers the execution of application adaptation and resource re-allocations in an attempt to maintain or improve the received QoS.

5.6 Summary

This chapter experimentally investigated the Q-Fabric approach to QoS management, which is used to integrate application adaptation and resource management closely via a kernel-based event service. Q-Channels are shared between resource managers and applications to exchange resource management information and requests, both asynchronously and anonymously. The advantages of having applications and resource managers share the same control path are several. First, distributed applications can interact freely to monitor and adapt their behavior according to the desired and achieved QoS. Second, applications are able to directly address resource managers, locally and globally. Previous approaches either prohibited the access to remote resource managers or only allowed this by crossing several interfaces, typically involving actions by underlying resource managers. Third, resource managers can interact with each other to support applications in achieving their desired QoS and to ensure fair distribution of resources. Fourth, resource managers can address applications directly by issuing monitoring and control events to them, therefore requesting application adaptations. Finally, QoS management policies can be developed that combine application adaptation with distributed resource management.

CHAPTER 6

ENERGY AS A FIRST CLASS RESOURCE

The management of multiple resources across multiple hosts is a challenging problem in end-to-end QoS management. While the device capabilities of modern mobile and wireless systems increase, energy is becoming the constraining resource for application domains such as mobile multimedia. Therefore, in mobile settings, the goal is to prolong battery life time to maximize the duration of device usage. Further, large-scale Internet data centers with hundreds or thousands of hosts cause large energy costs or require significant cooling efforts, e.g., to protect hosts from device or component failures. Making energy management even more challenging, energy takes an exceptional role as a resource in that it is closely linked to the utilization of all other resources of a system, e.g., increased CPU or network utilization translates to increased energy consumption. The goal of any QoS management is to maximize total system utility (and user-perceived quality), however, if energy has to be considered, the goal is also to minimize the system's energy consumption. This chapter introduces energy management techniques typically found in modern mobile devices and describes their use in the Q-Fabric infrastructure. Using the Q-Fabric integration interfaces, this chapter underlines the importance of information sharing and cooperation between multiple adaptive approaches in order to attain efficient QoS management results and to prevent conflicting adaptation decisions.

6.1 Introduction

A necessity for the acceptance of wireless devices and wireless applications is an acceptable life time of their finite energy sources. Batteries of wireless systems are drained by device display, network card, memory, and the processor. In the last few years, there has been significant work on the development of energy saving techniques, e.g., exploiting the voltage and frequency scaling capabilities of modern mobile processors [75, 130, 98, 45, 60] or application-level approaches that utilize the adaptivity of applications and the flexibility in acceptable user-perceived qualities, e.g., in video and audio streaming [102, 7, 19]. At the network level, previous work has investigated energy conserving mechanisms at different layers of the protocol stack, including the medium access control layer [153, 58, 88], transport protocols [2, 64], or routing [21, 42]. The key problems of wireless communication for multimedia applications are the provision of Quality of Service and energy efficiency. The goal is to balance both such that users are satisfied with the multimedia qualities and the energy consumption of a device is minimized. Energy management can be performed at multiple layers of a system: the physical layer, the protocol stack, the operating system, and at application-level. With a rising number of energy management techniques available at all these layers, it is increasingly important to provide an approach that involves all layers.

Frequently, energy management techniques are used to exploit resource idleness, e.g., resource utilization can be 'slowed down' such that a resource is fully utilized or unused resources can be switched off. The first approach is supported by the frequency and voltage scaling approaches of modern mobile processors or the dynamic modulation scaling approach of wireless connections. The latter approach is supported by timeout-based techniques, where after certain periods of inactivity, a device is turned off or put into a low-power sleep mode. The problem here is to decide when to wake up a device and to prevent frequent switches between shutdown and wake-up due to the high costs associated with these switches. While timeouts are a common approach to save energy during idle time, energy is wasted during these timeout periods. Predictive approaches, e.g., for hard disks, networks, or terminals, force a device into a low power state as soon as a predictor estimates an idle period of sufficient length. Wrong predictions can result in performance and energy penalties. Another method is to actively re-arrange resource requests (e.g., disk reads/writes, network transmissions) in order to better predict and to lengthen idle periods.

Communication over wireless or cellular links is often associated with costs, i.e., the bandwidth utilization translates to a financial burden to the user. Therefore, it is desirable to minimize the utilization of this resource. On the other hand, mobile devices are battery-operated and have to minimize the resource utilization in order to prolong battery life. While the approach for minimal network utilization is straightforward – e.g., low frame rates, small image size – the approach for minimal energy utilization is more difficult for two reasons. First, different resources have different energy costs associated, which have to be balanced. Consider, for example, the use of a media transcoder that reduces the size of a video image to be sent. Energy can be saved on the sender device if the costs of executing the transcoder are outweighed by the energy savings of transmitting a smaller-sized image. Second, energy management techniques have to be considered globally, e.g., in the previous example, employing the transcoder and therefore changing the image can have consequences for the resource utilization and therefore energy requirements of the receiver of the image. Depending on the global energy management goal, e.g., maximize the operational time of the application or minimize the energy requirements of a particular device, energy management has to be performed globally over all involved devices.

6.2 Media Transcoding

Previous work has pointed out the importance of *quality-aware* transcoding of multimedia [18] in order to provide differentiated services. In video transcoding, as an example, such transcoder functions can customize the video images to the restricted capabilities of mobile devices like handhelds or cellular phones. The proliferation of media streaming between resource-constrained wireless devices has raised the need for techniques that adapt these streams both to clients' Quality of Service requirements and to the energy restrictions of battery-driven mobile systems. First, the heterogeneity found in the resources of mobile devices (e.g., different specifications for displays, processors, or network cards) requires that media streams are adapted to a device's capabilities. Second, dynamic variations in resource demands and in user requirements necessitate the run-time customization of distributed applications and of the services they utilize. In particular, with client-specific service differentiation, each client requires services to be adapted to its individual needs, thereby better matching the resources expended on service provision with client requirements and capabilities.

Transcoding is the process of transforming information from one form into another, e.g., to convert large images to smaller ones that are suitable for the limited resources of handheld devices or cellular phones. Frequently, the transcoding of data at one end-host of a client-server communication has consequences on the processing and communication requirements for both end-hosts. More than one transcoding function or set of transcoder parameters can be used to transform data into suitable forms, making it necessary to compare transcoders with respect to their potential provision of quality of service and energy savings. These transcoders can be classified into *mandatory* transcoders, i.e., transcoders that have to be executed by at least one end-host, and *optional* transcoders. The goal is to use transcoders (a) to ensure that clients receive data in a form that corresponds to their QoS needs, such that (b) energy consumption can be reduced.

An interesting aspect of the approach introduced in this work is its ability to conserve energy by *adding* processing – in the form of transcoders – to a device. The intent is to reduce the energy consumption of a device by transforming large data elements into smaller ones, thereby reducing the costs of wireless data transmissions. However, such data transformations come at a price, i.e., the additional processing results in additional energy usage. As a result, data transformations reduce overall energy needs *only if* the additional energy consumed for the execution of a transcoder is outweighed by energy savings due to the transmission of smaller data items and the reduced processing needs at the other end-host. Fortunately, for applications like video streaming, past work has resulted in the creation of many useful transcoders [67, 24], and there has also been substantial work on the selection of suitable transcoder parameters [138]. By leveraging such results, this work can focus on maintaining a client's desired QoS characteristics, using different transcoders that result in varying energy savings, depending on QoS specifications, device and transcoder characteristics, and data content.

Though applicable to many areas with large-data communications, wireless multimedia applications are an important target for energy-aware transcoding, for three reasons: (1)

their increasing importance in mobile applications; (2) the fact that multimedia communications typically involve the long-running exchange of large data items, where data complexity or content change slowly over time; this justifies the potentially expensive deployment of transcoder functions; and (3) because these items can easily be changed in size and quality depending on resource availabilities and user needs. Video streams, as an example, can be customized in quality, including image size, color depth, resolution, or compression method. Here, different transcoders can be deployed and depending on a client's preferences, transcoders can be made *parameterizable*, e.g., a 'resolution-transcoder' can be tuned with different parameters for the desired resolution. As a concrete example, consider two

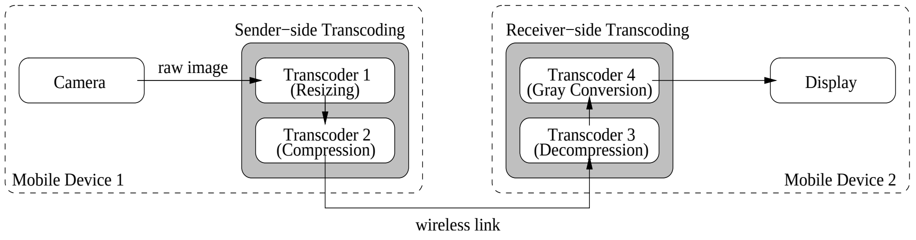


Figure 44: Video streaming example.

handhelds participating in a video communication, as shown in Figure 44. This could be part of an emergency communication system where firefighters and paramedics communicate in a disaster area using their handhelds, using visual communication to bring expertise wherever needed. Consider two devices used by medical personnel instructing each other in how to perform emergency care. One device captures raw images through its camera, which are then adjusted in size to fit the display of the receiver handheld and then compressed and transmitted to the receiver. At the receiver, these images are decompressed and further adjusted if required, e.g., converted to gray images if the device features a monochrome display only. In such cases, device battery life is critical and the decision about where transcoders are executed affects the energy requirements of both the sender and the receiver. Further, one cannot assume the availability of nearby servers or support infrastructure. As a result, it is assumed that transcoders can only be placed at the end-points of a communication, i.e., intermediate points such as wireless access points and base stations are not available to users for the deployment of application-specific functionality such as video transcoders (e.g., in wireless ad hoc networks).

6.2.1 A Transcoding Framework

The approach introduced here is based on the following observations. First, applications require that media streams have qualities that are suitable for the limited capabilities of mobile devices, where these qualities can be expressed as QoS ranges or as utility functions. Second, transcoding is an appropriate technique in media streaming to adapt media quality to suit the needs of the clients, where lower quality media typically results in reduced communication needs. Frequently, multiple transcoding techniques can be applied, resulting in different qualities and data sizes. Figure 45 shows the architecture of this approach. The

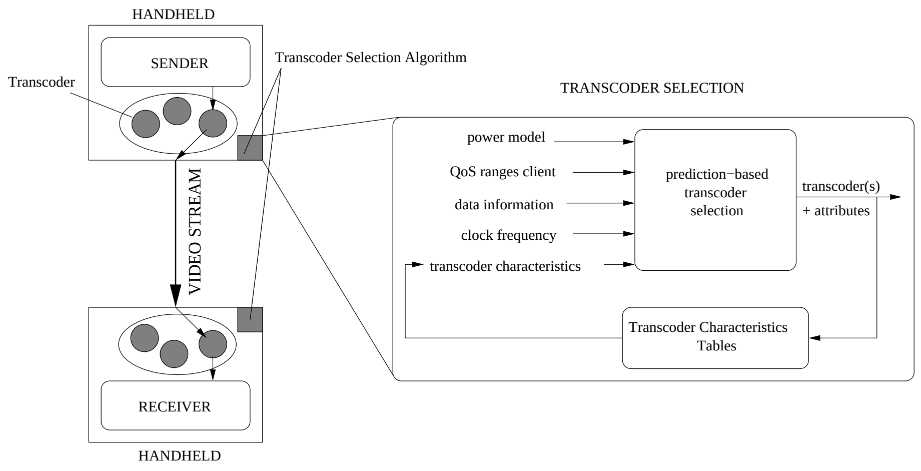


Figure 45: Transcoder selection.

sender transmits a media stream to a receiver, while the media stream is modified by a number of transcoders at either end-host. The transcoder selection algorithm collects a number of parameters in order to accurately predict the potential energy savings achievable by utilizing transcoder functions:

- **Clock Frequency (n).** The clock frequency used at a device is required information since it affects the transcoder run-time. In all measurements in this chapter, the clock frequency is kept constantly at the highest possible frequency.

- **Power Model.** The device's power model is obtained through off-line measurements of the relationship between run-time and processing energy for each available clock frequency ($K_r(n)$) and the relationship between data size and transmission energy ($K_d(n)$).

- **QoS Specification.** The user specifies the desired quality of service, which is translated into parameters for the transcoders. These QoS parameters include the minimum image width and height, minimum color depth, or the frame rate. Note that the approach in this section only addresses image-related qualities such as size or color depth; QoS parameters such as frame rate and jitter are managed separately by Q-Fabric's system-level resource managers.

- **Transcoder Characteristics.** Transcoder characteristics are determined off-line, i.e., the relationship between input data size and output data size (expressed as *ratio* r_d) and the relationship between input data size and transcoder run-time (expressed as *ratio* r_r), for each transcoder.

- **Data Sizes $size(d)$ and $size(d')$.** The input data size ($size(d)$) is the size of some data d to be transmitted before a transcoder function is applied, and the output data size ($size(d')$) is the predicted size of the output data (i.e., the size of the data after a transcoder is applied).

- **Transcoder Run-Time (rt_t).** This is the predicted execution time for each of the transcoders.

The transcoder selection is executed at the sender of a data stream, where the sender uses control events to inform the receiver whenever new transcoders (or new transcoder parameters) have been selected, allowing the receiver to decide if other transcoders need to be executed at the receiver host. The transcoder selection algorithm is executed periodically, however, other choices are possible, e.g., to invoke the algorithm for each frame or only when the user changes the QoS requirement or the video frames change significantly in size or content.

Obtaining Transcoder Characteristics. In order to compare transcoders, the relationship between input data size and output data size (r_d) as well as the relationship between input data size and transcoder run-time (r_r) are required. In this work, these relationships are measured off-line for sample input data and stored in tables. Each table entry contains the following information: *input data size*, *output data size*, and *transcoder run-time*. The transcoder selection framework predicts the output data size and the transcoder run-time by approximating these relationships based on the table entries. Different approximation methods exist, such as least squares regression. For simplicity, one can approximate these ratios by searching the table for the input data sizes closest to a given data size (i.e., the nearest entry higher and the nearest entry lower) and then compute the mean value for the output data size and the run-time of these two entries. When a transcoder is executed, the actual transcoder run-times and output data sizes are determined and can be used to update table entries. To ensure that the entries for transcoders that have not been used for a period of time are still accurate, unused transcoders can be executed periodically to obtain recent numbers for transcoder run-time and output data size, while considering these additional overheads in the transcoder selection process.

6.2.1.1 *Computation of Potential Energy Savings*

Next, the approach of this framework to determine the energy savings that can be achieved by deploying a transcoder at the sender is described. For each transcoder, the algorithm obtains an estimated transcoder run-time from the input data size and the ratio r_r:

$$rt_t = r_r * size(d)$$

Then the run-time - energy factor $K_r(n)$ is used to obtain the energy consumed by the execution of the transcoder at a particular clock frequency n:

$$E_t = K_r(n) * rt_t.$$

Next, the output data size is obtained from the ratio r_d:

$$size(d') = r_d * size(d).$$

The output data size ($size(d')$) is used along with the factor $K_d(n)$ to determine the energy consumption for the transmission of the output data:

$$E_{d'} = K_d(n) * size(d').$$

Further, the input data size $size(d)$ is used to determine the energy consumption for the transmission of the original data:

$$E_d = K_d(n) * size(d).$$

Finally, the *transcoder energy quality TEQ* is determined by subtracting the energy used for the transcoder execution (E_t) from the gains in energy caused by transmitting the output data instead of the input data ($E_d - E_{d'}$). The result is further corrected by E_a, the energy consumed by the transcoder selection algorithm. Similar to the monitoring of the transcoder run-time, the algorithm monitors its own run-time and uses this to determine the energy overhead of the algorithm. The resulting transcoder energy quality is:

$$TEQ = (E_d - E_{d'}) - E_t - E_a.$$

The resulting number, TEQ, is the energy that can be saved by executing the corresponding transcoder. Only those optional transcoders that have positive TEQs are eligible for data transformation. If more than one transcoder has a positive TEQ, the one with the largest TEQ is applied. If transcoders can be chained, all acceptable transcoder orderings (which can be specified by the client) are considered separately, i.e., TEQs are computed and compared for each suitable chain.

Algorithm Complexity. If the clock speed is determined by a separate approach, e.g., by a power-aware CPU scheduler, TEQs have to be computed for each available transcoder (and parameter setting) for the given clock frequency and input data size, resulting in $O(t * p)$ operations, where 't' indicates the number of transcoders and 'p' is the number of parameter settings. If the clock frequency can be changed by the transcoder selection algorithm, transcoders have to be evaluated at all clock frequencies. Here, the costs are $O(t * p * n)$, where n is the number of frequency levels. Typically, both the number of transcoders and the number of frequency levels are low (e.g., less than 10 transcoders for video streaming and less than 15 frequency levels for mobile devices).

6.2.2 Case Study: Video Transcoding

The mobile device under consideration is a Compaq iPAQ H3870 with a StrongARM SA1110 processor. This device supports dynamic frequency scaling and runs a modified version of the *familiar* Linux distribution (version 0.7.1), supporting 11 different clock frequency levels from 59MHZ to 206.4MHz. The iPAQ consumes about 1.32W of power in idle state (with disabled LCD screen) independent of the used clock frequency, and about 2W when active at the highest clock frequency. A Lucent Technologies Orinoco Gold 11Mbps wireless card is used for the wireless communication. This card causes a power consumption of 0.8W in receive mode and 1.3W in transmit mode. Energy measurements are performed with a Picotech ADC-100 PC Oscilloscope (2 channels, 100kS/second, 12-bit resolution). To facilitate energy measurement, the batteries of both the handheld device and its extension sleeve are unplugged, thereby forcing the device to draw its power from the DC adapter (5V, 2A max).

Image Transcoding. In the following measurements, images are read from disk in PPM format (Portable Pixel Map) and then transformed using a set of image transcoders (gray conversion, crop, reduce, Huffman and LZ77 compression). The first transcoder, gray, has no input parameters and converts the color coding of an existing image into a gray coding. Figure 46 (left) compares the energy consumption of transmitting unmodified data with the energy consumption of gray converting and transmitting the resulting smaller data. Figure 46 (right) shows a similar experiment for the crop transcoder. Here, the energy consumption for different ratios of output data size to input data size for three different input data sizes is compared. The horizontal line shows the costs of gray conversion of a 518kBytes image in comparison. Figure 47 (left) compares the reduce transcoder with three different input data sizes and the costs of gray conversion for data of size 518kBytes. The key result in these graphs is that the energy costs depend both on input data size

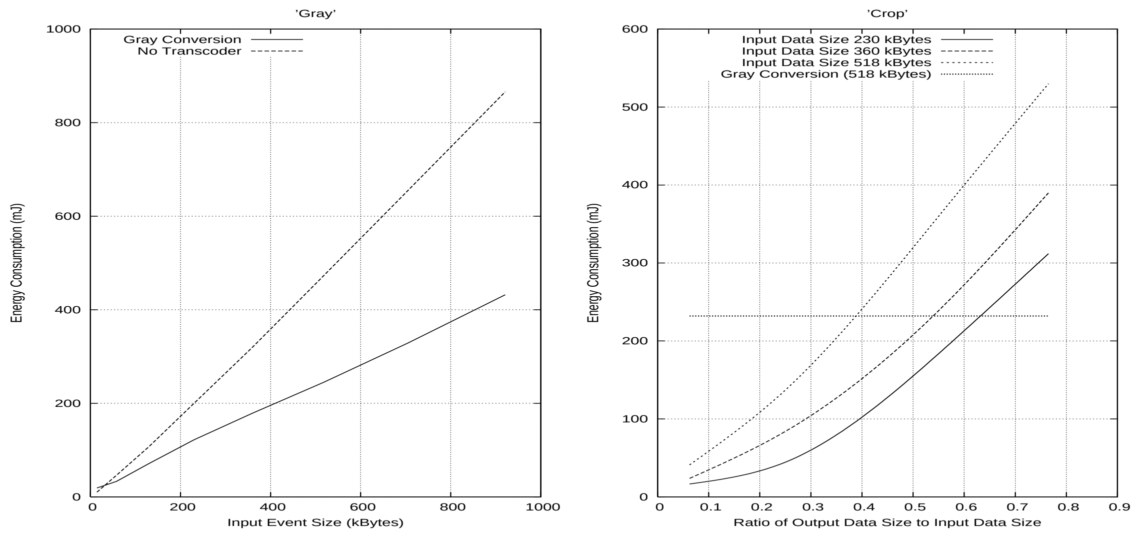


Figure 46: 'Gray' transcoder (left) and 'crop' transcoder (right).

and the parameters of the transcoder (e.g., the 'reducefactor' for the reduce transcoder). Finally, in Figure 47 (right), we compare the energy consumption for the two compression algorithms. The key result is that two different approaches to the same goal (compression of data) can have different energy requirements for different data, e.g., data transcoded with the Huffman compression and transmitted over the wireless link requires less energy than the LZ77 compression, except for input data size in the range from 140kBytes to 400kBytes.

Transcoder Chains. Transcoders can be chained together, e.g., a client may wish to receive gray-scale images with a certain size, i.e., both the gray transcoder and the crop or reduce transcoders are applicable. An important issue here is the **transcoder ordering**, i.e., the order with which transcoder chains are built. Figure 48 (left) compares combinations of two of the transcoders used in this paper with different orderings, namely the gray and the crop transcoders. The transcoders are first chained such that the gray transcoder is executed before the crop transcoder, then the ordering is changed so that the crop transcoder is executed before the gray transcoder. Here, the energy consumption of the crop-gray combination is lower than that of the gray-crop combination. The reason is that the gray transcoder leaves the image size untouched, and the crop transcoder's run-time

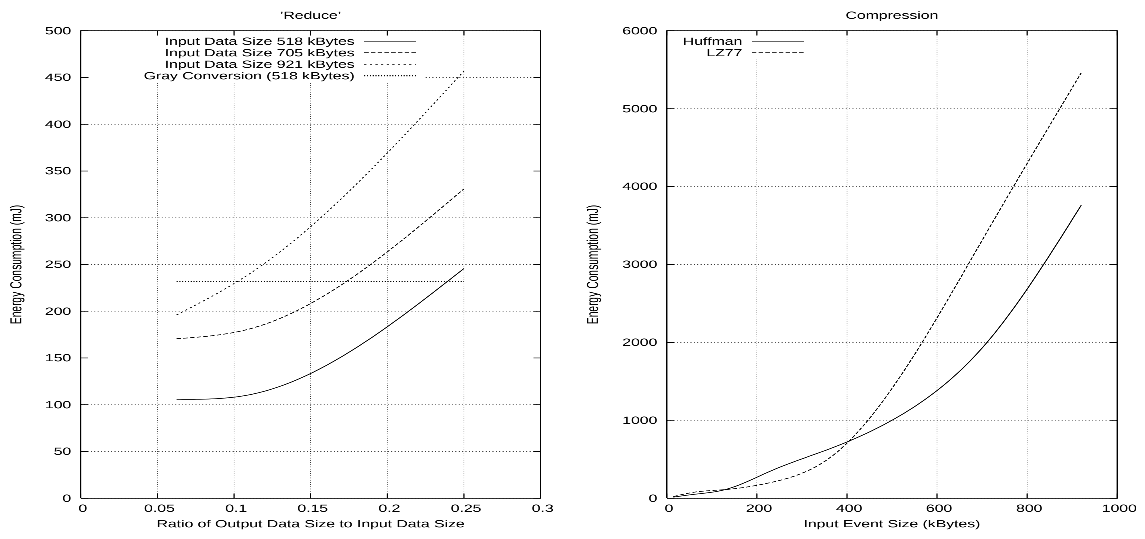
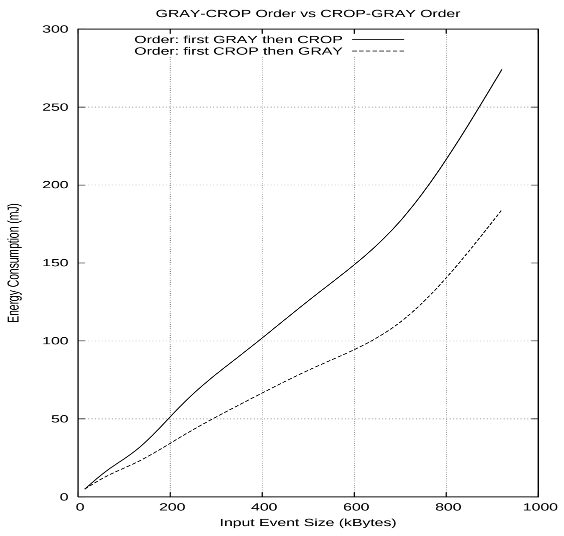


Figure 47: 'Reduce' transcoder (left) and Huffman and LZ77 compression (right).

is independent of the color depth of the image. On the other hand, the crop transcoder reduces the size of the image, thereby also reducing the amount of work required by the gray transcoder (for its pixel by pixel conversion of the image). Next, Figure 48 (right) compares the chains 'gray conversion - LZ77' and 'gray conversion - Huffman', showing how chaining can affect the energy requirements of transcoders. Here, the chain 'gray conversion - Huffman' outperforms (in terms of energy savings) the chain 'gray conversion - LZ77' for all image sizes.

Data Complexity. The *content* of data can have an influence on the amount of energy a transcoder function can save. In the case of the gray, crop, and reduce transcoders, image content has no affect on the transcoder run-time – and therefore energy requirements – or output data size (due to the pixel-by-pixel operation of these transcoders). However, the compression algorithms depend on the content of the images. Figure 49 (left) compares the energy consumption of the Huffman and LZ77 compression algorithms for four images with varying content but identical sizes. Figure 49 (right) compares the size of the transcoded images, which shows that variation in image content has an affect on the achievable image size and therefore the gains achievable with the transmission of smaller-sized data over the



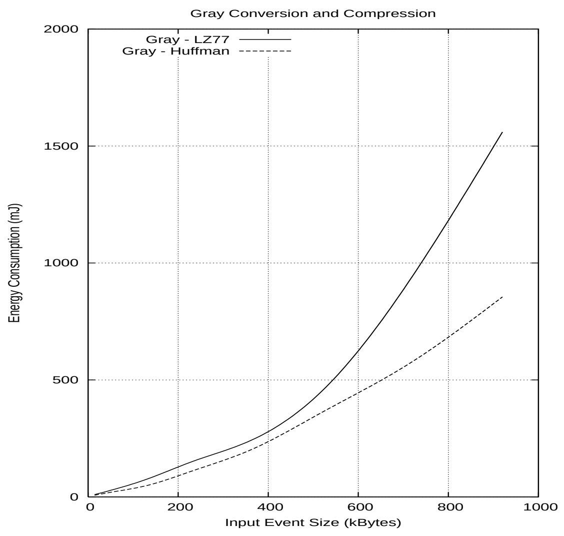


Figure 48: Ordering: gray and crop (left) and gray and compression (right).

wireless link. Both graphs indicate that the content of an image plays an important role in the energy requirements of a transcoding process, which indicates that off-line measurements of the energy characteristics of a transcoder are not sufficient. Although it can be assumed that in many scenarios video image content varies little from image to image, a dynamic approach to determining the characteristics of transcoders is required.

Power Model. Power models are used to describe the energy consumption behavior of system devices or software components. It is to expect that in the near future, 'on-board' power measurement mechanisms will be used to measure the energy consumption of a device, allowing for the automated generation and revision of power models. However, the results presented in this chapter rely on off-line obtained power models. Here, the processing costs of transcoders and the communication costs of submitting media streams are of interest. The iPAQ handheld device is examined using an oscilloscope while the transcoders are executed and media streams are transmitted. Figure 50 (left) depicts the measurements of the relationship between the run-time of a transcoder function and the energy requirements caused by the transcoder's execution. Note that this relationship is linear for all clock frequencies for the given architecture. This and all subsequent energy graphs depict the

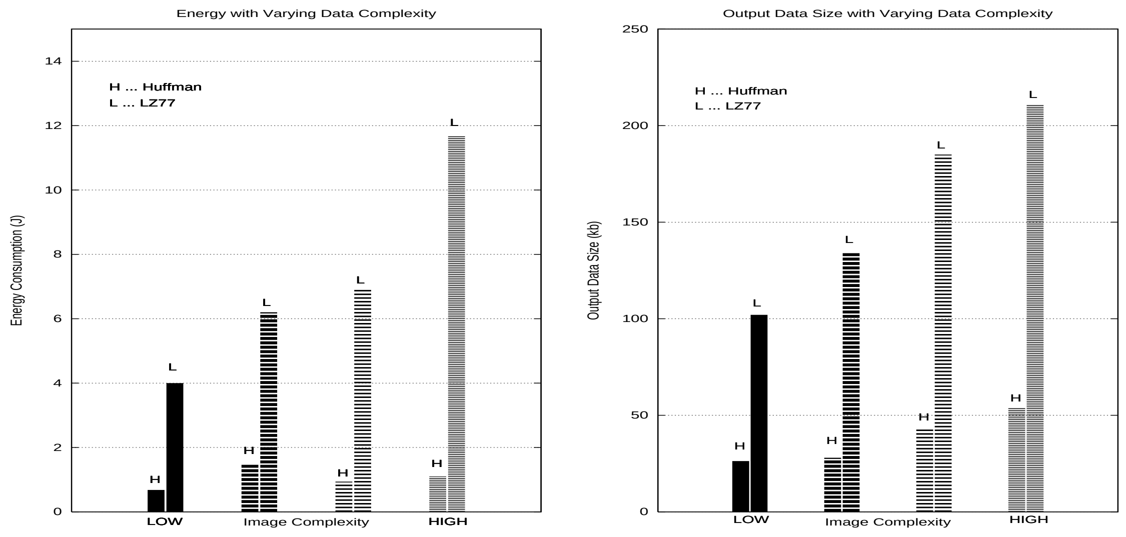


Figure 49: Effects of variation in data complexity on energy consumption (left) and output data size (right).

active energy, i.e., the total energy minus the idle energy. The results obtained allow us to derive the energy costs for transcoder executions. In comparison to these CPU-centric measurements, it is well-known that wireless network cards consume considerable power for message receipt and transmission. The specification of the Orinoco wireless card notes a receive power of 800mW and a transmit power of 1.3W. For Figure 50 (right), data of varying size is transmitted and the energy consumption for the wireless network card is measured as a function of data size. To utilize the obtained raw energy measurements, the results are turned into factors: $K_r(n)$ for the relationship between transcoder run-time and energy and for a given clock frequency n (Figure 50 (left)) and $K_d(n)$ for the relationship between data size and energy (Figure 50 (right)).

6.2.3 Results

The following measurements have been performed with the same setup as in Section 6.2.2. In the first experiment, the overheads associated with the transcoder selection mechanism are measured. Figure 51 (left) shows the overheads in microseconds for both the transcoder selection algorithm and the table updates with an increasing number of transcoders. Figure 51 (right) evaluates the sender-side computation of transcoder run-times for the gray, crop,

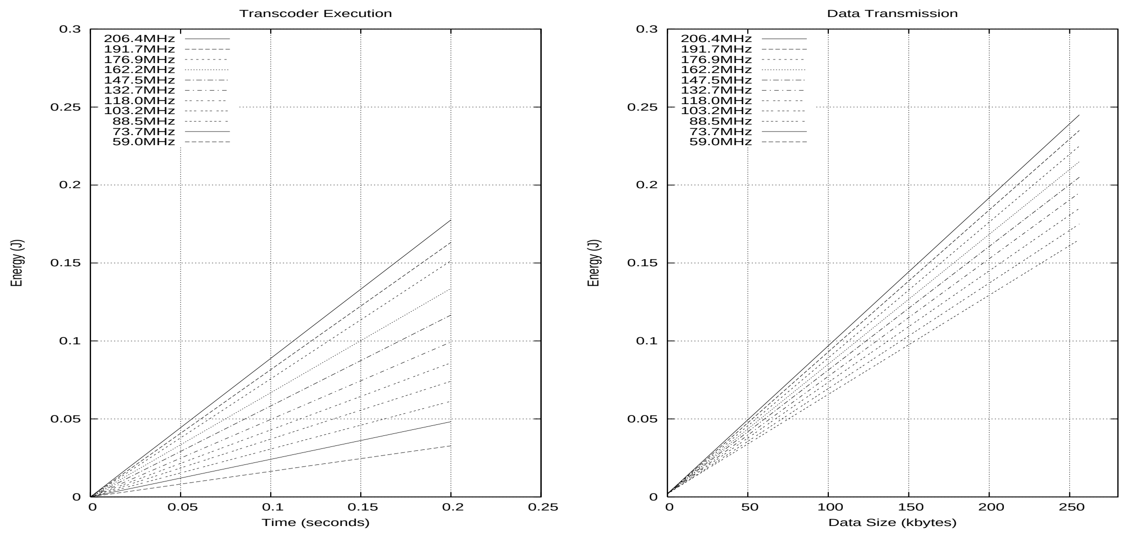
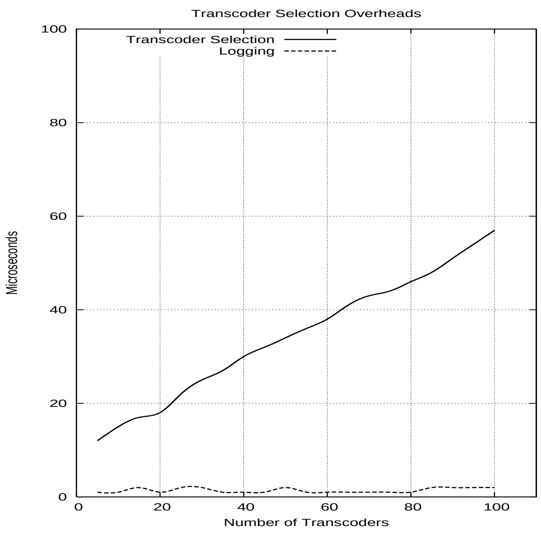


Figure 50: Energy cost of transcoder execution (left) and data transmission (right).

and reduce transcoders for 100 images with random sizes between 1kByte and 900kBytes. The graph shows the number of run-time predictions (in %) that deviate from the actual, measured run-times. The accuracy of the run-time predictions determines the accuracy of the energy predictions for the transcoder executions. For example, for an image of size 200kBytes, a deviation in 1ms in run-time prediction results in an error of less than 2% in the energy computation (TEQ) for the gray transcoder. In comparison, the predictions of the output data sizes for the gray, crop, and reduce transcoders were accurate within 1kByte in about 98% of all cases.

Figure 52 shows the results of the TEQ computations for the gray, crop, and reduce transcoders. The first graph compares the three transcoders for different input event sizes, where the parameters for the crop transcoder and the reduce transcoder where set to resize the original image to 1/4 of its original size. The second graph compares the crop transcoder for three different parameter settings with the gray transcoder for data sizes of 58, 230, and 518kBytes. These results indicate that for different parameters and different input data sizes, the TEQ computation will result in different transcoder selections, e.g., for data size of 518kBytes the crop transcoder achieves higher TEQs (and therefore energy savings) than the gray transcoder for images that are cropped to 40% or or less of the original size.



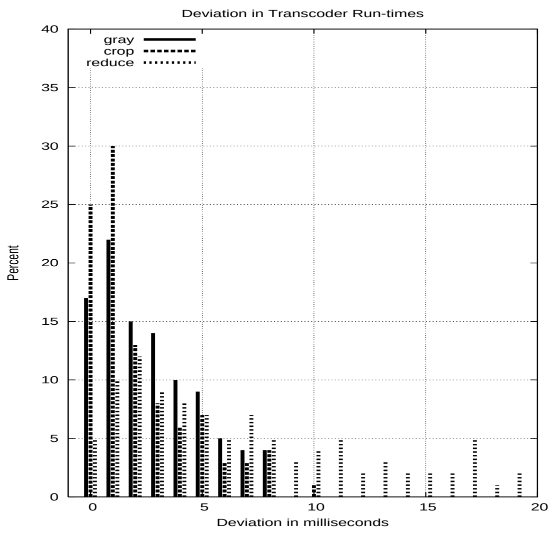


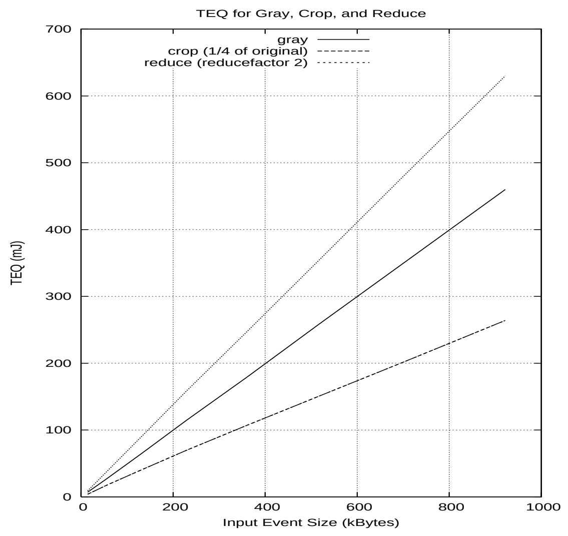
Figure 51: Overheads (left) and deviations of the predicted run-times from the actual run-times (right).

Discussion. Table 4 summarizes the mean deviations obtained with the framework introduced in this chapter. For example, the predictions for the run-times deviate by 3.8ms for the gray transcoder (corresponding to 5.4% of the total run-times), where the predictions for the output data sizes deviate less than 1kByte or 0.2%. These numbers result in a TEQ prediction for the gray transcoder that deviates 3.2mJ (or 5.6%) from the measured results. The predictions for the crop and reduce transcoders are only slightly worse than the ones for the gray transcoder. The approach evaluated here is comparable to the solution proposed

Table 4: Mean deviations.

Transcoder	Run-time	Data Size	TEQ
gray	3.8ms (5.4%)	1kByte (0.2%)	3.2mJ (5.6%)
crop	9.7ms (6.9%)	1kByte (0.2%)	7.3mJ (8.4%)
reduce	7.6ms (6.2%)	1kByte (0.2%)	6.5mJ (6.2%)

in [87], where video streams are retrieved from a multimedia server and a proxy between server and clients executes transcoders. Their middleware approach collects residual energy availability information on the client and uses this information on the proxy for real-time



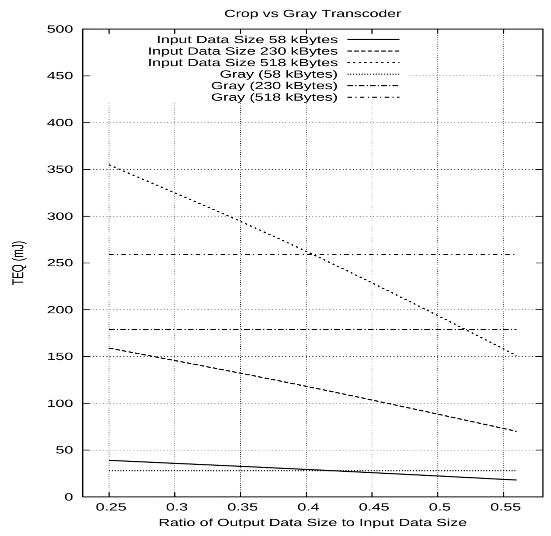


Figure 52: TEQ computation.

transcoding. This is similar to the approach introduced here in that relevant energy information of devices is shared and used to perform transcoding. However, the framework used in this chapter addresses situations where both client and server are mobile devices, and transcoder execution is only feasible at either end-host (e.g., in ad-hoc networks, access points without customization capabilities, etc.). This solution can easily be applied to the same setup as introduced in [87], i.e., where the transcoder selection algorithm is executed in an access point or proxy.

6.3 Exploitation of Network Idleness

Wireless cards consume a substantial percentage of a mobile device's energy. For instance, the Compaq iPAQ handheld with a StrongARM processor requires approximately 1.32W when idle, while an Orinoco Gold 11Mbps wireless card requires about 1.4W when transmitting. A key motivation for this research is the fact that this large contribution of network components to total system energy requirements is likely to increase in the future, due to current trends toward increasingly thin, continuously connected client devices.

Key approaches to energy management are: (a) reducing the amount of work (or avoiding work with little or no 'profit'), (b) 'smart' scheduling of resources (e.g., bursty disk or network accesses), and (c) reducing data reads and writes, e.g., to and from networks and disks. In the 802.11b standard, mobile devices can announce to an access point that they wish to switch to a low-power doze or sleep mode. The access point's task is to send out a beacon periodically (e.g., every 100ms), followed by a traffic indication map or 'TIM'. Each sleeping device has to wake up periodically to listen for the beacon in order to exchange control information with the access point. The TIM is used by an access point to indicate if it has data buffered for a sleeping device. In the case of the Orinoco wireless card, the idle mode requires 0.8W, however, the doze mode requires only 45mW, which suggest that maximizing the times a device can be placed in doze mode can be very effective for the preservation of energy.

The approach to energy management proposed in this work is summarized as follows:

- Real-time data streams have *ready times* t_r (earliest allowable transmission time) and *deadlines* t_d (latest allowable transmission time) associated with each packet. The task of a QoS packet scheduler is to transmit packets according to their scheduling attributes. Best-effort packets (i.e., no associated deadlines) are transmitted *after* real-time packets. If no schedulable packets are in the device's packet queue, the wireless network card is switched to a low-power 'doze' mode.

- When a packet is placed into the device's packet queue, the device's *watchdog timer* t_w is set to the packet's deadline minus an adjustable *offset* t_o: $t_w = t_d - t_o$. This delays the packet's transmission to the latest possible time, increasing the likelihood of *bursty* transmissions, i.e., if multiple packets are in a device's queue, t_w is set such that these packets can be transmitted together as one burst. The offset value is dynamically adjustable and ensures that data transmission is successful even under network medium contention: If packet deadlines are missed, the offset is increased and if packet deadlines are met consistently over a period of time, the offset is decreased.

- When t_w expires, the device is woken up and the packet scheduler transmits all queued packets with $t_r \leq t_{curr}$, where t_{curr} is the current time. After transmission, the device is switched back to doze mode (i.e., no timeout periods).

- Dynamic frequency scaling is coordinated with this approach such that when the packet scheduler's queue is empty, the lowest possible CPU clock frequency is used to delay the generation of new packets. As soon as the first packet is generated, the system switches to the highest possible clock frequency, in order to increase the number of packets queued (i.e., increasing the burstiness of data transmission) and to accelerate data transmission.

- The real-time CPU scheduler is adjusted such that tasks that have a low probability of packet generation are preferred while the packet scheduler's queue is empty, and tasks that have high probability of packet generation are preferred when the queue is non-empty, again, to increase the burstiness of data transmission.

The goals of these steps are twofold: (1) increase the burstiness of data transfer in order to reduce the number of switches between low- and high-power modes of the wireless card, and (2) minimize the potentially negative effects of dynamic frequency scaling on the energy efficiency of traffic shaping. The resulting advantages are increased burst sizes, a reduced number of costly switches between doze and idle modes, the elimination of costly timeouts, and increases in the communication device's sleep times by 'accelerating' data transmissions.

6.3.1 Bursty Packet Transmission

The first element of the integrated management approach introduced int this section increases the burst size of packet transmissions over a wireless link. Figure 53 (left) shows a scenario where 3 streams are submitted periodically, all with identical periods (T=200ms) and packet sizes (5kBytes). All measurements are performed on a Compaq iPAQ H3870 with an Orinoco Gold 11Mbps wireless card. In this scenario, as soon as a packet is submitted to the device's packet queue, the device immediately tries to deliver the packet at the earliest possible time (i.e., *ready time*). In keeping with standard energy management techniques, the device is kept in doze mode whenever possible. In this scenario, the network card needs to wake up 3 times each period and transmit a packet. Note the 'peaks' at 90 and 190ms, which occur when the device wakes up to receive a beacon signal from

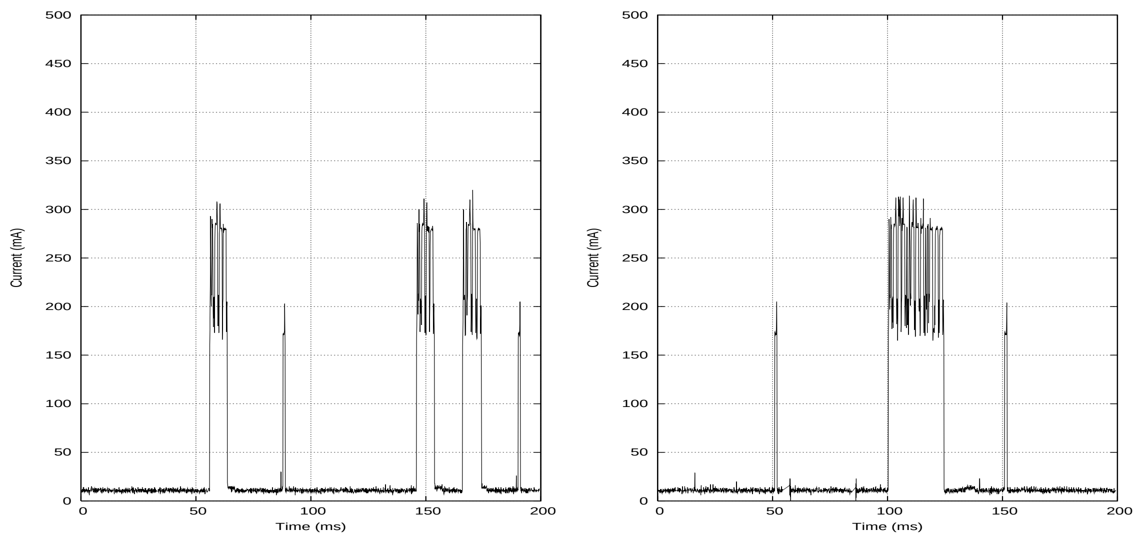


Figure 53: Packet transmission without traffic shaping (left) and with traffic shaping (right).

the access point. Figure 53 (right) shows the same scenario, but packet transmission is delayed in order to increase the likelihoods of large bursts. When a packet is entered into the packet queue, the transmission time is determined as shown in the following code segment, where t_w is the current watchdog time, t_o is the offset, t_x is the predicted transmission time, T_x is the sum of the predicted transmission times of all previously queued packets, and *q_len* is the the number of queued packets that will be transmitted before the newly arrived packet (according to the DWCS packet scheduling policy). As before, t_d denotes the packet's deadline and t_r the packet's ready time.

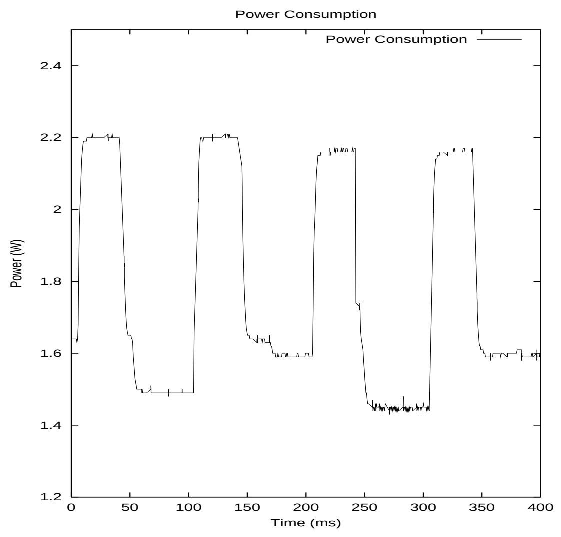
```
if (q_len == 0) {
  t_w = t_d - t_o;
} else {
  T_x = 0;
  for (i=0; i<q_len; i++) {
    T_x += queue[i].t_x;
  }
  t_x = t_w + T_x;
  if (t_d < t_x) {
    prev_t_w = t_w;
    t_w = t_w - (t_x - t_d);
    restore = FALSE;
13:       for (i=0; i<q_len; i++) {
14:         if (queue[i].t_x < queue[i].t_r) {
15:           restore = TRUE;
16:           break;
17:         }
18:       }
19:       if (restore == TRUE) {
20:         t_w = prev_t_w;
21:       }
22:     }
23:     if (t_d < t_w) {
24:        t_w = t_d - t_o;
25:     }
26:     enqueue(packet);
27:   }
```

Lines 1-2 describe the case when the packet queue is empty and t_w is set such that the newly arrived packet will be transmitted as late as possible. Lines 4-8 compute the predicted transmission times for all packets in the queue that will be transmitted before the newly arrived packet; this time will be used in the subsequent steps. If the scheduler would not be able to transmit the packet before its deadline, the algorithm attempts to adjust t_w such that the new packet will 'fit' into the current burst (lines 9-22). This is done by moving t_w forward by $t_x - t_d$, but only if this action will not cause any previously queued packet to be transmitted before its ready time (lines 14-17). Finally, if $t_d < t_w$, a new watchdog time is set (lines 23-25), i.e., the new packet will form a new burst. The adaptive offset is initially set to zero; when the packet scheduler observes that deadlines are missed beyond the 'allowable' deadline misses (expressed by x/y), it is increased in steps of 1ms.

Once all 'eligible' packets (i.e., the packet's ready time has expired) are transmitted, the watchdog time is set to the deadline (minus the offset) of the next packet in the queue, if the queue is non-empty. This approach can further be refined to coordinate data reception and data transmission, i.e., the watchdog time is set such that each burst is transmitted right before a beacon signal is expected from the access point. That is, a device wakes up, submits all queued packets and then remains awake to listen for a beacon signal and returns to doze mode if no data is pending at the access point. This optimization is not further considered in this dissertation. All packets that do not have deadlines associated are considered best-effort packets and are transmitted at the end of the current burst. Finally, this paper considers UDP streams, i.e., streams that support lossy transmission of data, e.g., video and audio streams. If either a TCP packet or a UDP packet that has been marked as 'urgent', is submitted to the device, the packet is transmitted immediately and all queued packets with $t_{curr} \geq t_r$ are transmitted.

6.4 DWCS and Dynamic Frequency Scaling

Modern mobile processors support multiple clock frequencies or voltages; this dissertation uses frequency scaling.



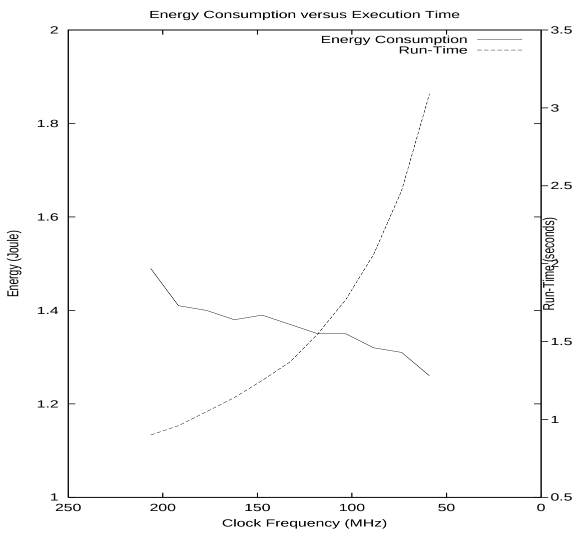


Figure 54: Video decoding on a mobile device (left) and energy consumption versus execution time for clock frequencies ranging from 59MHz to 206.4MHz (right).

Figure 54 (left) shows a snapshot of a video decoding process on the iPAQ H3870 handheld. A video stream is received at a rate of 10 frames per second, giving the decoder 100 ms for the display of each individual frame. If the device is under-utilized, frame decoding can be performed faster than that, resulting in 'idle times'. It is possible to reduce such idle times by reducing the CPU clock frequency, while still meeting each frame's soft deadline for decoding [82, 107]. To enable such per-frame device power management, measurements in Figure 54 (right) compare the energy consumption of the iPAQ with the execution time of a simple for-loop with 10^7 iterations (simulating a video decoding process) at 11 different clock frequencies. The iPAQ is run without any extension or network cards and with the LCD screen turned off. The energy consumption

$$E(Joule) = P_{active} * T_{active} + P_{idle} * T_{idle}$$

is the sum of the 'active' period of the device ($P_{active} * T_{active}$) and the 'inactive' (or idle) period of the device ($P_{idle} * T_{idle}$). The energy consumption depicted in Figure 54 (right) and in all subsequent energy graphs is computed over the period of the worst-case execution time (WCET) of the emulated function handling video frame decoding; in this experiment the handler function has a WCET of 3.09s running at 59MHz. The idle power of the iPAQ is 0.29W.

The key result depicted in Figure 54 (right) is that although the run-time of the examined code increases by more than 2 seconds when CPU frequency is scaled from 206.4MHz to 59MHz, energy consumption is reduced by 200mJ (P_{active} for 206.4MHz: 0.92W, P_{active} for 59MHz: 0.41W). These graphs indicate that there is a possibility to save energy by 'intelligently' slowing down processing on a mobile device. One can also observe that at some frequencies the energy consumption rises, however, this behavior is disregarded in this dissertation, but others (e.g., in [84]) have investigated this issue.

6.4.1 Dynamic Frequency Scaling Algorithm

The example considered in this section is that of a video player on a mobile device. Clock frequencies are typically computed such that the *rest utilization* of the CPU is exploited by slowing down task execution. With DWCS, a new clock frequency is computed whenever the system utilization changes, e.g., when tasks join or leave the run queue. The current utilization of all tasks is computed with:

$$U = \sum (1 - \frac{x_i}{y_i}) * \frac{C_i}{T_i}$$

C_i is the service time allocated to task i at the default clock frequency and this service time increases when the clock is slowed down. For each clock frequency n, a *scaling factor* k_n can be obtained by executing a sample processing-intensive code at both the default frequency

f_{max} and f_n and dividing the measured run-times: $k_n = C_n/C_{max}$. This is repeated for each available clock frequency (or core voltage) for a given processor. The goal of frequency scaling is to get as close to 100% utilization as possible, i.e.,

$$U_{100\%} = \sum (1 - \frac{x_i}{y_i}) * \frac{C_i * k'}{T_i}$$

where k' is the yet unknown scaling factor. To guarantee that best-effort tasks are not starved, one can replace $U_{100\%}$ with $U_{x\%}$, e.g., $U_{95\%}$. Then k' can be determined with:

$$k' = \frac{U_{95\%}}{\sum (1 - \frac{x_i}{y_i}) * \frac{C_i}{T_i}}$$

The resulting k' is compared to the previously obtained scaling factors, and the scaling factor k_n closest to k' ($k_n \leq k'$) is selected, and the clock frequency is adjusted to frequency n.

6.4.2 Run-Time Variations

In addition to computing the most appropriate clock frequency, Q-Fabric also measures application progress for deadline-constrained periodic activities. Here, if the execution is faster or slower than expected, Q-Fabric can adjust the clock frequency such that application termination before the deadline is ensured. Variations in the run-time can be large due to **external** and **internal** influences. External influences are outside of the control of Q-Fabric, e.g., tasks can be delayed or preempted by other tasks with higher priorities. Internal invariants are under the control of Q-Fabric and are caused by variations in data size or complexity (e.g., changing size or resolution of images).

6.4.2.1 Power Checkpoints

With *power checkpoints*, clock frequency adjustments can be made while a task executes, that is, adaptivity is introduced at a finer granularity. Two types of checkpoints are supported in this implementation: (i) code checkpoints and (ii) scheduler checkpoints.

Code Checkpoints. Code checkpoints (Figure 55) are placed directly in the task by the application developer or by a compile-time tool. Each time a checkpoint is reached, the

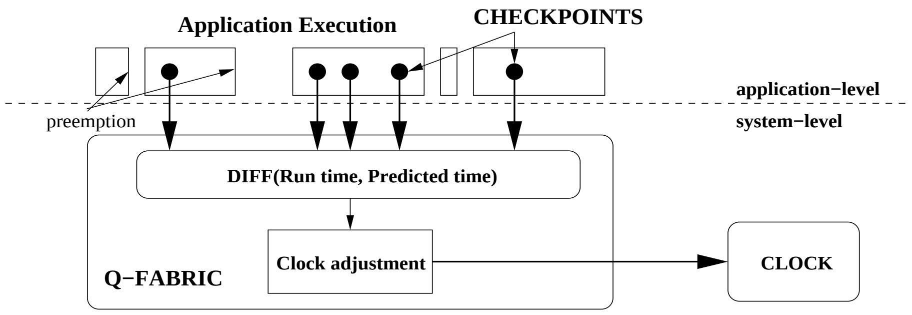


Figure 55: Code checkpoints.

task execution is interrupted and a call-back into Q-Fabric is performed. Q-Fabric then compares the actual run-time of the task with the predicted run-time and changes the clock frequency if necessary. This is useful for situations where external influences delay the execution of the task. The positioning of a code checkpoint has an influence on the usefulness of the checkpoint. For example, a checkpoint placed in the second half of the code may be more useful than a checkpoint in the first half. Further, a code checkpoint has one argument which is a simple integer in the range from 0 to 10. This argument allows the task to inform Q-Fabric about the complexity of the currently processed data, which then allows Q-Fabric to choose a more conservative frequency level if required. As an example, a video decoding task can 'inform' Q-Fabric about an image size or quality that deviates from the size or quality of previous images. Q-Fabric determines the clock frequency level as described before, however, it takes the checkpoint argument into consideration. For example, an argument of '2' indicates that Q-Fabric should choose a clock frequency of at least 2 levels higher than it would choose without this argument. This allows the task to force Q-Fabric to be more conservative in frequency scaling. Figure 56 compares the number of missed deadlines with different checkpoints. The first bar indicates that 215 executions out of 1000 miss their deadlines in this experiment. Note that some of these executions have such early deadlines that they are not able to meet them even running at the highest possible clock frequency (in this experiment about 60-70 of all executions fall into this category). The following bars then indicate the missed deadlines for 1,2,3, and 4 checkpoints, which are set at equal intervals in the code. It can be seen that the number of missed

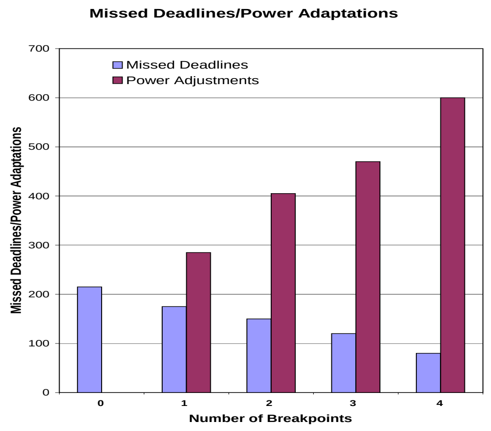


Figure 56: Missed deadlines (left bars) and number of frequency adaptations (right bars).

deadlines drops from 215 with no checkpoints to 80 with 4 checkpoints. The second set of bars show the number of power adjustments that were necessary, the more checkpoints the more adjustments are being made. Figure 57 shows that the average clock frequency increases with the number of checkpoints, however, the change here is only about 10MHz between 0 and 4 checkpoints. Note that in this implementation, only if a handler finishes without clock frequency adjustment during run-time, the measured run-time will be used for re-computation of the run-time average.

Scheduler Checkpoints. A second approach to the checkpoint solution relieves the application developer from finding appropriate places in the code for the checkpoint placement. Instead, the Linux CPU scheduler has been modified such that each time the scheduler is about to schedule a Q-Fabric-supported task, it first calls back into Q-Fabric. The advantage here is that the checkpoints are set 'automatically' by the scheduler. However, the code is not able to inform Q-Fabric about increased complexity of the processed data content. Further, with scheduler checkpoints the clock frequency is only re-considered when the scheduler runs and when the task actually is being preempted. Figure 58 shows this scenario. Each time the CPU scheduler selects the Q-Fabric-supported task as the next

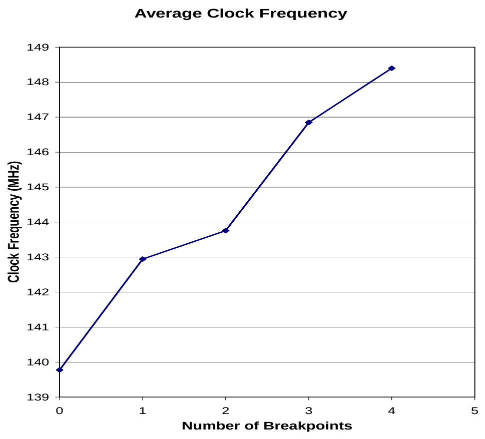


Figure 57: Average clock frequencies.

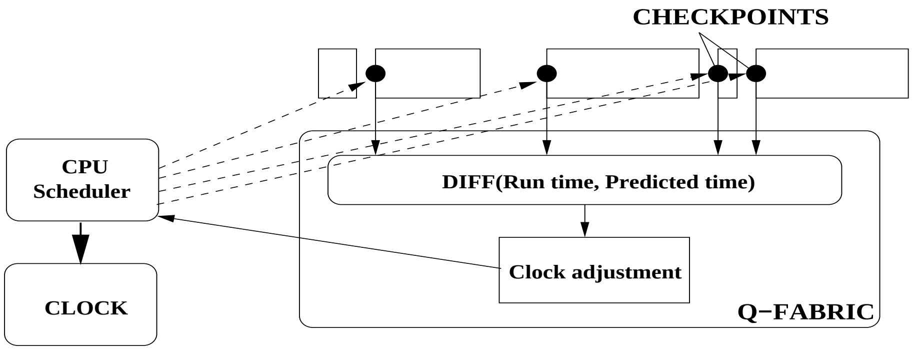


Figure 58: Scheduler checkpoints.

task to run (after it got previously preempted by another task), the scheduler calls back into Q-Fabric, which then re-adjusts the clock frequency if necessary. The call-back functionality is independent from the scheduling policy used. To enable any CPU scheduler to make use of per-task frequency levels and to make call-backs into Q-Fabric the following changes have been applied to the Linux kernel:

(a) The task structure in linux/sched.h (**struct task_struct**) has been extended with two new entries: **clock_frequency**, which is used to store the most recently used clock frequency in kHz for this task and **kecho_task**, which is a flag that indicates if a task is a Q-Fabric-supported task.

(b) After the CPU scheduler has selected the next task to be run it first inspects the

kecho_task variable in the task structure of this task and makes a call back into Q-Fabric if the entry indicates that the task is a Q-Fabric-supported task. This gives Q-Fabric the opportunity to re-compute the required clock frequency.
(c) Finally, the CPU scheduler checks the **clock_frequency** entry (which might have been modified by Q-Fabric in the previous step) and re-adjusts the clock speed if the entry differs from the current value of the clock speed.

In the case of scheduler checkpoints, the actual number of checkpoints can vary and depends on the run-time of the task, the number of preemptions, and the frequency of scheduler invocations.

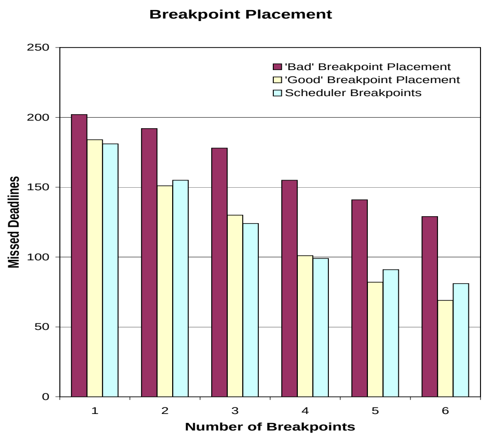


Figure 59: Comparison of 'bad' and 'good' code checkpoint placement, and scheduler checkpoint placement.

While in the case of scheduler checkpoints the checkpoint placement is 'performed' by the CPU scheduler, with code checkpoints the task developer has to identify appropriate places in the code for such call-backs (possibly with the help of compile-time tools). Figure 59 compares the effect of 'bad' checkpoint placement (e.g., situations where checkpoints are placed only in the first half of a task) with 'good' checkpoint placement (e.g., situations where the checkpoints are evenly distributed in the task). The latter case shows greater reductions in missed deadlines with larger numbers of checkpoints. In addition, the scheduler checkpoint approach is compared with the code checkpoint approach: the results are comparable to the 'good' placement policy for the code check points.

6.5 Summary

Limiting the energy consumption of mobile devices has evolved into a key issue for QoS management. This chapter introduced three different approaches to energy management supported with Q-Fabric: (a) media transcoding, (b) DWCS-based CPU scheduling coupled with dynamic frequency scaling, and (c) deferred packet transmission for wireless communication. In the first approach, energy-aware media transcoding carefully balances the costs associated with the use of different resources in order to find the 'cheapest' (in terms of energy) approach. Dynamic frequency scaling is an approach to slow down the operation of the device, reducing application performance, but preserving energy. This chapter further introduced an approach to compute the optimal clock frequency when a real-time scheduler such as DWCS is used. Finally, wireless network connections make significant contributions to the energy requirements of a mobile device. This chapter also introduced an approach to reduce the number of switches between doze and active modes by increasing the burstiness of packet transmission. It does so by delaying packet transmission up to their deadlines to increase the probability of large bursts. While each individual approach succeeds in reducing a system's energy consumption, only the cooperative management of all approaches results in maximum energy savings, which is further underlined in the following chapter.

CHAPTER 7

INTEGRATED ENERGY MANAGEMENT

Energy management can be performed at multiple layers of a system: the physical layer, the protocol stack, the operating system, and the application. With the rising number of energy management techniques available at all of these layers, it is increasingly important to integrate these techniques within and across layers and across devices. The previous chapter introduced three energy management techniques and their deployment in the Q-Fabric architecture. In this chapter, the combined use of these techniques is studied and the concept of global energy management directives is introduced. The goal is to underline the importance of integration in distributed QoS management and to describe this integration in the context of the Q-Fabric infrastructure. Using Q-Fabric, multiple energy management techniques residing on multiple hosts can efficiently cooperate and thereby ensure that energy consumption in a distributed system is minimized.

7.1 Energy-Aware Traffic Shaping

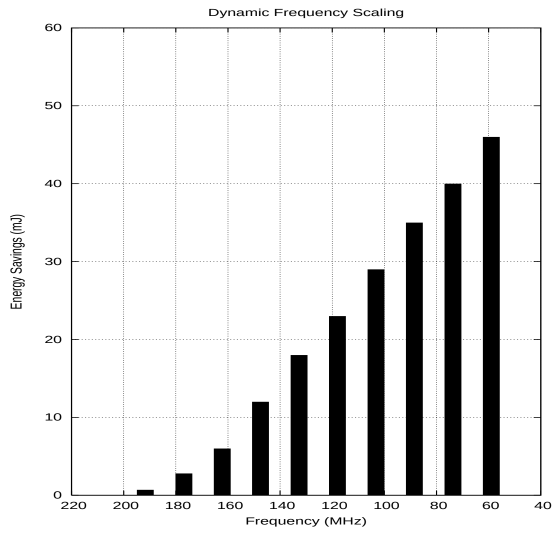
Besides communication, computation is another significant source of energy consumption. Clock frequencies and CPU voltages of modern mobile processors can be reduced with dynamic frequency scaling (DFS) and dynamic voltage scaling (DVS). A similar approach has been suggested for the use in wireless cards, called *Dynamic Modulation Scaling* or DMS [127]. Here, modulation frequencies are selected such that transmission times are prolonged while energy requirements are reduced, similar to the effects of dynamic voltage scaling.

The approach to energy management proposed in this section extends the approach introduced in Chapter 6 with the following steps:

- Dynamic CPU frequency scaling is coordinated with network-level traffic shaping such that when the packet scheduler's queue is empty, the lowest possible CPU clock frequency is used to delay the generation of new packets. As soon as the first packet is generated, the system switches to the highest possible clock frequency, in order to increase the number of packets queued (i.e., increasing the burstiness of data transmission) and to accelerate data transmission.

- The real-time CPU scheduler is adjusted such that tasks that have a low probability of packet generation are preferred while the packet scheduler's queue is empty, and tasks that have high probability of packet generation are preferred when the queue is non-empty, again, to increase the burstiness of data transmission.

The goals of these steps are twofold: (1) increase the burstiness of data transfer in order to reduce the number of switches between low- and high-power modes of the wireless card and (2) minimize the potentially negative effects of dynamic frequency scaling on the energy efficiency of traffic shaping. The resulting advantages are increased burst sizes, a reduced number of costly switches between doze and idle modes, the elimination of costly timeouts, and increases in the communication device's sleep times by 'accelerating' data transmissions. When multiple energy management techniques are deployed, the uncoordinated combination



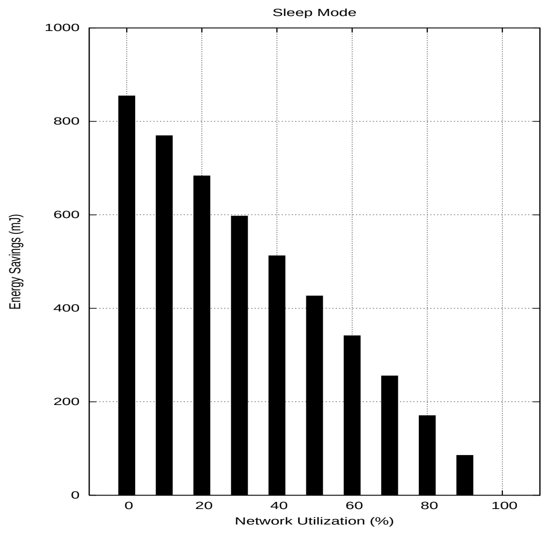


Figure 60: Energy savings for using dynamic frequency scaling (left) and sleep modes for network cards (right).

can have unintended adverse effects, i.e., the combined energy savings are suboptimal.

Figures 60 and 61 repeat the results introduced in the first chapter of this dissertation: the left graph in Figure 60 shows the energy savings achievable by using frequency scaling (DFS) on a sample processor for a snapshot of 1s and a CPU utilization of 25% (resulting in energy savings of up to 46mJ). In comparison, the right graph shows the energy savings achievable by exploiting the *sleep mode* of a wireless network card, i.e., when no communication occurs, the device is switched into a low-power mode. Depending on the network utilization, the savings can reach up to 850mJ. Figure 61 shows the expected energy savings when both

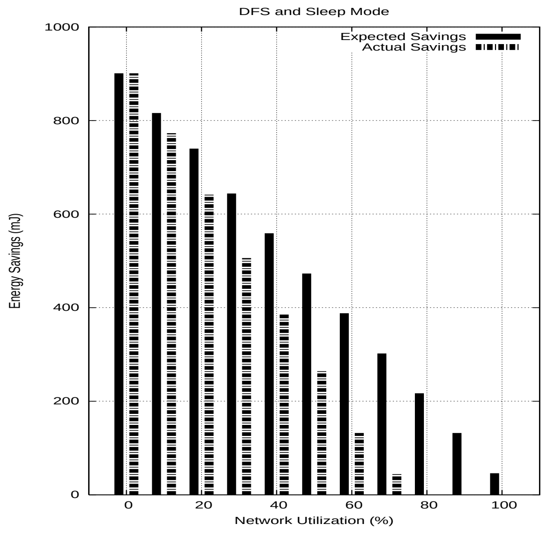


Figure 61: Expected energy savings (left bars) and actual energy savings (right bars) for the combined use of dynamic frequency scaling and the network card's sleep mode.

DFS and the network card's sleep mode are deployed. In this example it is assumed that the lowest clock frequency is used, resulting in energy savings of 46mJ from the use of DFS. Ideally, the total energy savings are the sum of the savings from using DFS and the network card's sleep mode; the expected results are shown as the left bars in Figure 61. However, the actual energy savings are significantly lower than the expected savings (right bars), e.g., at 50% network utilization, the expected savings are 473mJ, the actual (measured) savings are only 266mJ. This discrepancy is addressed in this section.

Scaling the clock frequency has been shown to be effective in saving energy when the processor has idle times [75, 130, 98, 45, 60]. Network activities that involve the CPU, including protocol processing and packet scheduling, particularly when packet fragmentation is used (such as in the 802.11b standard), affect how long a network card requires to transmit data and how long it can be kept in doze mode. Figure 62 (left) shows the

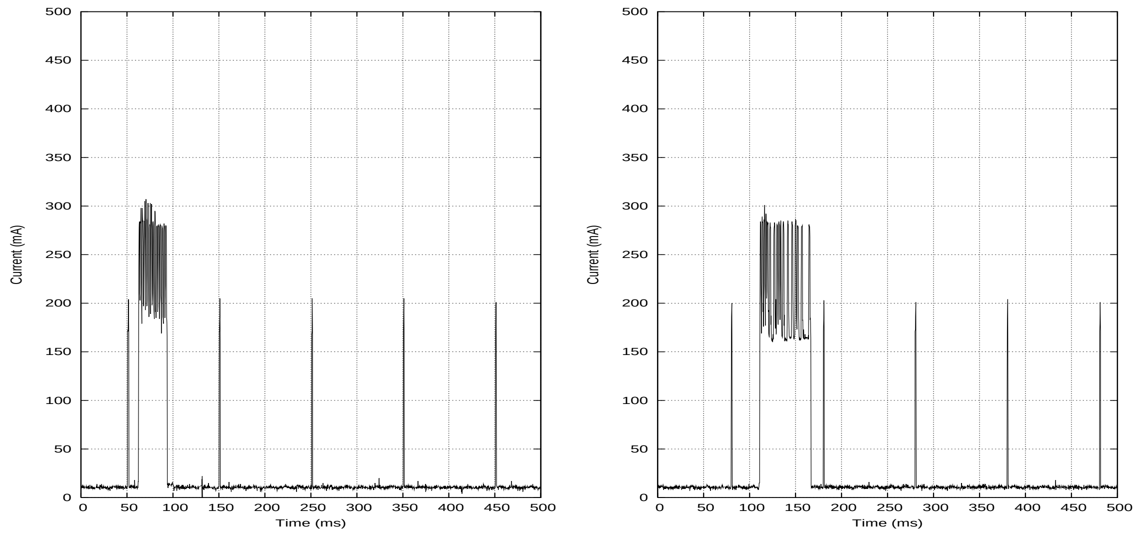


Figure 62: Packet transmission without (left) and with (right) dynamic frequency scaling.

current drawn by the network device when a packet of size 12kByte is transmitted over a wireless connection at the default clock frequency of 206.4MHz on the Compaq iPAQ, requiring 31ms. In contrast, Figure 62 (right) shows the same packet being transmitted when the clock frequency is scaled down to 59MHz, requiring 55.5ms. IEEE 802.11b supports fragmentation, where packets are transmitted in fragments of typically 256 to 2048 bytes. In the example shown here, at 59MHz, the network card has to remain active 78% longer than compared to the case with 206.4MHz. This means that frequency scaling – and similarly voltage scaling – affects adversely the doze mode approach for wireless network cards. Further, as shown in Figure 63, slowing down the clock frequency also reduces the burstiness of data transmission. The first half of Figure 63 shows the transmission of two packets as a single burst at 206MHz; after 200ms the clock frequency is changed to 59MHz, resulting in both packets being transmitted separately. The reason is that once the first packet is entered into a packet queue, it takes longer, possibly beyond the deadline of the first packet, until other applications can generate more packets. Figure 64 explains this

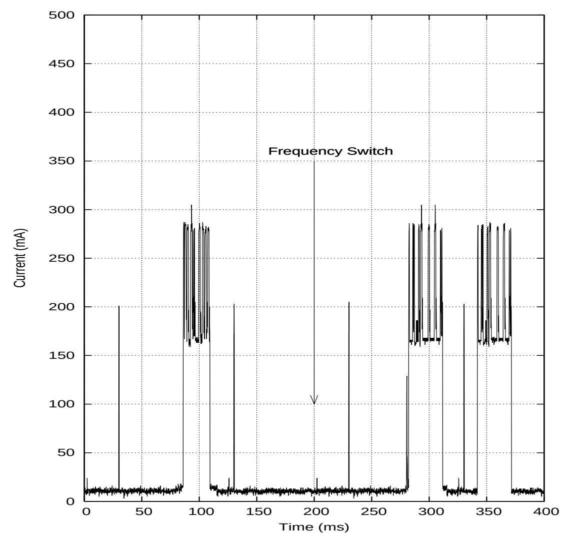


Figure 63: Effect of frequency scaling on transmission times and burstiness.

situation in detail. Consider two tasks that generate packets with identical periods, but with a phase delay. When task 1 submits a packet, the transmission is delayed until the packet's deadline. In the meantime, task 2 also generates a packet, which is placed in the scheduler queue behind task 1's packet and both are transmitted together. In the snapshot shown in the graph, the network is active 4 times. Now consider the case with a low clock frequency, the execution of tasks 1 and 2 requires more time, leading to a scenario where task 2 cannot generate the packet in time such that it could be send at the same time as task 1's packet, and therefore will be transmitted at its own deadline. The result is that the network has to be active twice as frequent. Again, this reduces the possibility of forming large bursts. This problem is addressed by modifying the way frequency scaling is used.

7.1.1 Modified Frequency Selection Algorithm

The frequency selection method introduced in the previous chapter is now adjusted in the following way. The clock frequency is kept at its maximum (f_{max}) while the packet scheduler queue is non-empty and while packets are transmitted. This ensures that packet transmission is executed at the highest clock speed and that the traffic burstiness is maximized. Once the queue is empty, Q-Fabric switches to a lower clock frequency to exploit idle CPU

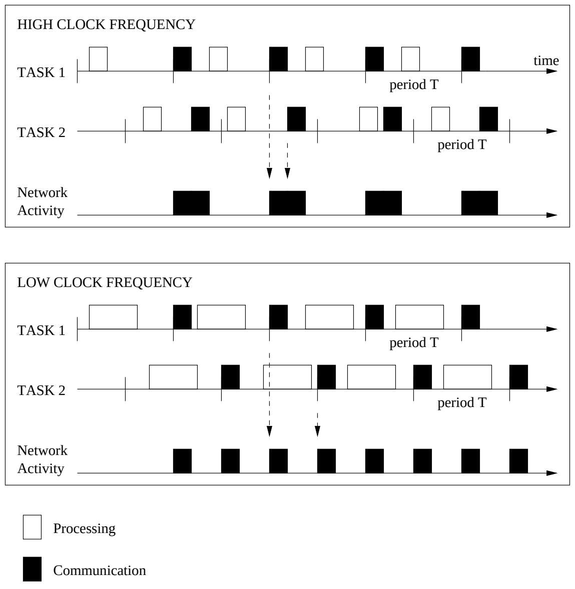


Figure 64: Effect of frequency scaling on burstiness.

time. Here, k' is calculated as follows:

$$k' = \frac{U - U'}{(\sum (1 - \frac{x_i}{y_i}) * \frac{C_i}{T_i}) - U'}$$

U' is obtained by measuring and averaging the time from the arrival of the first packet in the device's queue until the end of the transmission of the last packet in the next burst. It expresses the part of the total system utilization that is measured when f_{max} is used as the clock frequency. Therefore, k' and consequently k_n and f_n are determined for the period of time when the queue is empty; whenever the queue is non-empty, the CPU clock is run at f_{max}. Figure 65 (left) shows the transmission of one packet at f_{max} on the first y-axis and the used clock frequency on the second y-axis (in this case, 206.4MHz). Figure 65 (right) compares this to the transmission of a packet at f_{59} (59MHz). Finally, Figure 66 shows the packet transmission with the modified approach, where computations are performed with frequency f_{59} until packets are enqueued, then the clock frequency is switched to f_{max}.

Burst Sizes. If the average burst size is small (e.g., 1 packet), there is no advantage in switching the clock frequency at the time the packet is placed into the queue. Therefore, the actually achieved burst sizes are monitored and Q-Fabric switches between two modes:

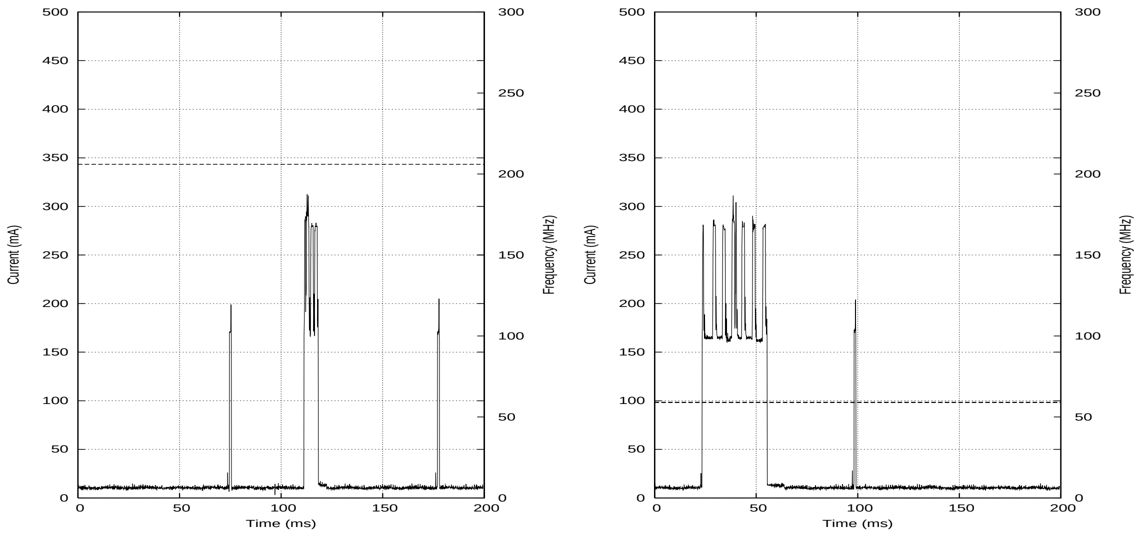


Figure 65: Packet transmission at maximum clock frequency (left) and minimum clock frequency (right).

(a) if the average burst size is 2 or higher, the clock frequency is switched to f_{max} at the time the first packet is enqueued, (b) otherwise, the clock frequency is switched to f_{max} at the time packet transmission starts.

Scheduler Queue Fill Level. The ready time of a packet determines the packet's earliest possible transmission time. Typically, ready times are either the current time or in the near future, e.g., for a live video streaming application where the CPU's and packet scheduler's attributes are synchronized, the ready time is at most the offset Θ ahead of the current time. However, it could happen that an application submits packets with ready times further in the future, preventing them from being transmitted in the current burst. In that case, the algorithm would not switch back to f_n after the burst is transmitted because packets are still pending in the queue. Here, the algorithm is modified to ensure that the CPU clock is executed at the lower frequency for at least 50% of the time, to exploit idle CPU times. This is achieved by setting the device clock to f_n after transmitting a burst and the CPU clock is set to f_{max} at packet arrival in the queue *only if* the CPU clock has run at f_n for at least t_b time units, where t_b is the measured average burst size.

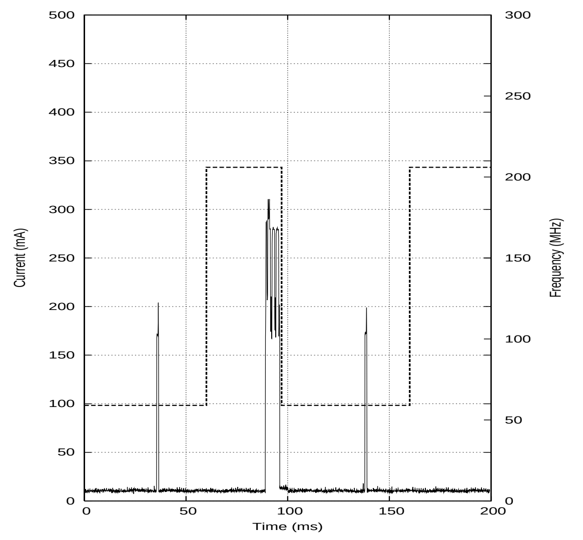


Figure 66: Packet transmission and clock frequency.

7.1.2 CPU Scheduling

With each task or task's time slice is associated a probability of network communication. For example, a probability of '0' may be assigned to processing-intensive tasks that do not use the network card, where a probability of '1' may be assigned to communication-intensive tasks. In the general case, such probabilities can be assigned to tasks through the use of online monitoring. For example, a video capture task may require 3 time slices for each frame. While the first and second slices in each period are required for camera operation, the processing, and the compression of a video frame, the third time slice may be the one during which the frame is actually submitted to the network connection. In that case, time slices 1 and 2 will have a low probability and slice 3 has a high probability of network communication. Consider Figure 67, where the top graph shows the task or time slice communication probabilities, and the bottom shows the network activity. While the network queue is empty, the CPU scheduler chooses low probability tasks in order to maximize the time the device is operated with a low clock frequency, and in addition, the network card can be kept in doze mode. When the first packet is generated and placed in the packet scheduler queue, the scheduler begins to prefer tasks with high probabilities of packet generation in order to ensure that as many packets as possible will arrive in the packet

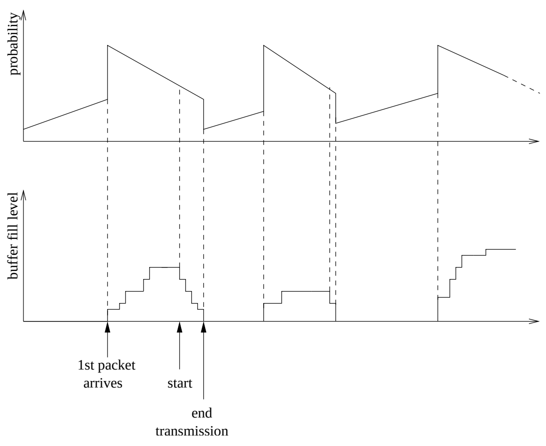


Figure 67: Probability of network activity and packet scheduler queue fill levels.

queue to maximize burstiness. Further, the DFS mechanism switches the clock frequency to the highest possible clock frequency. Once the packets are transmitted, the clock frequency is set to low and low-probability tasks are preferred. Table 5 shows the precedence rules used by the modified DWCS CPU scheduler to find the next process to be scheduled. The

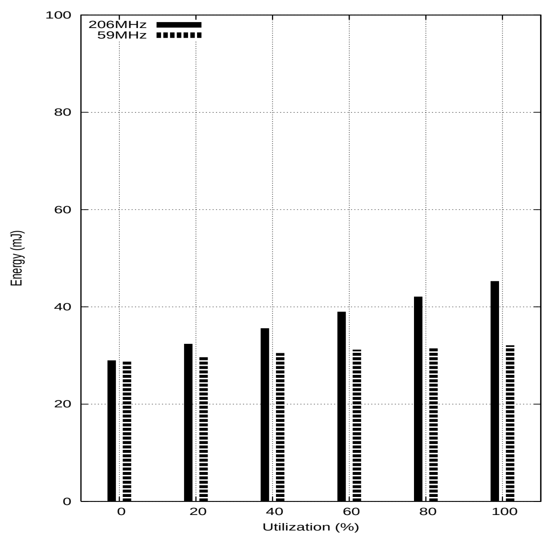
Table 5: Modified precedence rules.

Earliest deadline first (EDF)
Equal deadlines, then order lowest/highest probability of network activity first
Equal deadlines and equal probabilities, then order tightest window-constraint first
Equal deadlines and zero window-constraints, then order highest window-denominator first
Equal deadlines and equal non-zero window-constraints, then order lowest window-numerator first
All other cases: first-come first-serve

new rule (shown in bold) ensures that a task's probability of network activity is considered when multiple tasks share the same deadline. Note that this rule is executed frequently (i.e., task deadlines are identical), since (a) periods for multimedia streaming are likely to be 'close' to each other (e.g., corresponding to transmission rates of audio and video), (b) periods in the DWCS CPU scheduler are 'aligned' such that *hyper periods* can be formed, which facilitate the computation of CPU utilization or clock frequencies, and (c) the timing parameters in DWCS are not chosen randomly, but as multiples of ‘jiffies’, the unit of time used in Linux (e.g., 10ms).

7.1.3 Overhead Considerations

By running the device at two different clock frequencies (f_{max} and f_n), it is possible that CPU idle times are not fully exploited. However, this is acceptable when the potential loss in energy savings for running the CPU at higher clock frequencies is compared with the gains in energy savings for aggregating larger bursts and faster data transmissions. Consider the graphs in Figure 68. The left graph compares the energy costs of a mobile device at both the default clock frequency (206.4MHz) and the lowest possible frequency (59MHz), where the x-axis shows CPU utilization for a snapshot of 100ms. This could be the period of a video encoder for a frame rate of 10fps. If the utilization is 100%, then the difference in energy consumption is 13.1mJ for the execution of the same code at 206.4 and 59MHz. The handheld used in these experiments has 12 different clock frequencies, i.e., between two neighboring frequencies, the difference in energy consumption is only slightly more than 1mJ. Since the algorithm uses f_{max} for at most 50% of the time, the differences in energy consumption are 6.55mJ between the highest and lowest possible clock frequencies and 0.5mJ between neighboring clock frequencies in the worst case. In contrast, Figure 68 (right) shows the difference in energy costs for the transmission of data over a wireless link at the default frequency and the lowest clock frequency. For example, a packet of size 19.3kBytes takes 25ms to be transmitted, i.e., the wireless card is active for that period of time. At 59MHz, even though the costs for the actual transmission are identical, the card has to stay in idle mode longer because of the delays caused by the the lower clock frequency. Here, the card has to stay active for 80ms, resulting in an additional energy consumption of 44mJ. Again, the difference for neighboring clock frequencies would be about 3.7mJ. That is, if the clock frequency is set to a higher value than necessary, then the loss of energy savings for the CPU are outweighed by a gain in energy savings for the network transmission by a factor of at least 7.4 in this example (the higher the communication, i.e., more or larger packets, the higher this factor).



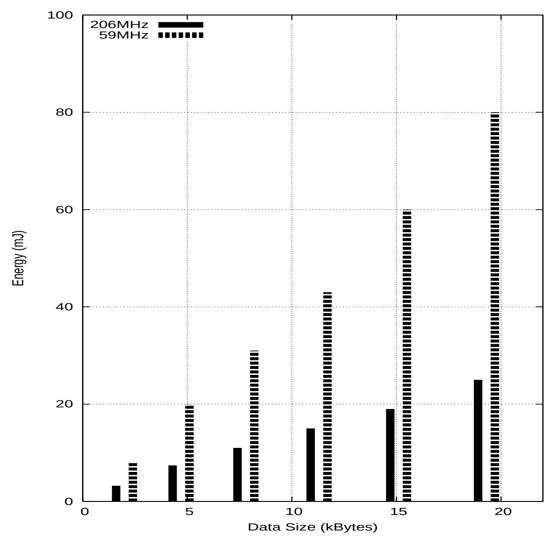
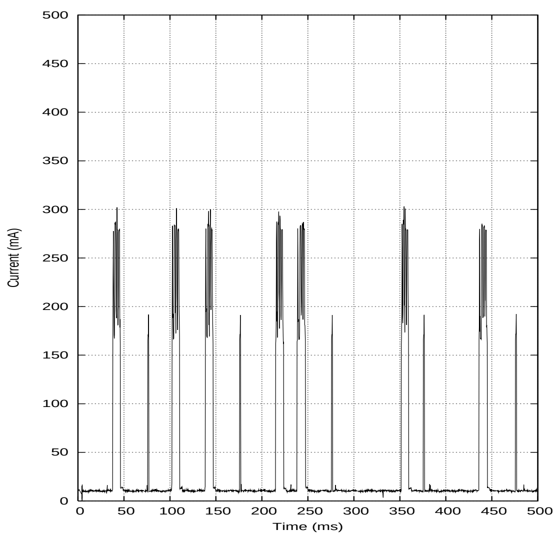


Figure 68: Energy savings of frequency scaling (left) and energy costs of network communication (right).

7.1.4 Evaluation: Video Streaming

The multi-resource scheduling approach described in this dissertation has been implemented as part of Q-Fabric. Significant kernel changes include the CPU scheduler, the packet scheduler, and the Orinoco device driver. In this section, an experiment with a simulated video capturing application is performed, i.e., images are read from disk instead of a camera and written back to disk at the receiver side. Clients specify the desired frame rate of the video stream, here a frame rate of 5 frames per second is used. The video streaming application shares the CPU with 3 other applications with the following packet deadlines: 500ms, 333ms, 250ms. The intent is to emulate a mobile device used in autonomous robotics scenarios, where robots collect sensor data in a search and rescue mission. These other applications may involve dynamic sensor data acquisition and transmission, monitoring and control information for QoS management purposes (as used in Q-Fabric), other multimedia applications, and other background applications running on the same device. Figure 69 (left) shows the scenario when no power management is performed, but the CPU uses frequency scaling. The device runs at the lowest possible frequency, 59MHz and the applications transmit together 7 packets per 500ms on average. In contrast, Figure 69 (right) shows the same scenario, however, power management has been enabled. Instead of waking



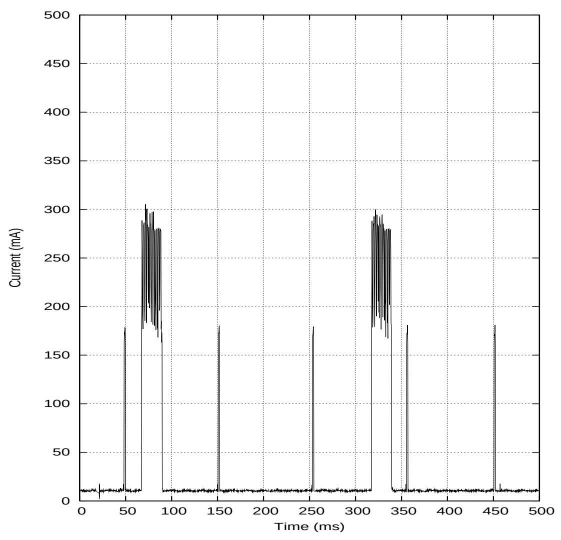


Figure 69: No traffic shaping (left) and energy-efficient traffic shaping (right).

up the device 7 times in the time frame of 500ms, it is woken up only twice. In addition, the total ‘awake time’ is 43.95ms, whereas in Figure 69 (left) the device has to be awake 68.15ms or 55% longer. In terms of energy savings, for the snapshot shown in this example (500ms), the energy savings related to frequency scaling are 25mJ. If the network device uses doze modes, but no frequency scaling is used, the energy savings are 389mJ. However, if both approaches are used together, the energy savings are 392mJ, only slightly more than using doze modes alone. Using the approach introduced here, the savings increase to 420mJ, a difference of about 7%. If the packet sizes are doubled, this increases to about 13%, i.e., the achievable savings depend on the number of packets transmitted and the size of these packets.

Discussion. The resulting energy savings depend on the amount of data and the number of packets being transmitted from a mobile device. The approach introduced here succeeds in improving energy savings when there are several communicating real-time applications residing on the same device. Further, the results depend on the task and packet scheduler parameters, i.e., on the ready times and the deadlines. If deadlines of different data streams are aligned such that the maximum burst size is always one, the number of switches between low- and high-power modes can not be reduced. The second part of this approach, coordinating frequency scaling with data transmissions, aims at decreasing the transmission times of packets. The energy savings are most significant if fragmentation is used, and here, if small fragment sizes (e.g., 256 bytes) are used (in this work's examples, fragment size was 2kbytes). If no fragmentation is used, retransmissions in the case of contention or errors are more expensive, however, the potential energy savings achievable with our approach are not as significant. Finally, the coordination of frequency scaling and data communications is useful if there is CPU idle time available, i.e., the clock frequency can be switched to lower speeds.

7.2 Integrated Media Transcoding, Frequency Scaling, and Traffic Shaping

If the previously introduced approaches (energy-aware media transcoding, frequency scaling, and the traffic shaping approach) are used simultaneously on a device, potential conflicts have to be studied. With energy-aware media transcoding, 'energy-expensive' resources are traded for 'energy-cheap' resources, thereby reducing the overall energy requirements. For the device under consideration, the previous chapter has showed that adding CPU resources allows the video streaming application to transform a given image into a smaller-sized image, and thereby reducing the network requirements. So far, these results have not considered that changing these resource allocations will also affect the energy management techniques that can be applied. More specifically, if CPU resource requirements are added in the form of media transcoders, the overall CPU utilization is increased. In Chapter 6, an approach to determine the desired clock frequency depending on the CPU utilization was introduced, that is:

$$k' = \frac{U_{max}}{\sum(1 - \frac{x_i}{y_i}) * \frac{C_i}{T_i}}$$

In other words, a scaling factor k' is determined based on the maximum allowable utilization (U_{max}) and the actual utilization. The resulting value k' is compared to the previously obtained scaling factors, and the scaling factor k_n closest to k' ($k_n \leq k'$) is selected, and

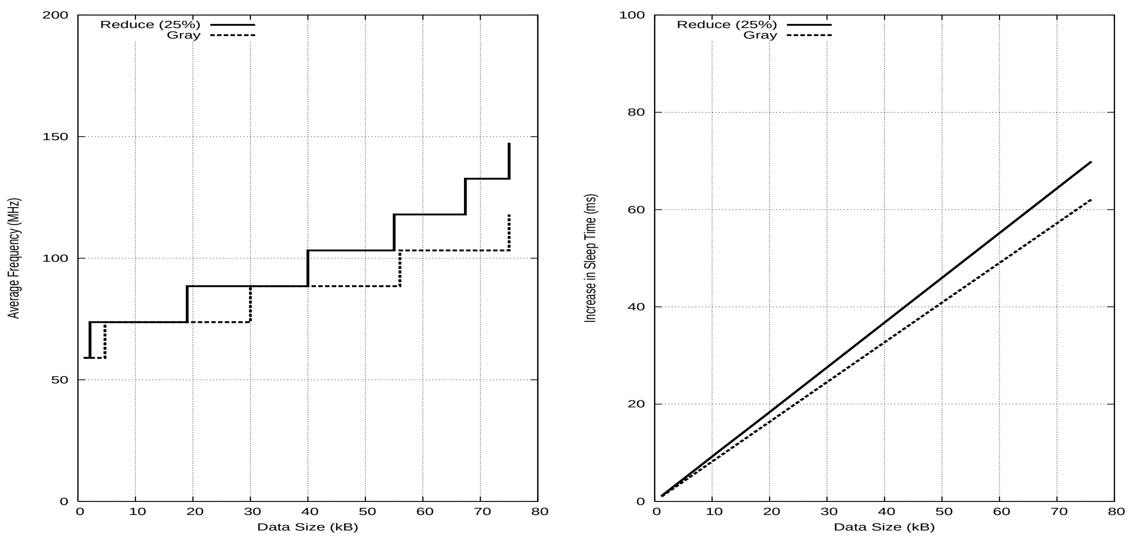


Figure 70: Effect of media transcoding on average clock frequencies (left) and increase in network card sleep times (right).

the clock frequency is adjusted to frequency f_n. However, when media transcoding is used, the CPU utilization is increased, decreasing the scaling factor and possibly the clock frequency. Figure 70 (left) shows this scenario for two different transcoders, the 'reduce' transcoder (reducing an image to 25% of its original size) and the 'gray' transcoder. Here, the average clock frequency is shown as a function of the size of the original image. Further, in both cases, the CPU utilization is 20% when no transcoder is used, resulting in DFS choosing the lowest possible clock frequency of 59MHz. However, when a transcoder is used, the clock frequency has to be adjusted – depending on the image size – to reflect the larger CPU utilization, e.g., for image sizes of 50kBytes, the computed clock frequency is 103.2MHz (instead of 59MHz). In contrast, using a transcoder typically decreasing the image size, resulting in reduced network overheads. Figure 70 (right) shows the increase in sleep times for the network card (per period) caused by the transmission of smaller-sized images. Summarizing, to accurately predict the potential energy savings that can be obtained from the use of transcoders, one has to consider the energy loss caused by running at a higher clock frequency and the energy gain caused by allowing the network card to remain in the sleep mode longer. The following sections will utilize these insights to accurately make global energy management decisions.

7.3 Energy Management Directives

While a significant amount of research has focused on the development of energy management techniques on individual devices, only little attention has been given to *global* or *system-wide* energy management. For example, in a distributed video conferencing application, the desired global goal is that all participants can communicate as long as possible, i.e., it is more desirable that all participants can communicate for the same amount of time than having one subset of participants that can communicate for only a significant amount shorter than another subset. In cluster servers, it is desirable to direct incoming requests to a minimal set of servers to maximize the number of servers that can be kept in low-power mode, in order to preserve energy and to reduce the energy or cooling costs. In other words, depending on the application and the user preferences, a global energy management goal has to be declared. In this work, these goals are described in *energy management directives*. The goal of QoS management is therefore (a) to ensure that clients receive data and application performance in a form that corresponds to their QoS needs, such that (b) a chosen energy management directive is observed. The following directives are supported:

1. Maximize Sender's Operational Time (MAX-SOT). The goal of this directive is to minimize the energy requirements of the *sender* of a media stream, e.g., in the case of transcoding, all applicable transcoder functions are evaluated in order to find the ones that minimize the energy costs for the sender, without considering the consequences to the receiver. Figure 71 shows the scenario when the sender can preserve energy by exploiting transcoders. The left graph shows the energy costs for the transmission of the original data (E_1) and the transmission of some smaller-sized version of the same data (E_2). On the other hand, the second graph shows the processing costs for the execution of a transcoder to obtain the smaller-sized version of the data (E_{cpu}). If $E_1 - E_2 > E_{cpu}$, then the transcoder execution will result in reduced energy consumption at the sender.

2. Maximize Receiver's Operational Time (MAX-ROT). Here, the goal is to minimize the receiver's energy consumption, e.g., the sender will perform (a) all *mandatory* transcoders and (b) all *optional* transcoders that result in reduced energy requirements at the receiver.

3. Maximize Application's Operational Time (MAX-AOT). Here, the battery load levels of both sender and receiver have to be compared periodically and depending on their current levels, either MAX-ROT or MAX-SOT has to be followed. The goal of this directive is to keep the battery loads of the sender and the receiver at about the same level, in order to ensure that the distributed application can run as long as possible (i.e., both sender and receiver will run out of battery power at about the same time).

4. Minimize System's Energy Consumption (MIN-SEC). The energy requirements of the entire system are to be minimized, i.e., the combined energy consumption of sender and receiver has to be kept low. This is particularly important wherever energy consumption translates into costs (e.g., power supply for large hosting centers and cooling costs). Again, consider the example of media transcoding. At the sender, this approach is similar to MAX-SOT, however, with one important difference: if the execution of a mandatory transcoder at the sender causes the sender to consume more energy compared to transmitting the original data, then the sender will still – unlike in MAX-SOT – execute the transcoder *if* the combined costs of the transmission of the original data and the costs for the execution of the transcoder at the receiver are greater than the processing and transmission costs at the sender.

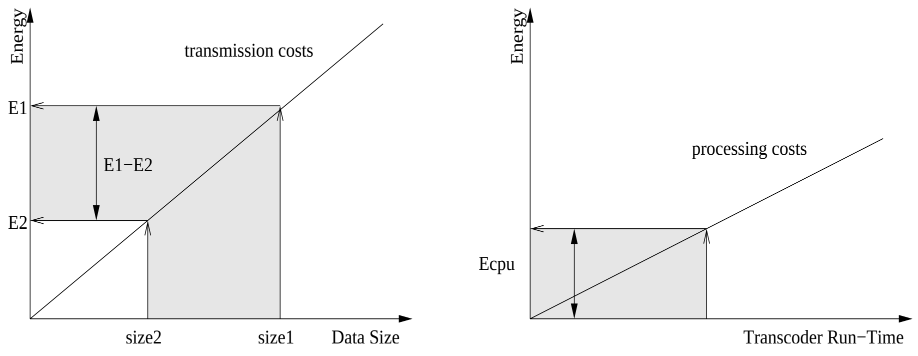


Figure 71: The MAX-SOT directive.

Depending on the directive chosen, energy management decisions have to be evaluated and compared in order to determine the optimal action for a given directive. These decisions have to be re-evaluated frequently due to changes in application conditions, including changes in user requirements, resource allocations, or data content. In order to decide where to focus energy management, the battery charge levels of all involved devices have to be obtained and compared periodically. In the devices under consideration in this dissertation, the battery monitor DS2760 is integrated in each device and returns the current charge level in mAh. The next section describes how the previously introduced energy management techniques are linked together in Q-Fabric to provide global energy management.

7.4 Global Energy Management

7.4.1 A Transcoding Framework

The transcoding framework introduced in Chapter 6 is now adjusted to consider energy management directives, i.e., the decision on where to execute a transcoder and which transcoder(s) to execute is made such that a energy management directive is observed. Figure 72 shows the architecture of the modified approach, where the new parameters are

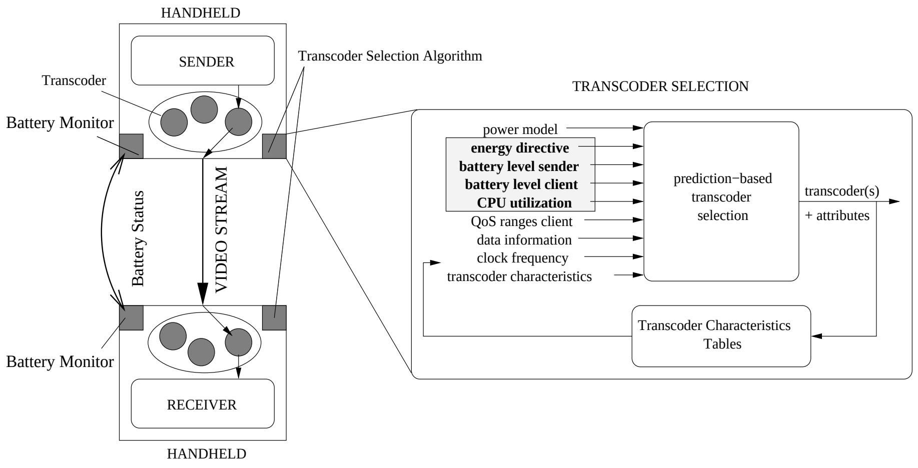


Figure 72: Modified transcoder selection.

shown in bold letters:

- **Energy Management Directive.** Sample directives are MAX-SOT, MAX-ROT, MAX-AOT, and MIN-SEC.

- **Battery Load Levels.** In the case of the MAX-AOT directive, the battery load levels of both sender and receiver have to be compared periodically (e.g., once per minute). The battery loads of the receiver and the sender are obtained using the */proc* virtual filesystem (in the file */proc/asic/ds2760*), which returns the battery charge level in mAh.

- **CPU Utilization.** This is required to let the transcoder selection approach consider the effects of the execution of a transcoder on the device's clock frequency (as discussed earlier in this chapter).

The transcoder selection is executed by a QoS manager at the sender of a data stream, and the receiver is informed about decisions via control events over a Q-Channel. The transcoder selection algorithm is invoked periodically, however, approaches are possible where the algorithm runs only in response to certain events, e.g., whenever the user changes the QoS requirements, or when the video frames change significantly in size or content. Further, resource monitors and both sender and client watch the current battery charge levels and exchange this information via monitoring events (see Figure 72).

As before, the transcoder algorithm has to determine the gains and costs of executing a transcoder at either the sender or the receiver. For the sender-side, for each transcoder one can obtain an estimated transcoder run-time from the input data size and the ratio r_r:

$$rt_t = r_r * size(d)$$

Then the run-time - energy factor $K_r(n)$ is used to obtain the energy consumed by the execution of the transcoder at a particular clock frequency n, adjusted by the potential overheads E_{dfs}:

$$E_t = K_r(n) * rt_t + E_{dfs}.$$

Here, E_{dfs} is the loss of energy caused by executing a transcoder that forces the DFS algorithm to select a higher clock frequency due to the increased CPU utilization. To obtain this energy, the transcoder algorithm compares the current clock frequency n with the required clock frequency by adjusting the CPU utilization (obtained via ECalls) with the costs of executing the transcoder. That is, a new scaling factor k'' is computed and compared to the list of scaling factors for the device and the new scaling factor is used to determine a new clock frequency m:

$$m = n + i \ (i >= 0).$$

In other words, the new clock frequency is i steps higher than the original frequency ($i >= 0$) and therefore the energy overhead E_{dfs} is computed as follows:

$$E_{dfs} = (i * E_{step})/rate.$$

Here, E_{step} is the energy difference between two successive energy levels for a duration of the transcoder execution period, which then multiplied with the number of steps (i) and divided by the streaming rate returns the energy loss by having to use a higher clock frequency.

One can then obtain the output data size from the ratio r_d:

$$size(d') = r_d * size(d)$$

and the output data size ($size(d')$) is used along with the factor $K_d(n)$ to determine the energy consumption for the transmission of the output data:

$$E_{d'} = K_d(n) * size(d') - E_{sleep}.$$

Here, $E_{d'}$ is adjusted by E_{sleep}, the energy gain caused by transmitting a smaller sized image therefore allowing the network card to remain in sleep mode longer. E_{sleep} is determined with:

$$E_{sleep} = ((t_x - t'_x) * P_{transmit})/rate.$$

In words, E_{sleep} is computed by multiplying the difference in transmission times for the original image and the transcoded image (obtained from the off-line measured network characteristics) and the power required to transmit data, divided by the streaming rate.

As before, the input data size $size(d)$ is used to determine the energy consumption for the transmission of the original data:

$$E_d = K_d(n) * size(d).$$

This leads us to the transcoder energy quality:

$$TEQ = (E_d - E_{d'}) - E_t - E_a.$$

At the receiver side, E_{dfs} is computed identically, however, the formula for computing E_{sleep} has to be adjusted as follows:

$$E_{sleep} = ((t_r - t'_r) * P_{receive})/rate.$$

That is, it is computed with the times required to receive a transcoded image and the power required to receive it.

Based on the directive chosen and the current batter charge levels, the algorithm decides which transcoders to execute and where to execute them, e.g., given two transcoder energy qualities TEQ_s (sender) and TEQ_r (receiver), two battery charge levels b_s (sender) and b_r (receiver), and E_{rcvr}, which is the energy overhead of executing a transcoder at the receiver, the simplified rules for mandatory transcoders are as follows:

```
if (MAX-SOT) {
  if (TEQs > 0)
    execute at sender;
  else
    execute at receiver;
} else if (MAX-ROT) {
  if (TEQr > 0)
    execute at sender;
  else if (Ercvr > abs(TEQr))
    execute at sender;
  else
    execute at receiver;
} else if (MAX-AOT) {
  if (bs > br)
    same as MAX-ROT;
  else if (bs < br)
    same as MAX-SOT;
  else
    same as MIN-SEC;
} else if (MIN-SEC) {
  if ((TEQs + TEQr) > Ercvr)
    execute at sender;
  else
    execute at receiver;
}
```

7.4.2 Case Study: Video Streaming

Consider a video conferencing application between multiple PDAs as described in the previous chapters. The appropriate energy management directive here is MAX-AOT, to maximize the application's operational time. In this example, video conferencing is simulated in the sense that images are read from disk at the sender and written back to disk at the receiver, instead of using camera and display for capture and replay.

QoS Specification. The system's managed resources are CPU, network, and energy, and the DWCS scheduler is used to schedule both CPU and network packets. The application's utility functions and QoS ranges are as follows:

$$U_{rate} = rate * 8;$$

$$U_{size} = (width + height)/2;$$

$$Q_{rate} = \{10, 25\};$$

$$Q_{width} = \{50, 100\};$$

$$Q_{height} = \{50, 100\};$$

In words, the frame rate has to be managed between 10 and 25fps (frames per second), where the maximum utility achievable is 200 ($rate * 8$). The image width and height are between 50 and 100 pixels, resulting in a maximum utility of 100, that is the frame rate has a weight $w_r = 2$ and the image size has a weight $w_s = 1$. The CPU and network resources are similarly managed as described in Chapter 5 and the energy management techniques described in the previous chapters are deployed.

Results. Figure 73 (left) shows the achieved utility for a snapshot of 1 minute. For the first 30 seconds, the goal of QoS management is to maximize the total application utility (maximum is 300), after 30 seconds, the goal is changed such that the energy consumption of the device is to be minimized, but the total application utility has to be kept above the minimum (130). Figure 73 (right) shows the energy consumption for the same experiment. The first 30 seconds show the energy consumption (for a period of 1s) when no power management techniques are used, after 30 seconds, the goal is to minimize the device's consumed energy, which is reduced to about 50% when coordinated energy management is used. However, when the energy management approaches are used without coordination,

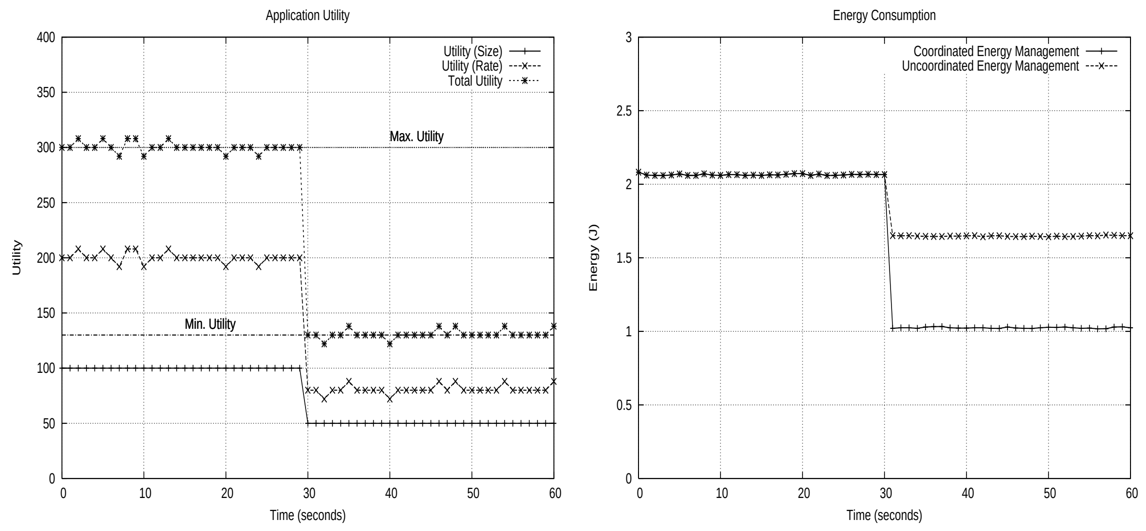


Figure 73: Utility graphs for image size, frame rate, and total application utility (left) and total device energy consumption (right).

the energy savings are significantly lower, i.e., compared to the 50% savings with coordinated energy management, the uncoordinated approach manages to save only 20%. This is mostly due to the effects of frequency scaling on the sleep mode times of the network card as described in the previous chapters. If broken down, 60% of the energy savings are due to the sleep modes of the network card, 34% are due to the media transcoding, and the rest is due to dynamic frequency scaling and possible other unaccounted effects.

7.5 Summary

This chapter concludes the work on integrated QoS management with a study on the conflicting effects of multiple energy management techniques and introduces approaches to carefully integrate them. Further, this chapter introduced the concept of energy management directives, which determine how to use local energy management techniques to save energy wherever needed. For example, media transcoding can be used in ways that reduce energy consumptions on either the sender- or receiver-side of video streaming. The results in this chapter underline the importance of careful integration and show the effects if multiple adaptation techniques are used with and without cooperation. For example, while the combined use of all techniques can achieve energy savings of up to 20%, the coordinated or integrated approach can achieve energy savings of up to 50%.

CHAPTER 8

RELATED WORK

8.1 Adaptive QoS Management

Previous research has used middleware to 'bind' the multiple machines, applications, and resource managers that implement QoS provisioning, resource management, and performance differentiation for distributed applications and platforms, sometimes enhanced by operating system (OS) extensions on individual machines [57, 81]. However, there remain some open problems with such middleware-based approaches. First, user-level approaches must rely on the voluntary participation of applications in QoS management, thereby enabling non-participants to threaten the guarantees made to participants. Second, the granularity at which resources can be managed and the fidelity of such resource management are not always sufficiently high to meet applications' performance needs. Reasons for this include (1) inappropriate delays of QoS management [117], often aggravated by the overheads of application-level QoS managers' interactions with the system-level mechanisms they must use to understand current resource usage and availability, and (2) inappropriate interfaces provided by operating systems that require managers to poll for changes in resource state or make unnecessary resource reservations, as also noted in [57, 105]. Earlier research principally considered adaptations performed at application-level [10, 25]. Cooperation between application-level adaptations and system-level resource management is implemented as middleware [37] or as point solutions for specific resources like computer networks [49]. Many operating system services have been proposed in the past which enhance these systems in order to make them suitable for multimedia applications [134, 93, 57].

In [69], the authors describe a hierarchical approach to Quality of Service management, where adaptations are performed within applications. This is achieved by requesting applications to export a control interface allowing QoS management to activate adaptive mechanisms on a set of tunable QoS parameters. In the EPIQ framework [129], a distributed

Quality of Service architecture is described, providing QoS specification, end-to-end QoS negotiation and establishment, and QoS adaptation, implemented as middleware solution. EPIQ provides the tools necessary for the integration and customization of QoS management and resource management for real-time applications. Further implementations include QLinux [136], which, like Q-Fabric, modifies a Linux kernel to provide QoS guarantees. The focus in QLinux, however, is on scheduling algorithms for CPU, packet, and disk scheduling. Q-RAM [40] is an analytical approach for QoS management with multiple resources and multiple dimensions of QoS requirements, which allows resources to be traded off against each other to obtain the same level of QoS. In the FARA framework [116], the authors use hierarchical adaptation at middleware level to utilize multiple adaptation approaches for real-time systems and the SWiFT project [41] resulted in a toolkit for constructing feedback loops from libraries, where the interaction between components is limited to a simple input/output model.

Other recent contributions include multi-resource solutions [113, 151] using libraries [116] and/or additional servers [51] for distributed adaptation and resource management [141]. Cooperation between application-level adaptations and system-level resource management is implemented as middleware [37] or as point solutions for specific resources like computer networks [49]. Other work, such as OMEGA [90] or QuO [115] introduce general QoS architectures to address the end-to-end management of QoS.

The research described here provides efficient system-level and architectural abstractions that enable the cooperative management of kernel-level resources with the run-time adaptation of user-level applications. Abstractions are implemented as low-overhead, kernel-level services, jointly termed *Q(uality)-Fabric*, which are based on the notions of distributed event services and event handlers. The issues addressed by Q-Fabric services concern the limitations experienced in current systems for cooperative resource management and application adaptation, which are due to (a) the specific interfaces defined between applications and resource managers and (b) the limited interactions permitted between distributed resource managers. Moreover, (c) cooperation is particularly difficult when resource management actions are performed at kernel-level. Finally, (d) in large-scale multimedia applications, the coordination of a large number of distributed resource managers for heterogeneous resources and the adaptation of all instances of an application can be overwhelming in complexity [116].

8.2 Energy Management

There has been substantial work on power management for mobile devices, including low-power modes for disks and networks [50, 20], power-aware scheduling policies [122, 80, 60, 101], and energy management techniques for wireless communication [2, 88, 102, 110]. The authors in [58] describe a modification to the power saving mechanism in the IEEE 802.11 Distributed Coordination Function. Their aim is to remove the overheads associated with the TIM windows and to increase the available bandwidth for data transmission. In the PAMAS [131] approach, separate control and data channels are used, where the control channel is used to determine when and how long to turn off a wireless network card. Dynamic Modulation Scaling (DMS) [127] has been introduced as the network variant of frequency and voltage scaling for processors. This approach is used in [111], where a WFQ CPU scheduler is modified to take advantage of DMS. In [157, 155, 86], the authors address the integration of resource management across different layers of a system.

Other approaches have off-loaded processing to other hosts, e.g., to extend the battery life on mobile devices [71] or to minimize energy costs in Internet data centers [99]. For mobile devices, researchers have developed energy management techniques, e.g., by addressing the energy costs of wireless data transmissions [110], by scaling device activity according to applications' resource needs [22, 122], or by switching between modes with different power characteristics [20]. The work introduced in this dissertation differs from previous work on quality-aware transcoding [18] in that the focus is on the reduction of energy consumption according to global energy management directives. This work is similar to the one proposed by the authors in [19] in that they also address the use of transcoding to reduce energy consumption, however they focus on storage and energy limitations in an image capture device. This work builds further on similar work presented in [7], which investigates the trade-offs between processing costs of lossless compression algorithms and networking costs of transmitting reduced-size data. In contrast to that, this thesis addresses more generally the transcoding of media streams in order to maintain application-specific QoS with particular focus on observing global energy management directives. Further, other approaches address the integration of multiple approaches to preserve energy [87, 124], e.g., by coordinating adaptation across protection boundaries [157, 121, 87, 33]. The GRACE project [157, 156] proposes coordinated adaptation of hardware, operating system, and application layers to achieve fine-grained tuning of system utility. In the Puppeteer project [32], a middleware framework is introduced that also uses transcoding to minimize energy requirements. The focus here is on closed-source applications, where the authors show that applications can significantly reduce energy usage by allowing applications and power management systems to incorporate knowledge of each other's activities. In this thesis, feedback is used to improve the predictions made of future resource requirements. This is similar to the work presented in [33, 3], where the authors adapt network and CPU resources based on history, or in [55], where MPEG decoders are used to maximize a system's lifetime. Also, in [87], the authors explore the use of transcoders for multimedia streaming, where transcoders reside on proxies. This dissertation is mostly concerned with fully mobile situations, i.e., both stream generation and replay are performed on battery-operated devices and devices cannot rely on support infrastructure such as extensible and customizable base stations. Finally, the approach introduced in this document complements work done for server-side traffic shaping, e.g., as in [20], and also has similarities to the approach in [97], where disk access pattern burstiness is increased in order to decrease the time a disk has to be kept in high-power active mode.

8.3 ECalls

Multimedia and real-time applications often require support from the underlying operating system to achieve their real-time and QoS guarantees. This has led to the development of operating system services that are responsible for process scheduling [81, 93, 95] and resource management tasks [56, 126]. When using such kernel-level services, applications interact with them via system calls or signals. Since such interactions can be costly, researchers have sought ways to control call overheads [26, 73], and they have attempted to reduce the frequency of system calls, e.g., by extending kernels with appropriate application-specific functionality [8, 29, 39]. Further, upcall primitives have been introduced [23, 53], to better integrate the kernel- and application-level actions carried out for certain requests. Real-time variants of upcalls address the specific needs of multimedia and real-time applications [43].

Common elements of these solutions are (1) the need to share information about events between kernel- and user-level facilities that are critical to application performance, and (2) to be able to act on such information in a timely fashion. For instance, communication rates can be adjusted based on information about buffer fill-levels [134], if such information is made available and acted upon with little delay. The same requirements exist when exploiting knowledge about the ACK/NAK behavior of communication protocols to alter the behavior of media streaming [63] or even scientific applications [128]. In fact, past work has shown that system quality may be reduced rather than improved by runtime adaptation if such actions are not performed within certain tolerances [117].

Particularly the poor scalability of system calls such as *select()* or *poll()* has been the topic of previous research. In [26], the authors implement an integrated buffer management and transfer mechanism optimized for high-bandwidth I/O. The goal is to achieve high throughput across protection domains by exploiting page remapping and shared memory techniques. The performance of *select* and *poll* system calls and POSIX.4 real-time signals is further analyzed in [17, 6]. In [152], the authors introduce an approach to efficiently transfer data and control between application and system domain and also provides rate-based flow control. This is achieved by using I/O efficient buffers and independently scheduled kernel threads. In [6], the authors introduce an event delivery system allowing applications to register interest in event sources like sockets. However, the application still has to poll for events, whereas ECalls is able to notify a process of pending events by executing a handler function and raising its scheduler priority. In [96], the authors introduce a new flexible and general I/O approach that avoids data copying with minimal overhead. I/O completion ports, supported in Windows NT, use a number of pre-forked threads to handle incoming events. A throttling mechanism limits the number of currently active threads to avoid large context switching overheads. Other implementations include the ones addressed in [68] and [6], where both support mechanisms to (a) register interest in events and (b) collect these events at a later time. However, these event delivery mechanisms are pull-based, i.e., applications have to scan some form of lists, flags, or queues. Similarly, the approach in [118] aims at reducing the need for system calls to obtain network connection states from the kernel. Although ECalls offers the tools to implement similar approaches, it also supports push-based delivery of events, e.g., through real-time signals or direct invocation of handler code – including dynamically generated code – in the system domain. Further, ECalls' event-aware CPU scheduling can enhance the CPU scheduler with knowledge about event receipt, e.g., which processes have events pending, and influences scheduling decisions correspondingly. This functionality builds on and extends earlier work. For example, in [78], the authors propose an integrated framework for interacting process and message schedulers for distributed real-time systems. Similarly, in [66], an architecture that is aware of the real-time characteristics of tasks sending and receiving network packets is introduced. The goal is to overcome the traditional deficiencies like FIFO ordering of incoming packets and processing in the kernel of all packets regardless of their priority to the receiving application.

8.4 KECho

KECho uses anonymous event-based notification and data exchange, thereby contrasting it to lower-level mechanisms like kernel-to-kernel socket communications, RPC [11], or the RPC-like active messaging developed in previous work [140]. Furthermore, compared to object-based kernel interactions [47] or to the way in which distributed CORBA, DCOM, or Java objects interact at the user level [94, 12, 150], KECho's model of communication provides improved flexibility, since its use of anonymous event notification permits services to interact without explicit knowledge of each others identities. Further, event exchange with KECho can maintain the time order between events as well as deadline- or priority-based event ordering.

The KECho kernel-level publish/subscribe mechanism shares several important attributes with its user-level counterparts. First, KECho events may be used to notify interested subscribers of internal changes of system state or of external changes captured by the system [77]. Second, it may be used to implement kernel-level coordination among distributed services, perhaps even to complement the application-level coordination implemented with user-level event notification architectures [119, 27, 48, 77]. Applications constructed with event-based architectures include peer-to-peer applications like distributed virtual environments, collaborative tools, multi-player games, and certain real-time control systems. Third, KECho's functionality is in part identical to that of known user-level event systems, sharing functionality such as real-time attributes associated with events, event filtering, and anonymous and asynchronous communication providing decoupled linkage of event publishers and subscribers.

8.5 Monitoring

Different monitoring tools operate at different levels of granularity with consequent trade-offs between the quality of the information monitored and the overhead associated with it. Cluster performance monitoring tools have been developed to allow system administrators to monitor cluster state. A typical tool consists of two major entities: a *server* that collects state information of a cluster and a *GUI-based* front-end, which provides a visualization of system activity. Parmon [15], Ganglia [120], Smile [137], and many others belong to this kind. These tools cannot deliver very frequent monitoring updates.

Paradyn [83] is a tool that does performance monitoring for long running parallel and distributed applications. It adapts the performance of these applications by dynamically instrumenting them at run-time using the monitoring information that it collects. The Pablo [114] toolkit focuses on collecting and doing statistical analysis of performance data in scalable parallel systems. Falcon [46] is an application specific on-line monitoring system that provides its own set of instrumentation libraries and controls which the developer of an application can use to tune its performance.

The Supermon [132] cluster monitoring system uses a modified version of the SunRPC remote-status *rstat* protocol [135] to collect data from remote cluster nodes. This modified protocol is based on *symbolic expressions* which allows it to operate in a heterogeneous environment. The Supermon kernel patch exports various kernel monitoring information via a *sysctl* call. Scalability can be a problem in Supermon because of the centralized data concentrator, which collects monitoring data from all cluster nodes.

HPVM's performance monitor [123] is targeted toward Windows NT clusters. Like Q-Fabric, HPVM has the ability to automatically adapt cluster applications. The SHRIMP performance monitor [72] makes a compromise between high level software monitoring and low level kernel monitoring to accurately monitor various resource information. MAGNeT [30] uses an instrumented kernel to export kernel events to user space. It maintains a circular buffer in the kernel where all events are recorded and other nodes can obtain these records by contacting a daemon, called *magnetd*. The kernel must be configured at compile-time to enable the monitoring, which increases the administrative overhead as monitoring needs change.

In comparison, Q-Fabric provides a low overhead, fine-grained, kernel level monitoring facility, with communication based on strict kernel-kernel messaging. Q-Fabric is extensible, i.e., new monitoring functionality can be added dynamically, e.g., through loadable kernel modules. Further, Q-Fabric is customizable, i.e., applications can fine-tune the distributed resource monitoring via parameters and dynamically generated code.

CHAPTER 9

CONCLUSIONS AND FUTURE WORK

9.1 Thesis

The goal of this research was twofold: (1) to provide the framework for efficient and low-overhead QoS management and (2) to provide the tools necessary for the efficient integration of multiple QoS management and adaptation approaches. The first is addressed by providing a system-level event-based implementation of QoS and resource management mechanisms, achieving low overheads, fine-grain access to system-level resources, and decoupled communication between multiple resource monitors and managers. The second problem is addressed by providing a unifying integration interface, using which multiple adaptation strategies at different system layers and different hosts can freely communicate, share information, or cooperate their adaptive measures. Careful integration of these management mechanisms is key to attaining effective adaptations and to prevent adverse and unintended effects, e.g., when the actions of one adaptive approach contradict the actions of another. This work resulted in Q-Fabric, a set of system-level tools, which if used cooperatively facilitate the deployment of QoS management approaches. The remainder of this chapter describes the contributions of this dissertation and discusses future research directions.

9.2 Research Contributions

9.2.1 Conceptual Contributions

The main contributions of this work are the implementation and verification of a quality management approach that addresses the needs of current and future complex distributed applications. It provides the performance and timeliness required of QoS-aware applications by using a full in-kernel solution, based on low-overhead communication across protection boundaries and between nodes. It supports the transparency commonly required in quality management approaches, while also being highly customizable, particularly through Q-Fabric's dynamic code generation ability, allowing it to dynamically extend and modify a kernel's functionality. The in-kernel approach also enforces the participation of all applications in quality management, and the event-based approach of Q-Fabric supports the flexibility necessary to address the dynamics of large-scale distributed applications. As a specific contribution, the Q-Fabric approach is used to develop and deploy quality management techniques with *energy* as a first-class resource. Here, the integrated management of multiple energy management techniques is essential to minimizing the system's energy requirements.

9.2.2 Artifacts

The following software artifacts are among this research:

- ECalls: a low-overhead alternative to existing system call and signal mechanisms, where applications can choose the method they desire. Novel elements include shared memory segments between protection boundaries, conditional system calls, or real-time events with dynamically generated event handlers in the operating system kernel.
- ECalls-based CPU scheduling: an approach to attain high responsiveness by allowing event and CPU scheduling to cooperate, as shown with a real-time CPU scheduler.
- Distributed extension and customization interface: operating system kernels can be modified and extended (locally and remotely) with Q-Fabric's dynamic code generation. This functionality is used to deploy resource monitors, utility functions, event handlers, or event filters.
- KECho: a kernel-level approach to the familiar publish/subscribe communications, including a number of enhancements, e.g., a network monitor layer (for higher responsiveness), asynchronous kernel-level socket communication, real-time events, and 'communicating' filters, i.e., the filter of a control channel (Q-Channel) can change

attributes of a filter in a data channel (application-to-application communication), thereby affecting data streams immediately without involvement of applications.

- Integration of the real-time CPU scheduler DWCS and the frequency scaling (DFS) capabilities of modern mobile processors: frequency scaling is a popular way to preserve energy on mobile devices; this dissertation introduced an approach to utilize DFS efficiently in conjunction with the DWCS real-time scheduler.

- Energy-aware media transcoding: media transcoding refers to the downsampling of video or audio data in order to reduce their size and therefore the communication overheads. Energy savings can be achieved if the computation overheads associated with the execution of a transcoder are smaller than the gains in transmitting data of reduced size.

- Energy-efficient traffic shaping: this approach carefully integrates the frequency scaling capabilities of the CPU and the low-power sleep mode of the network card, in order to increase the burstiness of network traffic (to avoid costly switches between the sleep and active modes of a wireless network card) and to prevent adverse effects of using both approaches simultaneously.

9.2.3 Evaluation Results

The results in this dissertation underline the advantages of a system-level event-based approach to QoS management and the importance of careful integration of multiple quality management techniques. Both microbenchmarks and real applications (such as video players or web servers) show that both ECalls- and KECho-based solutions have significant performance advantages compared to their user-level counterparts. Further, the real-time capabilities of these approaches make them attractive to applications with stringent timing requirements. Chapter 5 presented an example of a multi-peer video streaming scenario based on the Q-Fabric approach, showing that the best adaptation results are achieved if (1) application-level and system-level QoS management are used cooperatively and (2) end-to-end QoS management is used, e.g., by sharing monitoring information among all peers of a distributed application. Chapters 6 and 7 presented a comprehensive study of QoS management in mobile multimedia systems, where energy is the constraining resource. Here, the results underline the importance of integrated energy management, where energy management techniques cooperate within layers of a system, across layers, and across device boundaries. The specific example used in these chapter, a video streaming application, shows that energy savings can be as high as 50% if integration is used, as opposed to 20% without integration.

9.3 Future Research Directions

The Q-Fabric approach can be used to study a multitude of questions related to distributed quality management, including the development and verification of sample control policies integrating the management of multiple resources. Of particular interest here is the issue of the non-linearity in resource management, i.e., the re-allocation of one resource to a given process may affect the resource requirements of the same process for a different resource. If this is not taken into account, adaptive measures can aggravate resource shortages as previously identified in [117]. While the presented work focuses primarily on energy, future work will address other resources, such as disk, CPU, network, and memory. Q-Fabric offers a powerful set of tools for developers to implement efficient and application-specific control mechanisms in a distributed system, using off-the-shelf components. However, a certain knowledge of the system architecture of all involved nodes, of the applications involved, and of the potential resource requirements and adaptation possibilities is required to efficiently implement *working* solutions. Quality management toolkits, implemented as user libraries, can help in the development of management policies and monitoring/control components, e.g., such a toolkit could allow application developers to use *knobs* that can be turned 'manually' (command interface) or automatically by the applications in order to find optimal management policies. Results of such experimental policy development will be knowledge about the resource information to be monitored and adaptation strategies to control resource allocations. A first step into this direction will be the development of a graphical interface to develop, deploy, and verify QoS management policies. The ultimate goal will be that applications themselves can fine-tune coarse quality management strategies.

QoS management will continue to be an important area of research in Computer Science. Particular with the proliferation of new computing environments and applications, QoS-awareness has to be introduced into both applications and the underlying systems. Three areas are of particular interest because of their challenges due to heterogeneity and size. First, large-scale multimedia applications over the Internet will be used to perform video conferences, remote teaching, or multi-player games. While the work in this dissertation has focused on end-system management, future work will have to extend this to the networks linking these systems. For example, overlay networks promise to efficiently select communication paths based on bandwidth or latency requirements, which are essential for QoS-aware applications. Second, sensor networks are typically very resource-scarce, complicating the QoS management due to this limitations. Further, energy plays a particular role, and 'intelligent' resource sharing and communication can support the task of energy management. Finally, large scale Internet data centers are emerging, where QoS management is of importance to ensure that (a) service providers achieve optimal throughput, (b) clients receive their requested services with low latencies, and (c) thermal management is part of the load balancing decisions to prevent overheating or failure of parts of the data center.

REFERENCES

[1] Abdelzaher, T. F. and Shin, K. G., "QoS Provisioning with qContracts in Web and Multimedia Servers," in *Proc. of the IEEE Real-Time Systems Symposium*, December 1999.

[2] Agrawal, S. and Singh, S., "An Experimental Study of TCP's Energy Consumption over a Wireless Link," in *Proc. of the 4th European Personal Mobile Communications Conference*, February 2001.

[3] Anand, M., Nightingale, E. B., and Flinn, J., "Self-Tuning Wireless Network Power Management," in *Proc. of the 9th Intl. Conference on Mobile Computing and Networking*, September 2003.

[4] Aron, M., Sanders, D., and Druschel, P., "Scalable Content-aware Request Distribution in Cluster-based Network Servers," in *Proc. of the USENIX Annual Technical Conference*, June 2000.

[5] Balaouras, P., Stavrakakis, I., and Merakos, L., "Potential and Limitations of a Teleteaching Environment based on H.323 Audio-Visual Communication Systems," in *Proc. of the TERENA Networking Conference*, May 2000.

[6] Banga, G., Mogul, J., and Druschel, P., "A Scalable and Explicit Event Delivery Mechanism for UNIX," in *Proc. of the USENIX Annual Technical Conference*, June 1999.

[7] Barr, K. and Asanovic, K., "Energy Aware Lossless Data Compression," in *Proc. of the 1st Intl. Conference on Mobile Systems, Applications, and Services*, May 2003.

[8] Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M. E., Becker, D., Chambers, C., and Eggers, S., "Extensibility, Safety and Performance in the SPIN Operating System," in *Proc. of the 15th ACM Symposium on Operating System Principles*, December 1995.

[9] Bestavros, A., Crovella, M., Liu, J., and Martin, D., "Distributed Packet Rewriting and its Application to Scalable Web Server Architectures," in *Proc. of the 6th IEEE Intl. Conference on Network Architectures*, October 1998.

[10] Bihari, T. E. and Schwan, K., "Dynamic Adaptation of Real-Time Software," *ACM Transactions on Computer Systems*, vol. 9, pp. 143–174, May 1991.

[11] Birrell, A. D. and Nelson, B. J., "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems*, vol. 2, February 1984.

[12] Box, D., *Understanding COM.* Addison-Wesley, 1997.

[13] Brinthaupt, D., Letham, L., Maheshwari, V., Othmer, J., Spiwak, R., Edwards, B., Terman, C., and Weste, N., "A Video Decoder for H.261 Video Teleconferencing and MPEG Stored Interactive Video Applications," in *Proc. of the IEEE Intl. Solid-State Circuits Conference*, February 1993.

[14] Busse, I., Deffner, B., and Schulzrinne, H., "Dynamic QoS Control of Multimedia Applications based on RTP," *Computer Communications*, January 1996.

[15] Buyya, R., "PARMON: A Portable and Scalable Monitoring System for Clusters," *Software Practice and Experience Journal*, vol. 30, no. 7, pp. 723–739, 2000.

[16] Cardellini, V., Colajanni, M., and Yu, P. S., "Dynamic Load Balancing on Web-server Systems," *IEEE Internet Computing*, vol. 3, May/June 1999.

[17] Chandra, A. and Mosberger, D., "Scalability of Linux Event-Dispatch Mechanisms," in *Proc. of the USENIX Annual Technical Conference*, June 2001.

[18] Chandra, S., Ellis, C. S., and Vahdat, A., "Application-Level Differentiated Multimedia Web Services Using Quality Aware Transcoding," *IEEE Journal on Selected Areas in Communications*, 2000.

[19] Chandra, S., Ellis, C. S., and Vahdat, A., "Managing the Storage and Battery Resources in an Image Capture Device (Digital Camera) using Dynamic Transcoding," in *Proc. of the 3rd ACM Intl. Workshop on Wireless Mobile Multimedia*, August 2000.

[20] Chandra, S. and Vahdat, A., "Application-specific Network Management for Energy-aware Streaming of Popular Multimedia Formats," in *Proc. of the USENIX Annual Technical Conference*, June 2002.

[21] Chang, J.-H. and Tassiulas, L., "Energy Conserving Routing in Wireless Ad-hoc Networks," in *Proc. of the IEEE Infocom Conference*, March 2000.

[22] Choi, K., Kim, K., and Pedram, M., "Energy-Aware MPEG-4 FGS Streaming," in *Proc. of the 40th Design Automation Conference*, June 2003.

[23] Clark, D., "The Structuring of Systems Using Upcalls," in *Proc. of the 10th ACM Symposium on Operating Systems Principles*, December 1985.

[24] Cucchiara, R., Grana, C., and Prati, A., "Semantic Video Transcoding for Live Video Server," in *Proc. of the ACM Multimedia Conference*, December 2002.

[25] Diot, C., "Adaptive Applications and QoS Guarantees," *Multimedia and Networking*, 1995.

[26] Druschel, P. and Peterson, L. L., "Fbufs: A High-bandwidth Cross-domain Transfer Facility," in *Proc. of the 14th ACM Symposium of Operating Systems Principles*, December 1993.

[27] Eisenhauer, G., Bustamante, F. E., and Schwan, K., "Event Services for High Performance Computing," in *Proc. of the High Performance Distributed Computing Symposium*, August 2000.

[28] Eisenhauer, G., Bustamante, F. E., and Schwan, K., "A Middleware Toolkit for Client-Initiated Service Specialization," *ACM SIGOPS*, vol. 35, April 2001.

[29] Engler, D. R., Kaashoek, F. M., and O'Toole Jr., J., "Exokernel: An Operating System Architecture for Application-Level Resource Management," in *Proc. of the 15th ACM Symposium on Operating System Principles*, December 1995.

[30] Feng, W., Broxton, M., Engelhart, A., and Hurwitz, G., "MAGNeT: A Tool for Debugging, Analysis and Reflection in Computing Systems," in *Proc. of the 3rd IEEE/ACM Intl. Symposium on Cluster Computing and the Grid*, May 2003.

[31] Ferrari, G., "Applying Feedback Control to QoS Management," in *Proc. of the 7th CaberNet Radicals Workshop*, February 2002.

[32] Flinn, J., de Lara, E., Satyanarayanan, M., Wallach, D. S., and Zwaenepoel, W., "Reducing the Energy Usage of Office Applications," in *Proc. of the IFIP/ACM Intl. Conference on Distributed Systems Platforms (Middleware)*, November 2001.

[33] Flinn, J. and Satyanarayanan, M., "Energy-Aware Adaptation for Mobile Applications," in *Proc. of the 17th ACM Symposium on Operating Systems Principles*, December 1999.

[34] Florissi, P., *QuAL: Quality Assurance Language*. PhD thesis, Columbia University, 1996.

[35] Foster, I., Kesselman, C., Lee, C., Lindell, R., Nahrstedt, K., and Roy, A., "A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation," in *Proc. of the Intl. Workshop on Quality of Service*, June 1999.

[36] Foster, I., Kesselman, C., Nick, J., and Tuecke, S., "Grid Services for Distributed System Integration," *IEEE Computer*, 2002.

[37] Foster, I., Roy, A., and Sander, V., "A Quality of Service Architecture that Combines Resource Reservation and Application Adaptation," in *Proc. of the 8th Intl. Workshop on Quality of Service*, June 2000.

[38] Ganev, I., Schwan, K., and Eisenhauer, G., "Kernel Plugins: When a VM is too much," *Proc. of the 3rd Virtual Machine Research and Technology Symposium*, May 2004.

[39] Ghormley, D., Rodrigues, S., Petrou, D., and Anderson, T., "SLIC: An Extensibility System for Commodity Operating Systems," in *Proc. of the USENIX Annual Technical Conference*, June 1998.

[40] Ghosh, S., Rajkumar, R., Hansen, J., and Lehoczky, J., "Scalable Resource Allocation for Multi-Processor QoS Optimization," in *Proc. of the 8th Intl. Conference on Distributed Computing Systems*, May 2003.

[41] Goel, A., Steere, D., Pu, C., and Walpole, J., "SWiFT: A Feedback Control and Dynamic Reconfiguration Toolkit," in *Proc. of the 2nd Usenix Windows NT Symposium*, September 1998.

[42] Gomes, J., Campbell, A. T., Naghshineh, M., and Bisdikian, C., "PARO: Power-aware Routing in Wireless Packet Networks," in *Proc. of the 6th IEEE Intl. Workshop on Mobile Multimedia Communications*, November 1999.

[43] Gopalakrishnan, R. and Parulkar, G., "Efficient User Space Protocol Implementations with QoS Guarantees using Real-time Upcalls," *IEEE/ACM Transactions on Networking*, 1998.

[44] Goyal, P., Vin, H. M., and Cheng, H., "Start-Time Fair Queueing: A Scheduling Algorithm for Integrated Services Packet Switching Networks," in *Proc. of the ACM SIGCOMM Conference*, August 1996.

[45] Grunwald, D., Levis, P., Morrey III, C. B., Neufeld, M., and Farkas, K. I., "Policies for Dynamic Clock Scheduling," in *Proc. of the 4th Symposium on Operating System Design and Implementation*, October 2000.

[46] Gu, W., Eisenhauer, G., Schwan, K., and Vetter, J., "Falcon: On-line Monitoring and Steering of Large-Scale Parallel Programs," *Concurrency: Practice and Experience*, vol. 6, April–June 1998.

[47] Hamilton, G. and Kougiouris, P., "The Spring Nucleus: A Microkernel for Objects," in *Proc. of the Summer 1993 USENIX Conference*, Summer 1993.

[48] Harrison, T. H., Levine, D. L., and Schmidt, D. C., "The Design and Performance of a Real-time CORBA Object Event Service," in *Proc. of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, October 1997.

[49] He, Q. and Schwan, K., "IQ-RUDP: Coordinating Application Adaptation with Network Transport," in *Proc. of the Intl. Symposium on High Performance Distributed Computing*, July 2002.

[50] Helmbold, D. P., Long, D. D. E., and Sherrod, B., "A Dynamic Disk Spindown Technique for Mobile Computing," in *Proc. of the Intl. Conference on Mobile Computing and Networking*, November 1996.

[51] Huand, J., Jha, R., Heimerdinger, W., Muhammad, M., Lauzac, S., Kannikeswaran, B., Schwan, K., Zhao, W., and Bettati, R., "RT-ARM: A Real-Time Adaptive Resource Management System for Distributed Mission-Critical Applications," in *Proc. of the Workshop on Middleware for Distributed Real-Time Systems and Services*, December 1997.

[52] Huang, J., Wang, Y., and Cao, F., "On Developing Distributed Middleware Services for QoS- and Criticality-Based Resource Negotiation and Adaptation," *Journal of Real-Time Systems*, 1998.

[53] Hutchinson, N. C. and Peterson, L. L., "The x-Kernel: An Architecture for Implementing Network Protocols," *IEEE Transactions on Software Engineering*, vol. 17, pp. 64–76, January 1991.

[54] International Standards Organization, Geneve, Switzerland, *Information Technology - Software Product Evaluation - Quality Characteristics and Guidelines for their Use*, 1991.

[55] Iranli, A., Choi, K., and Pedram, M., "Energy-Aware Wireless Video Streaming," in *Proc. of the Workshop on Embedded Systems for Real-Time Multimedia*, October 2003.

[56] Jones, M., "Adaptive Real-Time Resource Management Supporting Modular Composition of Digital Multimedia Services," in *Proc. of the 14th Intl. Workshop on Network and Operating System Support for Digital Audio and Video*, November 1993.

[57] Jones, M. B., Rosu, D., and Rosu, M.-C., "CPU Reservations: Efficient Predictable Scheduling of Independent Activities," in *Proc. of the 16th ACM Symposium on Operating System Principles*, October 1997.

[58] Jung, E. and Vaidya, N., "An Energy-Efficient MAC Protocol for Wireless LANs," in *Proc. of the IEEE Infocom Conference*, June 2002.

[59] Kapadia, N. H. and Fortes, J. A. B., "On the Design of a Demand-Based Network-Computing System: The Purdue University Network Computing Hubs," in *Proc. of the IEEE Intl. Symposium on High Performance Distributed Computing*, July 1998.

[60] Kim, W., Shin, D., Yun, H.-S., Kim, J., and Min, S. L., "Performance Comparison of Dynamic Voltage Scaling Algorithms for Hard Real-Time Systems," in *Proc. of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium*, September 2002.

[61] Kon, F., Yamane, T., Hess, C. K., Campbell, R. H., and Mickunas, M. D., "Dynamic Resource Management and Automatic Configuration of Distributed Component Systems," in *Proc. of the 6th USENIX Conference on Object-Oriented Technologies and Systems*, January/February 2001.

[62] Kravets, R., Calvert, K., Krishnan, P., and Schwan, K., "Adaptive Variation of Reliability," in *Proc. of the Conference on High Performance Networking*, April 1997.

[63] Kravets, R., Calvert, K., and Schwan, K., "Payoff Adaptation of Communication for Distributed Interactive Applications," in *The Journal for High Speed Networking: Special Issue on Multimedia Networking*, Winter 1999.

[64] Kravets, R. and Krishnan, P., "Application-driven Power Management for Mobile Communication," *Wireless Networks*, vol. 6, no. 4, pp. 263–277, 2000.

[65] Lee, C., Lehoczky, J., Rajkumar, R., and Siewiorek, D., "On Quality of Service Optimization with Discrete QoS Options," in *Proc. of the IEEE Real Time Technology and Applications Symposium*, June 1999.

[66] Lee, C., Yoshida, K., Mercer, C., and Rajkumar, R., "Predictable Communication Protocol Processing in Real-Time Mach," in *Proc. of the IEEE Real-time Technology and Applications Symposium*, June 1996.

[67] Lei, Z. and Georganas, N. D., "Rate Adaptation Transcoding for Precoded Video Streams," in *Proc. of the ACM Multimedia Conference*, December 2002.

[68] Lemon, J., "A Generic and Scalable Event Notification Facility," in *Proc. of the FREENIX Track of the USENIX Annual Technical Conference*, June 2001.

[69] Li, B., Kalter, W., and Nahrstedt, K., "A Hierarchical Quality of Service Control Architecture for Configurable Multimedia Applications," *Journal on High Speed Networks, Special Issue on Management of Multimedia Networking*, vol. 9, no. 3-4, 2001.

[70] Li, L., Lizheng, F., Guixing, L., and Ping, W., "A Hierarchical Architecture for Distributed Network Management," in *Proc. of the Intl. Conference on Internet Computing*, June 2001.

[71] Li, Z., Wang, C., and Xu, R., "Computation Offloading to Save Energy on Handheld Devices: A Partition Scheme," in *Proc. of the Intl. Conference in Compilers, Architecture, and Synthesis for Embedded Systems*, November 2001.

[72] Liao, C., Martonosi, M., and Clark, D. W., "Performance Monitoring in a Myrinet-connected Shrimp Cluster," in *Proc. of the 2nd SIGMETRICS Symposium on Parallel and Distributed Tools*, August 1998.

[73] Liedtke, J., "Improving IPC by Kernel Design," in *Proc. of the 14th ACM Symposium on Operating Systems Principles*, December 1993.

[74] Lin, Y., Guo, K., and Paul, S., "Sync-MS: Synchronized Messaging Service for Real-Time Multi-Player Games," in *Proc. of the IEEE Intl. Conference on Network Protocols*, November 2002.

[75] Lorch, J. R. and Smith, A. J., "Improving Dynamic Voltage Scaling Algorithms with PACE," in *Proc. of the ACM SIGMETRICS Conference*, June 2001.

[76] Lu, S., Lee, K.-W., and Bharghavan, V., "Adaptive Service in Mobile Computing Environments," in *Proc. of the 5th Intl. Workshop on Quality of Service*, May 1997.

[77] Ma, C. and Bacon, J., "COBEA: A CORBA-Based Event Architecture," in *Proc. of the 4th USENIX Conference on Object-Oriented Technologies*, April 1998.

[78] Manimaran, G., Shashidhar, M., Manikutty, A., and Murthy, C., "Integrated Scheduling of Tasks and Messages in Distributed Real-time Systems," in *Proc. of the IEEE Workshop on Parallel and Distributed Real-time Systems*, April 1996.

[79] McNamee, D., Walpole, J., Pu, C., Cowan, C., Krasic, C., Goel, A., Wagle, P., Consel, C., Muller, G., and Marlet, R., "Specialization Tools and Techniques for Systematic Optimization of System Software," *Transactions on Computing Systems*, vol. 19, no. 2, pp. 217–251, 2001.

[80] Mejia-Alvarez, P., Levner, E., and Mosse, D., "Power-Optimized Scheduling Server for Real-Time Tasks," in *Proc. of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium*, September 2002.

[81] Mercer, C. W., Savage, S., and Tokuda, H., "Processor Capacity Reservation for Multimedia Operating Systems," in *Proc. of the IEEE Intl. Conference on Multimedia Computing and Systems*, May 1994.

[82] Mesarina, M. and Turner, Y., "Reduced Energy Decoding of MPEG Streams," in *Proc. of the Multimedia Computing and Networking Conference*, January 2002.

[83] Miller, B. P., Callaghan, M. D., Cargille, J. M., Hollingsworth, J. K., Irvin, R. B., Karavanic, K. L., Kunchithapadam, K., and Newhall, T., "The Paradyn Parallel Performance Measurement Tool," *IEEE Computer*, vol. 28, pp. 37–46, November 1995.

[84] Miyoshi, A., Lefurgy, C., Hensbergen, E. V., Rajamony, R., and Rajkumar, R., "Critical Power Slope: Understanding the Runtime Effects of Frequency Scaling," in *Proc. of the 16th Annual Intl. Conference on Supercomputing*, June 2002.

[85] Mogul, J. and Ramakrishnan, K., "Eliminating Receive Livelock in an Interrupt-driven Kernel," in *Proc. of USENIX Annual Technical Conference*, January 1996.

[86] Mohapatra, P., Li, J., and Gui, C., "QoS in Mobile Ad hoc Networks," *Special Issue on QoS in Next-Generation Wireless Multimedia Communications Systems in IEEE Wireless Communications Magazine*, June 2003.

[87] Mohapatra, S., Cornea, R., Dutt, N., Nicolau, A., and Venkatasubramanian, N., "Integrated Power Management for Video Streaming to Mobile Handheld Devices," in *Proc. of the ACM Multimedia Conference*, November 2003.

[88] Monks, J. P., Bharghavan, V., and Hwu, W. W., "A Power Controlled Multiple Access Protocol for Wireless Packet Networks," in *Proc. of the IEEE Infocom Conference*, April 2001.

[89] Mosberger, D. and Jin, T., "httperf - A Tool for Measuring Web Server Performance," in *Proc. of the Workshop on Internet Server Performance*, June 1998.

[90] Nahrstedt, K., Chu, H., and Narayan, S., "QoS-aware Resource Management for Distributed Multimedia Applications," *Journal on High-Speed Networking, IOS Press*, vol. 7, no. 3,4, pp. 227–255, 1998.

[91] Nahrstedt, K. and Smith, J., "The QoS Broker," *IEEE Multimedia*, vol. 2, no. 1, pp. 53–67, 1995.

[92] Nieh, J., Hanko, J. G., Northcutt, J. D., and Wall, G. A., "SVR4 UNIX Scheduler Unacceptable for Multimedia Applications," in *Proc. of the 4th Intl. Workshop on Network and Operating System Support for Digital Audio and Video*, November 1993.

[93] Nieh, J. and Lam, M., "SMART: A Processor Scheduler for Multimedia Applications," in *Proc. of the 15th Symposium on Operating Systems Principles*, December 1995.

[94] Object Management Group, *CORBAservices: Common Object Services Specification*, July 1997. http://www.omg.org/.

[95] P. Goyal, X. G. and Vin, H. M., "A Hierarchical CPU Scheduler for Multimedia Operating Systems," in *Proc. of the 2nd Symposium on Operating Systems Design and Implementation*, October 1996.

[96] Pai, V., Druschel, P., and Zwaenepoel, W., "Flash: An Efficient and Portable Web Server," in *Proc. of the USENIX Annual Technical Conference*, June 1999.

[97] Papathanasiou, A. E. and Scott, M. L., "Energy Efficiency through Burstiness," in *Proc. of the 5th IEEE Workshop on Mobile Computing Systems and Applications*, October 2003.

[98] Pillai, P. and Shin, K. G., "Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems," in *Proc. of the 18th ACM Symposium on Operating Systems Principles*, October 2001.

[99] Pinheiro, E., Bianchini, R., Carrera, E., and Heath, T., "Load Balancing and Unbalancing for Power and Performance in Cluster-Based Systems," in *Proc. of the Workshop on Compilers and Operating Systems for Low Power*, September 2001.

[100] Plagemann, T., Goebel, V., Halvorsen, P., and Anshus, O., "Operating System Support for Multimedia Systems," *The Computer Communications Journal*, vol. 23, pp. 267–289, February 2000.

[101] Poellabauer, C. and Schwan, K., "Power-Aware Video Decoding using Real-Time Event Handlers," in *Proc. of the 5th Intl. Workshop on Wireless Mobile Multimedia*, September 2002.

[102] Poellabauer, C. and Schwan, K., "Energy-Aware Media Transcoding in Wireless Systems," in *Proc. of the 2nd IEEE Intl. Conference on Pervasive Computing and Communications*, March 2004.

[103] Poellabauer, C., Schwan, K., Eisenhauer, G., and Kong, J., "KECho - Event Communication for Distributed Kernel Services," in *Proc. of the Intl. Conference on Architecture of Computing Systems*, April 2002.

[104] Poellabauer, C., Schwan, K., and West, R., "Coordinated CPU and Event Scheduling for Distributed Multimedia Applications," in *Proc. of the 9th ACM Multimedia Conference*, October 2001.

[105] Poellabauer, C., Schwan, K., and West, R., "Lightweight Kernel/User Communication for Real-Time and Multimedia Applications," in *Proc. of the 11th Intl. Workshop on Network and Operating Systems Support for Digital Audio and Video*, June 2001.

[106] Poletto, M., Engler, D., and Kaashoek, M. F., "tcc: A Template-based Compiler for 'C," in *Proc. of the 1st Workshop on Compiler Support for Systems Software*, February 1996.

[107] Pouwelse, J., Langendoen, K., Lagendijk, R., and Sips, H., "Power-Aware Video Decoding," in *Proc. of the Picture Coding Symposium*, April 2001.

[108] Provos, N. and Lever, C., "Scalable Network I/O in Linux," in *Proc. of the FREENIX Track of the USENIX Annual Technical Conference*, June 2000.

[109] Provos, N., Lever, C., and Tweedie, S., "Analyzing the Overload Behavior of a Simple Web Server," in *Proc. of the 4th Annual Linux Showcase and Conference*, October 2000.

[110] Qiao, D., Choi, S., Jain, A., and Shin, K. G., "MiSer: An Optimal Low-Energy Transmission Strategy for IEEE 802.11a/h," in *Proc. of the ACM/IEEE Intl. Conference on Mobile Computing and Networking*, September 2003.

[111] Raghunathan, V., Ganeriwal, S., Schurgers, C., and Srivastava, M., "E2WFQ: An Energy Efficient Fair Scheduling Policy for Wireless Systems," in *Proc. of the Intl. Symposium on Low-Power Electronics and Design*, August 2002.

[112] Rajkumar, R., Lee, C., Lehoczky, J., and Siewiorek, D., "A Resource Allocation Model for QoS Management," in *Proc. of the IEEE Real Time Systems Symposium*, December 1997.

[113] Rajkumar, R., Lee, C., Lehoczky, J., and Siewiorek, D., "Practical Solutions for QoS-based Resource Allocation Problems," in *Proc. of the IEEE Real-Time System Symposium*, December 1998.

[114] Reed, D. A., Aydt, R. A., Noe, R. J., Roth, P. C., Shields, K. A., Schwartz, B. W., and Tavera, L. F., "Scalable Performance Analysis: The Pablo Performance Analysis Environment," in *Proc. of the Scalable Parallel Libraries Conference*, April 1993.

[115] Rodrigues, C., Loyall, J. P., and Schantz, R. E., "Quality Objects (QuO): Adaptive Management and Control Middleware for End-to-End QoS," in *Proc. of OMG's 1st Workshop on Real-Time and Embedded Distributed Object Computing*, July 2000.

[116] Rosu, D., Schwan, K., and Yalamanchili, S., "FARA - A Framework for Adaptive Resource Allocation in Complex Real-Time Systems," in *Proc. of the 4th IEEE Real-Time Technology and Applications Symposium*, June 1998.

[117] Rosu, D., Schwan, K., Yalamanchili, S., and Jha, R., "On Adaptive Resource Allocation for Complex Real-Time Applications," in *Proc. of the 18th IEEE Real-Time Systems Symposium*, December 1997.

[118] Rosu, M.-C. and Rosu, D., "Kernel Support for Faster Web Proxies," in *Proc. of USENIX Annual Technical Conference*, June 2003.

[119] Rowstron, A., Kermarrec, A.-M., Druschel, P., and Castro, M., "SCRIBE: The Design of a Large-scale Event Notification Structure," in *Proc. of the 3rd Intl. Workshop on Networked Group Communications*, November 2001.

[120] Sacerdoti, F. D., Katz, M. J., Massie, M. L., and Culler, D. E., "Wide Area Cluster Monitoring with Ganglia," in *Proc. of the IEEE Cluster Conference*, December 2003.

[121] Sachs, D. G., Adve, S., and Jones, D. L., "Cross-Layer Adaptive Video Coding to Reduce Energy on General-Purpose Processors," in *Proc. of the Intl. Conference on Image Processing*, September 2003.

[122] Saewong, S. and Rajkumar, R., "Practical Voltage-Scaling for Fixed-Priority RT-Systems," in *Proc. of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, May 2003.

[123] Sampemane, G., Palkin, S., and Chien, A., "Performance Monitoring on an HPVM Cluster," in *Proc. of the Intl. Conference on Parallel and Distributed Processing Techniques and Applications*, June 2000.

[124] Sasanka, R., Hughes, C. J., and Adve, S. V., "Joint Local and Global Hardware Adaptations for Energy," in *Proc. of the 10th Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[125] Schmidt, D. C., Gokhale, A., Harrison, T., Levine, D., and Cleeland, C., "TAO: A High-performance Endsystem Architecture for Real-time CORBA," *IEEE Communications Magazine*, February 1997.

[126] Schreier, L. and Davis, M., "System-Level Resource Management for Network-Based Multimedia Applications," in *Proc. of the 5th Intl. Workshop on Network and Operating System Support for Digital Audio and Video*, April 1995.

[127] Schurgers, C., Aberthorne, O., and Srivastava, M. B., "Modulation Scaling for Energy Aware Communication Systems," in *Proc. of the Intl. Symposium on Low-Power Electronics and Design*, August 2001.

[128] Schwan, K. and Bo, W., "Topologies - Distributed Objects on Multicomputers," *ACM Transactions on Computer Systems*, vol. 8, pp. 111–157, May 1990.

[129] Shankar, M., DeMiguel, M., and Liu, J. W. S., "A End-to-End QoS Management Architecture," in *Proc. of the IEEE Real-Time Applications Symposium*, June 1999.

[130] Simunic, T., Benini, L., Acquaviva, A., Glynn, P., and Micheli, G. D., "Dynamic Voltage Scaling and Power Management for Portable Systems," in *Proc. of the Design Automation Conference*, June 2001.

[131] Singh, S., Woo, M., and Raghavendra, C. S., "Power-aware Routing in Mobile Ad Hoc Networks," in *Proc. of the 4th Intl. Conference on Mobile Computing and Networking*, October 1998.

[132] Sottile, M. and Minnich, R., "Supermon: A High-Speed Cluster Monitoring System," in *Proc. of the IEEE Intl. Conference on Cluster Computing*, September 2002.

[133] Stankovic, J. A., Lu, C., Son, S. H., and Tao, G., "The Case for Feedback Control Real-Time Scheduling," in *Proc. of the 11th Euromicro Conference on Real-Time Systems*, June 1999.

[134] Steere, D., Goel, A., Gruenberg, J., McNamee, D., Pu, C., and Walpole, J., "A Feedback-Driven Proportion Allocator for Real-Rate Scheduling," in *Proc. of the Third Symposium on Operating Systems Design and Implementation*, February 1999.

[135] Sun Microsystems Inc., Network Working Group RFC 1057, *RPC: Remote Procedure Call Protocol Specification Version 2*, 1988. http://www.ietf.org/rfc/rfc1057.txt.

[136] Sundaram, V., Chandra, A., Goyal, P., and Shenoy, P., "Application Performance in the QLinux Multimedia Operating System," in *Proc. of the 8th ACM Conference on Multimedia*, November 2000.

[137] Uthayopas, P., Phaisithbenchapol, S., and Chongbarirux, K., "Building a Resources Monitoring System for SMILE Beowulf Cluster," in *Proc. of the Third Intl. Conference/Exhibition on High Performance Computing in Asia-Pacific Region*, 1998.

[138] Vetro, A. and Chen, C. W., "Rate-Reduction Transcoding Design for Wireless Video Streaming," in *Proc. of the IEEE Intl. Conference on Image Processing*, September 2002.

[139] Vetter, J. S. and Reed, D. A., "Real-time Performance Monitoring, Adaptive Control, and Interactive Steering of Computational Grids," *The International Journal of High Performance Computing Applications*, vol. 14, no. 4, pp. 357–366, 2000.

[140] von Eicken, T., Culler, D., Goldstein, S., and Schauser, K., "Active Messages: A Mechanism for Integrated Communication and Computation," in *Proc. of the 19th Intl. Symposium on Computer Architecture*, May 1992.

[141] Waddington, D. and Hutchison, D., "A General Model for QoS Adaptation," in *Proc. of the 6th Intl. Workshop on Quality of Service*, May 1998.

[142] Wallach, D. A., Hsieh, W. C., Johnson, K. L., Kaashoek, F. M., and Weihl, W. E., "Optimistic Active Messages: A Mechanism for Scheduling Communication with Computation," in *Proc. of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, July 1995.

[143] Welch, L. R., Werme, P. V., Fontenot, L. A., Masters, M. W., Shirazi, B., Ravindran, B., and Mills, D. W., "Adaptive QoS and Resource Management using A Posteriori Workload Characterizations," in *Proc. of the IEEE Real Time Technology and Applications Symposium*, June 1999.

[144] West, R. and Gloudon, J., "'QoS Safe' Kernel Extensions for Real-Time Resource Management," in *Proc. of the 14th EuroMicro Intl. Conference on Real-Time Systems*, June 2002.

[145] West, R. and Poellabauer, C., "Analysis of a Window-Constrained Scheduler for Real-Time and Best-Effort Packet Streams," in *Proc. of the 21st IEEE Real-Time Systems Symposium*, November 2000.

[146] West, R. and Schwan, K., "Dynamic Window-Constrained Scheduling for Multimedia Applications," in *Proc. of the 6th Intl. Conference on Multimedia Computing and Systems*, June 1999.

[147] West, R. and Schwan, K., "Quality Events: A Flexible Mechanism for Quality of Service Management," in *Proc. of the 7th IEEE Real-Time Technology and Applications Symposium*, May 2001.

[148] West, R., Schwan, K., and Poellabauer, C., "Scalable Scheduling Support for Loss and Delay Constrained Media Streams," in *Proc. of the 5th Real-Time Technology and Applications Symposium*, June 1999.

[149] Wolf, L. C., Griwodz, C., and Steinmetz, R., "Multimedia Communication," *Proceedings of the IEEE*, vol. 85, no. 12, pp. 1915–1933, 1997.

[150] Wollrath, A., Riggs, R., and Waldo, J., "A Distributed Object Model for the Java System," *USENIX Computing Systems*, vol. 9, November/December 1996.

[151] Xu, D., Wichadakul, D., and Nahrstedt, K., "Multimedia Service Configuration and Reservation in Heterogeneous Environments," in *Proc. of the Intl. Conference on Distributed Computing Systems*, April 2000.

[152] Yau, D. and Lam, S., "An Architecture Towards Efficient OS Support for Distributed Multimedia," in *Proc. of the IS&T/SPIE Multimedia Computing and Networking Conference*, January 1996.

[153] Ye, W., Heidemann, J., and Estrin, D., "An Energy-Efficient MAC Protocol for Wireless Sensor Networks," in *Proc. of the IEEE Infocom Conference*, June 2002.

[154] Youssef, A., Abdel-Wahab, H., and Maly, K., "A Scalable and Robust Feedback Mechanism for Adaptive Multimedia Multicast Systems," in *Proc. of 8th Intl. Conference on High Performance Networking*, September 1998.

[155] Yuan, W. and Nahrstedt, K., "A Middleware Framework Coordinating Processor/Power Resource Management for Multimedia Applications," in *Proc. of the IEEE Globecom Conference*, November 2001.

[156] Yuan, W. and Nahrstedt, K., "Process Group Management in Cross-Layer Adaptation," in *Proc. of the SPIE/ACM Multimedia Computing and Networking Conference*, January 2004.

[157] Yuan, W., Nahrstedt, K., Adve, S., Jones, D., and Kravets, R., "Design and Evaluation of a Cross-Layer Adaptation Framework for Mobile Multimedia Systems," in *Proc. of the SPIE/ACM Multimedia Computing and Networking Conference*, January 2003.

[158] Zhang, R., Lu, C., Abdelzaher, T. F., and A.Stankovic, J., "ControlWare: A Middleware Architecture for Feedback Control of Software Performance," in *Proc. of the 22nd Intl. Conference on Distributed Computing Systems*, July 2002.

VITA

Christian Poellabauer was born in Oberwart, Austria on December 6, 1972. After attending a 5-year Technical School for Electrical Engineering from 1987-1992, he studied Computer Science at the Technical University of Vienna, where he received the degree of 'Diplom-Ingenieur' in 1998. That fall, he joined the Systems Group of the College of Computing at the Georgia Institute of Technology.

In April 2004, under the supervision of Prof. Karsten Schwan, Christian completed his Ph.D. dissertation entitled "Q-Fabric: System Support for Continuous Online Quality Management". His research – which was partially supported by an IBM Ph.D. Research Fellowship – ranges from system-level work to support QoS-aware applications, to communication infrastructures for on-line quality management, to methods for energy management in mobile systems.