THESIS
2
2007

This is to certify that the
dissertation entitled

DESIGN AND EVALUATION OF ADAPTIVE SOFTWARE
FOR MOBILE COMPUTING SYSTEMS

presented by

ZHINAN ZHOU

has been accepted towards fulfillment
of the requirements for the

Ph. D. degree in Computer Science

[illegible]
Major Professor's Signature

7/27/06
Date

DESIGN AND EVALUATION OF ADAPTIVE SOFTWARE FOR MOBILE COMPUTING SYSTEMS

By

Zhinan Zhou

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Department of Computer Science and Engineering

2006

ABSTRACT

DESIGN AND EVALUATION OF ADAPTIVE SOFTWARE FOR MOBILE COMPUTING SYSTEMS

By

Zhinan Zhou

Increasingly, software must adapt to a changing environment during execution. One of the key driving forces behind the need for adaptation is the advent of the "Mobile Internet," where software on portable computing devices must adapt to several, potentially conflicting, concerns, including quality of service, security, and energy consumption. Moreover, mobile systems often comprise multiple heterogeneous applications, each of which might support different types of adaptation. Such situations motivate the need for comprehensive approaches to designing adaptive mobile systems, in which multiple software components, possibly at different system layers, collaborate to achieve overall system goals. In this dissertation, we investigate software adaptation for mobile computing.

Composing a single adaptive system from existing adaptive/non-adaptive applications requires an adaptation infrastructure to orchestrate the behavior of adaptive systems and guide the collaboration among system participating applications. We propose a new concept called *expressive orchestration*, which refers to the techniques that enable system designers to specify the system requirements, generate infrastructure for interaction among participating applications, and codify logic for the run-time management of the system. This dissertation addresses three aspects of design and evaluation of adaptive software for

mobile computing systems.

First, we evaluate the tradeoffs that exist among concerns (such as energy consumption and quality of service) in mobile devices. Understanding these tradeoffs is a precursor to designing adaptive systems. This investigation, which includes experimentation on a mobile computing testbed, has produced several results that are directly applied to other aspects of this research.

Second, we investigate the use of message-based communication to facilitate the integration and collaboration of adaptive/non-adaptive applications. As a proof of concept, we develop COCA (**CO**mposing **C**ollaborative **A**daptation), an infrastructure for collaborative adaptation in composite systems. COCA provides a set of development utilities to aid system designers in specifying system configuration and adaptation logic, as well as automatically generating the corresponding code. In addition, COCA provides a set of run-time utilities to enforce the collaborative adaptation execution. The methods used in COCA are general and can be extended to other distributed computing models that require collaborative adaptation.

Third, we propose ASSL (**A**utonomic **S**ervice **S**pecification **L**anguage), an XML-based approach to specifying and realizing adaptation in distributed service-oriented systems. Focusing on system integration, configuration, and run-time interaction management, ASSL is an extension of COCA that provides a unified platform to describe and support interactions among different parties in the development and execution of autonomic systems.

Combined, these contributions provide the research and development communities with a better understanding of the opportunities for adaptation in mobile system and the means to realize such systems from existing, non-adaptive software components.

To my dear wife, *Xu*, and my lovely son, *Vincent*.

Thank you for encouragement, support, and love!

ACKNOWLEDGMENTS[1]

My advisor and guidance committee chairperson, Dr. Philip K. McKinley, supervised this work and guided me through this research area. I would like to express my thanks to him for his invaluable advice and the unlimited time he spent to correct my mistakes. Other members of my guidance committee, Dr. Betty H.C. Cheng, Dr. Sandeep Kulkarni, and Dr. Jonathan I. Hall, were always available for all my questions. I would like to thank them for their help and contributions to this work. I am grateful to my colleagues in the Software Engineering and Network Systems Laboratory and in the Computer Science and Engineering Department of Michigan State University for the insightful discussions we had during the course of this research. Especially, I am very thankful to Dr. Seyed Masoud Sadjadi, Ji Zhang, Zhenxiao Yang, Farshad Samimi, Dr. Chiping Tang, Dr. Peng Ge, Eric Kasten, Dave Knoester, and Min Deng.

Last but not least, I would like to thank my family: my wife, Xu, who encouraged me to start this Ph.D. program; my son, Vincent, who motivated me to graduate; and my parents, parents-in-law, and other family members, who have always been there for me.

[1]This work has been supported in part by the U.S. Department of the Navy, Office of Naval Research under Grant No. N00014-01-1-0744, and in part by National Science Foundation grants CCR-9912407, EIA-0000433, EIA-0130724, and ITR-0313142.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

Increasingly, software must adapt to a changing environment during execution. One of several driving forces behind the need for adaptation is the advent of the "Mobile Internet," where software on portable computing devices must adapt to several, potentially conflicting, concerns, including quality of service (QoS), security, and energy consumption. For example, achieving an acceptable quality of service on a video stream might reduce the lifetime of a battery-powered device to an unacceptably low level.

While some types of adaptation can be realized in individual standalone applications, other situations require coordinated responses from multiple components[1] within a composite system running either on a single platform or distributed across multiple platforms. For example, a communication application transmitting a video stream between two nodes might be able to mitigate high channel loss rates by simply increasing the level of forward error correction (FEC) on a wireless channel. However, if the stream is part of a

[1]Here, we use the term *component* loosely, referring to standalone applications as well as software modules developed and deployed by third parties.

teleconferencing system, then above a certain loss rate, it may be preferable to reconfigure the entire system, for example, by switching from an audio/video configuration to an audio-only system. Such situations motivate the need for comprehensive approaches to adaptation, where an *adaptive system* comprises multiple adaptive/non-adaptive components, possibly spanning multiple system layers, that collaborate to achieve overall system goals. In recent years, several cross-layer (and collaborative) adaptation frameworks have been proposed; examples include Odyssey [8], DEOS [9], Chisel [10], and GRACE [6]. In these systems, collaboration is realized either by constructing applications within a common framework [6, 8, 9], or by transparently augmenting applications with interfaces to such a framework [10].

In a distributed system, however, the collaboration problem is further exacerbated. Interactions need to take place across a network among heterogenous platforms and applications. Moreover, many adaptive computing systems are constructed from pre-existing and otherwise independent applications running on separate nodes. This trend poses three important challenges to the design of adaptive systems. First, the individual applications might have different adaptation policies that produce competition for limited system resources and conflicts in satisfying overall system needs. Second, even if they are compatible in behavior, the applications might have been developed by different organizations, using different languages and/or different middleware platforms, and using different (and likely incompatible) approaches to adaptation. Third, system-wide adaptations require some means to specify and coordinate the collaboration among different applications.

We contend that supporting collaborative adaptation among existing adaptive/non-adaptive components should be based on a model that (1) requires little or no modification of existing applications; (2) can be easily extended to accommodate new platforms and services; and (3) leverages existing middleware services whenever possible. To explore the issues involved in realizing such a model, this dissertation proposes *expressive orchestration*, a new concept which refers to techniques that enable system designers to construct distributed adaptive systems by specifying system requirements, managing the interaction among system participating components, and codifying the necessary adaptation logic. Here, we focus on providing a framework for the design, development, and run-time management of adaptive mobile computing systems. Moreover, we also provide an adaptation infrastructure to support collaborative adaptation among components that were not necessarily designed to interoperate.

Thesis Statement. *By providing a means to specify the system composition and configuration, manage the interaction between system components, and codify the adaptation logic, expressive orchestration offers an effective solution to the design, development, and run-time management of adaptive mobile systems. Expressive orchestration can be applied to individual applications, composite systems, and fully distributed systems.*

The major contributions of this dissertation are summarized as follows.

1. We evaluate tradeoffs that exist among concerns in mobile systems, focusing primarily on energy consumption and quality of service (QoS). The results of these experiments can be used as a basis for developing adaptive software to manage these tradeoffs in the presence of highly dynamic wireless environments. As a case study, we evaluate the energy consumption of forward error correction (FEC) as used to improve QoS on wireless devices, where encoded audio streams are multicast to multi-

ple mobile computers. Our results quantify the tradeoff between improved QoS, due to FEC, and additional energy consumption, delay, and bandwidth usage caused by reception and decoding of redundant packets.

2. Using the above results, we investigate the use of message-based communication to facilitate the integration and collaboration of adaptive and non-adaptive components. As a proof of concept, we develop COCA (**CO**mposing **C**ollaborative **A**daptation), an infrastructure for collaborative adaptation among components that were not necessarily designed to interoperate in the composite systems. COCA provides a set of development utilities to aid system designers in specifying system configurations and adaptation logic, as well as automatically generating the corresponding code to realize collaborative adaptation among existing components. COCA also provides a set of run-time utilities to enforce the collaborative adaptation execution, and a Web services infrastructure to support the corresponding interaction among components. The methods used in COCA are general and can be extended to other distributed computing models that require collaborative adaptation.

3. We investigate specification techniques that can help design, development, deployment, and management of distributed service-oriented autonomic systems. We propose ASSL (**A**utonomic **S**ervice **S**pecification **L**anguage), an XML-based technique that enables specification of an autonomic distributed system, focusing on system integration, configuration, and run-time interaction management. ASSL is an extension of COCA that provides a unified platform to describe and support interactions among different parties in the development and execution of autonomic systems. We

apply ASSL in a service-oriented infrastructure, called Service Clouds, providing interactive design support and run-time adaptation management.

This research is part of an Office of Naval Research sponsored project called *RAPID-ware* [11], which addresses design of adaptive middleware to support interactive applications in dynamic, heterogeneous environments. Several results of this research have already been published or are planned for publication. First, we have completed our study of the basic adaptation characteristics and how to manage those tradeoffs in an individual adaptive system. This work is published in [12–14]. Based on these preliminary results, we have published a paper [15] about the formal specification of timing properties in adaptation. Second, two papers have been published, describing message-oriented adaptation mechanisms [16] and the COCA framework [17]. Finally, we have applied the expressive orchestration concept to a fully distributed, service-oriented mobile computing environment and are currently preparing a paper on ASSL for publication.

The remainder of this dissertation is organized as follows. Chapter 2 provides background on mobile adaptation techniques, surveys collaborative adaptation in mobile computing, and motivates the need for our research. Chapter 3 introduces a test bed for the study of adaptation characteristics, provides the experimental results for understanding tradeoffs in adaptation behaviors and logics, and demonstrates the potential of implementing dynamic adaptation through rule-based management. Chapter 4 introduces the COCA framework and describes a case study that demonstrates the use of COCA to realize an adaptive mobile multimedia conferencing system constructed from legacy components. In Chapter 5, we introduce ASSL and show how it can be used to orchestrate adaptive services atop the distributed Service Clouds infrastructure. Finally, Chapter 6 offers conclusions and discusses future research directions.

Chapter 2

BACKGROUND AND RELATED WORK

2.1 Adaptive Mobile Computing Systems

Interest in adaptive computing systems has grown dramatically in the several few years [18], driven primarily by two ongoing revolutions: ubiquitous computing, which removes traditional boundaries for how, when, and where humans and computers interact; and autonomic computing, which refers to the ability of systems to manage and protect their own resources with only high-level human guidance.

In the past decade, the number of mobile computing devices has grown dramatically, as shown in the Figure 2.1 [3]. This increase is driven primarily by the rapid growth in the use of the Internet and the wide deployment of wireless networks. The need for adaptation for mobile systems arises in part because conditions in wireless environments are highly variable, and available computing resources are strictly limited. Hence, the software on mobile computing devices must balance several, potentially conflicting, concerns, including quality of service, security, and energy consumption. In this chapter, we explore the issues involved in adaptation for mobile computing and discuss key issues in proposed approaches to adaptation.

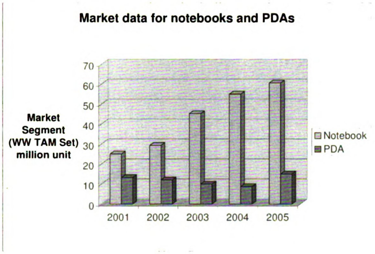


Figure 2.1: Market data for the portable computing devices by 2005 [3].

2.2 Reducing Energy Consumption

A characteristic that distinguishes mobile computing system from many other types of systems is the need to minimize energy consumption. Advances in rechargeable battery technologies have not kept pace with the development of other hardware components (for comparison, see Figure 2.2). Unlike other system resources, such as memory, energy that has already been consumed cannot be "released" and "reallocated." This property motivates both the need to increase energy efficiency (more work per unit of consumed energy) and the need to extend battery lifetime (work longer under a given load).

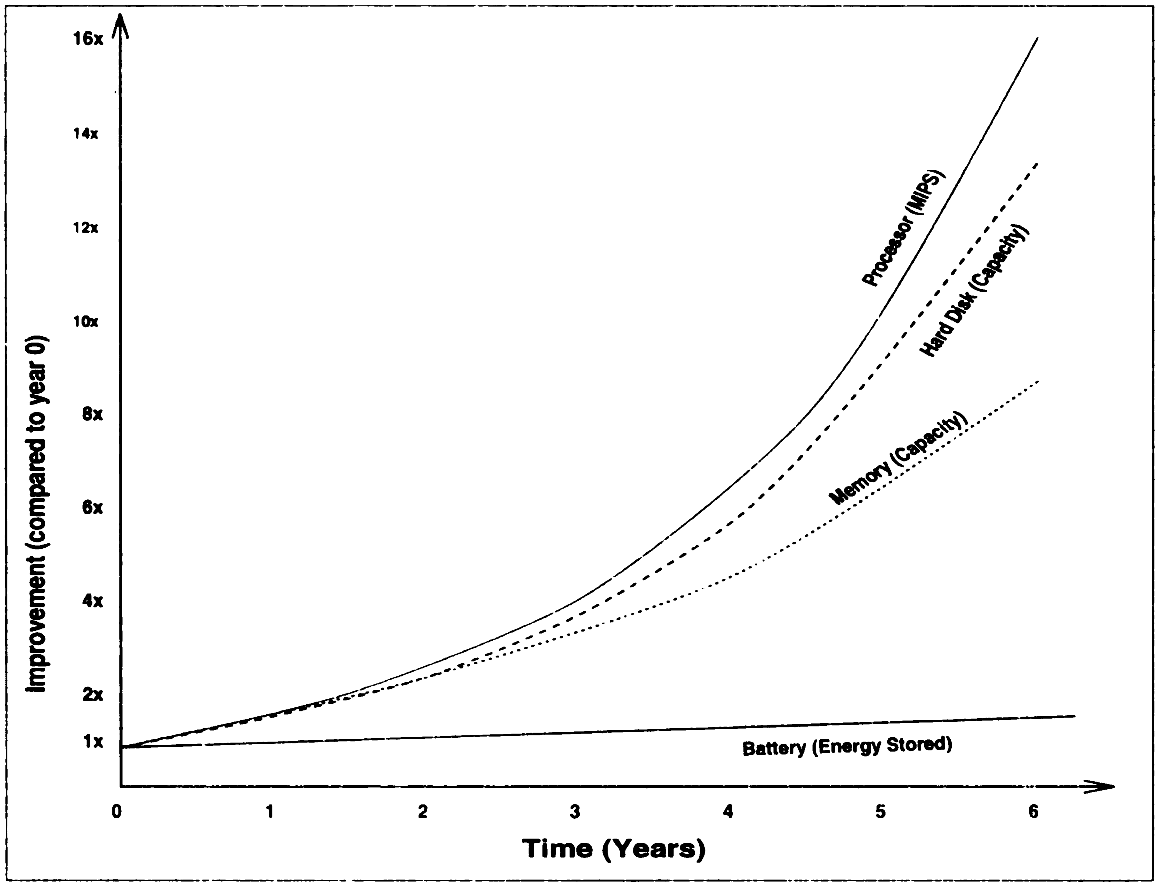


Figure 2.2: Approximate performance/capacity growth of major laptop components [4].

Many adaptive energy management strategies reduce the energy consumption of hardware components by way of the operating system. We can group the adaptive software control issues at the operating system level as shown in Table 2.1: *transition, load-change, and adaptation* [1]. Almost all subsystems (CPU, wireless interface, hard disk, display and so on) can achieve energy savings by adopting one or more of these strategies. For example, the operating system may use the *transition* strategy by slowing down the CPU speed when the computational load is low, producing a corresponding reduction in the voltage needed by the CPU. Similarly, displays usually have one or more low-power modes. On the other hand, the *load-change* strategy can be used to reduce energy consumption of a

Table 2.1: Categories of energy-related software problems on the OS level [1].

Category	Description
Transition	When should a component switch between available execution modes?
Load-change	How can the load on a component be modified so that it can be put in low-power modes more often?
Adaptation	How can software permit novel, energy-saving uses of components?

wireless interface by compressing the transmitting data; doing so can reduce the packet size and thus reduce the communication activity on the wireless client. An example of an *adaptation* strategy is to use the wireless network as the replacement for hard disk: offloading storage to a fixed workstation results in energy savings on the hard disk, which is also a major energy consumer. We emphasize that these strategies are not mutually exclusive, but can be used in combination. Furthermore, each strategy has its own advantages and disadvantages, so tradeoffs need to be considered in their selection. Next, let us examine how these strategies have been used to conserve energy via hard disk, CPU, display, and wireless network interface.

The two main strategies here are to put devices into sleep mode (e.g., hard disk and wireless network interface) and to reduce output power (e.g., CPU and display). A key to the effectiveness of both strategies is the inactivity threshold. Numerous prediction algorithms have been proposed for both fixed and adaptive thresholds. A fixed inactivity threshold method is simple to implement. If the device is inactive for the threshold time, it is assumed that there will be no activity in the near future and the device can be switched into low power mode. An adaptive inactivity threshold method, however, attempts to adjust the threshold according to the device usage pattern distribution. Tradeoffs exist among these strategies. Switching devices into low power mode always introduces delays, which can inconvenience the user and potentially harm the application, such as those with real-time requirements. Moreover, low power mode can sometimes introduce new energy consumption that cancels part of the energy savings.

2.2.1 Hard Disk

Depending on the rotation speed, buffer size and disk usage pattern, the hard disk typically consumes 15-30% [19] of the total energy in a mobile computer. Although the power/MByte ratio, which represents the energy efficiency of a hard disk, has fallen with technology advances, the absolute energy consumed by a typical hard disk has remained approximately constant. Many researchers have investigated how to achieve energy savings by spinning down the hard disk during periods of inactivity [19–24], while others [25, 26] have proposed reducing energy consumption through remote execution. We review each method below.

Table 2.2 lists five hard disk operation modes in order of decreasing energy consumption. Li et al. [27] showed that spinning down the disk after idling a few seconds can save about 90% of the energy compared to *never* spinning down. However, there exist tradeoffs in this approach. First, hard disks are mechanical devices, so frequent spin-up/spin-down may cause hardware failure (normally, a hard disk has spin-up/spin-down life of 40,000-60,000 cycles) [27]. Second, the disk will use considerable time and energy in the *startup* mode. If this consumed energy is greater than the energy saved by spinning down the motor, the overall energy consumption may increase rather than decrease. Third, spinning down the motor will introduce some delays, since the motor must return to full speed to satisfy the next disk request.

Table 2.2: Hard disk operation modes [2].

Mode	Description
Startup	the motor accelerated from rest to rated speed
Active	seeking, reading, or writing
Idle	not seeking, reading or writing, but the motor is still spinning
Standby	the motor is not spinning and the heads are parked, but the controller electronics are active
Sleep	the host interface is off except for a logic circuit to sense a reset signal

How to select an inactivity threshold, either fixed or dynamic, before which the hard disk can enter the *sleep* mode, is a key problem in hard disk mode transition. Most manufacturers suggest a fixed inactivity threshold of 3-5 minutes. But some researchers have found that a more fine-grained approach (as low as 1-10 seconds) can save more energy than a coarse-grained approach [24, 27]. Since this "fixed inactivity threshold" method is simple to implement, it is the most widely used at present, even though the corresponding energy savings is limited. If the hard disk is inactive for the threshold time, it is assumed that there will be no disk accesses in the near future and the motor is spun down until next read/write request. If the threshold is too low, the user may experience the spin-up delay too often; if the threshold is too high, the energy savings will be small since the motor remains spinning most of the time. Li et al. showed that the optimal threshold is about 6 seconds [27], which means if there is no disk activity for greater than 6 seconds, then spinning down the disk will save energy.

In most cases, the hard disk access patterns change with time, and thus a fixed inactivity threshold may be insufficient. An *adaptive* inactivity threshold attempts to adjust the threshold according to the access pattern distribution [21]: undesirable spin-up delays indicate that the threshold is too short and should be increased; if the delays are acceptable, the threshold is long enough and can possibly be decreased. It is worth pointing out that many parameters affect the performance (e.g., how to increase/decrease the threshold and by how much, limits to the maximum/minimum threshold, etc), and no single set of parameters accommodates all workloads. Therefore, to our knowledge this approach has not yet been incorporated into products.

Another way to reduce the energy consumption of a hard disk is to modify its workload. Such modification is usually effected by changing the configuration or usage of the cache above it. Li et al. [27] found that increasing cache size can produce a large reduction in energy consumption. In that study, using a 1 MB cache reduced energy consumption by 50% compared to using no cache, but further increases in cache size had a smaller effect on energy consumption, presumably because cache hit ratio increases slowly with increased cache size [28].

Finally, offloading storage through a wireless network can also be considered as an adaptation strategy in hard disk energy management. The advantage of this strategy is that the wired storage device can be large and power-hungry without affecting the weight and energy consumption of the portable device. Disadvantages include increased energy consumption by the wireless communication system, increased use of the limited wireless bandwidth, and higher latency for file system accesses. Rudenko et al. [25] proposed a model that performs all processing on a wired server. In this model, the portable device is merely a terminal that transmits and receives low-level I/O information, so that the energy for general processing and storage is consumed by the wired server instead of the mobile device. In this way, portable storage and CPU energy consumption is traded for high processing request latency, network bandwidth consumption, and additional energy consumption at the wireless network interface.

2.2.2 Processor

The energy consumed by the CPU is directly related to the CPU clock frequency and supply voltage, which can be controlled and adjusted at run time. The basic method for reducing the CPU energy consumption is to lower the supply power, which results in slowing down the CPU clock. The amount of power P used by a CMOS circuit can be given as $P \propto CV^2f$, where C is load capacitance, V is the power-supply voltage, and f is the clock frequency. The time t for the CPU to finish a task is inversely proportional to the clock frequency as $t \propto 1/f$. Because the total energy E consumed for the CPU to complete a task is $E = P \times t$, it indicates that the total energy E is proportional to the square of the voltage V: $E \propto CV^2$. Hence even a small decrease in CPU supply voltage can produce a large decrease in the total energy consumed by the system.

Most researchers investigating this problem [29–33] focus on how to schedule CPU usage to achieve the maximal energy savings by reducing CPU idle time or trading energy savings for acceptable performance. A primary strategy to adjust the CPU speed is to "stretch" activities from busy periods into subsequent adjacent idle periods, thereby balancing CPU usage between periodic bursts of high CPU utilization and the remaining periods of idle time [29]. For example, when the CPU is running at the full speed, it may take 0.001 second to respond a user's command followed by an idle period. However, if the CPU is running at one-tenth speed, the same task can be completed in 0.01 second without inconveniencing the user while the corresponding energy consumption will decrease. One approach to examine the CPU utilization is to divide CPU time into fine-grained windows (e.g., 50 ms), and at the beginning of each window, examine the CPU utilization of previous windows. Under the assumption that the CPU utilization of adjacent windows will be similar, if the utilization is high, the CPU speed will be increased; if the utilization is low, the CPU speed will be decreased. The performance of this method is highly dependent on the design of the *prediction algorithm*, which predicts the near term CPU utilization. Govil et al. [30] proposed several prediction algorithms according to different utilization patterns. For example, one prediction algorithm looks up the last twelve utilization values. The three most recent values constitute the short-term past while the remaining nine values constitute the long-term past. The prediction for the coming utilization is then a weighed sum of these twelve values.

The lowest CPU energy consumption comes with the lowest possible CPU speed, however, slowing down the CPU speed achieves energy savings at the expense of performance. Furthermore, reducing CPU performance may cause an increase in the energy consumption of other components, since they may need to remain active longer. For example, reducing CPU speed may result in the slowdown of the processing of incoming/outgoing packets in a wireless mobile device, which may increase the active time of WNIC and consume more energy. Another problem with switching the processor speed is that the system will experience more frequent changes in temperature [34], which may increase the stress at the chip interface and reduce the CPU reliability.

2.2.3 Display

The display subsystem is the largest energy consumer in a stand-alone mobile computer [4], and approximately 80% of the energy consumed by the display subsystem is for the backlight [35]. Because the energy consumed by the backlight is roughly proportional to the luminance delivered, one general strategy is to reduce the backlight brightness level or turn off the backlight entirely when it is not needed [2]. Switching from color to monochrome or reducing the update frequency can also reduce the energy consumption of display subsystem [2]. A simple approach is to turn down or off the backlight and display after an idle period without any user input. The rationale is that since if the user has not performed any input recently, he or she may not be looking at the screen any longer [1]. A variation on this strategy is not to turn off the backlight immediately, but rather to dim it progressively. If the user is indeed looking at the screen, he or she has the option to restore the backlight back by prompting the system.

In addition, different display patterns have different loads on the display subsystem. For example, most LCDs are naturally white, which means that the display pixels are white when they are unselected and black when they are selected, so lighter color consumes less energy [36]. Furthermore, lighter color looks brighter, and thus encourages the user to use dimmer backlight. According to these characteristics, software may be designed to dynamically increase or decrease the display brightness to satisfy the user's activity requirements. Iyer et al. found that darkening the unused windows and simplifying the display contents can reduce the system energy consumption while not affecting the user's normal activities [37].

Table 2.3: Wireless communication devices operation modes.

Mode	Description
Transmit	transmitting data
Receive	receiving data
Idle	is neither sending or receiving but scanning for a valid signal, which is like the *receive mode*
Sleep	the transceiver circuitry is powered down except for some small timing parts which allows for a fast bring-up
Off	completely switched off

2.2.4 Wireless Network Interface

Lastly, let us consider the wireless network interface card (WNIC), which makes mobility possible, but is also a major consumer of energy. Wireless communication devices typically have five operation modes, as listed in Table 2.3. The main difference between *idle* and *off* mode is the presence of the WNIC. In *idle* mode, the WNIC continually listens to the network and exchanges control messages (e.g., beacon messages) with the access point or other mobile hosts. Furthermore, in idle mode, the system has to process incoming traffic and maintain the data exchange between the network interface and the operating system. Hence, the difference in energy consumption between idle mode and off mode can be considered as the energy needed by the system to maintain network connection.

Transition strategies for wireless communication devices entering sleep mode are similar to those for hard disks, so like solutions can be applied. However, two features of WNICS suggest different approaches to determining the inactivity threshold selection. First, the energy needed to put the WNIC into sleep mode and to reawaken it is very small. Second, it is necessary for the WNIC to exit sleep mode periodically in order to maintain its connection with the access point or other peer hosts [1]. One proposed strategy is to monitor the host's network activities (e.g., HTTP, SMTP, FTP, etc), and if there is no significant network traffic during the threshold period, it implies that the user may be in the *think time* (e.g., browsing the contents of the web page) and the WNIC can be put into sleep mode until the user sends a connection request (e.g., a HTTP request) again [38].

An alternative approach to reducing energy consumption is to reduce the load (e.g., packet number, packet size, or both) on wireless interface. Xu et al. [39] investigated the tradeoff between (1) compressing transmitting data, which can reduce the packet size and thus reducing the transmission time, and (2) the corresponding computation workload, which increases CPU energy consumption. Another strategy is to reduce or stop the data transmission when the wireless channel is temporarily poor, i.e. the packet loss rate is high [40], so as to reduce packet retransmissions.

Communication and computation are two main sources of energy consumption in wireless networks. Besides the energy saving mechanisms provided by the IEEE 802.11 protocol, error control schemes and compression techniques can reduce energy consumption by avoiding unnecessary processing and reducing the amount of data traffic. However, it is important to consider tradeoffs. First, compression can reduce traffic, but compressor selection would increase energy consumption instead of saving energy due to the extended idle time during decompression. Second, switching WNIC into sleep mode can conserve energy but also introduces delays that reduce quality of service. Third, forward error control can reduce retransmissions, but the computational load of encoding/decoding redundant packets is not negligible. These issues are addressed in Chapter 3.

2.3 Collaborative Adaptation

As we have seen, different parts of a mobile system can be adapted individually in order to conserve energy. However, these techniques might be in conflict with other system goals, for example, maximizing quality of service. Numerous frameworks have been proposed to address the need for a coordinated, system-wide approach to software adaptation [5–10, 29, 41–54]. Supporting adaptation usually involves intercepting and redirecting interactions among software entities: encapsulating these actions within a particular system layer provides transparency to higher and lower layers. Several projects address adaptation at the operating system level [55, 56]. Many others place adaptive behavior in middleware which, in addition to its traditional role in hiding resource distribution and platform heterogeneity, can be used to address concerns such as quality of service, energy management, fault tolerance, and security policy [45, 57–65]. Finally, several projects focus on dynamic recomposition within the application itself, either directly by using a language that supports recomposition [66, 67] or indirectly by modifying code as it is loaded by a virtual machine [68], or dynamically weaving new behavior into running programs [45,69–72]. Here, we review several projects targeted primarily or exclusively at mobile systems [6, 8, 9, 73].

Odyssey. Because mobile hosts are resource-poor relative to static hosts and rely on a finite energy source, it is suggested that the client-server architecture is desirable [47]. In this kind of architecture, servers are the home of data and clients retrieve data from servers [48]. *Odyssey* [8], developed at Carnegie Mellon University, is a relatively early cross-layer framework, supporting interaction between the operating system and applications to meet user-specified goals for battery duration. In Odyssey, the role of the operating system is to sense the external environment (such as network connectivity and physical local changes), and monitor and allocate resource (such as network bandwidth, disk space, battery power, etc); in contrast, the role of individual applications is adapting to the changing environment with the information and resource provided by the operating system. In this way, a well-defined collaborative partnership between the operating system and individual applications is established [48].

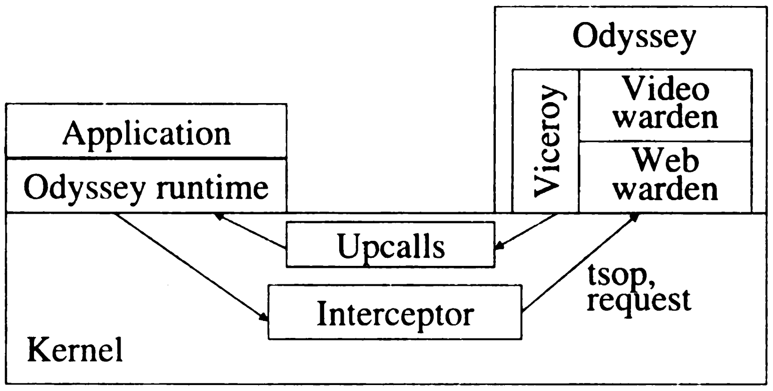


Figure 2.3: Odyssey architecture [5].

The architecture of Odyssey is shown as Figure 2.3. Although it is implemented in user space, it could be thought of as part of the operating system and implemented directly in the kernel or as a middleware [8]. The adaption in Odyssey is trading *fidelity*, which is defined as the degree to which a presented item matches the reference copy, for performance. When an application chooses a fidelity, it issues a *resource request*, which is forwarded by the *interceptor* to the *viceroy*. The viceroy is responsible for monitoring the availability of system resources. Once the viceroy receives a resource request, it compares the resource's current availability with any established window of tolerance (since the adaptation only takes place when certain range of changes happens). If a resource is out of the window bounds, each affected application is notified via an *upcall*, and then the application responds to the notification by changing the fidelity of the data. These changes are done through *wardens*, which are responsible for all operations on data items of their type and communications between the clients and servers.

Based on the assumption that lowering data fidelity yield significant energy savings, Flinn et al. [49] used Odyssey to trade data quality for energy savings. With the help of PowerScope [50], an energy profiling tool that maps energy consumption to specific software components, Odyssey can calculate residual energy. When predicted demand exceeds residual energy, Odyssey issues upcalls so that applications can adapt themselves to reduce energy usage by decreasing the data fidelity. For example, a media player application can request the low-quality black-white media data instead of the high-quality colorful copy to conserve energy. When multiple applications are requesting the same resource concurrently, Odyssey allocates the resource according to user-defined priorities, i.e. always tries to degrade a lower-priority application before degrading a higher-priority one. Flynn citereduce.energy.office extended this concept in the design of Puppeteer, a proxy-based system that dynamically adjusts fidelity of documents delivered to mobile systems.

GRACE. Most existing energy-aware adaptation techniques utilize the OS to facilitate application adaptation or focus on adapting in a single layer (network layer or application layer) at a time. The goal of the Global Resource Adaptation through CoopEration (GRACE) [6] project at the University of Illinois at Urbana-Champaign is to develop an integrated cross-layer adaptive system to maximize user satisfaction within the constraints of energy, time, and bandwidth. To achieve this system design goal, hardware and all software layers cooperatively adapt to the changing system resources and application demands. As shown in Figure 2.4, all parts of the existing system cooperatively adapt as a community and achieve a globally optimal utilization of resources.

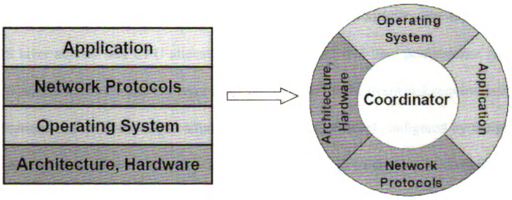


Figure 2.4: The GRACE approach [6].

GRACE introduces the concepts of combination of global and local adaptation. The target of global adaptation is large and long-term changes while the local adaptation reacts to small and temporary variations. So the global adaptation is a negotiation among different applications for resources. Once the resources have been allocated fairly to all applications, different system layers can adapt locally as long as they do not exceed the provided reservations.

Figure 2.5 shows the architecture of GRACE-2 framework, the latest GRACE prototype. GRACE-2 supports application QoS under CPU and energy constraints via coordinated adaptation in cross layers (hardware, OS and application). Specifically, the *global controller* resides in the OS layer and has full access to the system states (e.g., task resource demands, energy availability, etc). According to task utilities, CPU demands observed by the CPU *monitor*, and energy availability observed by the battery *monitor*, the global controller mediates task QoS levels, CPU processing allocations, and CPU frequency to meet the QoS and energy requirement. The global controller interacts with different adaptors which reside in different layers to adjusts their tasks which achieves in cross-layer adaptation. For example, the CPU *adaptor* in hardware layer dynamically adjusts the CPU frequency to save energy using dynamic voltage scaling (DVS) [29]. The OS *scheduler* in OS layer adjusts task CPU allocation to deliver a soft real-time performance guarantee. When the hardware and OS layer adaptation cannot meet current task requirement, the application *adaptor* is evoked to adjust its task to the QoS level configured by the global controller at the application layer. GRACE has been used to develop ReCalendar [52], which allow users to arrange application activities and request energy reservation via CPU frequency/voltage adaptation and soft real-time scheduling.

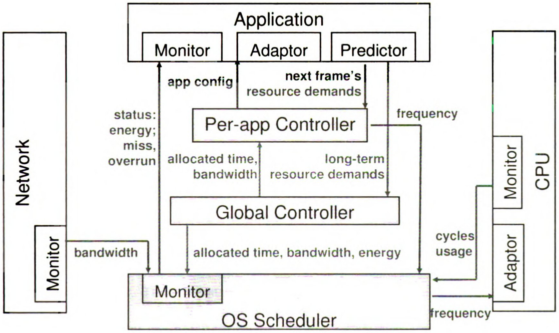


Figure 2.5: Overview of the GRACE-2 cross-layer adaptation architecture [7].

PADS. The goal of the Power Aware Distributed Systems (PADS) project [73] is to provide a framework for assessing of power-aware design strategies in sensor network environments. The project also investigates strategies for intra-node power-aware management and network-wide power-aware management that realize the tradeoffs between quality and energy [73]. One key area of study in PADS is power-aware resource scheduling in real-time operating system (RTOS), which yields an adaptive tradeoff between energy consumption and system fidelity/quality [74]. The basic approach is to exploit slack time in the use of a device by shutting it down, or operating it at a lower-power or lower-speed setting. In many systems, even if all task instances run for their worst case execution time (WCET), the CPU utilization is often far lower than 100% and thus generates idle intervals (slack). This slack time can be exploited to reduce energy consumption by slowing down the CPU and operating at a lower voltage, extending the task execution time to its WCET.

2.4 Specifying Adaptation

From aforementioned example adaptive systems, we know that adaptive computing is a promising but also challenging computing model, and it is extremely difficult to build adaptive systems from scratch. In order to simplify adaptive system design and management, the implementation of adaptive functionalities often relies on the collaboration among individual components. The relationship among components must be based on agreement, in which a component can precisely specify its service to other components and the interactions with other components. To validate the agreement, a component must not only understand and abide the terms of its agreement, but also be capable of negotiating to establish agreements. With the help of these expressive and functional agreements, it might be possible to change the system administration from passive monitoring and human based intervention to active management that requires only high-level human guidance.

In the past several years, numerous approaches have been proposed to specify software composition and govern software adaptation. Examples include QoS specification and contract [75–79], adaptive QoS control and management [80–86], software architecture approaches to adaptation [87–90], and policy-oriented adaptation [91–94]. In this dissertation, the concept of expressive orchestration is most closely related to three classes of projects: those that use architecture description languages (ADLs) to describe how an adaptive system is composted [53, 54, 87–90], those that use a policy-oriented approach to guide the adaptation process during execution [10, 60, 79, 91–99], and those that use the concept of contract [100, 101] to specify and manage the collaborative relationship among components and guide their interactive behaviors.

2.4.1 ADL

Architectural Description Languages (ADLs) are notations for expressing and representing architectural designs and styles. They describe the high level structure of a system in terms of components and component interactions. Using an ADL, a system developer can specify the system functional composition through component selection, and attach to it particular module interaction contracts. ADLs are useful in enabling component reuse and product line development, formalizing component relationships and tailoring related components to specific application domain. Wright [102] is an ADL that focuses on formally specifying protocols of interaction among components in an architecture. Darwin [103] is intended to be a general purpose notation for specifying the structure of distributed systems composed from diverse components using different interaction mechanisms. It divides the description of structure between computation and interaction in order to provide a clear separation of concerns. Darwin allows distributed programs to be specified as a hierarchic construction of components, and components interact by assessing services. Each inter-component interaction is represented by a binding between a required service and provided services. The ADLs are now adapted to XML. The XML-based ADL, xADL 2.0 [104], clearly defines a structural instance schema, describing the topological organization of components.

2.4.2 Policy

The concept of "policy" is being widely used in enterprizes for defining strategies for quality of service management, storage backup, system configuration as well as security authorization and management [105]. Policies and QoS specifications are often specified at design time and enforced at run time. However, most of them also provide run-time modification mechanism which brings dynamic specification support.

QML [95] is a quality of service modeling language designed by HP laboratories, and it can be used to construct QoS-based quantitative specifications, allowing users to specify non-functional aspects of services separate from the interface definition. QML is a general-purpose QoS specification language capable of dealing with any QoS aspects (e.g., reliability, availability, performance, security, and timing) and any application domains. QML allows detailed descriptions of the QoS associated with operations, attributes, and operation parameters of interfaces. This level of detail is essential to clearly specify and divide the responsibilities among service clients and service implementations.

2.4.3 Contract

Contract-based techniques have been widely used in the research of software engineering, programming language, and distributed systems. The concept of "contract" can be considered as the extension and combination of the concept of ADL and policy. Beugnard et al. defined a general model of software contracts [100]. According to their definition, there are four classes of contracts in the software component world according to increasingly negotiable properties: basic or syntactic, behavioral, synchronization, and quantitative. Basic contracts are normally implemented in Interface Definition Languages (IDLs) and O-O languages, specifying the input/output parameters, operations, and possible exceptions of a component. Behavioral contracts specify precisely the effect of operation executions, and behavioral contracts are designed to restrict the conditions of operations and express the outcomes of executions. Synchronization contracts show concrete and specific ways in which the component serves its clients. Specifically, it is important for the system developer to describe the relations among component elements and how they interact with each other. Quantitative contracts quantify the expected behaviors of a component and provide the means to negotiate the offered services. Quantitative contracts also encapsulate the customer expectation of quality of service to the service provider. In the rest of this section, we briefly review examples about each class of contract.

An Interface Definition Language (IDL) is a formal language used to define object interfaces independent of the programming language used to implement the those methods. Many software vendors use IDL to enable distributed computing architectures, for example, OMG IDL and Microsoft WIDL. An intuitive property of IDL is that the interface definition is independent of hardware, operating system, and programming language. The interface to a class of objects contains the information that a caller must know to use an object, specifically, the names of its attributes and the signatures of its methods. An IDL does not contain any mechanism for specifying computational details.

Behavioral contracts are designed to restrict the operation conditions and express the execution outcomes. *Design by contract* is the collaboration-level specification and design approach [101], and it is supported in the Eiffel language [106]. It views each interaction between two objects as a legal contract between a service client and a service provider. Each such contract documents the respective obligations and benefits of each party, and the obligations of one party result in benefits for the other party. An operation's behavior is specified by boolean assertions, called pre- and post-conditions. Each contract specifies the following important aspects of behavioral compositions. Firstly, the contract identifies the participants and their contractual obligations. Contractual obligations includes type obligations (supporting certain variables and external interfaces) and causal obligations (performing an ordered sequence of actions by requests and making certain conditions true). Secondly, the contract defines invariants that participants cooperate to maintain. Lastly, the contract specifies pre-conditions on participants to establish the contract and the methods which instantiate the contract. Besides these building blocks, contracts also provide constructs for the refinement and inclusion of behavior defined in other contracts.

Coordination contracts [107] are modeling primitives that facilitate the evolution of software systems by encapsulating the coordination aspects, i.e., the way components interact. A coordination contract fulfils a role similar to that of a connector in ADLs, and it consists of a prescription of coordination effects that will be superposed on a collection of partners. The use of coordination contracts encourages the separation of computation from coordination aspects.

One of the most extensive examples of quantitative contracts is Quality Objects (QuO) [60] developed by BBN, which provides an adaptable framework to support QoS in CORBA applications. QuO use Aspect-Oriented approach to weaves QoS aspects, referred to as *qoskets*, into the applications at compile time by wrapping stubs and skeletons with specialized delegates, which intercept requests and replies for possible modifications. QuO extends the CORBA functional IDLs with a QoS Description Language (QDL) consisting of three sub-languages, the Contract Description Language (CDL), the Structure Description Language (SDL), and the Resource Description Language (RDL). CDL is used for specifying a QoS contract, which consists of four major components: a set of *nested regions*, each representing a possible state of QoS; *transitions* for each level of regions, specifying behavior to trigger when the active region changes; *system condition objects*, gathering run-time information for measuring and controlling QoS; and *callbacks*, notifying the client or object. While CDL is used for describing the QoS contract between a client and an object, SDL allows programmers to specify the structural aspects of the QoS application, including adaptation alternatives and strategies based on the QoS measured in the system.

2.5 Toward Expressive Orchestration

The concept of expressive orchestration is intended to provide a comprehensive development toolkit and infrastructure to specify the system requirement, facilitate the system integration and configuration process, and manage the run-time collaborative adaptation. Ultimately, the high-level expressive specification (compositional architecture, possible states of operation, and actions recommended with respect to the state transitions) can be used to orchestrate the behaviors of possibly incompatible and potentially conflicting components. In the next chapter, however, we focus on a preliminary step, understanding the basic adaptation behaviors and logics.

Chapter 3

EMPIRICAL ASSESSMENT

3.1 Introduction

While wireless communication brings mobility to the user, the network subsystem is also one of the largest consumers of energy in a mobile device. This problem is exacerbated in noisy environments, where error control strategies generate additional network traffic. Traditional error control methods are based on retransmissions of lost packets, while others involve forward error correction (FEC) [108]. FEC introduces redundancy in the data stream in the form of parity packets, enabling recovery of lost packets at the receiver without retransmissions. FEC is particularly well-suited for use with interactive, real-time communication streams, where waiting for retransmissions introduces unacceptable delay and jitter. However, transmitting and receiving parity packets consumes additional energy.

In this chapter, we investigate the relationship between quality of service (QoS) and energy consumption characteristics when FEC is used in communication with wireless devices. The work is experimental and focuses on FEC support for interactive audio multicasting to handheld computers and laptops in wireless local area networks (WLANs). In this study, we focus on WLANs that extend wired LANs, that is, they are used in infrastructure mode. One dimension of our ongoing work addresses energy management and QoS in mobile ad hoc networks. Two FEC protocols are investigated, one using block erasure codes and the other using the GSM 06.10 encoding algorithm for cellular telephones [109].

The main contributions of this chapter are threefold. First, the study helps to quantify the tradeoff between improved packet delivery rate, due to FEC, and additional energy consumption caused by receiving and decoding redundant packets. Second, we assess the effectiveness of periodically putting the wireless network interface card (WNIC) into sleep mode to save energy while satisfying QoS requirements. Third, we demonstrate how these results can be used as a basis for the development of adaptive software mechanisms that "manage" the energy consumption in the presence of highly dynamic environments.

The remainder of this chapter is organized as follows. Section 3.2 provides the background and related work. In Sections 3.3 and 3.4, respectively, we describe the experimental environment and software configuration used in this study. Section 3.5 describes experiments to evaluate energy consumption characteristics under different FEC configurations. In Section 3.6, we assess the quality of audio communication using various FEC protocols and parameters. Section 3.7 shows how an adaptive software framework can respond dynamically to changes in the environment. Conclusions are given in Section 3.8.

3.2 Related Work

Before describing our experimental study, let us first review other research aimed at reducing the energy consumption associated with the communication subsystem. We focus on three general issues: (1) use of a power-save mode, (2) the energy consumption characteristics of error control protocols, and (3) energy-aware adaptation.

3.2.1 Power Saving Modes

The sources of energy consumption in wireless communication can be classified into two categories: *communication related* and *computation related* [110]. Correspondingly, two basic principles to achieve energy savings are (1) avoiding unnecessary network activities, and (2) reducing the amount of data traffic. Researchers have investigated the main cause of unnecessary energy consumption and the corresponding energy saving mechanisms in a wireless [110]. For example, Packet collisions produces retransmissions in reliable protocols (e.g., TCP/IP), and retransmissions lead to unnecessary energy consumption and possibly unbounded delays. Hence, reducing collisions can reduce energy consumption [111]. Switching from transmit to receive mode and vice versa also consumes additional energy [112], so if possible, a mobile device should be allocated contiguous slots for transmission or reception so as minimize this effect. Moreover, poor channel conditions generate high error rates, and the energy used to process and transmit that will later be lost, is wasted. Hence, avoiding transmission while the channel quality is poor, or adopting effective error control schemes, can save energy. Finally, significant energy is consumed at a mobile host when it either transmits a packet or when it receives a packet, and a transmission from one host to another is potentially overheard by all the neighbors of the transmitting host. So all these overhearing nodes consume energy even though the transmission is not directed to them [113, 114]. Periodically putting the mobile hosts into sleep modes can avoid overhearing problems.

Researchers at the Technical University of Berlin [112] further investigated the energy consumption of an IEEE 802.11 WLAN interface card under different working modes (idle, sleep, receive, transmit) and wireless network conditions. They found that the energy consumed by the WNIC is significantly affected by the data rate, transmission power and packet size. In particular, the energy consumed per bit of data successfully transmitted over the medium decreases as the packet size and data rate increased.

The power management mechanism is one of the most complicated parts of wireless protocols such as the IEEE 802.11 standard [115]. The primary power saving mechanism in the IEEE 802.11 protocol is to switch a mobile station into Power Save (PS) mode, which enables mobile stations, in either an infrastructure network or in an ad-hoc network, to save energy by periodically turning off the WNIC transmitter and receiver. All stations (STAs) in PS mode are synchronized to wake up at the same time. At this time, the sender announces whether there are buffered frames, a.k.a MSDU (MAC service data unit), for the receiver (when the receiver is in the sleep mode, the sender buffers all frames destined to the sleeping receiver). A station that receives such an announcement will remain awake until the buffered frames are delivered. It is easy for infrastructure networks to implement this mechanism, since the Access Point (AP) is able to buffer packets and synchronize all mobile stations. However, in ad-hoc networks, the situation is more complicated because of the absence of a trusted synchronization authority.

In addition, periodic sleep has been proposed in the design of energy-efficient MAC protocols for wireless networks. For example, PAMAS [113] puts a host into sleep mode during transmissions of other hosts and schedules the wake-up process with the help of an extra so-called *wake-up radio*, which operates on a different frequency than the radio used for communication and consumes much less energy. Inspired by PAMAS, S-MAC [114] uses single-frequency signaling and divides the time into fairly large frames. Each frame has two parts: a sleep part in which a node turns off its radio, and an active part in which a host can communicate with other nodes and send out messages buffered during the sleep part. Because all messages are sent out at a burst, instead of being "spread out" over the whole frame, energy wasted on idle listening is reduced. Different from the fixed active/sleep duty cycle in S-MAC, T-MAC [116] introduces an adaptive duty cycle in which the active part is ended dynamically. This modification not only further reduces the energy wasted on idle listening, but also outperforms S-MAC in the scenario with variable load. Unlike contention-based protocols, TDMA protocols have a natural advantage of energy conservation [117], because the duty cycle of the radio is reduced and there is no contention-introduced overhead and collisions. ER-MAC [118] is a TDMA-based protocol, but also uses the periodic listen and sleep mechanism introduced in S-MAC in a way of using *energy-criticality* to determine the host duty cycle. In our experiments, we investigate the feasibility and effect of periodic sleep during real-time audio streaming.

3.2.2 Energy Consumption vs. Error Control

As discussed earlier, retransmissions in a wireless channel always lead to unnecessary energy consumption and possibly unbounded delay, so a possible way to reduce the energy consumption is to reduce the retransmissions. Three main approaches to error control have been used in wireless packet networks [119, 120]: retransmission based ARQ (Automatic Repeat reQuest) [121–123], pure FEC (Forward Error Correction) [124, 125] and hybrid FEC-ARQ [126, 127].

Error control schemes are effective for loss recovery, however, the overall quality of service is determined by the combination of packet loss, delay and perceptual quality. Recent works show that performance gains can be expected by coupling of the delay-oriented adaptation and the error control schemes. Rosenberg et al. [128] investigated the problem of the delay introduced by FEC. They pointed out that waiting for all the redundant information is inappropriate when network loss rate is low and proposed a number of new algorithms to implement packet buffers and absorb delays observed by users. On the other hand, Dempsey et al. [123] proposed S-ARQ, which performs timely retransmission of lost packets by controlling the playback time for the first packet in each "talkspurt."

Lettieri et al. [129] used theoretical analysis and simulation to compare how different error control strategies (FEC, ARQ, and hybrids) affect energy consumption in wireless networks. The comparison is based on the mean power consumed versus the actual computational load and the delay introduced for different methods. The FEC cost is independent of the channel condition and "pre-paid," but in return, FEC can reduce the probability of retransmission. ARQ has good performance when the channel is clear, but as the loss rate increases, ARQ retransmissions adversely affect energy consumption. From the results of their study, the authors argue that the system should be able to select an energy-efficient error control strategy according to QoS requirements, channel quality and packet size. It is likely that no single method can fit all the environment requirements, so a combination of different schemes may be needed. Havinga [130] conducted an extensive experimental study of both ARQ and block-based FEC in a WaveLAN network. Havinga found that receiving of parity packets by the WNIC is a major consumer of energy, relative to the encoding/decoding work of the processor. Our results confirm this observation and quantify the tradeoff between energy consumption and QoS for FEC-based error control.

3.2.3 Energy-Aware Adaptation for Mobile Systems

The need for adaptive energy management extends beyond communication protocols. An energy-aware system should respond effectively and dynamically to the changing conditions. Specifically, these decisions involve the state of various hardware components, the operating system, and the currently running applications. To achieve this level of adaptation, a collaborative relationship between different parts of the system (e.g., operating system, middleware, and application) should be established. Thus, the application should be able to gather real-time information about system resources and the environment, select proper tradeoffs between energy consumption and other system requirements, and finally modify the subsystem behavior dynamically to conserve energy. Several recent projects have addressed adaptive energy management.

Generally, most energy-aware adaptations are cross-layer adaptation, which means different layers of a system (application, middleware, operating system, and hardware) coordinate and cooperate each other to achieve system wide energy efficiency. According to where the energy-aware adaptation behavior takes place, we can categorize energy-aware adaptations into hardware and operating system layer-specific adaptation, application layer-specific adaptation, and multiple-layer adaptation. In hardware and operation system adaptation [9, 73, 74, 131–136], adaptation actions generally change the hardware runtime parameters, such as hard disk rotation, CPU speed, and display backlight, etc., by way of interaction between operating system and hardware components. In application layer adaptation [8, 47–50, 137–139], the application itself changes its behavior or processing data without affecting the system configuration and runtime parameters. For example, an mobile application can offload its computation tasks to a wired host and retrieve the results after the processing complete to save energy. For the multiple-layer adaptation [6, 29, 52], if any single layer adaptation cannot satisfy the energy requirement, adaptations in other layers may be invoked to further reduce energy consumption and help to achieve energy saving goal.

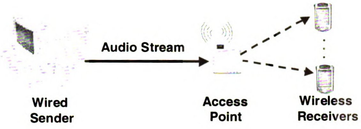
3.3 Experimental Environment

This study was conducted on a mobile computing testbed that includes various types of devices: laptop computers, iPAQ handheld systems, and Xybernaut Mobile Assistant V wearable computers. These systems communicate via an 11Mbps 802.11b WLAN. The local wireless cell is also connected to a multi-cell WLAN that covers many areas of the Michigan State University Engineering Building and its courtyard. To monitor the wireless traffic and help interpret experimental performance results, we execute the WildPackets Airopeek network analyzer on a laptop in the wireless cell.

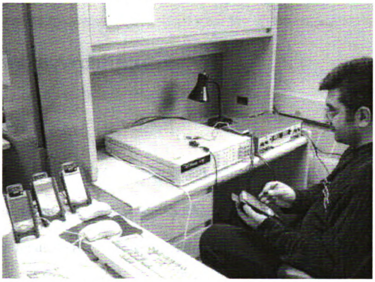
The interconnection of the systems is depicted in Figure 3.1(a). A live audio stream is multicast from a wired desktop computer to multiple mobile devices via the WLAN. Effectively, the receivers are used as multicast-capable Internet "phones" participating a conferencing application. Most experiments in this research used iPAQs as receivers. Each iPAQ is a model H3650 or H3870, with a 206 MHz StrongARM processor and 64 MB memory. Each is configured with the Familiar Linux distribution and Blackdown Java [140], and each system has a dual-slot expansion pack to support a PCMCIA wireless card (Cisco Aironet 350 Series) and an IBM 1.0 GB Microdrive. In some experiments we used laptop computers, each with a 2.0 GHz P4 processor and 1.0 GB memory, running RedHat 9.0 Linux.

A key aspect of the experimental environment involves measurement of energy consumption. Such mechanisms are specific to the particular battery configuration on a given system. For example, the iPAQ main unit and the expansion pack have separate batteries that operate independently, unless the voltage value of the main unit battery becomes lower than that of the external battery. In this situation, the main unit battery will draw power from external battery through an activated internal trickle charge until the voltage value exceeds that of the external battery. However, the external battery will never draw power from the main unit [141].

We measure energy consumption using both a hardware method, which is more accurate, as well as a software method, which is the only option in a deployed mobile system that needs to adapt its behavior based on the current state. For the former case, we remove



(a) physical experimental configuration



(b) multimeter and power supply connected to iPAQ

Figure 3.1: Testbed configuration.

the system batteries and use a power supply (Elenco Model XP-760) to power the system. We use an Agilent 3458A multimeter to measure the current drawn from the power supply. Figure 3.1(b) shows a photograph of the lab environment; the multimeter in the center, and the power supply on the right, are connected to the iPAQ held by the user. Because the iPAQ main unit and the expansion pack can share power, this configuration supplies power to both the iPAQ main unit and the expansion pack. For software measurements in Linux, we record the drop in battery voltage or capacity provided by the APM (Advanced Power Management) through the /proc file system. Specifically, a program reads from /proc/apm five times per minute, and uses the mean of these samples to represent the voltage or capacity drop in one minute. As noted, this measurement includes only the main unit battery, which can draw power from the expansion pack battery. As we shall see later, however, the expansion pack battery drain much faster than the main battery under communication-intensive scenarios.

3.4 Software Architecture

3.4.1 MetaSockets

Our experiments make use of *MetaSockets* [142], which are adaptable communication components that we developed earlier. MetaSockets (short for *metamorphic sockets*) can be used in place of regular Java sockets, providing the same imperative functionality, including methods for sending and receiving data. However, their internal structure and behavior can be adapted at run time in response to changes in their environment.

MetaSockets are implemented in Adaptive Java [143], an extension to Java that supports run-time modifications to components using computational reflection. Although using a Java-based language (Adaptive Java is source-to-source compiled into Java) introduces some processing overhead, its support for dynamic loading of code is very useful to our investigation of adaptive software. Moreover, even our modest 206 MHz iPAQs can support real-time audio streaming in Java. Figure 3.2 illustrates the internal architecture of the particular type of MetaSocket used in this study. Packets are passed through a pipeline of Filter components, each of which processes the packets. Example filter services include:

auditing traffic and usage patterns, transcoding data streams into lower-bandwidth versions, encrypting and decrypting data, and implementing forward error correction (FEC) to make data streams more resilient to packet loss. Figure 3.2 shows that the MetaSocket also supports special types of methods to insert and remove filters, as well as retrieve their status. Details of MetaSocket architecture and operation can be found in [142].

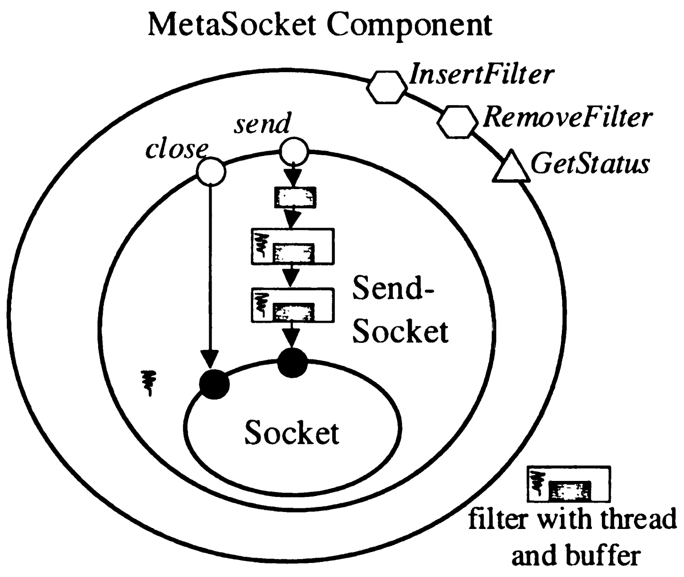


Figure 3.2: Structure of a MetaSocket.

3.4.2 Block-Oriented FEC Encoder/Decoder

In this study, we first evaluate the energy consumption characteristics of a particular FEC method based on (n, k) *block erasure codes*, which were popularized by Rizzo [108] and are now used in many wired and wireless distributed systems. Figure 3.3 depicts the basic operation of these codes. An encoder converts k source packets into n encoded packets, such that any k of the n encoded packets can be used to reconstruct the k source packets [108]. In this research, we use only *systematic* codes, which means that the first k of the n encoded packets are identical to the k source packets. We refer to the first k packets as *data* packets, and the remaining $n - k$ packets as *parity* packets. Each set of n encoded packets is referred to as a *group*.

The advantage of using block erasure codes for multicasting is that a single parity packet can be used to correct independent single-packet losses among different receivers [108]. We implemented MetaSocket filters for block-oriented FEC encoding and decoding using an open-source Java implementation of Rizzo's C library. In the remainder of the research, we will refer to the block-oriented FEC simply as "FEC (n, k)."

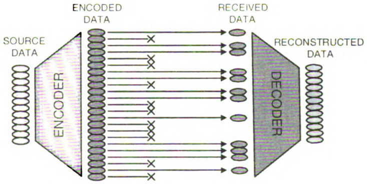
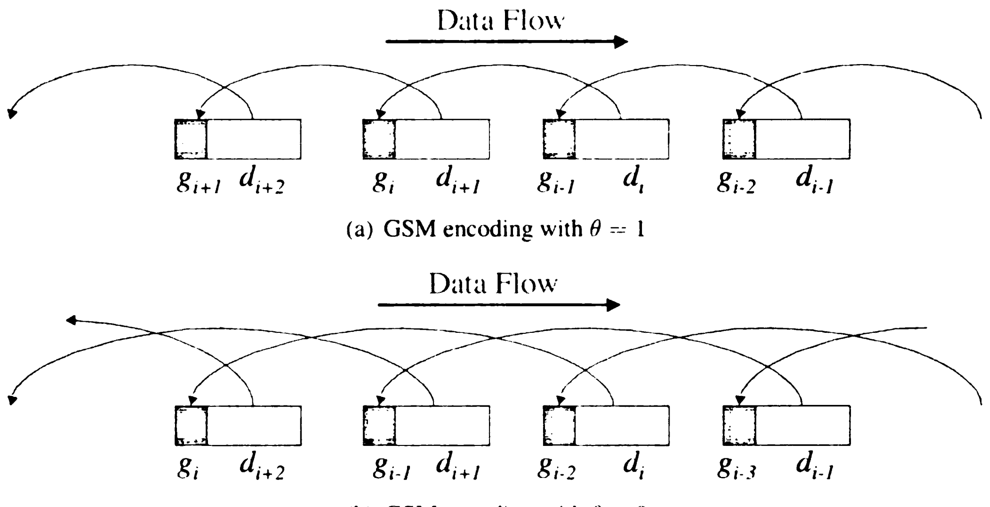


Figure 3.3: Operation of block erasure code.

While block-oriented FEC approaches are effective in improving the quality of interactive audio streams on wireless networks [144], the group sizes must be relatively small in order to reduce playback delays. In our studies, we typically use (n, k) values of (6,4) or (8,4). Hence, the overhead in terms of parity packets is relatively high.

3.4.3 GSM-Oriented FEC Encoder/Decoder

An alternative approach with lower delay and lower overhead is *signal processing based FEC (SFEC)* [127, 145], in which a lossy, compressed encoding of each packet i is piggy-backed onto one or more subsequent packets. If packet i is lost, but one of the encodings of packet i arrives at the receiver, then at least a lower quality version of the packet can be played to the listener. The parameter θ is the offset between the original packet and its compressed version. Figure 3.4 shows two different examples, one with $\theta = 1$ and the other with $\theta = 2$. As mentioned, it is also possible to place multiple encodings of the same packet in the subsequent stream, for example, using both $\theta_1 = 1$ and $\theta_2 = 3$.



(a) GSM encoding with $\theta = 1$

(b) GSM encoding with $\theta = 2$

Figure 3.4: Different ways of using GSM encoding on a packet stream.

We use GSM 06.10 encoding [109] for generating the redundant copies of packets. Although GSM is a CPU-intensive coding algorithm [145], the bandwidth overhead is very small. Specifically, the GSM encoding creates only 33 bytes for a PCM-encoded packet containing up to 320 bytes (160 samples in our experiments). We use the Tritonus Java version of the GSM codec , a freeware package available under GNU public license. Unfortunately, this Java version is unable to satisfy real-time audio encoding and decoding requirements on iPAQs with low processing power, so all the GSM-related experiments were conducted on laptop computers. In the remainder of the research, we will refer to the GSM-oriented FEC simply as "GSM (θ, c)," which means copies of the coded packet p are placed in c successive packets, beginning θ packets after p.

3.4.4 Audio Streaming Application

To investigate adaptation in interactive audio communication, we developed an audio streaming application (ASA), depicted in Figure 3.5. ASA uses MetaSockets instead of regular Java sockets, enabling dynamic insertion and removal of FEC filter pairs, as well as filters to measure and report packet loss characteristics. As shown, the ASA comprises two main parts. On the sending station, typically a desktop computer, the *Recorder* reads live audio data from a system's microphone. The Recorder multicasts this data to the receivers via a MetaSocket. If the MetaSocket is configured to introduce FEC on the data stream, it invokes an FEC encoder and transmits the modified audio stream on the network. On each receiving node, the stream arrives on a MetaSocket, where it is decoded as necessary, and delivered to the *Player* component. When executing on an iPAQ, the Player delivers the stream to the speaker using the Java Native Interface (JNI), necessary due to a known problem with audio in Blackdown Java [140].

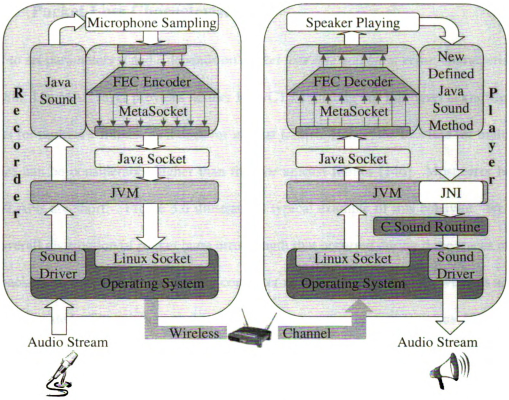


Figure 3.5: Software component interaction.

3.5 Experiments and Results

We first conducted a set of baseline experiments designed to evaluate the effect of FEC/GSM parameter values on energy consumption. For interactive audio streams, the values of k and θ must be relatively small to limit the playback delay to an acceptable level. For example, in many of our experiments we used 8-bit samples and placed 200 samples, or 25 milliseconds of audio, in each packet. If an FEC $(8, 4)$ code is used and the first data packet of a group is lost, then at least 75 milliseconds additional delay will be introduced between the last packet arrival of the preceding group and playing the (decoded) data packet. Therefore, in most experiments we set $k = 2$ or $k = 4$, although in some cases we used $k = 8$ for comparison purposes.

3.5.1 Packet Loss Characteristics

How to set parameters n and c depends on *packet loss rate and burst error* characteristics. The 802.11b MAC layer provides neither RTS/CTS signaling nor link-level acknowledgements for multicast frames, as it does for unicast frames. Hence, the loss rate for multicast frames can be considerably higher than that for unicast frames [146]. Most error bursts in WLANs are short. Figure 3.6 illustrates a typical example of this behavior. We plot the overall distribution of packet burst error length that occurred during three traces of audio packets, as recorded by a receiving computer near the room where our wireless access point is located. The average packet loss rate, across the three traces, was 17%. Also plotted in the figure is the distribution produced by a simulation using a two-state Markov model [147], which is widely used to model losses in wireless networks.

Two characteristics of Figure 3.6 are important to this study. First, while some large bursts occur, the vast majority are under 4 packets long, and most "burst" errors comprise a single packet loss. Such results are encouraging because they imply that a relatively small amount of FEC information is likely to correct most errors, that is, $n - k$ or c can be small. Second, we note that the simulation is reasonably accurate in modeling the loss distribution. Being able to reproduce environmental conditions is notoriously difficult in wireless networks [148], so simulating losses provides a way to test different protocols and parameter values under the same loss conditions. Therefore, many of the experiments described in this section and in Section 3.6 use iPAQs and laptops located near the access point, but with emulated packet losses produced by a two-state Markov model and a specified overall loss rate. The results given in Section 3.7, however, were collected under real packet loss conditions.

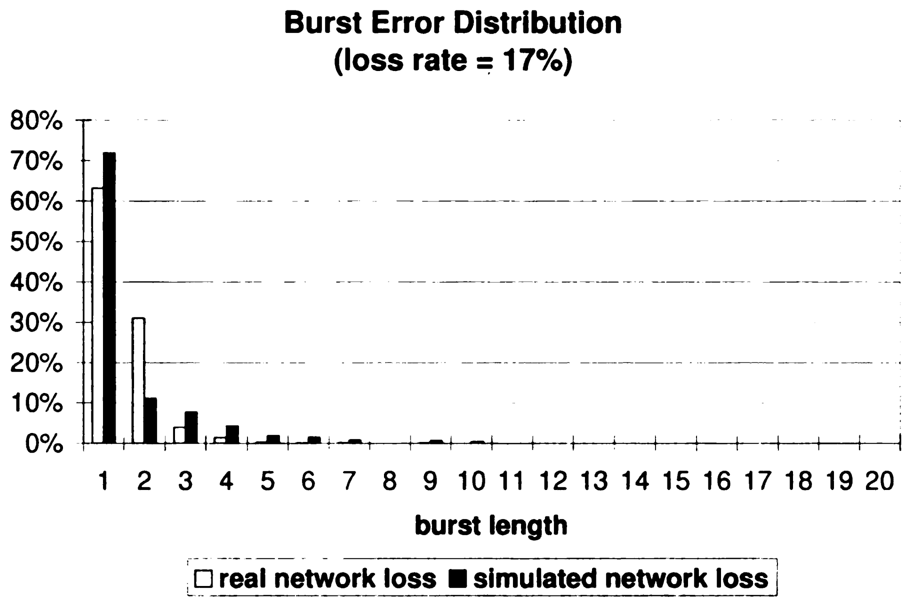


Figure 3.6: Burst error distribution (experiments and simulation).

3.5.2 Effect of n, k Values

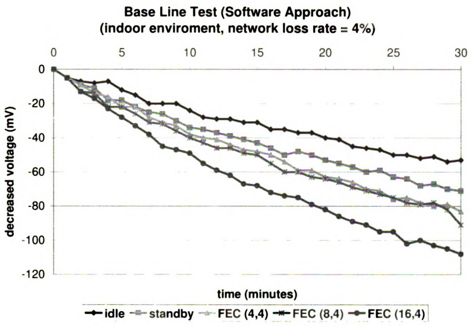
In the first set of experiments, we tested iPAQ receivers in three different execution modes: *idle*, *standby*, and *working*, listed in Table 3.1. The main difference between idle and standby mode is the presence of the WNIC. In standby mode, the WNIC continually listens to the network and exchanges control messages with the access point. The system has to process incoming traffic and maintain communication between the network interface and the operating system. Hence, the difference in energy consumption between idle mode and standby mode can be considered as the energy needed to remain connected to the network. In working mode, the WNIC operates in constantly awake mode (CAM), as opposed to a power saving mode discussed later. In CAM, the WNIC is always listening to the channel. During the experiments, we executed only the ASA application and a very simple power-

Table 3.1: iPAQ execution modes.

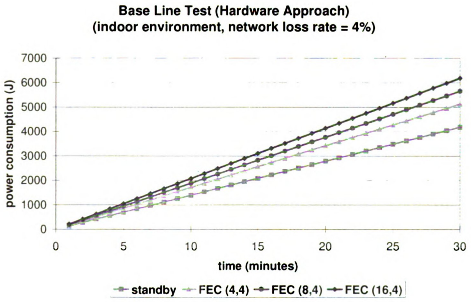
Mode	Description
Idle	Only the system processes are executing; no application; WNIC is not inserted.
Standby	Only the system processes are executing; no application; WNIC is inserted and operating in CAM.
Working	The ASA application is running. The iPAQ receives a continuous stream of audio (data and parity) packets on its WNIC, invokes FEC decoder as needed, and delivers decoded data to application.

sampling program on each iPAQ; we also shut down the backlight to minimize its effect on energy consumption. We varied the (n, k) FEC parameters, as discussed below. Each experiment was conducted three times, and the mean values are reported.

First, we investigate the energy consumption characteristics of block-oriented FEC under the real packet loss conditions. Figure 3.7(a) plots the results of using software to measure the voltage drop (an important variable to represent energy usage) as the iPAQ executed for 30 minutes in different modes. These experiments were conducted near the room containing the wireless access point, where the (actual) packet loss rate is approximately 4%. In this experiment, we set $k = 4$ and evaluate the energy consumption at the receiver side with different n values: 4, 8, and 16. Increasing the value of n causes the voltage level of the battery to decrease faster. The drop is due to receiving and processing additional parity packets (by the WNIC, the operating system, and application software) and, when needed, invoking the FEC decoder. Figure 3.7(b) shows the results using the multimeter to measure energy consumption on the device. The results are commensurate with those in Figure 3.7(a); the more voltage drops in Figure 3.7(a) correspond to the higher energy consumption in Figure 3.7(b) (although as we mentioned earlier, the hardware approach measures the total energy consumption of the system, including the expansion battery of



(a) software measurement



(b) hardware measurement

Figure 3.7: Baseline energy consumption tests.

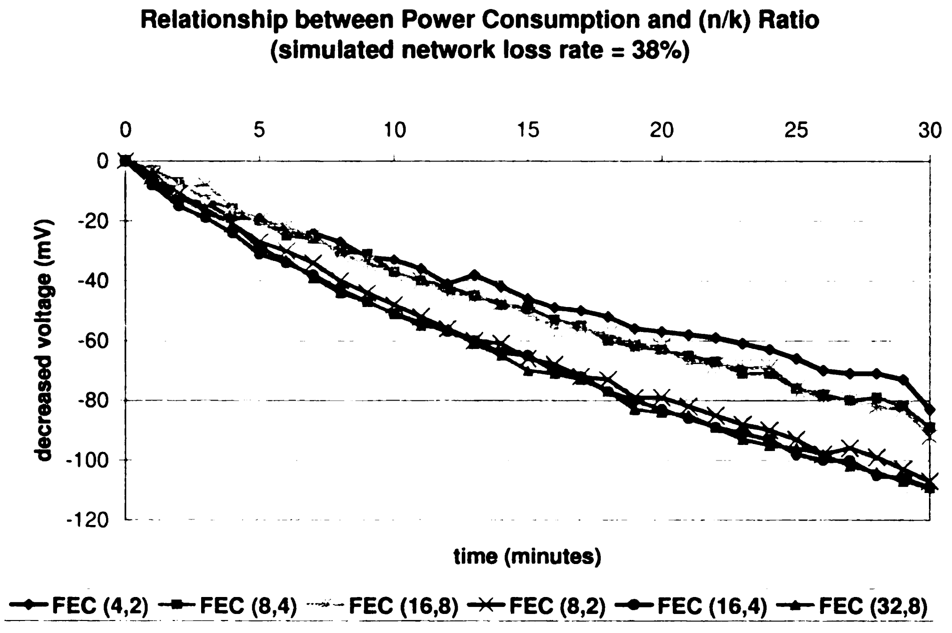
iPAQ).

Figure 3.8(a) shows voltage drop for different (n, k) values under emulated loss conditions, with a mean packet loss rate of 38%. As shown in the figure, the curves are grouped approximately according to the (n/k) ratio: the curves for the $(16, 8)$, $(8, 4)$ and $(4, 2)$ cases, where $n/k = 2$, are relatively close together, as are the curves for the $(32, 8)$, $(16, 4)$ and $(8, 2)$ cases, where $n/k = 4$. This result indicates that the total *number* of incoming (data and parity) packets dominates the energy consumption, at least on the main unit. Other factors, such as how often the FEC decoder is invoked, appear to be less important.

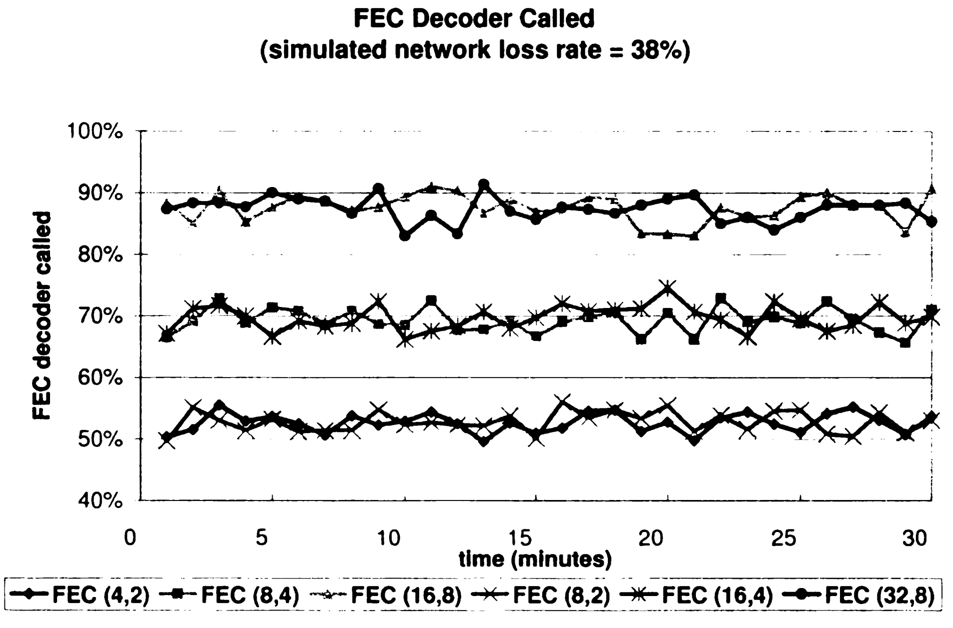
This conclusion is supported by Figure 3.8(b), which plots the percentage of time that the FEC decoder is invoked for different (n, k) pairs, under the same packet loss conditions. The probability of invoking the decoder depends primarily on the value of k, rather than the (n/k) ratio, and we see that the curves are grouped in that manner. However, despite the fact that FEC decoding is computationally intensive, Figure 3.8(a) shows that decoding has little effect on the overall behavior of the voltage drop curves, which are linear in the number of incoming packets.

3.5.3 Effect of Power Saving Mode

Adjusting block-oriented FEC parameters, n and k, is not the the only way to manage the energy consumption. The IEEE 802.11 specification also provides a *power saving (PS) mode*, which can be used to switch the WNIC periodically between "sleep" state and "active" state, in order to conserve energy. In the sleep state, power is shut off to most parts of the WNIC, except the timing circuit. For 802.11 WLANs operated in infrastructure



(a) energy consumption



(b) decoder invocations

Figure 3.8: Baseline experiments.

mode, the access point (AP) buffers frames destined for hosts in PS mode. All PS hosts are synchronized by the beacon from the AP. Each PS host will wake up to listen to the beacon, which contains a delivery traffic indicator message (DTIM). The DTIM identifies those PS hosts for which buffered frames are waiting to be delivered. Those identified nodes will remain awake until the next beacon. After the AP transmits the DTIM message, it transmits any buffered data. Other researchers [149] have investigated how to exploit the 802.11 PS mode, for example, to support energy efficient routing in mobile ad-hoc networks (MANETs). To our knowledge, however, the interaction between 802.11 PS mode and FEC in streaming audio has not been studied previously.

Our first step in understanding how PS mode affects block-oriented FEC audio streams is to observe the traffic pattern created when buffered frames are transmitted by the AP. Those frames are delivered after the DTIM, and the interval between DTIMs is a multiple of the beacon period (100 msec). In the default AP configuration, the DTIM interval is set to 2, which equates to the AP transmitting buffered data every 200 msec. In the following experiments, we set the DTIM to 1, 2, and 4, respectively, and used our wireless network analyzer to monitor the channel and trace the traffic patterns during the transmission.

Figure 3.9 shows a sample of the results, where the AP forwards buffered multicast frames to a single iPAQ in PS mode. The block-oriented FEC parameters in this trace are $(8, 4)$, and the packet size is 200 bytes. The AP transmits the buffered data only after each DTIM, so the number of audio packets transmitted as a "groups" increases with the DTIM interval. The result is a "stairstep" pattern, where each step comprises the packet transmissions following a DTIM. The packets sent between groups are apparently beacon packets. We observe that the number of packets sent between DTIMs varies. The number

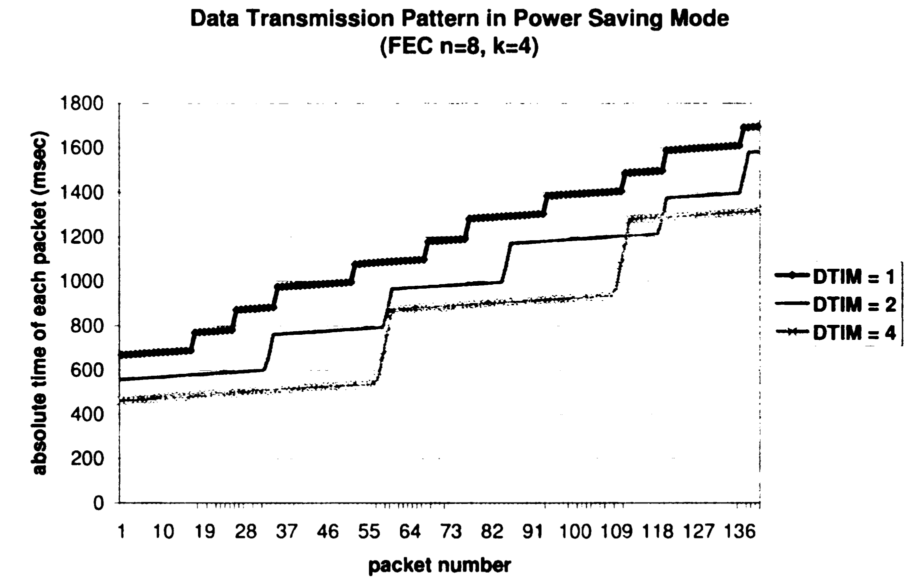


Figure 3.9: Sample trace of packet arrival pattern in power saving mode.

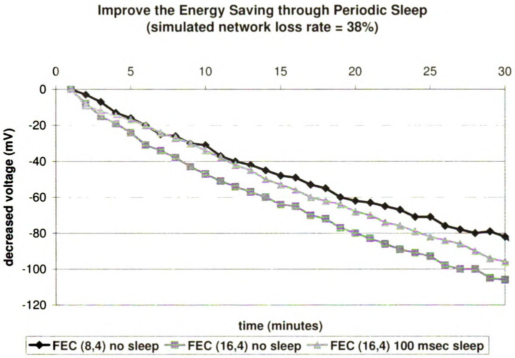
of packets depends on how many FEC groups are sent during each DTIM interval and the spacing between packets relative to the beacon at the AP. For example, if DTIM = 2, then every 200 msec the AP transmits 16 buffered packets.

Next, let us assess the energy savings. Figure 3.10(a) plots the voltage drop over a half-hour experiment, as measured by software, for two different FEC parameters. Use of periodic sleep provides a noticeable, albeit somewhat modest, energy savings on the main unit. However, Figure 3.10(b), which shows energy consumption as measured by hardware, provides a more complete representation of the situation. Power saving mode combined with a $(16, 4)$ code reduces energy consumption by 42% compared to a $(16, 4)$ code without PS mode, and by 37% compared to the $(8, 4)$ code. This result indicates that much of the energy being saved is from the battery in the iPAQ expansion pack, rather than from the battery in the main unit. Since the main unit can draw power from the expansion pack, but not vice versa, the expansion pack battery can drain completely before that of the main unit, leaving the iPAQ operational but disconnected from the network. Indeed, the use of PS mode not only reduces energy consumption, but in doing so, also makes practical use of FEC codes with higher n/k ratios. The effect of PS mode on delay is discussed in Section 3.6.

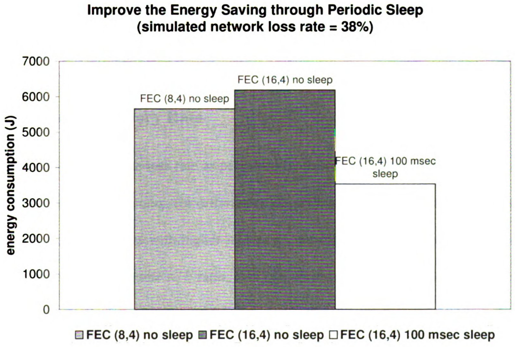
3.5.4 Effect of GSM Coding

From observing the energy consumption characteristics of block-oriented FEC, we conclude that the total number of incoming packets dominates the energy consumption. In contrast, the piggyback method in GSM does not increase the total number of transmitted packets. Therefore, we might expect better energy performance with GSM. This hypothesis is supported by Figure 3.11, which compares the estimated battery lifetime under various FEC configurations. All the GSM configurations are significantly more energy-efficient than block-oriented FEC.

Of course, reporting energy consumption tells only part of the story. Other factors, such as bandwidth usage, packet delivery rate, and delay, must also be considered in assessing audio streaming communication. Considering the examples illustrated in this section, the use of FEC introduces bandwidth overhead that depends on the values of n, k, θ, and c. Given the same packet size, GSM not only is more energy-efficient, but also consumes much less bandwidth. However, in the next section we will see that these savings have a clear effect on QoS and that packet loss rate is not always the most important factor in determining QoS.



(a) main unit (software measurement)



(b) entire system (hardware measurement)

Figure 3.10: Energy savings through periodic sleep.

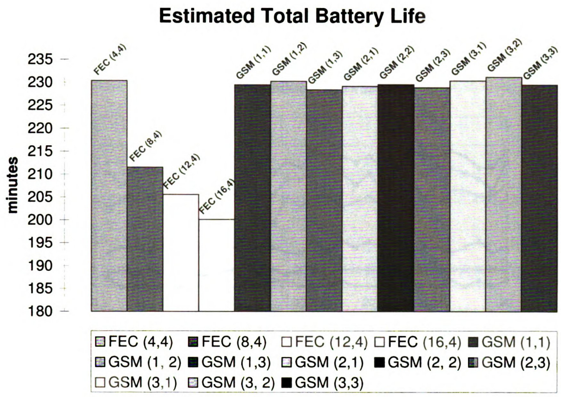
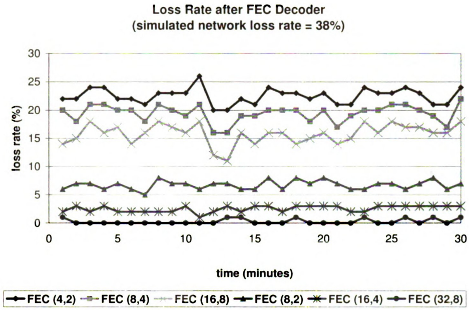


Figure 3.11: Energy consumption for FEC and GSM.

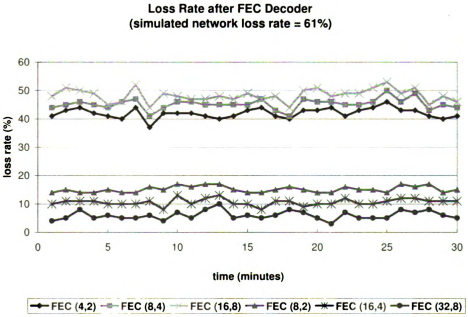
3.6 QoS Assessment

3.6.1 Packet Delivery Rate

Figures 3.12(a) shows the loss rate as perceived by the receiving application, that is, *after block-oriented FEC decoding*, for different (n, k) settings. The mean network loss rate is 38%. As expected, codes with higher n/k ratios are more effective in correcting losses. Among codes with the same n/k ratio, loss rate decreases as n increases. For example, the $(32, 8)$ code results in a lower packet loss rate than the $(8, 2)$ code, even though the two codes consume approximately the same amount of energy. Both codes do well in correcting single packet losses and short burst errors, but the $(32, 8)$ code can handle any burst error



(a) simulated packet loss rate = 38%



(b) simulated packet loss rate = 61%

Figure 3.12: Loss rate after FEC decoding.

of 24 or fewer packets. However, we need to decrease the packet size to compensate for the jitter introduced by the large value of k.

On the other hand, in some high-loss situations, a smaller value of n can produce better results. For example, let us consider the results in Figure 3.12(b), where the mean loss rate is very high, 61%. When $n/k = 4$, a larger n value produces a lower loss rate, as in Figure 3.12(a). However, when $n/k = 2$, a smaller n value is more effective than a larger one. Effectively, since at least half a group's packets must arrive in order to recover the data, and since errors are bursty, a smaller group size is more likely to achieve this goal. For example, consider four groups using a $(4, 2)$ code, compared to one group of the $(16, 8)$ code. Although the number of packets is the same for each, because the loss rate is 61%, on average they will both lose 10 packets. The $(16, 8)$ code can not recover such a loss, but due to the short burst length, in some cases, the $(4, 2)$ can recover one or more groups, yielding a higher packet reception rate.

Next, let us consider the combination of PS mode and block-oriented FEC. The results presented in Section 3.5 confirmed that a lower n/k ratio consumes less energy than a higher ratio. However, in some situations a low n/k ratio FEC cannot meet QoS expectations due to high loss at the network layer, and a higher n/k ratio FEC is needed. Figure 3.13 shows that using a $(16, 4)$ code, with and without a periodic sleep of 100 msec, is very effective in reducing losses, compared to an $(8, 4)$ code. Specifically, the loss rate drops from near 20% to only 3%.

We tested the GSM-oriented FEC by setting θ to different values and using 1, 2 and 3 copies of the encoded data. Table 3.2 shows that using multiple copies produces a clear advantage in terms of packet delivery rate. However, the loss recovery performance for

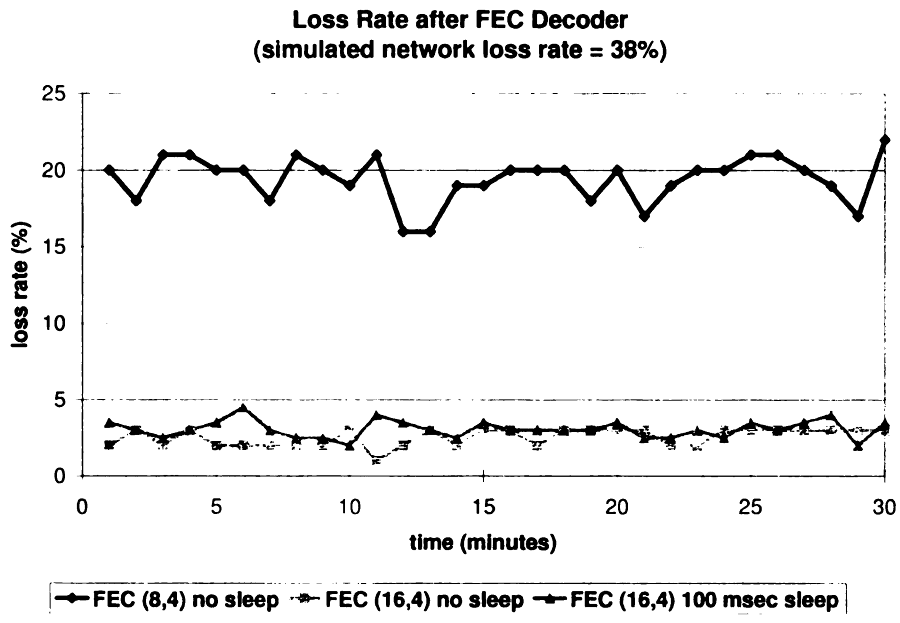


Figure 3.13: Effect of sleep mode on loss rate.

different GSM parameters highly depends on the actual loss distribution. For example, the loss rates of GSM (1, 1), GSM (2, 1) and GSM (3, 1) are not monotonically decreasing as expected.

Table 3.2: Loss rate comparison of different FEC codes.

Code	Raw	GSM (1,1)	GSM (1,2)	GSM (1,3)	GSM (2,1)	GSM (2,2)	GSM (2,3)	GSM (3,1)	GSM (3,2)	GSM (3,3)	FEC (4,4)	FEC (6,4)	FEC (8,4)
%	28	11.05	6.28	3.53	13.88	6.8	3.78	10.27	4.8	2.75	28.20	16.29	9.16

Table 3.3: Delay comparison of different FEC codes.

Code	GSM (1,1)	GSM (1,2)	GSM (1,3)	GSM (2,1)	GSM (2,2)	GSM (2,3)	GSM (3,1)	GSM (3,2)	GSM (3,3)	FEC (4,4)	FEC (6,4)	FEC (8,4)
Delay (msec)	19.95	39.95	59.9	40.83	60.37	80.10	62.55	82.52	102.87	17.79	37.73	52.33

3.6.2 Delay

Another factor important to real-time communication is the additional delay introduced into the packet stream. Table 3.3 calculates the worst case delay introduced by different FEC codes to wait for the encoded packets. For example, considering FEC (8, 4) and GSM (3, 1), if the first data packet is lost, then the receiver will need to wait for at least 3 packets until the first parity packet or piggybacked packet arrives to recover the loss. In order to satisfy the real-time audio requirement, the delay should not exceed 150 msec [128]. Table 3.3 shows that all these codes satisfy this requirement.

Although use of the PS mode introduces delay, we note that most of the delay can be hidden by the use of FEC. For example, considering FEC (8, 4), if each packet contains 25 msec of audio data, then the 75 msec delay incurred while waiting for data buffered at the AP is largely subsumed by the (possible) delay incurred by waiting for parity packets. Specifically, if the first data packet of a group is lost, then the receiver will need to wait for at least 100 msec until the first parity packet arrives and it can decode and play the data. On an 11 Mbps network, the transmission time for the packet is only about 0.3 msec, so sleeping for 100 msec, then retrieving both data and parity packets, actually introduces only a small delay.

3.6.3 Bandwidth

Use of FEC introduces bandwidth overhead that depends on the values of n, k, θ, and c. Considering the FEC (8, 4) and GSM (3, 1) illustrated in Table 3.2 and Table 3.3, which has close loss rate and delay, if the size of the data packets is 320 bytes, then the overhead for FEC (8, 4) is approximately 100 percent since this code doubles the number of packets transmitted; and the overhead for GSM (3, 1) is 10 percent since this method introduces only payload bytes, but no new packets. Based on aforementioned energy consumption and bandwidth comparison, we conclude that the GSM-oriented FEC is not only more energy-efficient but also less bandwidth consuming than the block-oriented FEC.

3.6.4 Audio Quality

Although packet delivery rate, delay, and bandwidth are important objective factors to evaluate the QoS, the most important factor is how the played audio stream sounds to the human ear. Since, the assessment of audio quality by individuals is inherently subjective, we need an objective method. Perceptual Evaluation of Speech Quality (PESQ), defined by ITU-T recommendation P .862 [150], is used to determine voice quality in the telecommunication networks. The PESQ score is mapped to a MOS (Mean Opinion Score) like scale, a single number in the range from -0.5 to 4.5.

Figures 3.14(a) and 3.14(b) use PESQ to compared the audio quality of FEC (8, 4) and GSM (3, 3), which achieves the highest packet delivery rate among the block-oriented and GSM-oriented codes respectively. Although GSM (3, 3) has a higher packet delivery rate under different simulated network loss rates, its PESQ score is lower since the recovered packets are generated from the highly compressed, lossy encodings. Considering that PESQ score of 2.0 and above corresponds to acceptable audio quality [151] and Figure 3.14(c) shows that approximately 20% of the audio data is GSM-quality, we can conclude that GSM-oriented FEC is still suitable for voice communication over wireless

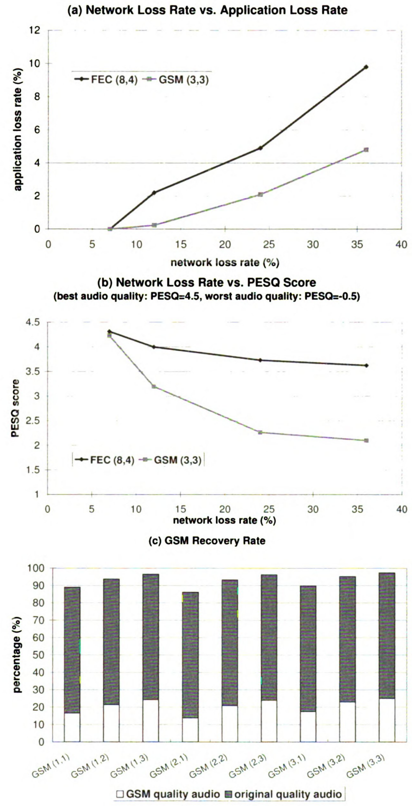


Figure 3.14: Audio quality assessment.

networks. However, in situations where a higher quality audio stream is needed, block-oriented FEC may be worth the additional costs in bandwidth and energy.

From above energy consumption characteristics and QoS assessment analysis, we can build a basis for tradeoffs between energy consumption and QoS. When a user encounters the decision on the selection of FEC configuration under energy constraints, these tradeoffs and user preferences play very important roles in the decision making process. If the user has critical QoS requirement, block-oriented FEC is more effective; otherwise, GSM-oriented FEC is a good candidate since it is apparently more energy-efficient. When using block-oriented FEC, higher (n/k) ratio is more efficient to recover loss but consumes more energy, however, PS mode can help a lot in energy saving; lower (n/k) ratio is energy efficient but error-prone, however, higher n and k can help in increasing packet delivery rate.

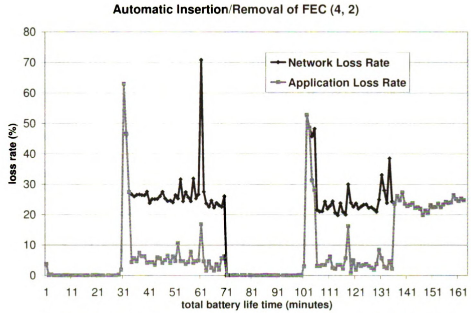
3.7 Toward Dynamic Adaptation

Experimental results such as those presented above can be used to develop rules for dynamic adaptation in mobile computers. Although this aspect of our project is ongoing, we present a sample of the results here. We have conducted a series of experiments in which we used MetaSockets to provide adaptive error control for interactive audio streaming.

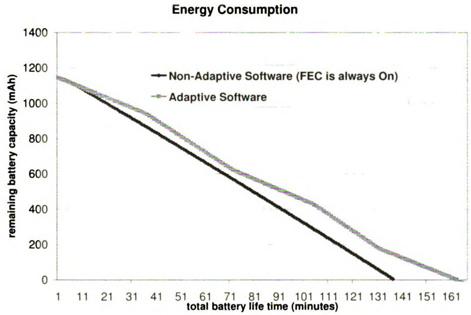
In our implementation, two MetaSocket filters, SendNetLossDetector and RecvNetLossDetector, cooperate to monitor the raw loss rate of the wireless channel. Similarly, the SendAppLossDetector and RecvAppLossDetector filters are used to monitor the packet loss rate as observed by the application, which may be lower than the raw packet loss rate due to the use of FEC. At present, a small set of rules is used by a decision maker (DM) component to govern changes in filter configuration. For example, if the loss rate observed by the application rises above a specified threshold, then the DM can decide to insert an FEC filter in the pipeline or modify the (n, k) parameters of an existing FEC filter. On the other hand, if the raw packet loss rate on the channel drops below a lower threshold, then the level of redundancy may be decreased, or the FEC filter may be removed entirely.

Figure 3.15(a) shows a trace of an experiment using the ASA described earlier, running in ad hoc mode. A stationary user speaks into a laptop microphone, while another user listens on an iPAQ as he moves among locations in the wireless cell. In this particular test, the iPAQ user remains in a low packet loss area for approximately 30 minutes, moves to a high packet loss area for another 40 minutes, moves back to the low packet loss location for another 30 minutes, then reenters the high packet loss location. He remains there until the iPAQ's external battery drains and the WNIC is disconnected. In this experiment, the upper threshold for the RecvAppLossDetector to generate an UnAcceptableLossRateEvent is 20%, and the lower threshold for the RecvNetLossDetector to generate an AcceptableLossRateEvent is 5%. As shown in Figure 3.15(a), the FEC $(4, 2)$ code is effective in reducing the packet loss rate as observed by the application.

Figure 3.15(b) plots the remaining battery capacity as measured during the above experiment. The overlaid slope curve clearly shows the changes in battery capacity expectancy. Depending on conditions or the criticality of some other applications, if this slope indicated that the remaining battery capacity is not enough to keep FEC working, another rule might dictate a change in FEC parameters or removal of the FEC filter (as the 174th minutes shown in Figure 3.15(a)) to maintain the communication even though the QoS de-



(a) MetaSocket packet loss behavior with dynamic FEC filter insertion and removal



(b) trace of energy consumption during experiment (software measurement)

Figure 3.15: Adaptation between energy and QoS.

creased. Figure 3.15(b) also compares the energy performance of non-adaptive software versus adaptive software. If the audio streaming application is not adaptable, the FEC filter has to be present all the time, resulting in energy waste when the network condition is good. Contrarily, adaptive software can change the FEC configuration dynamically according to available energy resource and user preference, taking advantage of the tradeoffs between energy consumption and QoS. As a result, the adaptive version extends the battery lifetime by approximately 27 minutes.

3.8 Conclusions

In this chapter, we evaluate the energy consumption of forward error correction on wireless devices, where encoded audio streams are multicast to multiple mobile computers. Our results quantify the tradeoff between improved packet delivery rate, due to FEC, and additional energy consumption, delay, bandwidth usage caused by receipt and decoding of redundant packets. We also study the impact of the 802.11 power saving mode on system energy consumption and compared two different FEC approaches. These results are promising and indicate that significant savings are possible through appropriate adaptive management of system resources. In the remaining research, we use these studies as a basis for the development of adaptive software mechanisms that attempt to manage these tradeoffs in the presence of highly dynamic wireless environments.

From the experience of understanding the basic adaptation characteristics, we know that an adaptive system often consists of three basic functional units: sensing unit, decision making unit, and execution unit. Thus, achieving acceptable quality of service in highly dynamic computing environments requires not only adaptation and reconfiguration of individual components of the composite system, but also collaboration among these components. To address the integration and collaboration of adaptive computing components, in the next chapter we propose COCA, a message-based collaborative adaptation infrastructure. COCA provides a set of development utilities and run-time utilities that enable different legacy components to be integrated into an adaptive system.

Chapter 4

REALIZING COLLABORATIVE ADAPTATION FOR MOBILE SYSTEMS

4.1 Introduction

Software runs in a changing environment. Some types of changes might be anticipated, such as those associated with battery lifetime, CPU load, memory usage, or available network bandwidth. Other changes, such as new security threats, might be unknown at development time. One approach to addressing unanticipated adaptation is to take the system off-line, modify it, and then restart the system. However, some software, such as that used to manage critical infrastructures (e.g., financial networks and power grids) cannot afford downtime for reconfiguration. In other cases, such as sensor networks used to monitor remote geographic locations, the system may be physically inaccessible. Compositional adaptation techniques address this problem by enabling software to change its structure and behavior dynamically in response to external conditions [18, 152].

In recent years, adaptive behavior has been investigated for different parts of the computing environment. Many approaches introduce adaptive behavior in middleware [45, 57, 58, 60, 61, 63–65, 153, 154], exploiting information hiding to enhance portability while taking into account application-specific requirements and constraints. Finally, some approaches integrate context-awareness into the application itself, either explicitly in the application business code [155] or by "weaving" new behaviors transparently into the application at compile- or run-time [71, 156–158].

Supporting adaptation in individual parts of the system can address many aspects of dynamic execution environments. However, some situations require coordinated responses from multiple system components. Even a relatively simple multimedia conferencing application for mobile users might need to balance quality of service against other concerns, such as energy consumption, security, and fault tolerance. This need has fueled increasing interests in more holistic approaches to adaptation, where an *adaptive system* comprises multiple adaptive components, possibly spanning multiple system layers, that collaborate to achieve overall system goals. Example cross-layer (and collaborative) adaptation frameworks include Odyssey [8], GRACE [6], DEOS [9], and Chisel [10]. In these systems, collaboration is realized by either constructing components specifically for integration in the common framework [6, 8, 9], or by transparently augmenting components with interfaces to the framework [10].

Increasingly, however, many distributed computing systems are constructed from preexisting and relatively independent components. For example, a conferencing system might integrate existing components for streaming audio and video, displaying images and graphics, and managing access to a shared whiteboard. This trend poses three important challenges to the design of adaptive systems. First, the individual components might not support adaptive behavior at all, or might not support the type of adaptation needed in the target environment. Second, even if they are individually adaptive, the components might have been developed by different organizations, using different languages and/or different middleware platforms, and using different (and likely incompatible) approaches to adaptation. Third, some method is needed to specify and coordinate the collaboration among the components in order to realize system-wide adaptations.

The first two problems can be addressed using a variety of techniques that enable new behavior to be woven into existing components transparently with respect to the original code [159–163]. Our group has previously developed a set of such techniques, called *transparent shaping* [159], to enable collaborative adaptation in composite systems [160, 164]. In this chapter, we focus on the third problem. In coordinating adaptation among components, it is desirable that the system be to some extent *autonomic*, that is, capable of self-management with only limited human guidance [165]. Ultimately, we would like systems to be capable of *learning* how to adapt to changing situations. In this work, however, we focus on an intermediate step, the use of message-based communication to guide collaborative adaptation. Our focus here is on providing an infrastructure to support collaborative adaptation among components that were not necessarily designed to interoperate.

We propose COCA (**CO**mposing **C**ollaborative **A**daptation), an infrastructure for collaborative adaptation in composite systems. The main contributions of this work are threefold. First, COCA provides a set of development utilities to aid system designers in specifying system architecture and adaptation policy, and automatically generating the corresponding code to realize collaborative adaptation among existing components. Second, COCA provides a set of run-time utilities to enforce the collaborative adaptation execution. Third, COCA provides a Web services infrastructure to support the corresponding interaction among components.

The remainder of this chapter is organized as follows. We briefly introduce the background of this research in Section 4.2. Section 4.3 provides an overview of the architecture and operation of COCA. In Section 4.4, we review M^2 [164], a communication protocol used to realize interaction among COCA clients and components of the COCA infrastructure. To help illustrate various aspects of COCA, we use a running example on the use of COCA to construct an adaptive multimedia conferencing system from legacy applications; we describe the composition of this system in Section 4.5. Section 4.6 discusses the details of COCA specifications, including their structure and the set of tools used to construct them, translate them into code and enforce them during execution. In Section 4.7, we present experimental results demonstrating the ability of the COCA-enabled conferencing system to detect and respond to changing conditions. Conclusions are given in Section 4.8.

4.2 Background and Related Work

COCA is most closely related to two classes of projects that use contracts in QoS adaptation: those that use architecture description languages (ADLs) to describe how components in an adaptive system interact with one another [53,54,87–90], and those that use a policy-oriented approach to guide the adaptation process during execution [10,60,79,91–99].

Quality Objects (QuO) [60] is a mature project at BBN Technologies that provides support for QoS adaptation in CORBA applications. QuO enables weaving of QoS aspects, referred to as *qoskets*, into the applications at compile time by wrapping stubs and skeletons with specialized delegates, which intercept requests and replies for possible modification. COCA complements such functionality by enabling collaborative adaptation among components designed for *different* platforms. A suite of tools, discussed later, is used to "shape" existing applications so that they can interact with the COCA infrastructure. Indeed, QuO applications could be plugged into a COCA framework very easily by simply defining the appropriate qoskets for such interaction.

To dynamically reconcile QoS conflicts among components at run-time, GlueQoS [79] provides a mediation mechanism to support the dynamic management of QoS features between two components. GlueQoS policy mediators (GPMs) are added to each component and cooperate to configuration QoS features and policies for run-time adaptation. The GPM on each end oversees the configuration of QoS features at that end and evaluates policy based on runtime conditions; it then communicates with its counterpart GPM at the other end to compute an intersection of their policies to find a composition agreeable to both ends. If the compatible QoS feature composition cannot be found, the interoperation between these two components is refused to prevent malicious operation. The police is described in GlueQoS policy language (GPL), a declarative language used in GlueQoS for specifying the QoS feature preferences. As an extension to the Web Services Policy approach [166], GlueQoS addresses the interaction between Web services providers and requesters. In contrast, COCA supports policy-based collaboration between *general* service providers and requesters, which could be individual software applications, middleware or

Web services.

The Contract-based Adaptive Software Architecture (CASA) framework [167–169] addresses enabling the development and operation of adaptive applications in the way of providing "resource awareness" and "dynamic adaptability" to the applications. To achieve the system-wide adaptation, each application runs on an instance of the CASA run-time system, which consists of the Contract-based Adaptation System (carrying out dynamic adaptation on behalf of its associated application), the Resource Manager (monitoring the value and availability of resources), and the Contract Enforcement System (comparing the application resource requirement with the available resource, and selecting appropriate configuration according the pre-defined contract). The adaptability provided by CASA is based on component recomposition, which requires each application to provide various sets of components to constitute the application. However, different applications should be free to adopt their own adaptation techniques. In the contrast, the implementation and run-time management of COCA is set up with respect to the original code and adaptation techniques of the compositional applications. Furthermore, the contract used in CASA is policies similar to that used in QuO. It only defines different operation "zones" in response to different change environments, and it does not provide formal reasoning mechanism and enforcement support as COCA.

Rainbow [53] addresses architecture-based self-adaptation issues by providing a reusable infrastructure. The reusable infrastructure here is based on the "external models," which separates concerns of problem detection and resolution from the system that is being adapted. In this way, the general infrastructures provided by Rainbow can be easily reused by different systems that have the similar adaptation requirements. At this point, COCA and Rainbow adopts the same concept of enabling adaptation functionalities in a legacy system. The reusable Rainbow units include: (1) system-layer infrastructure which measures and probes various system states for problem detection; (2) architecture-layer infrastructure which aggregates the information from the system-layer infrastructure and makes the adaptation decision; (3) translation infrastructure which maps the system model to the concrete implementation; (4) system-specific adaptation knowledge which can be used to guide the system adaptation. Unlike COCA, which provides an adaptation infrastructure to serve the collaborative adaptation among various elements, Rainbow is inherently centralized, focusing on the adaptations within a single Rainbow instance. This inherent characteristic determines that the "reuse" of Rainbow infrastructure is conditional: only when two Rainbow instances has the same concerns, can they share/reuse the existing implementations of all or partial infrastructures. For example, when two systems share the concerns about the system bandwidth usage, they can share the system-layer infrastructure. However, if one system cares about the bandwidth while the other one is interested in the CPU usage, their system-layer infrastructures cannot be reused by each other. Thus, the reusable infrastructures proposed by Rainbow are system specific and may be used more in system modeling rather than real system construction.

The Chisel project [10] provides an open framework for dynamic adaptation that leverages the advantages of existing commonly-used middleware while supporting collaboration among elements in a distributed system. To support dynamic adaptation behavior, Chisel uses IguanaJ meta types [45], which provide a mechanism to associate non-functional behaviors to base-level objects and classes, as the adaptation mechanism. Based on this reflective programming model, the particular aspects of a service object can be decomposed into multiple possible behaviors, and the service object can be adapted at run time as the execution environment, user context, and application context changes. To support collaboration among adaptive applications and service objects, Chisel uses a policy-based approach to control the dynamic adaptation behaviors by incorporating user and application specific semantic knowledge and intelligence. However, the policy-based control provided by Chisel lacks reasoning capability so that it might difficult for the application to deal with complicated decision making process. COCA, however, adopts Jess rule base engine as the decision maker for the adaptation enforcement service, which provides the effective adaptation reasoning.

In summary, the related approaches described above have been shown to be effective in solving specific adaptation problems. However, most of these systems either target a specific middleware platform or require components to be designed explicitly to interact with an adaptation infrastructure. By using a suite of tools to transparently weave COCA interfaces into existing applications, in language- or platform-specific ways, COCA is not constrained in this manner. Indeed, COCA complements many other approaches because it can easily integrate applications designed for other adaptation infrastructures. Finally, since COCA not only facilitates application integration by generating "glue code," but also generates rules to govern the adaptation process during execution, it provides a significant step toward automating the construction of the decision structure for adaptive systems.

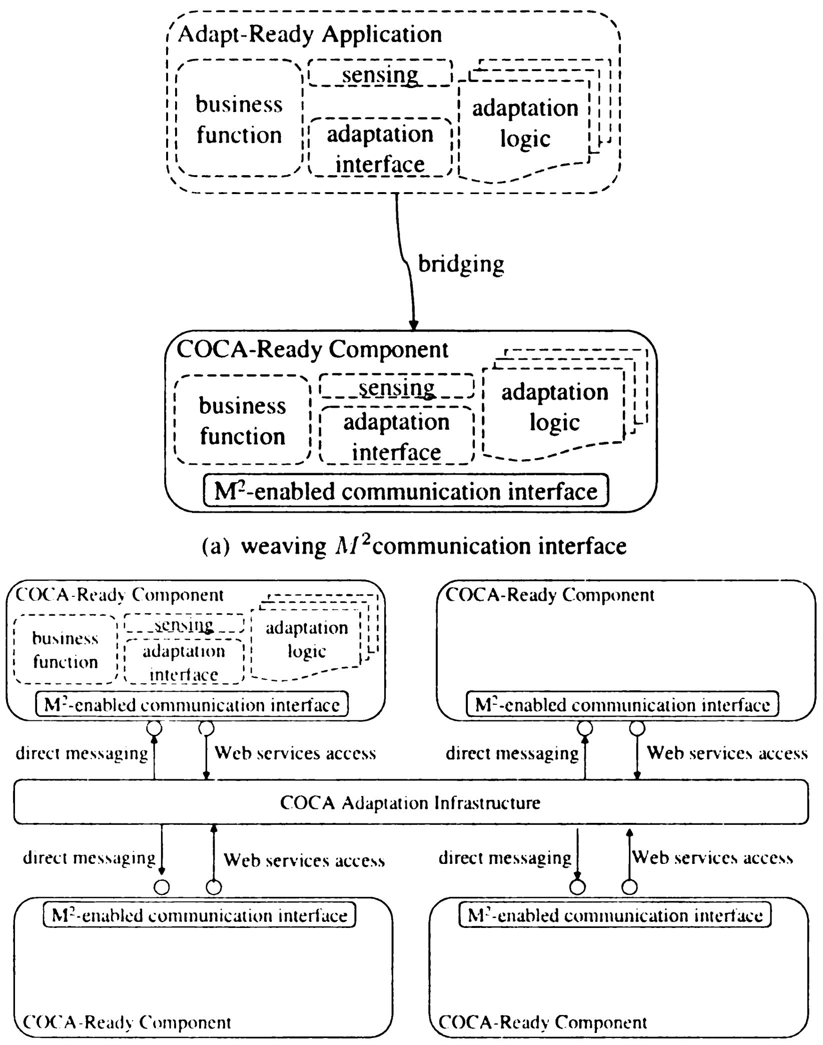
4.3 COCA Overview

In this section, we provide an overview of the COCA architecture and its operation. We first discuss the process of *bridging*, in which an existing (legacy) component is tailored to interact with the COCA infrastructure. And then, we introduce the general architecture and key components of COCA infrastructure.

4.3.1 Bridging Existing Applications

The COCA infrastructure is based on Web services, which provide a standardized way to integrate applications over the Internet by means of XML, SOAP, WSDL and UDDI. *Adaptation services clients* are adaptive components that collaborate through the COCA infrastructure. Of course, many legacy components considered for integration in an adaptive system may not support a Web services interface. We use the term *bridging* to refer to the process of weaving a Web services interface for COCA-related communication into an existing component. The interface supports a COCA protocol called M^2, discussed later in Section 4.4.

Figure 4.1(a) shows the bridging process, which produces a *COCA-ready* component. Figure 4.1(b) shows a collection of four bridged components, interacting via the COCA adaptation infrastructure. The COCA interface enables a bridged component to (1) report events of interest to COCA, and (2) make its local adaptation mechanisms accessible to the COCA infrastructure. One could modify the component manually to support such functionality, but a better approach is to weave in the communication interface transparently with respect to the existing business code. The mechanism(s) used on a particular component depends on the characteristics of the component, including the programming language and any middleware platform used in its implementation.



(a) weaving M^2communication interface

(b) communication with COCA infrastructure

Figure 4.1: Bridging an existing application to work with COCA.

In the past few years, our group has developed several techniques that can be used to implement bridging. These techniques are referred to collectively as *transparent shaping* [159]. Although primarily intended to enable new adaptive behavior to be added to individual components, transparent shaping also provides a means to enable existing adaptive components to interact with COCA adaptation infrastructure. In this case, the new "adaptive" behavior is the support for COCA-related operations. Transparent shaping tools developed by our group include TRAP/J [69], a generator framework that enables automatic generation of the necessary aspects; TRAP/C++ [170], a C++ version of TRAP that uses OpenC++ instead of AOP to define adaptation hooks; and ACT [65], a framework that uses CORBA portable interceptors to support transparent adaptability and interoperability of CORBA applications. In addition, frameworks such as IguanaJ [45] and QuO [60], designed to add new behavior to existing applications, can also be used to implement bridging in COCA.

4.3.2 COCA Architecture

Figure 4.2 shows an example of the COCA architecture. This example includes only a minimal set of adaptation-related services supported by COCA; additional services can be added easily. Included in Figure 4.2 are a Messaging Service, a Naming Service, a Specification Processing Service and an Adaptation Enforcement Service. The COCA Messaging Service provides a message interchange center for the entire system. The COCA Naming Service provides a tree-like directory for component references. The COCA Specification Processing Service maintains high-level specifications of system goals and constraints, and maps these specifications onto low-level behaviors of each system component. The COCA Adaptation Enforcement Service is a rule engine that provides formal reasoning support for checking conditions and selecting corresponding actions. The Specification Processing and Adaptation Enforcement Services are discussed further in Section 4.6.

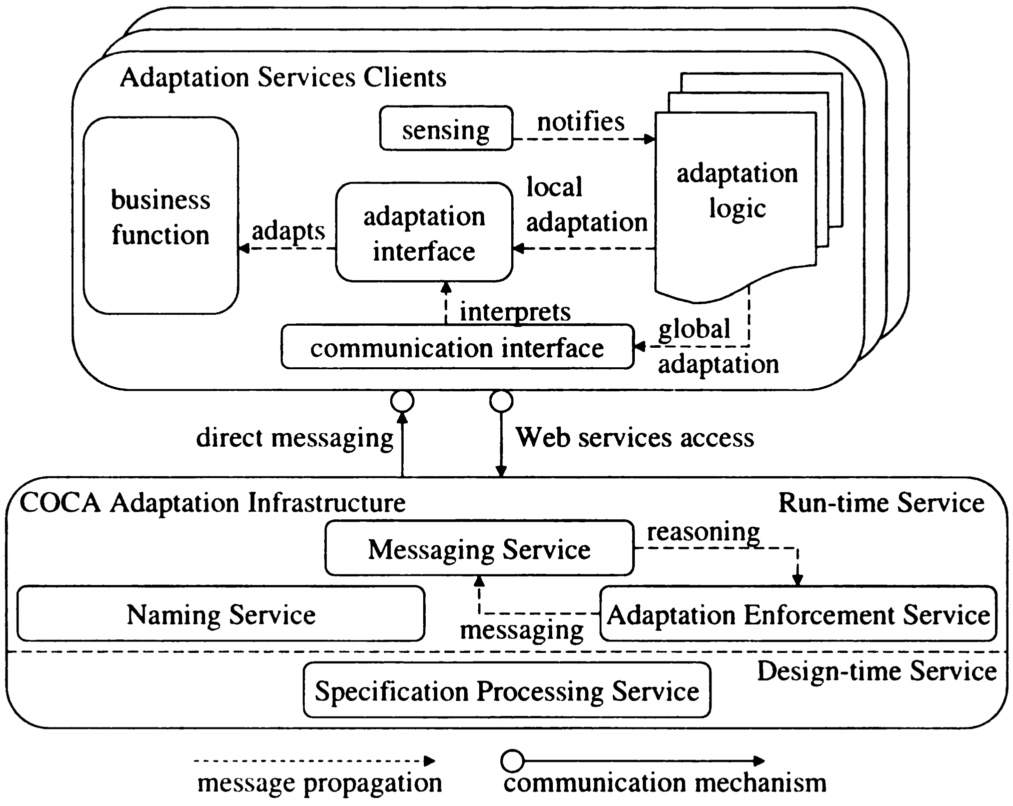


Figure 4.2: COCA architecture and operation.

Figure 4.2 also depicts message propagation in a COCA-based system. When the sensing unit of a system component detects a run-time environment change that could trigger system adaptation, it first notifies the local adaptation decision maker. This component (if there is one) decides whether it can handle the adaptation locally. If not, it passes the adaptation request to the COCA Adaptation Enforcement Service, which manages the interactions and collaborations among adaptive components, thereby implementing global adaptation. All these decision making processes and adaptations are governed by policies, as described in subsequent sections. The selected adaptation action will be sent to the target components by means of M^2, a collaborative adaptation protocol we developed in a preliminary study [164].

4.4 The M^2 Communication Infrastructure and the M^2 Protocol

4.4.1 Supporting Communication among Compositional Components

COCA components communicate with one another through M^2-based messages, and wraps M^2communication infrastructure with a Web services interface. M^2 uses two types of techniques to deliver messages among components in an adaptive system. First, M^2 supports existing distributed middleware platforms: CORBA, .NET, and Java RMI. This approach enables M^2 to take full advantages of existing distributed middleware techniques and avoids a lot of trivial details such as marshaling/unmarshaling, type safety, etc. Second, some resource-constrained mobile computing devices cannot afford or do not support the aforementioned distributed middleware platforms. For those devices, M^2 propagates messages directly through TCP/IP support from the operating system. Depending on system configurations, M^2 in an adaptive system may support a subset of the aforementioned communication techniques. For example, a copy of M^2 middleware on a Linux laptop may support CORBA-, Java RMI-, and TCP/IP-based message delivery.

In order to send and receive messages, the source component has to be able to locate the target component. M^2 defines a hierarchical universe to solve this problem. The entire M^2 universe comprises a set of sites, each of which contains multiple adaptive components. Components here can be adaptive applications that achieve system functions, platform brokers that gets context information from and reconfigure platforms such as operating systems and middleware, services that coordinate other adaptive components, and M^2 itself. In this universe, as shown in the upper portion of Figure 4.3, M^2 defines its own hierarchical addressing mechanism to locate each individual component, including, the communication protocol used by this component (e.g., `m2c` for M^2 over CORBA, `m2r` for M^2 over Java RMI, `m2n` for M^2 over .NET, `m2m` for M^2 over TCP/IP), the location of the component (i.e., IP address and corresponding port number), path of this component on the site including type of the component (i.e., `app` for applications, `plt` for platform brokers, `sev` for adaptation service components, or `msq` for M^2 itself), and the name of the component. For example, `m2r://copland.cse.msu.edu:1099/app/audio` is the M^2 universal address for an application (`app`) named `audio`. This application runs on host `copland.cse.msu.edu`, listening to port `1099` (rmiregistry port). This component uses Java RMI to communicate with other components.

```
URL-Based Element ID Format:
   comm_protocol://host:port/path_of_component
   comm_protocol is m2m|m2r|m2c|m2n

XML-based Message Format:
 <msg name=msg name>
   <source>source id</source>
   <target>target id</target>
   <timestamp>timestamp</timestamp>
   <params>
     <param name="param name">
       <type>param type</type>
       <value>param value</value>
     </param>
     ....
   </params>
 </msg>
```

Figure 4.3: The M^2 XML message format.

As shown in the lower portion of Figure 4.3, an M^2 message contains five fields: *name*, *source*, *target*, *time-stamp*, and *params*. The *name* field indicates the name of this message. The *source* and *target* fields are the unique universal addresses as described above. The *time-stamp* field indicates when this message was created. The variable-length *params* field indicates the parameters of this message.

In order to pass messages, M^2 has a set of *message routers* that listen to specific ports (depending on the communication techniques supported, for example, Java RMI message router may listen to port 1099), collect, and distribute messages between various components. M^2 administrator designates these port numbers when M^2 starts. Each component is equipped with a *message gateway* to send and receive messages for the component. Currently there are four types of message gateways: CORBA message gateways that communicate with the CORBA message router in M^2; Java RMI message gateways that communicate with the RMI message router in M^2; .NET message gateways that communicate with the .NET message router in M^2; and TCP/IP message gateways that communicate with the TCP/IP message router in M^2. The message gateways represent their corresponding local elements, communicate with the message router in M^2 middleware, and exchange messages with other components. In order to deliver messages across different distributed platforms, there is a inter-communication protocol router (ICPR) that exchanges messages among the CORBA message router, RMI message router, .NET message router, and TCP/IP message router. For example, if the RMI message router gets a message from one of the message gateways and the target element of that message uses CORBA instead of RMI, it forwards this message to the the ICPR and the ICPR forwards this message to the CORBA message router, and the CORBA message router delivers this message to its target which supports CORBA.

The message gateways, message routers, and the ICPRs coordinate and deliver messages across the system. As shown in Figure 4.4, each time the message gateway receives a message from the local component, it forwards it directly to its corresponding message router (lines 1-4). When a message router gets messages from its corresponding gateways or from the ICPR, it unwraps the message (lines 5-6). The message router first checks the communication protocol used by the target component. If the target component uses the same communication protocol as the source component (line 7), then the message router checks the location of the target component (line 8). If the target is a local component (line 9), then the message router forwards the message to the message gateway of the target component (lines 10-11); if the target component is on a remote adaptive system that has a different message router, then the message is forwarded to the corresponding message router (lines 12-14). If the target of this message is using a different communication protocol as this message router and the source component, the message router forwards this message to the ICPR and the ICPR dispatches the message to the corresponding message router that supports the communication protocol the target component uses (lines 15-20). The target component then gets the message from its gateway and processes the message. In M^2, we do not constrain how components are attached to their corresponding gateways. One possible implementation is through the observer design pattern [171], where the component works as an observer of the message gateway. When new messages arrive, the message gateway notifies the component as defined by the observer pattern.

```
1. An adaptive component sends a message to its local message gateway; 2. The local
message gateway forwards the message to the local
   message router with the same communication protocol (CORBA,
   .NET, RMI, or TCP/IP);
3. The message router checks the communication protocol of the target
   of this message;
   if(target uses the same protocol){
     check the target host address;
     if(target is on local host){
       send this message to the target message gateway;
       the target gateway forwards the message to the target element;
     }else{
       send this message to the remote message router;
     }
   }else{
     //target uses a different communication protocol
     forward the message to inter-comm protocol router (ICPR);
     the ICPR checks the target comm-protocol;
     the ICPR forwards the message to corresponding message router;
   }
4. Message successfully passed from source to target
```

Figure 4.4: Passing messages in M^2.

4.4.2 Adaptive Message Protocol

Using the previously described message passing mechanism, we defined a message protocol to support the collaboration among adaptive components in an adaptive system. This message protocol defines messages used for handling the responses of an adaptive component upon receiving a message and performing specific actions for dynamic adaptation.

Four categories of messages are defined. The first category is system topology and adaptive component interface messages. The topology of an adaptive system is important information to dynamic adaptation. For example, the decision making component has to know which adaptive components are currently connected in the system (system topology), what kind of context information can be extracted from them, and how they can be reconfigured (via adaptation interfaces). The second category of messages handles the context acquisition and propagation. In order to achieve dynamic adaptation, adaptive components need to obtain system-wide run-time context information. This category of messages obtains context information, and sends context information to interested components. The third category is the system reconfiguration messages that are used to achieve dynamic adaptation. The last category contains miscellaneous messages that serve all other purposes such as status updates. Each of these categories is described in further detail below.

System topology and component interface messages are used to pass system topology and component interface information among components. The system topology contains information about which components are currently in the adaptive system. Component interface includes two types of information: what kind of context information each component can retrieve and what kind of reconfiguration commands an adaptive component supports. A connection message (`conn`) is used by a source component to notify the target component that the system topology or element interface related to the specified name has changed. A disconnection message (`disconn`) is used by the source component to notify the target component that the system topology or the component interfaces related to the specified name has changed. Register messages (`reg`) are used by a source component to notify M^2 that it is interested in the topology of the adaptive system.

The M^2 collaboration protocol requires each component to send `conn` messages when connecting into the adaptive system and to send `disconn` messages when leaving the adaptive system so that interested component can maintain the system topology and use the topology information for adaptation purposes. Once a component registers its interest in system topology and component interface, a copy of system topology and component interface related message will be forwarded to this component so that it will have a complete view of the current topology of the adaptive system: which components are in the adaptive system, what kind of context information can be retrieved from which component, and how components can be reconfigured.

Context acquisition and dispatching messages are used to pass context information among various components. A context acquisition message (`get`) is used by a source component to request the value of the context variable with the given name parameter from the target component. A context dispatching message (`put`) is used by a source component to send the value of the context variable to the target component. Register messages (`reg`) are used by a source component to notify the target component with its interest in the particular context variable.

The M^2 collaboration protocol requires the target component of a `get` message to send the value of the context variable on receiving a `get` message from a source component. Once a component registers its interest in a context variable, this component shall be notified when the value of that context variable changes.

Component reconfiguration messages are used to notify another component in the adaptive system to perform a specific adaptive action. A component reconfiguration message (`reconf`) is used by a source component to request the target component to perform a reconfiguration action with the given name and argument parameters.

The M^2 collaboration protocol requires that an adaptive component perform the specified adaptive action once receiving a `reconf` message.

Miscellaneous messages include messages that serve all other purposes such as status updates. A notification message (`notify`) is used by a source component to convey some information to the target component, such as reconfiguration result, etc.

The message protocol described above defines the rights and responsibilities of individual components and regulates their behavior if they are connected through the M^2 communication infrastructure.

4.5 Case Study Application: Mobile Multimedia Conferencing

To evaluate COCA, we have used it to support collaborative adaptation in a multimedia conferencing system comprising video, audio, and textual caption components. In this section we review the main components of the system and how they can be adapted individually. In Section 4.6, we use this system to help demonstrate (1) how to compose a COCA specification document that characterizes the structure and adaptation logic of an adaptive system; (2) how to use the COCA specification document to create the adaptive system; and (3) how COCA adaptation services realize the desired behavior of the adaptive system. In Section 4.7, we demonstrate the results of experiments with the COCA-enhanced system.

The conferencing system comprises three existing applications, which interact via the COCA adaptation services. Table 4.1 summarizes the adaptive behaviors of the three applications. The first, *Vplayer*, is a Java application developed using Sun Microsystems JMStudio. Vplayer transmits video and audio streams over the network and can be adapted in two ways: (1) changing the frame rate of the video stream, and (2) switching the video stream off (audio-only mode) or on (audio-video mode). Vplayer is also equipped with a network detector for sensing the network connection changes.

The second component, *ASA*, is a Java audio streaming application developed atop

Table 4.1: The system architecture description of the adaptive conferencing system.

Component	Interface	Action	Constraint
Vplayer	changeFR	change the transmission frame rate	LAN and video is ON
	audioOnly	turn off the video transmission and switch to ASA	video is ON
ASA	insertFEC	insert a FEC facility to reduce the loss rate	WLAN and FEC synchronization
	caption	turn off the audio transmission and switch to Echo	audio is ON
Echo	insertFEC	insert a FEC facility to reduce the loss rate	

MetaSockets [172], which are adaptable sockets whose behavior can be changed dynamically by reconfiguring a chain of packet filters. For example, MetaSocket filters can be used to dynamically change the quality of transmitted streams through techniques such as forward error correction, encryption, and compression. In our study, we use the insertion/removal of FEC filters to accommodate variable packet loss rate on the wireless channel. ASA is equipped with a detector for sensing the observed packet loss rate. Figure 4.5 depicts the configuration of the ASA application, including control flow used to realize adaptive behavior. When the loss rate detector observes a high packet loss rate, it will send a event message to the Adaptation Enforcement Service through the M^2 communication interface, triggering a global adaptation (no local adaptation in this case). Once an adaptation decision is made, the corresponding messages will be propagated to ASA, where they are interpreted as concrete adaptation commands, and producing the corresponding actions.

The third component, *Echo*, is a "closed caption" tool that converts speech to text and transmits it over the network. Echo uses the CMU Sphinx Speech Recognition En-

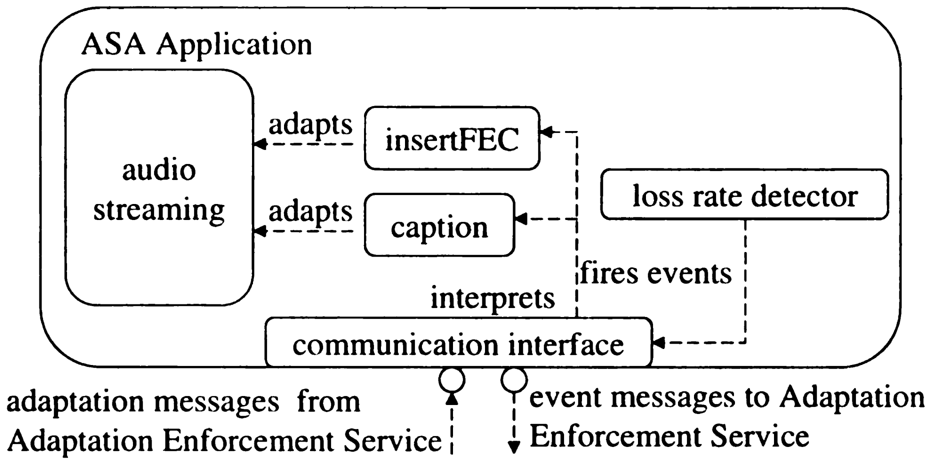


Figure 4.5: An example of components used in the case study.

gine [173] to recognize the speech from a microphone at the sender, and uses the FreeTTS speech synthesizer [174] to reconstruct the speech at the receiver, while also displaying the text. In order to reduce bandwidth consumption, or to make communication more tolerant of high packet loss rates, the Echo application can be used to replace the ASA application. (Live audio is converted to text, sent across the network, and synthesized back into speech.) Moreover, the Echo application itself can be adapted by adding or removing FEC on the text stream.

Figure 4.6 shows the physical configuration used in our experiments, and Figure 4.7 shows the class diagram of the components in this mobile multimedia conferencing system. The environment contains a collection of servers and a collection of clients, running on a 100 Mbps wired LAN and a 802.11b wireless LAN, respectively. For simplicity, we used one Windows desktop and one Windows laptop to represent the whole connection in the case study we will discuss in Section 4.7. The client/server parts of the above three adaptive components (Vplayer, ASA, and Echo) are running on the corresponding client/server subsystem, creating a multicast conferencing system. This multimedia conferencing system will automatically adapt its behavior in response to the changing bandwidth usage and QoS requirements. Specifically, a client will transmit collected information at the highest quality possible. When the channel conditions are good, both the video and audio stream will be used for interactive communication. When the packet loss rate becomes too high, however, video quality will be poor, so only the audio stream will be transmitted, with FEC applied to the stream as necessary. If the packet loss rate increases further and strong error correction coding is unaffordable, Echo will be activated and the textual version of the speech, with strong error correction coding, will be used to replace the audio stream. When the channel conditions improve, these actions will be reversed. The interactive adaptation is supported by the COCA adaptation infrastructure, which comprises Messaging Service, Naming Service, and Adaptation Enforcement Service.

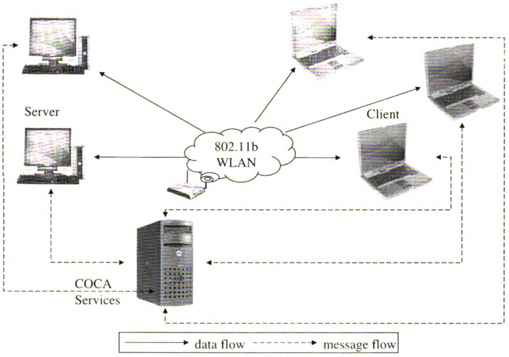


Figure 4.6: Physical configuration of the case study system.

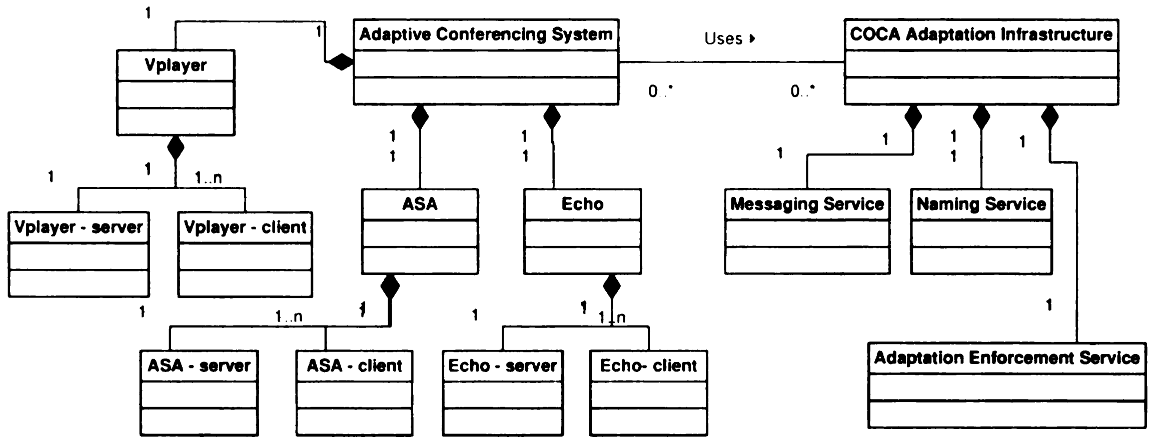


Figure 4.7: Class diagram of the mobile multimedia conferencing system.

4.6 COCA Specification Documents

Using the COCA framework to construct and execute an adaptive application centers around a COCA specification document, whose processing data flow is shown in Figure 4.8. At design time, the application developer uses the Specification Document Composer to write the COCA specification document, describing the system architecture and policies to govern adaptation. The processing of the CSD involves following steps:

Step 1: The developer uses the Specification Checker to validate the correctness of the specification content.

Step 2 and Step 3: The Specification Compiler is used to automatically generate "glue code" skeletons and adaptation rules from the architecture description and the policy description, respectively.

Step 4: In our current implementation, we use AspectJ [71] to transparently shape the application by weaving in COCA communication interfaces (hook points). Thus, the application developer should complete the "glue code" Aspects by mapping the concrete implementation for adaptation with the COCA message processing logic, and use AspectJ compiler to re-compile the business code with the final Aspects to produce COCA-ready code. The end user can then directly execute the COCA-ready code.

Step 5: The generated adaptation rules will be fed into the the Adaptation Enforcement Service, which is the front-end of the Jess rule engine. During execution, the Adaptation Enforcement Service monitors the run-time environment and carries out the adaptive behavior according to the events-actions list derived from the adaptation rules.

In this section, we describe each of these activities in detail, using the conferencing system as a running example.

4.6.1 Composing and Checking COCA Specification Documents

A COCA specification document is an XML file to store the necessary specification information. One of the main benefits of using XML as the basis for COCA specification documents is the abundance of development tools available for constructing, manipulating and checking XML documents. In our COCA prototype, for example, we instantiated the Specification Documents Composer by combining XML Designer in Microsoft Visual Studio .NET [175] and Altova XML Spy [176]. These tools enable the developer to easily create a well-formed specification documents and validate them. As shown in Figure 4.9, a COCA specification document comprises two main parts: the architecture description (used for constructing an adaptive computing system at design time) and the policy de-

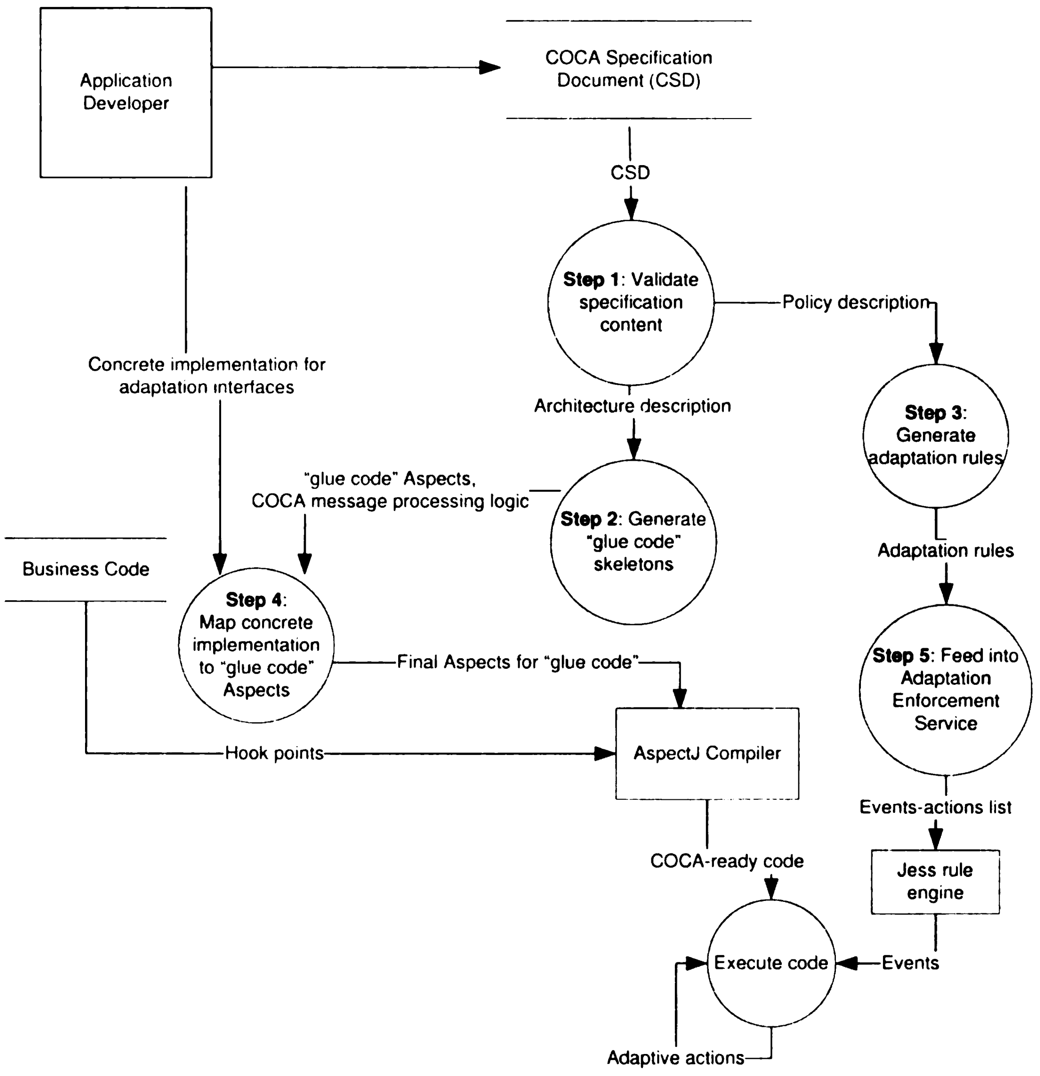

Application Developer
COCA Specification Document (CSD)
CSD
Step 1: Validate specification content
Policy description
Concrete implementation for adaptation interfaces
Architecture description
Step 3: Generate adaptation rules
"glue code" Aspects, COCA message processing logic
Step 2: Generate "glue code" skeletons
Adaptation rules
Business Code
Step 4: Map concrete implementation to "glue code" Aspects
Final Aspects for "glue code"
Step 5: Feed into Adaptation Enforcement Service
Hook points
AspectJ Compiler
Events-actions list
COCA-ready code
Jess rule engine
Execute code
Events
Adaptive actions

Figure 4.8: Data flow diagram for processing the COCA specification document.

scription (used for governing the adaptation behaviors of an adaptive computing system at run time). To handle the possible conflicts between adaptation rules in the policy description, developers can assign different priorities to different rules; the rules with the higher priorities will be used when conflicts occur.

Another purpose of using XML is that the specified information can adhere to a particular set of structural rules and data constraints, ensuring the syntax correctness of the specification documents. However, XML cannot ensure the logical and functional correct-

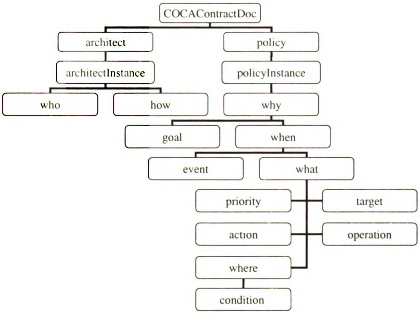


Figure 4.9: COCA specification document.

ness of the contents in the specification documents. Thus, an external, semantically-aware Specification Checker is needed. For example, as shown in Figure 4.10 (left), the component of *app.asa1* needs to be equipped with a communication interface for exchanging messages with other components. Transparent shaping can be used to weave this communication interface into ASA without modifying the original source code. To do so, a hook point (usually the main file of the application) needs to be provided in the specification documents. The Specification Checker checks the correctness of this information in the specification documents, in this example, whether file *Microphone.java* exists or not. In our current implementation, besides verifying such compositional information, the Specification Checker also checks the consistency of the adaptation interfaces and the possible adaptation actions.

The architecture description addresses two main questions regarding adaptation: *Who* are the software components involved in the adaptation? *How* do they interact with one another, that is, through which interfaces? This part of the specification document is used in bridging at design time to generate "glue code" that enables existing components to interact with one another for the purpose of adaptation. The *who* description includes information such as the component name (used to identify the component), communication information (used for message to interchange with other components) and the development language (used to decide the suitable means to generate glue code). The *how* description exposes the interfaces through which the individual components can adjust their behavior to achieve the desired adaptation.

Figure 4.10 (left) shows the architecture description of the ASA application we used in the case study. This server side ASA application is identified as *app.asa1* in the conferencing system, and it exchanges M^2 collaborative adaptation messages through a specified communication port. This topology information can be accessed by other components through COCA Naming Service. The component *app.asa1* has one reconfigurable interface *InsertFilter*, which can insert a FEC facility to reduce the observed packet loss rate. This reconfigurable interface can be considered as an adaptation service provided by *app.asa1*, and other components in the system can invoke this adaptation service by sending an M^2 reconfiguration message to *app.asa1*.

The policy description defines the conditions under which the system should adjust its behavior and the corresponding concrete actions. Specifically, the *why* element groups concrete adaptation activities of the adaptive system in response to the runtime environment; the *when* element identifies the events that will trigger the adaptations; the *what* element defines an action list that will guide the system behavior in response to the trigger events; and the *where* element specifies any constraints that validate the policy rules.

```
...
<architect>
  <architectInstance>
    <who elementName=app.asa1>
      <elementType>
        app
      </elementType>
      <hostAddress>
        copland.cse.msu.edu
      </hostAddress>
      <hostCommPort>
        8888
      </hostCommPort>
      <developLanguage>
        Java
      </developLanguage>
      <mainFile>
        Microphone.java
      </mainFile>
    </who>
    <how interfaceNumber=2>
      <interfaceInstance
          interfaceName=InsertFilter
          type=reconfiguration
          parameterNumber=2>
        <interfaceParameter
            parameterName=FilterName
            parameterType=string>
        </interfaceParameter>
        ...
      </interfaceInstance>
      ...
    </how>
    ...
  </architectInstance>
  ...
</architect>
```

```
...
<policy>
  <policyInstance>
    <why>
      <goal>
        reduce high loss rate
      </goal>
      <when>
        <event>
          high_loss_rate_alert
        </event>
        <what>
          <priority>
            3
          </priority>
          <target>
            app.asa1
          </target>
          <action>
            InsertFilter
          </action>
          <operation>
            FECEncoder
          </operation>
          <where>
            <condition>
              ifFECSynchronized=Yes
            </condition>
          </where>
        </what>
        ...
      </when>
      ...
    </why>
  </policyInstance>
  ...
</policy>
```

Figure 4.10: Excepts of an example COCA specification document: (left) architecture description of ASA; (right) policy description of ASA.

Figure 4.10 (right) shows the policy description of the ASA application. When the conferencing system experiences high network loss rate, a *high_loss_rate_alert* will be generated, and the COCA Adaptation Enforcement Service will be notified. Since the policy agreement states that the conferencing system must adapt its behavior in response to the such an event, the COCA Adaptation Enforcement Service will select from the action list the matching adaptation action with the highest priority. If the *InsertFilter* adaptation of *app.asa1* is selected, its pre-condition (*ifFECSynchronized*) must be checked and satisfied before a reconfiguration message is sent to *app.asa1*.

4.6.2 Translating COCA Specification to Code

An important feature of COCA is its support for automatically translating a COCA specification to code. Such automation is intended not only to lessen the workload on developers, but also to improve the quality of the software by introducing fewer bugs. The Specification Compiler is a code generation tool that produces various parts of the system from corresponding parts of the specification documents. Example products include: communication interface code and (if needed) an adaptive code skeleton woven into legacy components, execution scripts for decision-making by the Adaptation Enforcement Service, and various configuration files. In our prototype, the Specification Compiler includes two main sub-components: the Bridge Generator and the Rule Generator, which translate the architecture description and policy description, respectively.

Figure 4.11 shows the generated code that corresponds to the specification in Figure 4.10. Figure 4.11 (left) gives the glue code produced by the Bridge Generator for connecting existing ASA application to the COCA infrastructure. Besides mapping the architecture description onto a concrete implementation that weaves communication interfaces into existing legacy components, the translating process also generates a code skeleton for weaving new adaptation behavior into the system. The developer can later fill in concrete implementations into this skeleton, if needed.

Figure 4.11 (right) shows the translated policy description of *app.asa1*, as produced by the Rule Generator. In our prototype, we used the Jess rule engine [177] as the basis in the Adaptation Enforcement Service. Hence, the policy descriptions are transformed to Jess scripts, which are managed at run time by the Adaptation Enforcement Service. A Jess

```
public aspect Bridging_Microphone {
  // listen to the COCA messages
  declare parents: Microphone
    implements Observer;
  // connects to the COCA
  // infrastructure
  after() returning(Microphone mjc):
    call(Microphone.new(..)) {
    // register topology information
    // defined by who description
    MadaptHelper.connRemote(
                  "app.asal",
                  "copland.cse.msu.edu",
                  8888);
    MsgGateway.getInstance()
              .addObserver(mjc);
    // register reconfiguration
    // interfaces defined by
    // how description
    ReconfCmd reconfCmd =
              new ReconfCmd("");
    reconfCmd.setCmdName(
              "InsertFilter");
    reconfCmd.addCmdArg(
                 "FilterName",
                 "string");
    MadaptHelper.conn(reconfCmd);
    reconfCmd.clearCmdArgs();
  }
  // process reconfiguration message
  public Microphone.update(
                   Observable arg0,
                   Object arg1) {
    if(arg1 is a InsertFilter msg)
      insert FEC facilities
    process other messages }
}
```

```
...
(deffunction insertFEC ()
  (store msgName "reconf")
  (store targetName "app/asal")
  (store msgParams
    "name=/app/asal/cmd/
    InsertFilter&cmdargs=FilterName:
    string:FECEncoder")
)
(deffunction ifFECSynchronize ()
  (store msgName "reconf")
  (store targetName "app/asal")
  (store msgParams
    "name=/app/asal/cmd/
    IFFECSynchronize&cmdargs=Status:
    string:Yes")
)
...
(defrule reduce_loss_rate_condition
  (declare (salience 3))
  ?adaptFact <-
     (high_loss_rate_alert)
  =>
  (ifFECSynchronize)
)
(defrule reduce_loss_rate
  (declare (salience 2))
  ?adaptFact <-
     (high_loss_rate_alert)
  ?whereFact <-
     (ifFECSynchronize_Yes)
  =>
  (insertFEC)
  (retract ?adaptFact)
  (retract ?whereFact)
)
...
```

Figure 4.11: Code generated from the example COCA specification document: (left) glue code for bridging ASA to COCA; (right) rules for governing ASA adaptation.

rule is similar to an *if...then* statement in a procedural language, however, Jess rules are executed whenever their *if* parts (LHSs) are satisfied, instead of at a specific time and in a specific order as they were programmed. Due to this characteristic, the information of the *when* element in the policy description is used to define the LHS events and conditions of a rule, while the information of the *what* element is used to define RHS actions of a rule. The *where* element defines pre-conditions and constraints under which the policies are valid and enforced, and is often combined with other LHS events and conditions. For example, to ensure software consistency during adaptation [178], when a *high_loss_rate_alert* event message is sent to the Adaptation Enforcement Service by ASA, the condition of the rule *reduce_loss_rate_condition* is satisfied so that the corresponding action of checking the precondition for inserting the FEC facility is selected. Upon receiving this action command, ASA will check if the FEC facility is ready for use and respond to the Adaptation Enforcement Service with an event message *ifFECSynchronize_Yes*. Since the two conditions of rule *reduce_loss_rate* are now both satisfied, the adaptation action of inserting the FEC facility will be invoked by sending a reconfiguration message to ASA.

4.6.3 Enforcing COCA Adaptation

The main responsibility of the Adaptation Enforcement Service is to interpret the policies in the COCA specification and guide the system behavior according to the dynamic runtime environment and the corresponding policies. In general, the implementation of the COCA Adaptation Enforcement Service could be based on any reasoning engine that is able to connect to the COCA infrastructure through M^2-enabled communication interface. In our implementation, we used the Jess rule engine as the enforcement processor. Jess decides on actions using information supplied in the form of declarative rules. As mentioned above, the Rule Generator part of the Specification Compiler generates Jess rule scripts from the policy description defined in the COCA specification document, and feeds the generated rule scripts to the Jess engine for processing and decision making. Jess maintains a collection of knowledge units called *facts*, and Jess *rules* define actions based on the contents of one or more facts.

Once an adaptation action is selected, the corresponding operation commands will be sent to the target components using M^2 collaborative adaptation messages. For example, when the loss rate detector in the *app.asa1* detects a high packet loss in the network, it fires a *high_loss_rate_alert* event to notify the Adaptation Enforcement Service. Upon receiving this notification, a *high_loss_rate_alert* fact will be declared. The Adaptation Enforcement Service will select the adaptation action with the highest priority in the action list. If the adaptation policy needs to be modified at run time, system developers simply need to re-generate Jess scripts with the Rule Generator and feed them to the Jess engine.

4.7 Demonstration

Having shown how a COCA specification document can be used to introduce adaptive behavior to a distributed system, in this section we demonstrate the operation of our COCA-based conferencing system in a wireless environment. We used the Specification Compiler to produce glue code and adaptation rules for the application, as described in Section 4.6. We then compiled individual components using their respective compilers, and ran experiments using the physical configuration shown in Figure 4.6. Wireless networks are notorious for making it difficult to conduct repeatable experiments. To address this problem, we used a packet loss emulator to drop packets according to a packet loss model [179]. In an earlier study [12], we demonstrated the high accuracy of this model.

The initialization of the conferencing system proceeds as follows. We first start the Vplayer application and then the ASA application. Once each application starts, it registers itself with the Naming Service, which is part of the COCA infrastructure running on a separate Web Server. The adaptation goal of the system is set by defining appropriate policies in the COCA specification documents. In this case study, we want the conferencing system to adapt its behaviors according to network channel conditions (a loss rate sensor will fire *low_loss_rate_alert*, *high_loss_rate_alert*, and *extreme_loss_rate_alert* events based on observed packet loss rate of lower than 20%, 20%-40%, and higher than 40%, respectively), so we can specify the actions to be taken in response to different events (e.g. we give the ASA adaptation rules in response to the *high_loss_rate_alert* event in Figure 4.10 (right)). For each individual event, there may be several matching actions and the selection among these actions is based on their priorities.

Figure 4.12 shows a trace of an experiment in which we set the conferencing system goal to autonomically provide suitable quality of service on interactive communication according to the environment changes. The plot shows both the (emulated) packet loss rate on the network, as well as the packet loss rate observed by the system after FEC error control is applied. This particular trace represents the following scenario: a conference participant with a laptop computer remains close to the wireless access point for approximately 30 seconds, then begins walking, arriving at location of high packet loss (approximately 30%) at time 60. He remains there for approximately 60 seconds, then walks to a location of extremely high packet loss (50%) remains there for another 60 seconds, and finally returns to a location of relatively low packet loss (10%).

At the beginning of the experiment, and as the user begins walking, the network packet loss rate and the application packet loss rate are identical, since FEC is not applied to the data stream, which comprises both audio and video. When the client system enters the area of high loss rate at time 60, the platform loss rate sensor detects the increase in network packet loss rate (approximately 30%) and notifies the Adaptation Enforcement Service of

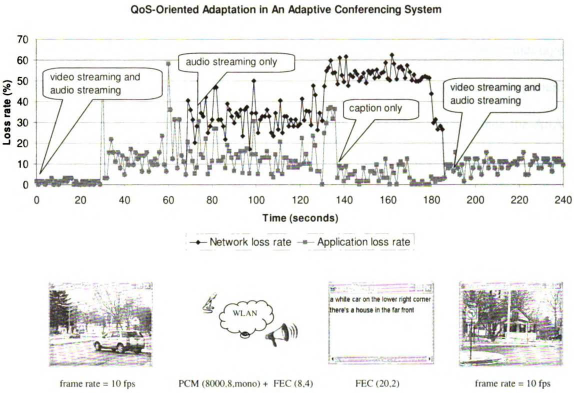


Figure 4.12: Trace of a COCA-based adaptive multimedia conferencing system.

a *high_loss_rate_alert* event. Once the Adaptation Enforcement Service receives this event, it consults the Jess rule base shown in Figure 4.11 (right) and selects the highest priority adaptation rule whose conditions are satisfied. This results in pausing of the Vplayer application (highest priority) and switching to audio-only mode using ASA. Shortly thereafter, since the loss rate is still high (30%), the loss rate detector of the ASA application fires another *high_loss_rate_alert* event. Since the possible adaptation may involve interactions between ASA and other components, this event is propagated to the Adaptation Enforcement Service via the *MsgGateway* communication interface of the ASA application, shown in Figure 4.11 (left). Since inserting FEC facility can reduce the loss rate in this case study, the Adaptation Enforcement Service first checks whether the pre-condition (*ifFECSynchronized*) is satisfied or not. If so, it sends a *insertFEC* reconfiguration message to the ASA application through the Messaging Service. Before the reconfiguration message is sent to the target component, the physical topology information of the target component is retrieved from the Naming Service and embedded in the message body. Once the ASA application receives the message, it inserts the FEC filter into the MetaSocket used to transmit the stream.

So far, the scenario of a complete message propagation and adaptation policy enforcement cycle was demonstrated. However, since the FEC facility will increase the bandwidth usage, perhaps beyond that available, there is a constraint in the selection of FEC parameters. Therefore, when the network loss rate is too high, limited FEC capacity cannot meet the QoS requirement. For example, when the client system enters an extremely high packet loss (50%) location, the FEC (8,4) coding for audio stream is not sufficient to correct the error, but stronger FEC coding will consume much more bandwidth and introduce jitter. Therefore, at time 130, the Adaptation Enforcement Service decides to switch conferencing system from the ASA application to the Echo application, which consumes much lower bandwidth consumption and thus imposes less constraint in the selection of FEC parameters. When the client system returns to a location with good network conditions, the Adaptation Enforcement Service is notified of a *low_loss_rate_alert* event, which results in resuming the Vplayer and ASA components.

4.8 Conclusions

In this chapter, we introduce COCA, a collaborative adaptation infrastructure for mobile computing systems. We describe how to compose a COCA specification document, which describes both architectural information as well as adaptation logic. By translating these parts, respectively, into communication interface glue code and rules to guide adaptive behavior, COCA enables the construction of adaptive mobile systems from non-adaptive legacy components. We apply COCA to a composite multimedia conferencing system, enabling it to adapt to changing network conditions in multiple, coordinated ways.

The methods used in COCA are general and can be extended to other distributed computing models that require collaborative adaptation. In this chapter, we introduce the use of COCA to realize an adaptive mobile system from a collection of legacy applications. In the next chapter, we will demonstrate how COCA cooperate with other distributed computing frameworks to construct autonomic communication services and support collaborative adaptation in a fully distributed computing environment. We will also introduce how to generalize the COCA specification and facilitate the process of specifying and managing this type of specification.

Chapter 5

ORCHESTRATING DISTRIBUTED AUTONOMIC COMMUNICATION SERVICES

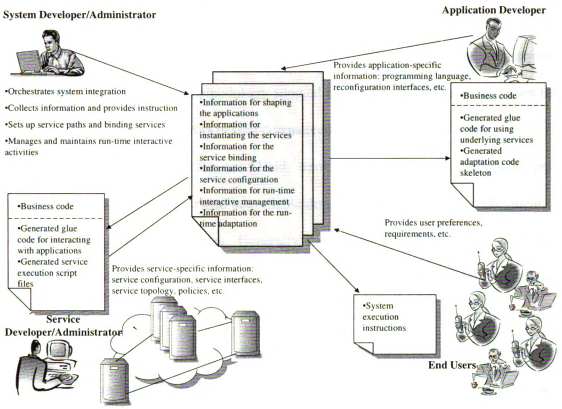
Autonomic computing refers to self-managed systems that require only high-level human guidance. The evolution of autonomic computing systems will be a long-term process [165]. When the system self-management capability improves, the interaction between humans and systems will progressively decrease. As automation technologies mature and humans gain more confidence in them, autonomic systems can independently make low-level decisions and take appropriate actions. Currently, however, human interaction with such systems is still necessary. As we have seen, the implementation of autonomic system functionality often relies on collaboration among individual components. In the previous chapter, we demonstrated the use of COCA to integrate several legacy components into an adaptive system. In this chapter, we extend expressive orchestration to distributed, service-oriented architectures, facilitating the development of autonomic computing systems based on external services.

5.1 Introduction

Future autonomic systems are likely to be large-scale information systems implemented atop heterogeneous hardware platforms, operating systems, programming languages, and networking protocols. The architecture of an autonomic system needs to address this heterogeneity in the run-time environment, as well as the interoperability among components. One of the great challenges in building autonomic systems is to manage their components correctly and effectively [180]. Techniques such as COCA are intended to aid this process by providing developers with the tools needed to realize collaborative adaptation among *software components*.

However, building autonomic systems will also be a collaborative effort between different *developers* and *organizations*. Thus the development process should involve different roles and separate concerns. Different interested parties may require different views of the system architecture, configuration, and run-time management. Specifically, application developers focus on implementing the business logic of individual components, adaptation developers put effort into making individual components autonomic, and system developers work on integrating individual components into an autonomic system. When autonomic systems are constructed as collections of interacting components (or services), it is necessary that the behaviors of individual components and the interaction among these components be specified precisely for later integration, configuration, and run-time management. Moreover, developing large scale service-oriented autonomic systems will require a means to enable developers in different organizations to specify and realize these interactions. Thus, application developers should have the means to specify the behaviors of individual components and deliver the corresponding specifications to adaptation developers. The same routine should also be applied between adaptation developers and system developers. Furthermore, from the view of integration, the life cycle of these interactions should start from the design time of individual components and continue through the evolution of the entire system. Thus, a comprehensive specification model is needed in business code and "glue code" development, system integration, run-time management, and system evolution.

Figure 5.1 depicts a scenario in which different parties involved in an autonomic system development need to collaborate. Given a collection of applications and autonomic services needed to build an autonomic system, the system developer/administrator has the most complete knowledge about each part of the system and acts as the coordinator who is responsible for orchestrating system integration, providing instructions to other parties for shaping applications, setting up service paths, and binding services. To oversee these tasks, the system developer/administrator needs inputs from the application developer, the service developer/administrator, and the end users to specify the high-level system compositions and map these specifications onto low-level behaviors of each component. The collected information includes application-specific information (e.g., programming language, reconfiguration interfaces, etc.), the service-specific information (e.g., service configuration, service interfaces, service topology, policies, etc.), and the user preferences and requirements. Unfortunately, in most cases, different parties involved in building an autonomic system lack a unified platform and infrastructure to enable such collaboration. Such a unified platform and infrastructure would facilitate rapid prototyping, system development, run-time management, and maintenance. The lack of such a platform will certainly hinder the development of autonomic systems.



System Developer/Administrator
•Orchestrates system integration
•Collects information and provides instruction
•Sets up service paths and binding services
•Manages and maintains run-time interactive activities
Application Developer
Provides application-specific information: programming language, reconfiguration interfaces, etc.
•Information for shaping the applications
•Information for instantiating the services
•Information for the service binding
•Information for the service configuration
•Information for run-time interactive management
•Information for the run-time adaptation
•Business code
•Generated glue code for using underlying services
•Generated adaptation code skeleton
•Business code
•Generated glue code for interacting with applications
•Generated service execution script files
Provides service-specific information: service configuration, service interfaces, service topology, policies, etc.
Service Developer/Administrator
Provides user preferences, requirements, etc.
•System execution instructions
End Users

Figure 5.1: Interactions among different parties involved in the autonomic system development.

To address these needs, we propose ASSL (**A**utonomic **S**ervice **S**pecification **L**anguage), an XML-based technique for specifying distributed, service-oriented autonomic systems, focusing on integration, configuration, and run-time interaction management. ASSL extends COCA to provide a unified platform to support the development and execution of distributed autonomic systems developed by multiple organizations. The resulting composite specifications not only facilitate system integration, but also can be used in the later control of the system workflow. The possible outputs of the scenario depicted in Figure 5.1 may include the generated "glue code" for the applications to use the underlying services, the generated adaptation code skeleton for the application developer to introduce adaptive functionalities to the legacy non-adaptive applications, the generated "glue code" for the services to interact with applications, the generated service execution script files, and the system execution instructions for the end users. Given that the unnecessary details are abstracted away, the methodology adopted by ASSL can substantially lessen the number of upstream system errors passed downstream at the source code level.

This work is a part of the Service Clouds framework [181], which attempts to provide a framework that enables rapid but reliable design, development, and deployment of service-oriented autonomic systems. Generally, the Service Clouds framework includes: (1) The Service Clouds infrastructure, which provides the necessary service components in a service-oriented autonomic computing architecture, and provides the concrete service implementation mechanisms. (2) The Service Graphs model provides a means to abstract autonomic systems based on layered graphs, which represent the connectivity of distributed entities using multiple layers of abstraction. (3) The ASSL specification and development toolkit. ASSL attempts to provide an abstract and succinct means of capturing and expressing the logic behind collaborative processes related to the development and management of autonomic systems. ASSL intends to serve two objectives: to serve as a source, based on the Service Clouds infrastructure, for constructing a running system and setting up an execution environment; and to serve as a target for the high level Service Graphs model. In a case study, which is conducted using the Service Clouds infrastructure and executed on the PlanetLab distributed computing testbed, we demonstrate the utility of ASSL in the composition, deployment, configuration, and management of distributed autonomic communication services.

The remainder of this chapter is organized as follows. We briefly introduce the background for this work in Section 5.2. To help illustrate various aspects of ASSL, we introduce an autonomic video streaming service in Section 5.3 as an example application. Section 5.4 discusses the detail use of ASSL for service specifications, service binding, and interactive management. Section 5.5 presents the empirical results obtained from building a video streaming service with ASSL. Conclusions are given in Section 5.6.

5.2 Background and Related Work

Since this work is conducted in the autonomic computing domain, we firstly introduce the background of autonomic computing in Section 5.2.1 and the service-oriented architecture to support autonomic computing in Section 5.2.2. As we have stated that this work is a part of the Service Clouds framework, in Section 5.2.3 we describe the architecture design of the Service Clouds infrastructure. Similar to the existing Architecture Description Languages (ADLs), ASSL can be used to specify the interaction between the compositional software components of an autonomic system in the solution space. Moreover, ASSL can be also used to specify the extra-functional properties that go beyond the structural behavior of the system, which are necessary for system integration, configuration, and run-time interaction management. In Section 5.2.4, we survey the existing ADLs, focusing on their contents and representations.

5.2.1 Autonomic Computing

An architectural approach to building an autonomic computing system has been proposed by White et al. [182]. In order to simplify system management, the implementation of system functionalities in an autonomic computing system relies mostly on the collaboration among the individual *autonomic elements*. Thus an autonomic element itself must also be self-sustaining, which means that it must handle, as far as possible, all problems locally. The relationships among autonomic elements are based upon agreements, in which an element can describe its service to other elements. To validate the agreement, an autonomic element must not only understand and abide by the terms of its existing agreements, but also be capable of negotiating new agreements. The autonomic elements can collaborate to implement autonomic computing functionality by establishing and maintaining relationships, providing services, and receiving directives.

In addition, the autonomic elements must also implement additional interfaces to achieve interoperability in the system. These interfaces include: *monitoring and testing interfaces*, which expose the run-time status of an element to any other element interested in it; *life cycle interfaces*, which enable administrative elements to determine and change the life cycle state of an element; *policy interfaces*, which can be used to determine the current policies of an element and send new policies to the element; and *negotiation and binding interfaces*, which allow elements to establish relationships by sending or receiving requests for services between each other.

Finally, an autonomic system requires *infrastructure elements* that support the operation of the autonomic system as a whole. For example, a *registry* helps elements to broadcast their services and establish relationships with other elements. A *sentinel* provides monitoring services to other elements. An *aggregator* combines two or more elements to provide improved services to other elements. A *broker* facilitates elements to express their demands and locate the required services. A *negotiator* assists elements with complex negotiations to establish stable relationships and solve conflicts. All these infrastructure elements facilitate the interactions among autonomic elements. Together, the autonomic elements and the infrastructure elements cooperate to implement the system functionalities.

In general, White et al. argue that to build an autonomic system, we should be able to map desired system-wide behaviors to a set of behavioral actions and interaction rules embedded within the individual elements. This mapping process is not a simple collection of local behaviors from individual elements; rather, it should be a mixture result from effective negotiations among the autonomic elements.

5.2.2 Service-Oriented Architecture for Autonomic Computing

Since autonomic systems will be interactive collections of autonomic elements and infrastructure elements, a distributed, service-oriented architecture needs to accurately support the interaction among elements. The OASIS SOA Reference Model group [183] defines Service Oriented Architecture (SOA) as "Service Oriented Architecture is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. It provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations." SOA allows individual elements to hide their implementation details and expose a consistent interface for communication, thus the interaction and collaboration can be achieved through negotiation between service providers and service requesters.

The service-oriented architecture is hierarchical. An autonomic element will typically consist of one or more managed elements coupled with a single *autonomic agent*. An autonomic agent is an aggregator that controls and represents the coupled autonomic elements. At the highest level, the managed element could be a hardware resource, an application service, or even an individual business.

There are two types of autonomic agents: *business agent* and *infrastructure agent*. The business agent is responsible for capturing and encapsulating the business logic of a collection of autonomic elements to implement one system functionality. The infrastructure agent is responsible for integrating those infrastructure elements which provide system-wide autonomic related functionalities, such as registry, monitoring, negotiation, and decision making. Since the functionalities provided by these infrastructure elements cross-cut the needs of other autonomic elements, it would relieve application developers' workload if the services from these infrastructure elements were available for use and they did not need to re-invent and re-develop these utilities. Moreover, it would be helpful if developers could leverage existing techniques and products provided by other vendors. With the integrated management of autonomic agents, the managed elements could reside in the same host, or be fully distributed across the Internet.

The emergence of Web services helps to address many of these integration problems, due to its loose-coupling nature and wide support among software vendors. Web services describe a standardized way of integrating applications using the XML [184], SOAP [185], WSDL [186], and UDDI [187] open standards over the Internet. Here, XML is used to tag the data; SOAP is used to transfer the data between a service provider and a service requester; WSDL is an XML document for describing the available services; and UDDI is used for publishing and locating the available services. Because all communication is in XML, Web services technology is independent of platform, operating system, programming language, middleware, and networking protocol. Web services allow applications to be integrated through message interchange, so the system developers only need to focus on service semantics description and organization without worrying about the intimate knowledge underlying each application. Furthermore, we can use Transparent Shaping tools to expose legacy applications to Web services. All these techniques facilitate the integration and the development of autonomic systems.

5.2.3 Service Clouds Infrastructure

Our study of ASSL is conducted with the support of a service-oriented infrastructure, Service Clouds infrastructure [181], which provides autonomic communication services to mobile devices. Figure 5.2 shows a conceptual view of the Service Clouds infrastructure. This infrastructure creates and terminates distributed overlay services at run time through a collection of hosts. In this view, deep service clouds comprise hosts on an Internet overlay network (such as wired nodes on the PlanetLab testbed), and mobile service clouds comprise hosts close to the wireless edge. The federation of clouds cooperates to provide autonomic communication services.

The Service Clouds infrastructure enables dynamic composition, instantiation, and reconfiguration of services on an overlay network. For example, when a mobile user uses his

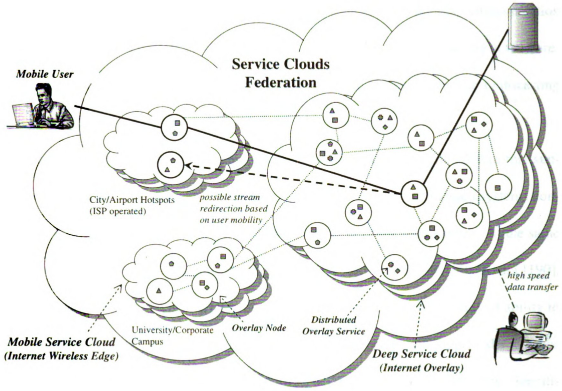


Figure 5.2: Conceptual view of the Service Clouds infrastructure.

PDA to view a video stream from a video server, he may move from one hotspot to another hotspot, and his IP address may change from time to time. In such a scenario, how can the communication infrastructure ensure that the mobile user can receive a continuous video stream? The solution provided by Service Clouds is to deploy autonomic communication services at the edge of the wired Internet, in support of wireless devices. These autonomic communication services are self-managed, and they can monitor user behaviors and adjust the services and resources dynamically to provide the best services to mobile users. The idea behind the implementation is similar to the concept of DNS and DHCP services in the traditional Internet, but focuses on QoS support of wireless communication. The Service Clouds infrastructure is primarily intended to facilitate rapid prototyping and deployment of autonomic communication services. Examples of such services include communication path resiliency, improvement of TCP throughput, and fault-tolerance streaming at the wireless edge [181], as well the example presented in Section 5.3 on supporting multicasting and user mobility.

The Service Clouds infrastructure is a typical service-oriented infrastructure that supports autonomic computing. As the complexity and scale of service-oriented systems continue to grow, they become increasingly difficult to administrate and manage. At the same time, service deployment technologies (e.g., Nixes [188], SmartFrog [189], Radia [190], etc.) are still based on the low-level scripts and configuration files with minimal ability to express dependencies, document configurations, and verify setups. ASSL provides a means to describe more complex autonomic behavior in service-oriented environments. Specifically, ASSL can be used to specify and support the interactions among the compositional components and the interested parties in three levels: component-component interaction in the application level, component-service interaction in the system level, and service-service interaction in the service level.

5.2.4 Architecture Description Languages

Architecture description languages (ADLs) represent a language-based design methodology, which are used to define and model system architecture prior to system implementation. According to Vestal [191], an ADL for software applications "focuses on the high-level structure of the overall application rather than the implementation details of any specific source module." There are several ADLs, such as Acme [192], Rapide [193], MetaH [194], C2 [195], xADL [104], Darwin [103], and Wright [102]. In general these ADLs differ from requirement languages because they are rooted in the solution space, whereas requirements describe problem spaces. Moreover, ADLs also differ from programming languages because ADLs do not bind architectural abstractions to specific solutions.

Medvidovic and Taylor [90] summarized that the essential building blocks of an ADL include *components*, *connectors*, and *architectural configurations*. Components are units of computation or data stores while connectors are architectural building blocks used to model interactions among components and rules that govern those interactions. Architectural configurations that are also known as topologies are connected graphs of components and connectors that describe architectural structure. Addressing the *structural properties* of a composite system, most existing ADLs provide computational models of constructing such a system and deal with the ways components interact. Thus in principle ADLs concentrate on the functional behavior, and can be used to specify how to compose systems from smaller parts so that the interactive result meets system requirements. In addition to the structural properties, Shaw and Garlan [196] indicated that other *extra-functional properties*, such as performance, reliability, security, capacity, environmental assumption, and so on are also (or even more) important. Unfortunately, however, existing ADLs have not been applied to address these aspects. Moreover, challenges remain in finding formal systems to handle and reason about these properties that go beyond the structural behavior of the system.

Like other requirements languages and modeling languages, to aid understanding and communication about a software system among different interested parties is one of key roles of an explicit representation of an architecture [90]. Many ADLs provide formal syntax and semantics, powerful analysis, model checkers, and so on. The formal notations for ADLs are useful both for system construction as well as verification support. On the other hand, it is also important that architectural descriptions be simple and understandable, with well understood, but not necessarily formally defined, semantics. The notations for ADLs fall into the following categories [197]: graph-based approaches, process algebra approaches, logic-based approaches, code-oriented approaches. Graphs are natural approaches to represent the software architecture and the relationship between the compositional components. Example graph-based approaches include Multiset [198], Hypergraph [199], Distributed [200], COMMUNITY [201], and CHAM [202]. To study concurrent systems, process algebras are commonly used by specifying and verifying concurrent systems with algebras and calculi. Commonly used process algebras include the Calculus of Communicating Systems (CCS), Communicating Sequential Processes (CSP), and the π-calculus. Dynamic Wright [102], Darwin [103], LEDA [203], PiLar [204] are typical ADLs that adopt process algebra approaches. First-order logic and temporal logic are also used as a formal basis for software architecture specification, especially for those dynamic software architectures. Example logic-based approaches include Aguirre-Maibaum [205] and ZCL [206]. There are some approaches exist that do not have a formal semantics based on graph theory, process algebra, or logic; however, these code-oriented approaches often provide code synthesis tools to support component-based development by utilisers architecture definitions as the development framework. For example, Rapide's [193] compiler generates executable simulations of Rapid architectures. xADL [104] (formerly C2 [195]), on the other hand, provides a tool (*Apigen*) that generates implementation API from an architecture model, providing completion guidelines for developers. From the point of view of rapid prototyping, the code-oriented approaches are more suitable for facilitating the system integration and development.

5.3 A Running Example

To evaluate our approach to specifying the interactive behaviors between service providers and service clients, we have conducted an experimental study to demonstrate the autonomic communication services specification, binding, and interaction. We will also use this example application to help describe ASSL. In this example, mobile client nodes receive a multimedia stream (e.g., in an interactive video conference or in a live video broadcast) from a video server. In this case, the infrastructure fulfills the following requirements. First, stream delivery should not be interrupted when a user relocates and connects to a new network domain, gaining a new IP address. Second, the quality of the received stream must remain acceptable as a wireless link experiences packet loss.

Figure 5.3 shows the configuration used in the experiments: three PlanetLab nodes in a deep service cloud, two workstations in a mobile service cloud on the Michigan State University intranet, and two Windows laptops to request a video stream from a video server on the Internet. Subnet *A* is a wired LAN and subnet *B* is wireless. The middleware software (*SC Enabler*) on a client connects to a Service Gateway node (*N1*) and requests the desired service. Gateway nodes are the entry points to the Service Clouds infrastructure. They accept requests for connection to the Service Clouds infrastructure and designate a primary proxy to coordinate the requested service. Upon receiving the request, the gateway identifies a node to act as the primary proxy (*N4*), and informs the client of the selection.

The primary proxy receives detailed requests of the desired service, sets up a service path, and coordinates monitoring and automatic reconfiguration of the service path during the communication.

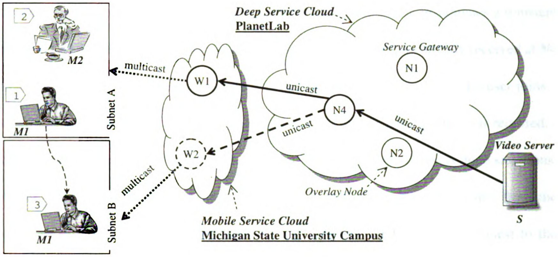


Figure 5.3: The experimental testbed and example scenario.

Besides the primary proxy in subnet *A*, there are several transient proxies in subnet *B*. In this example, the transient proxy deploys two functionalities: multicasting and forward error correction (FEC). Since multicasting is not readily available on the Internet (deep service cloud), the stream is unicasted toward the wireless edge, where the transient proxy multicasts it towards the wireless clients. Moreover, to maintain the quality of the video stream (especially since unlike unicast UDP packets, there is no MAC-layer retransmission for multicast packets on wireless link), the transient proxy applies FEC on the stream when a wireless client detects high packet loss.

In addition to providing multicast and QoS streaming, the mobile service cloud supports continuous streaming by the dynamic instantiation of the transient proxies, while users roam among different subnets. For example, when user *M1* moves from subnet *A* to subnet

B, the *SC Enabler* on the client detects the change of IP address and notifies the primary proxy (*N4*). The primary proxy checks the current service path and notices that the video is not being streamed in subnet *B*. Thus, it extends the service path by constructing a path that delivers the stream to the new subnet *B*. This service path extension instantiates a transient proxy (on *W2*) for the new domain and unicasts a copy of the stream already received at *N4* towards *W2*, where the proxy multicasts it in subnet *B*. On the other hand, if a user joins a subnet where the stream is already being multicast, no service path extension is required.

The running example scenario is depicted in Figure 5.3, which depicts three situations. At the beginning, user *M1* on the wired subnet A requests to receive a video from the video server. Accordingly, the *SC Enabler* on the client sends a service request to the gateway node *N1*, which chooses *N4* as the primary proxy and informs the client. Thus, the client software sends a primary proxy service request to *N4*, which constructs a service path comprised of UDP relays on itself and a unicast-to-multicast proxy on *W1*. Next, another user *M2* requests the same video on the wired subnet *A*. Since the video is already being multicast to the subnet *A*, the Service Clouds infrastructure simply assigns the same primary proxy to *M2* and configures it to receive the same video as user *M1*. Finally, user *M1* walks away and switches from the wired connection on the subnet *A* to the wireless connection on the subnet *B*. The *SC Enabler* detects this roaming between the subnets because the IP address of the laptop changes when it joins a new subnet. At this point, a dynamic service path extension, as explained earlier, makes the stream available in the subnet *B* via a proxy on *W2*. Moreover, since the connection to *W2* is wireless, if packet loss rate becomes intolerable, the transient proxy provides the FEC service to compensate for the packet loss.

5.4 Autonomic Service Specification Language

5.4.1 Introduction

When we incorporate a software component into a service-oriented infrastructure (e.g., Service Clouds infrastructure) to construct an autonomic system, we need to support the interactions between the application and the underlying autonomic services. The interactions may occur at system composition time, service deployment and configuration time, and system run time. Furthermore, the interactions may be extended from software components to humans, including system developers, system administrators, and end users. In our running example, an existing video streamer application, which is unaware of the presence of the Service Clouds infrastructure in the design and development time, needs to utilize the Service Clouds to provide the robust video streaming. To complete such an integration and configuration process, the application developer needs to know what kind of software modifications or configurations are necessary in order to use the underlying autonomic services, and how the system can conduct further interactions with services during the run time. On the other hand, the system administrator needs to know the concrete platform-specific and application-specific information to complete the services deployment, configuration, and binding. Furthermore, the system administrator may also need to know the user preferences regarding QoS and other system requirements (e.g., security policy) in order to manage the run-time adaptation.

Generally, to support interactive activities in such a service-oriented architecture, we need to:

- Modify the application to use the underlying services if it was not implemented specifically atop the underlying service infrastructure. To do so, the system developer could modify the application manually, but a better approach is to transparently shape the application with respect to the existing business code. The mechanism(s) used on a particular application depend on the characteristics of the application, including the programming language and any middleware platform used.

- Configure and bind services. Although a well-designed service should be general to any application, the system administrator still needs means to customize application-specific configurations in order to set up and maintain services. Only after the service binding is established, can the application start using the underlying services.

ASSL is intended to facilitate the rapid deployment and configuration of such services. The ASSL specification contents cover a range that include service capabilities and responsibilities, application requirements, and user preferences. ASSL is a highly-extensible XML-based language, and its main strength is extensibility; it can act as the basis for composing the domain/project-specific interaction specification. In our current design, ASSL is a collection of XML schemas, which are used to specify various aspects of interaction processes in an autonomic system. The core schemas can be extended to add new features or increase its expressiveness. Details of ASSL core schemas and extension schemas are presented in Section 5.4.2 and Section 5.4.3, respectively.

Figure 5.4 provides a conceptual view of the use of ASSL. The interactive process of construction, configuration, and management of an autonomic system centers around a Service Specification Document (SSD), which is an XML instance written in ASSL. An SSD contains three sections: information section, binding section, and interaction section. The information section lists physical host-specific and application-specific information. This information can be used to describe the system architecture and generate "glue code" for incorporating applications to communicate with a service-oriented infrastructure *(transparent shaping)*. In the binding section, the service side and the application side exchanges information about service composition and sets up the service path *(bootstrapping)*. The interaction section contains the application requirements and user preferences according to the parameterized services resource for the run-time system reconfiguration *(adaptation)*. All this information can be updated periodically or updated according to the change of the run-time conditions, for example, the change of the IP address. With the reusability and extensibility provided by XML, users can easily customize their SSDs by leveraging existing schemas as well as introducing new notations.

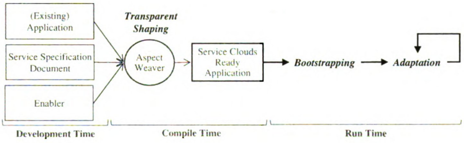


Figure 5.4: Conceptual view of the the use of ASSL.

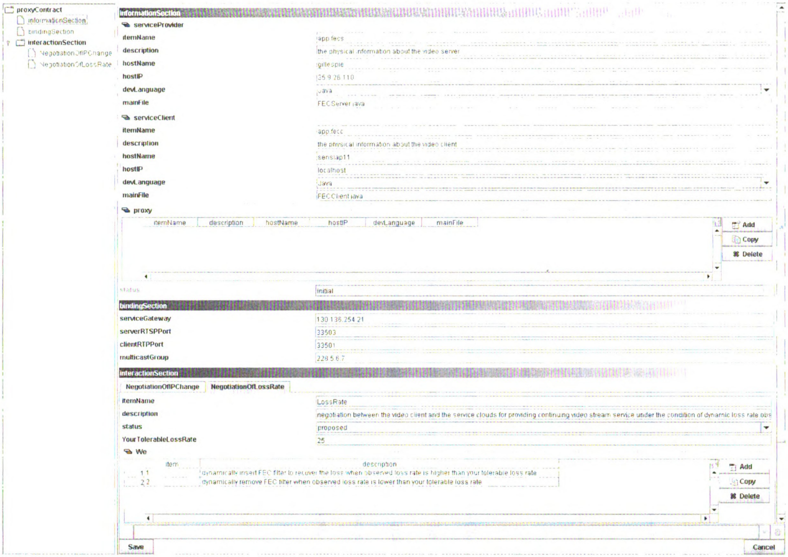
Different from other XML-based specification techniques, ASSL uses XUI techniques [207] to analyze the XML schemas and generate a Java graphical user interface, called the *SSD console*, visualizing the SSD at run time as shown in Figure 5.5. To generate an SSD console, a valid XML schema is required. Based on an XML schema, an SSD console can be generated automatically, without writing one line of code. The generated

SSD console enables a sophisticated way of editing the underlying XML instance or creating a new one. All modifications done with the SSD console will be validated on-the-fly against the XML schema immediately, greatly reducing the possible errors in composing XML instances. Only valid changes will affect the underlying XML-based SSD. By the dynamic generation of the graphic user interface on the basis of an XML schema, a noticeable shortening of the development cycles and a loose coupling between the SSD and the actual application development can be reached. This has the consequence that any changes in the SSD instance schema are reconstructed directly by the presentation logic. Thus, the developer only needs to focus on the SSD specification syntax, greatly increasing the expressiveness of SSD and the convenience of SSD-based interaction.

Furthermore, the Service Clouds infrastructure can provide a Web services function unit to maintain and manage the SSD-based interactions. Before an application can begin to use the underlying services, the system administrator or the user first needs to use the SSD console to complete the SSD and generate the "glue code" for shaping the application. The SSD console can also be used as the front-end interaction platform for services deployment, configuration, and binding. At run time, the user can use the SSD console to monitor the system execution, receive event notification, and conduct further system management.

5.4.2 ASSL Core Schemas

The core of ASSL is the SSD instance schemas, which can be extended to add new features or increase its expressiveness. The SSD instance schema is shown in Figure 5.6. In our current design, we define only three basic sections (*informationSection*, *bindingSection*,



informationSection

serviceProvider

itemName	app.fecs
description	the physical information about the video server
hostName	gillespie
hostIP	35.9.26.110
devLanguage	java
mainFile	FECServer.java

serviceClient

itemName	app.fecc
description	the physical information about the video client
hostName	senslap11
hostIP	localhost
devLanguage	java
mainFile	FECClient.java

proxy

itemName	description	hostName	hostIP	devLanguage	mainFile

status	initial

bindingSection

serviceGateway	130.136.254.21
serverRTSPPort	33503
clientRTPPort	33501
multicastGroup	228.5.6.7

interactionSection

NegotiationOfIPChange | NegotiationOfLossRate

itemName	LossRate
description	negotiation between the video client and the service clouds for providing continuing video stream service under the condition of dynamic loss rate obs
status	proposed
YourTolerableLossRate	25

We

item	description
1.1	dynamically insert FEC filter to recover the loss when observed loss rate is higher than your tolerable loss rate
2.2	dynamically remove FEC filter when observed loss rate is lower than your tolerable loss rate

Figure 5.5: An example of an SSD shown in the SSD console.

and *interactionSection*), which are sufficient for the proof of concept. To add more sections and create a new SSD instance schema, one can simply extend the XML complex type of *SSDInstance*.

```
<xsd:complexType name="SSDInstance">
  <xsd:sequence>
    <xsd:element name="informationSection" type="SSDInfoSectionType"></xsd:element>
    <xsd:element name="bindingSection" type="SSDBindingType"></xsd:element>
    <xsd:element name="interactionSection" type="SSDInteractionSectionType">
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
```

Figure 5.6: The SSD instance schema.

The basic elements in the information section are *serviceProvider* and *serviceClient*, as shown in Figure 5.7, which are also the basic roles in a service-oriented system. The information section lists physical host-specific and application-specific information, including IP address, screen size and color depth, programming language, character set, and operating system. This information is defined by the XML complex type of *HostInfoType*, which is introduced in Section 5.4.3.

```
<xsd:complexType name="SSDInfoSectionType">
  <xsd:sequence>
    <xsd:element name="serviceProvider" type="HostInfoType"></xsd:element>
    <xsd:element name="serviceClient" type="HostInfoType"></xsd:element>
  </xsd:sequence>
</xsd:complexType>
```

Figure 5.7: The information section schema.

One can introduce more elements (other service participants) by extending the XML complex type of *SSDInfoSectionType*. For example, if we need to specify a system with a type of component *serviceProxy* in addition to the *serviceProvider* and *serviceClient*, we can define a new XML complex type of *SSDInfoSectionExtType* by extending the XML complex type of *SSDInfoSectionType* as shown in Figure 5.8. The new type of *SSDInfoSectionExtType* consists of three elements: the *serviceProvider* and *serviceClient* defined in *SSDInfoSectionType*, and *serviceProxy* defined in *SSDInfoSectionExtType*.

```
<xsd:complexType name="SSDInfoSectionExtType">
  <xsd:complexContent>
    <xsd:extension base="cop:SSDInfoSectionType">
      <xsd:sequence>
        <xsd:element name="serviceProxy" type="cop:HostInfoType"
          maxOccurs="unbounded" minOccurs="0">
        </xsd:element>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Figure 5.8: An example of extending the information section schema.

In the binding section, the service side and the application side exchanges information about service composition and setting up the service path. The key elements for service binding in the Service Clouds infrastructure include *serviceGateway* and *primaryProxy*. More elements can be added by extending the XML complex type of *SSDBindingType*. The binding section schema is shown as Figure 5.9.

```
<xsd:complexType name="SSDBindingType">
  <xsd:sequence>
    <xsd:element name="serviceGateway" type="xsd:string"></xsd:element>
    <xsd:element name="primaryProxy" type="xsd:string"></xsd:element>
  </xsd:sequence>
</xsd:complexType>
```

Figure 5.9: The binding section schema.

The interaction section contains the application requirements and user preferences according to the parameterized services resource and run-time conditions (e.g., throughput, hand-off, packet loss, delay, data rate, etc.). This information can be used for the decision of service composition and run-time adaptation. The basic element in the interaction section is *InteractionItem*, which usually can be used to describe the adaptation policies. In each policy, we define the responsibilities of the participating parties who are involved in the adaptation processes and the interfaces through which they carry out the adaptation actions. Specifically, the responsibility specification is composed by extending the XML complex type of *InteractionItemType.* The basic information included in *InteractionItemType* could be text guides for other developers and end users, for example, the description of the necessary actions that should be taken by the end user in response to the adaptation request. The information included in *InteractionItemType* could also be other communication types, for example, the COCA messages we introduced in Chapter 4. For example, one can re-define or extend *InteractionItemType* by using *EventItemType* and *MessageItemType.* To customize other interaction policies, one can extend the XML complex type of *SSDInteractionSectionType* or completely override it. The interaction section schema is shown as Figure 5.10

5.4.3 ASSL Extension Schemas

Besides the ASSL core schemas, we have also developed some extensions to the ASSL core schemas. Our current set of extensions is hierarchical complex types, which complements the expressiveness of those more fundamental ones. These extensions are summarized below.

```
<xsd:complexType name="SSDInteractionSectionType">
  <xsd:sequence>
    <xsd:element name="InteractionItem" type="InteractionItemType"
      maxOccurs="unbounded" minOccurs="0">
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="InteractionItemType">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"></xsd:element>
    <xsd:element name="description" type="xsd:string"></xsd:element>
    <xsd:element name="status" type="InteractionStatusT"></xsd:element>
    <xsd:element name="providerReaction" type="SSDInteractionItemDescriptionType"
      maxOccurs="unbounded" minOccurs="0"></xsd:element>
    <xsd:element name="clientReaction" type="SSDInteractionItemDescriptionType"
      maxOccurs="unbounded" minOccurs="0"></xsd:element>
    <xsd:element name="interactionInterface" type="xsd:string" maxOccurs="unbounded"
      minOccurs="0"></xsd:element>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="SSDInteractionItemDescriptionType">
  <xsd:sequence>
    <xsd:element name="item" type="xsd:positiveInteger"></xsd:element>
    <xsd:element name="description" type="xsd:string"></xsd:element>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="EventItemType">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"></xsd:element>
    <xsd:element name="description" type="xsd:string"></xsd:element>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="MessageItemType">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"></xsd:element>
    <xsd:element name="description" type="xsd:string"></xsd:element>
    <xsd:element name="source" type="xsd:string"></xsd:element>
    <xsd:element name="target" type="xsd:string"></xsd:element>
    <xsd:element name="message" type="xsd:string"></xsd:element>
  </xsd:sequence>
</xsd:complexType>

<xsd:simpleType name="InteractionStatusT">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="proposed"></xsd:enumeration>
    <xsd:enumeration value="accepted"></xsd:enumeration>
    <xsd:enumeration value="denied"></xsd:enumeration>
  </xsd:restriction>
</xsd:simpleType>
```

Figure 5.10: The interaction section schema.

Information related types. These information related XML types can be used as the basic building blocks for specifying physical information of system components as shown in Figure 5.11. Here, we give the examples of *HostInfoType* and *ScreenSizeType*, showing the physical information about the running platform and application. One can easily define other similar types.

```
<xsd:complexType name="HostInfoType">
  <xsd:complexContent>
    <xsd:extension base="InfoItemType">
      <xsd:sequence>
        <xsd:element name="hostName" type="xsd:string"
          minOccurs="0" maxOccurs="1">
        </xsd:element>
        <xsd:element name="hostIP" type="xsd:string"
          minOccurs="1" maxOccurs="1">
        </xsd:element>
        <xsd:element name="devLanguage" type="DevLanguageT"
          minOccurs="0" maxOccurs="1">
        </xsd:element>
        <xsd:element name="mainFile" type="xsd:string"></xsd:element>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="InfoItemType">
  <xsd:sequence>
    <xsd:element name="itemName" type="xsd:string" maxOccurs="1"
      minOccurs="1">
    </xsd:element>
    <xsd:element name="description" type="xsd:string"
      maxOccurs="1" minOccurs="1">
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="ScreenSizeType">
  <xsd:sequence>
    <xsd:element name="screenWidth"
      type="xsd:nonNegativeInteger" minOccurs="1" maxOccurs="1">
    </xsd:element>
    <xsd:element name="screenHeight"
      type="xsd:nonNegativeInteger" minOccurs="1" maxOccurs="1">
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
```

Figure 5.11: The information related types.

```
<xsd:complexType name="LossRateType">
    <xsd:sequence>
      <xsd:element name="minLossRate" type="LossRateT"></xsd:element>
      <xsd:element name="maxLossRate" type="LossRateT"></xsd:element>
      <xsd:element name="curLossRate" type="LossRateT"></xsd:element>
    </xsd:sequence>
</xsd:complexType>

<xsd:simpleType name="LossRateT">
    <xsd:restriction base="xsd:nonNegativeInteger">
      <xsd:minInclusive value="0"></xsd:minInclusive>
      <xsd:maxInclusive value="100"></xsd:maxInclusive>
    </xsd:restriction>
</xsd:simpleType>
```

Figure 5.12: The QoS related types.

QoS related types. QoS-oriented adaptation is a very important fact in autonomic computing systems. These QoS related XML types can be used as the basic building blocks for specifying QoS parameters of user preferences and system run-time conditions. Here, Figure 5.12 is an example of network packet loss rate. More types can be defined for other QoS parameters such as throughput, hand-off, delay, data rate, etc.

Interaction related types. These interaction related XML types as shown in Figure 5.13 can be used as the basic building blocks for specifying adaptation rules and actions regarding to the system requirements or user preferences. Here, we give an example of the interaction section schema showing the adaptation policies regarding the change of IP address (*InteractIPChangeType*) and the network packet loss rate (*InteractLossRateType*). These two adaptation policies are based on the event-action pattern adopted by COCA. Thus they extend the basic *InteractionItemType* by introducing element *event* and element *action*, whose types are *EventItemType* and *MessageItemType*, respectively. Moreover, for the policy regarding the network loss rate, we also need the users to provide their preferences on

```
<xsd:complexType name="InteractIPChangeType">
  <xsd:complexContent>
    <xsd:extension base="cop:InteractionItemType">
      <xsd:sequence>
        <xsd:element name="event" type="cop:EventItemType">
        <xsd:element name="action" type="cop:MessageItemType">
        </xsd:element>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="InteractLossRateType">
  <xsd:complexContent>
    <xsd:extension base="cop:InteractionItemType">
      <xsd:sequence>
        <xsd:element name="YourTolerableLossRate" type="cop:LossRateT">
        <xsd:element name="event" type="cop:EventItemType">
        <xsd:element name="action" type="cop:MessageItemType">
        </xsd:element>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Figure 5.13: The interaction related types.

the tolerable loss rate for decision making purpose. Thus *YourTolerableLossRate* is introduced in *InteractLossRateType*. Other complicated adaptation policies can be specified in the similar way.

5.5 Empirical Results: Autonomic Services Specification, Binding, and Interaction

Having described the use and language basis of ASSL in the service specification, binding, and interaction, in this section, we will show the empirical results of using ASSL to compose SSDs and build an autonomic video streaming service atop the Service Clouds infrastructure. All the activities of service deployment, system configuration, and run-time management are guided under SSDs with the help of the SSD console.

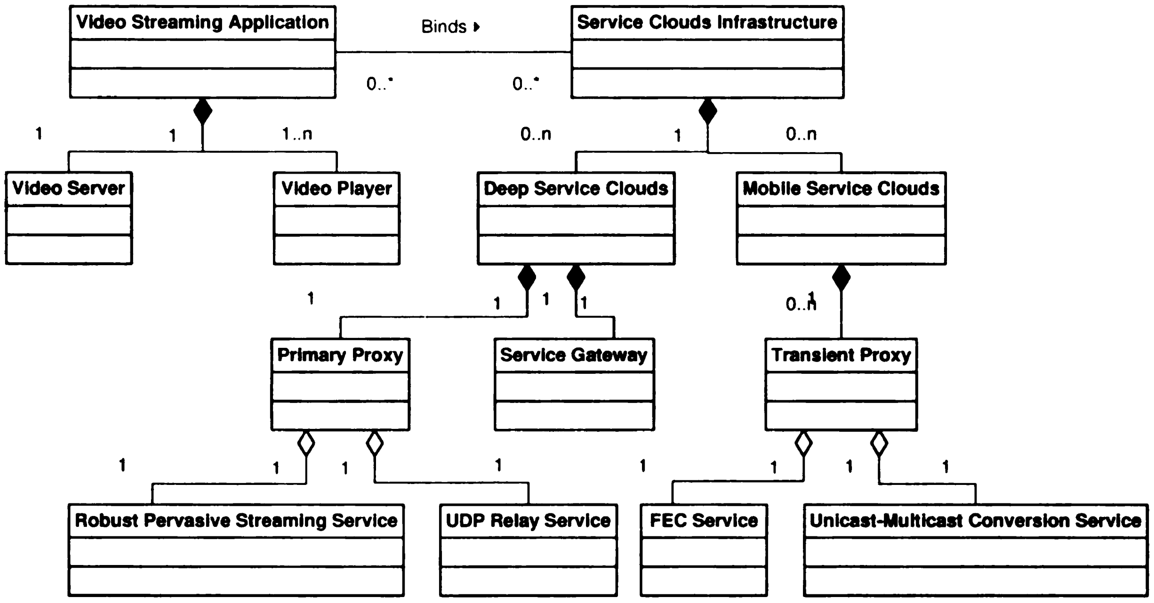


Figure 5.14: Class diagram of the video streaming application and the Service Clouds infrastructure.

In the scenario depicted in Figure 5.3, RTP-based video players (*app.fecc*) of user *M1* (*M1.cse.msu.edu*) and user *M2* (*M2.cse.msu.edu*) request a video stream from the video server *S* (*S.cse.msu.edu*). For test purposes we stream a recorded motion-JPEG 30 fps video (although it could be a live video). The Service Clouds infrastructure comprises deep service clouds and mobile service clouds. The entry point of the Service Clouds infrastructure is the service gateway *G* running on *G.planetlab.org*. There are four autonomic communication services used in this case study: UDP relay service that identifies the fast primary proxy, i.e., *N2* (*N2.planetlab.org*) or *N4* (*N4.planetlab.org*), to be used; robust pervasive streaming service that supports continual video stream; unicast-multicast conversion service and FEC service provided by the transient proxies running on the wireless edge, i.e., *W1* (*W1.cse.msu.edu*) and *W2* (*W2.egr.msu.edu*). Figure 5.14 shows the class diagram of the software components and the underlying services in such an autonomic video streaming system.

To build and maintain an autonomic video communication system from the given collection of aforementioned applications and services, the tasks for the system developer are (1) shaping the applications and the underlying services to enable them to communicate with one another; (2) configuring and deploying the applications and the underlying services according to user requirements and available system resources; (3) providing means to the end user for monitoring and interactively managing the system. In these activities, the system developer coordinates the interactions among application developers, service developers, and end users. Sometimes these roles are carried out by the same person or within the the same organization. However, in many cases, different parties are involved in these activities. This motivates the need for a collaborative environment for the system integration, configuration, and management.

Again, all these activities center around an SSD, and Figure 5.15 illustrates the data flow for processing the SSD, which involves following steps:

Step 1: The system developer uses the ASSL core and extension schemas to compose the SSD instance schema, and publishes it to other interested parties. The SSD instance schema defines that the XML instance document (SSD) must adhere to a particular set of data structural and data constraints, ensuring the syntax correctness of the SSD.

Step 2: Every interested party that obtains the SSD instance schema can automatically generate the SSD console to visualize the SSD instance locally.

Step 3: With the help of the SSD console, the application developer, the service developer, and the end user can retrieve and collaboratively complete the SSD by providing the application-specific information, service-specific information, and user preferences.

Step 4: Similar to the COCA specification document processing we introduced in Section 4.6, the final SSD can be used to generate Aspects code skeletons for shaping applications and services.

Step 5: The application developer and the service developer can then complete the Aspects by mapping the concrete implementation for application/service interfaces with the COCA message processing logic. The Service Clouds-ready code can be produced by using the AspectJ compiler to re-compile the business code for applications and services with the final Aspects.

Step 6 and 7: The final SSD can be also used to generate the service configuration and execution scripts, and the system execution instructions and scripts for the service administrator and the end users, respectively. All these generated codes and scripts will facilitate the system integration, configuration, and run-time interactive management.

5.5.1 Service Specification and Transparent Shaping

Transparently shaping applications. The first step of building such an autonomic system is to shape the applications with COCA. Integrating existing applications into the Service Clouds infrastructure requires that they are Service Clouds ready, that is, able to communicate with Service Clouds services. Thus, we need to augment existing applications so that (1) they can report events of interest to the Service Clouds infrastructure and (2) their local configuration and adaptation mechanisms are accessible to the Service Clouds

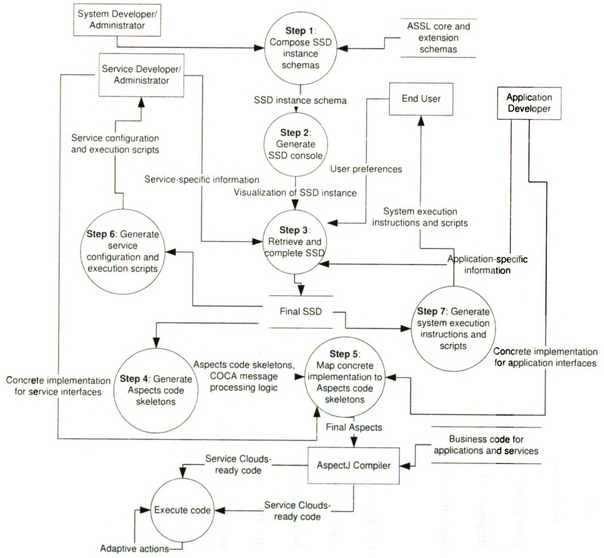

System Developer/ Administrator
Step 1: Compose SSD instance schemas
ASSL core and extension schemas
Service Developer/ Administrator
SSD instance schema
End User
Application Developer
Step 2: Generate SSD console
Service configuration and execution scripts
User preferences
Service-specific information
Visualization of SSD instance
System execution instructions and scripts
Step 3: Retrieve and complete SSD
Step 6: Generate service configuration and execution scripts
Application-specific information
Final SSD
Step 7: Generate system execution instructions and scripts
Step 5: Map concrete implementation to Aspects code skeletons
Aspects code skeletons, COCA message processing logic
Concrete implementation for application interfaces
Concrete implementation for service interfaces
Step 4: Generate Aspects code skeletons
Final Aspects
Business code for applications and services
Service Clouds-ready code
AspectJ Compiler
Execute code
Service Clouds-ready code
Adaptive actions

Figure 5.15: Data flow diagram for processing the SSD.

infrastructure. In our current design, we bridge the existing applications under the COCA framework [208]. COCA provides a means to specify the relationships among different system components, generates code that provides the collaborative adaptations, and governs the system-wide adaptive behavior during execution. COCA also provides a set of reusable adaptation-supporting services that enable legacy components to be integrated into an adaptive system. Specifically, we weave in a COCA communication interface transparently with respect to the existing business code of each application. With the COCA communication interface, an application can communicate with the Service Clouds infrastructure and other peer applications by exchanging and interpreting XML-formatted messages.

Most of the necessary information to support the transparent shaping is specified in the SSD information section. The information section addresses two main questions regarding integration activities: (1) Which software components are involved in the adaptation? (2) How do they interact with one another, that is, through which interfaces? This part of SSD is used at design time to generate "glue code" that enables existing applications and underlying services to interact with one another. The collected information includes the component name (used to identify the component), communication information (used to exchange messages with other components), the development language (used to decide the suitable means to generate "glue code"), and the interfaces through which the individual components can adjust their behaviors.

Figure 5.16 and 5.17 show an example SSD information section and the generated "glue code" skeleton, respectively. The interactive activities between the system developer and the application developer as well as the involved software components are illustrated in Figure 5.18. Specifically, the system developer specifies that an application named *app.fecc*

```
<informationSection>
  <serviceClient>
    <itemName>app.fecc</itemName>
    <hostName>M1.cse.msu.edu</hostName>
    <devLanguage>Java</devLanguage>
    <mainFile>FECClient.java</mainFile>
    <interactionInterface>reConnect</interactionInterface>
  </serviceClient>
</informationSection>
```

Figure 5.16: An example SSD information section for the application *app.fecc*.

running on *M1.cse.msu.edu* can be reconfigured through one interface of *reConnect*. When *app.fecc* receives the COCA *reConnect* message, it should invoke re-connection actions to react to the changes of IP address. The system developer publishes this SSD through a Web service interface, and the application developer can retrieve this SSD through an SSD console. After reviewing this SSD, the application developer should complete the application-specific information, e.g., the development language is *Java* and the main file is *FECClient.java*. As a result, the application developer can generate the "glue code" skeleton with the help of the COCA development toolkit provided by the SSD console. The skeleton comprises the code for weaving in a COCA communication interface to the application with the Aspect Java technique and processing the incoming COCA reconfiguration messages. The only thing left to the application developer is to map the message handling functionalities with the concrete reconfiguration implementations. For example, the dynamic proxy instantiation service will dynamically instantiate a new transient proxy on the wireless edge when a mobile client roams into a new subnet and its IP address changes. In this situation, the dynamic proxy instantiation service will notify the application *app.fecc* to re-connect to the new instantiated proxy by sending a COCA *reConnect* message. Upon receiving this message, the application should re-establish the socket con-

```
public aspect Bridging_FECClient {
// listen to the COCA messages
  declare parents: FECClient implements Observer;
// connects to the COCA infrastructure
  after() returning(FECClient fecc): call(FECClient.new(..)) {
// register topology information
   COCACoreEnv.connLocal("app.fecc");
   MsgGateway.getInstance().addObserver(fecc);
// register reconfiguration interfaces
   ReconfCmd reconfCmd = new ReconfCmd("");
   reconfCmd.setCmdName("reConnect");
   MadaptHelper.conn(reconfCmd);
 }

// process reconfiguration message
// this code skeleton is automatically generated based on COCA framework
  public FECClient.update(Observable arg0, Object arg1) {
    // receive and interpret  message
    MsgMessage msg = (MsgMessage) arg1;
    if (msg.checkMsgName(MadaptMessage.MSG_NAME_RECONF)) {
      ReconfCmd reconfCmd = new ReconfCmd(msg.getMsgParams());
      String cmdName = reconfCmd.getCmdName();

      if (cmdName.equalsIgnoreCase("reConnect")) {
       // here, the concrete implementation should be completed
       // by the application developer
       try {
         RTSPsocket = new Socket(ServerIPAddr, RTSP_server_port);
         // reset socket:
         RTSPBufferedReader = new BufferedReader(new InputStreamReader(
                 RTSPsocket.getInputStream()));
         RTSPBufferedWriter = new BufferedWriter(new OutputStreamWriter(
                 RTSPsocket.getOutputStream()));
         // re-join multicast group
         if (isMulticast) {
           RTPsocket_Video.leaveGroup(multicastRcvGroupIP);
           RTPsocket_Audio.leaveGroup(multicastRcvGroupIP);
           RTPsocket_Video.joinGroup(multicastRcvGroupIP);
           RTPsocket_Audio.joinGroup(multicastRcvGroupIP);
         }
       } catch (IOException e) {
         e.printStackTrace();
       }
      }
    }
 }
}
```

Figure 5.17: An example of "glue code" skeleton generated for the application *app.fecc* to use the dynamic proxy instantiation service.

nection. The application developer can complete this functionality within the generated "glue code" skeleton and ship the shaped application to the system developer for further integration. With the help of ASSL and the SSD console, the two parties involved in this system integration process can clearly specify their requirements and exchange information.

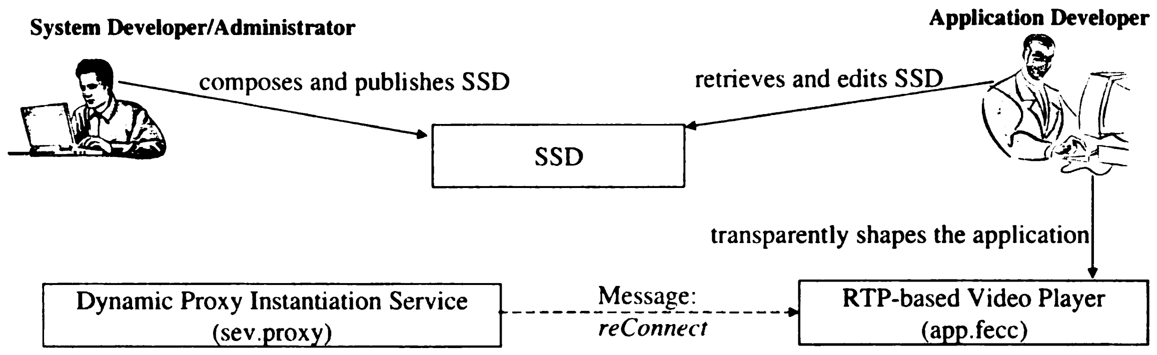

System Developer/Administrator
Application Developer
composes and publishes SSD
retrieves and edits SSD
SSD
transparently shapes the application
Dynamic Proxy Instantiation Service (sev.proxy)
Message: reConnect
RTP-based Video Player (app.fecc)

Figure 5.18: Interactive activities for transparently shaping applications.

Service shaping and configuration. Besides shaping the application to use the underlying services, the system developer also needs to specify how the services interact with the applications. The interactive activities between the system developer and the service developer as well as the involved software components are illustrated in Figure 5.19. In the demonstration example, the application *app.fecc* is capable of monitoring the network packet loss rate. When *app.fecc* detects the loss rate as being higher than a user pre-defined threshold, it can notify the interested parties of this event by sending COCA messages. Since the transient proxies running on the wireless edge provide the FEC services to compensate for the packet loss, *app.fecc* can request the transient proxies to instantiate the FEC services when the network packet loss is high and terminate the FEC services when the network packet loss is low. Thus, these interactions can be specified in an SSD information section by the system developer, and the service developer can retrieve this SSD through an SSD console. After reviewing this SSD, the service developer should complete the service-specific processing of service instantiation and termination upon receiving COCA messages from the applications. Figure 5.20 shows an example SSD information section for FEC services. Specifically, an FEC service named *sev.fec* running on a transient proxy *W1* can be reconfigured through two interfaces of *insertFEC* and *removeFEC*. When *W1* receives *insertFEC* or *removeFEC* COCA messages, it will invoke FEC service instantiation/termination actions to react to the changes of network packet loss.

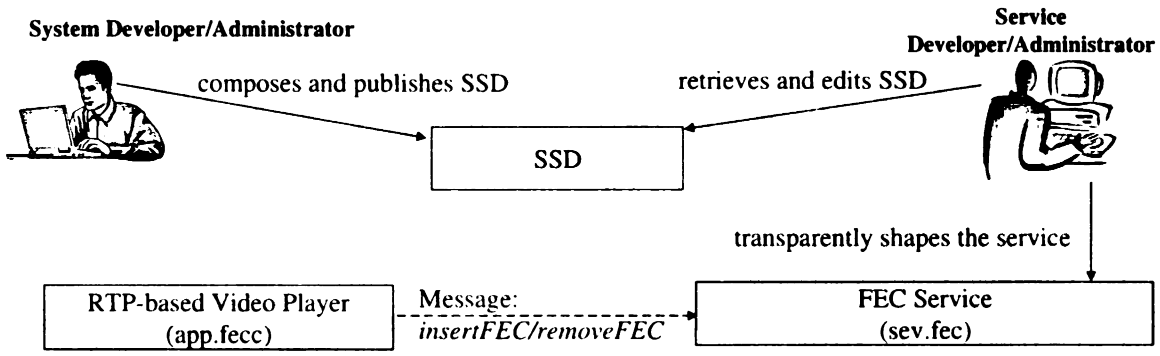

System Developer/Administrator
Service Developer/Administrator
composes and publishes SSD
retrieves and edits SSD
SSD
transparently shapes the service
RTP-based Video Player (app.fecc)
Message: insertFEC/removeFEC
FEC Service (sev.fec)

Figure 5.19: Interactive activities for service instantiation and termination.

```
<informationSection>
  <serviceProvider>
    <itemName>sev.fec</itemName>
    <hostName>W1.cse.msu.edu</hostName>
    <devLanguage>Java</devLanguage>
    <mainFile>AutoMMCMobileNode.java</mainFile>
    <interactionInterface>insertFEC</interactionInterface>
      <interfacePara-1>8</interfacePara-1>
      <interfacePara-2>4</interfacePara-2>
    <interactionInterface>removeFEC</interactionInterface>
  </serviceProvider>
</informationSection>
```

Figure 5.20: An example SSD information section for FEC services.

5.5.2 Service Binding

In the running example, in order to use the underlying services, the applications need to connect to the Service Clouds infrastructure, set up service path, and establish service binding. For example, the application *app.fecc* needs to know the entry point (service gateway) of the Service Clouds infrastructure so that it can initialize the service binding process. On the other hand, the Service Clouds infrastructure needs to know the video streaming specific communication interfaces (e.g., RTP and RTSP port) to set up the proxy service path. Thus, the system developer, the application developer, and the service developer can exchange this information in the binding section of an SSD.

Binding with the UDP relay service. Our previous studies indicate that application level relays in an overlay network can actually improve network throughput for long-distance bulk transfers [181]. For example, due to the dependence of TCP throughput on round trip time (RTT), splitting a connection into two (or more) shorter segments can increase throughput, depending on the location of the relay nodes and the overhead of intercepting and relaying the data. By using application layer entities to emulate network-layer functionality, such TCP relays can be more easily deployed and managed than some other approaches to improving TCP throughput, such as using advanced congestion control protocols, which requires either router support or kernel modifications. Similar to this concept, in the demonstration example, we deploy UDP relay services on the primary proxies. To develop a practical UDP relay service, key issues to be addressed include identification of promising relay nodes for individual data transfers, and the dynamic instantiation of the relay service. The example selection rules for relay nodes can be based on RTT or other

QoS and security parameters.

```
<bindingSection>
  <serviceGateway>G.planetlab.org</serviceGateway>
  <server>S.cse.msu.edu</server>
  <serverRTSPPort>33503</serverRTSPPort>
  <clientRTPPort>33501</clientRTPPort>
  <UDPRelaySelection>RTT</UDPRelaySelection>
  <primaryProxy>N2.planetlab.org</primaryProxy>
  <primaryProxyPort>57794</primaryProxyPort>
</bindingSection>
```

Figure 5.21: An example SSD binding section for UDP relay services.

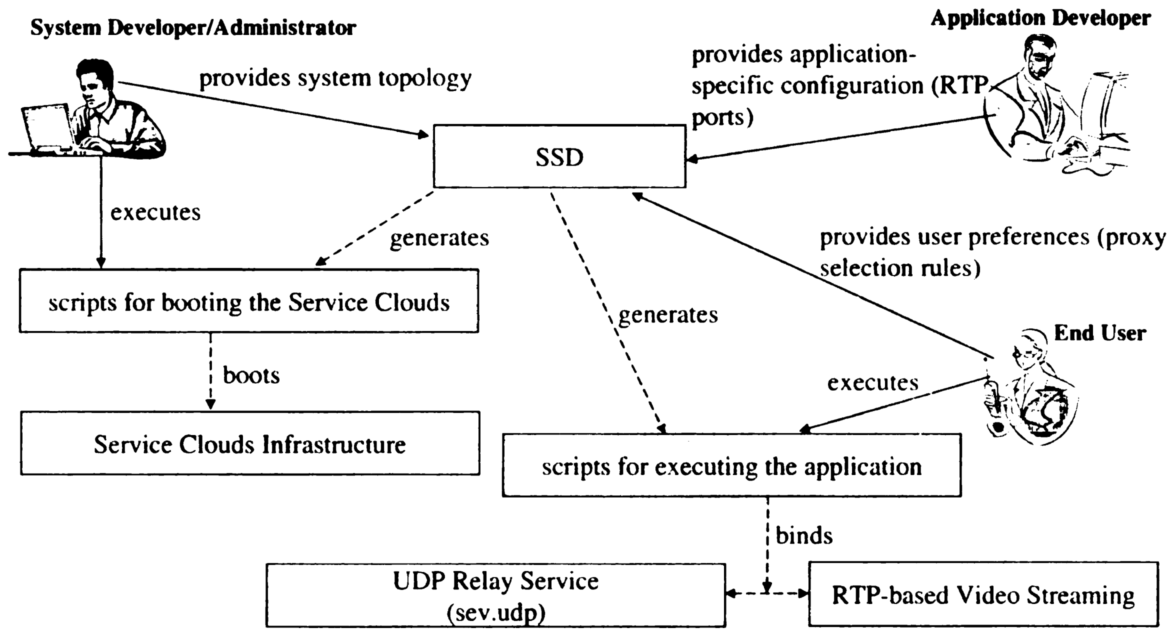

System Developer/Administrator
provides system topology
Application Developer
provides application-specific configuration (RTP ports)
SSD
executes
generates
provides user preferences (proxy selection rules)
scripts for booting the Service Clouds
generates
End User
boots
executes
Service Clouds Infrastructure
scripts for executing the application
binds
UDP Relay Service (sev.udp)
RTP-based Video Streaming

Figure 5.22: Interactive activities for binding with the UDP relay service.

Figure 5.21 shows an SSD binding section for the UDP relay service. The interactive activities between the system developer, the application developer, and the end user as well as the involved software components are illustrated in Figure 5.22. The system developer assigns the service gateway to be running on *G.planetlab.org*. The application developer specifies that the video server *S* runs on *S.cse.msu.edu* with the RTSP port *33503* and the application uses the RTP port *33501*. The user can specify that the selection of the primary proxy for the UDP service is based on RTT. Based on this collected information, the system developer can generate an execution script file, as shown in Figure 5.23, to boot the Service Clouds infrastructure. After the Service Clouds infrastructure is booted, the application *app.fecc* can connect to the service gateway running on *G.planetlab.org* to request the UDP relay service. The primary proxy is selected based on the RTT between the video server *S* and the primary proxy candidates *N2* and *N4*. The experimental computation results shows that the RTT between *S* and *N2* is 70.071 ms, whereas it is 97.188 ms between *S* and *N4*. Thus, the service gateway assigns the node *N2* running on *N2.planetlab.org* as the primary proxy, and notifies the application *app.fecc* of the selection result by updating the SSD. This primary proxy will open the port *57794* for receiving RTP commands from the application *app.fecc*. This primary proxy will also open the port *33501* for receiving RTSP video packets from the video server.

```
java autommc -sc-gateway G.planetlab.org -mm-server S.cse.msu.edu
 -client-video-port 33501 -server-mmc-port 33503 -relay-selection rtt
```

Figure 5.23: An example execution script for booting the Service Clouds infrastructure.

Moreover, based on the result of identifying the service path for the video stream, the system developer can generate another execution script file, as shown in Figure 5.24, to bind the application *app.fecc* with the UDP relay service. The end user can use this script file to start the application *app.fecc*. After execution, the application *app.fecc* can communicate with the primary proxy *N2*, instead of the original video server *S*, to obtain the video stream.

```
java videostreamer.FECClient -server N2.planetlab.org -port 57794
 -resource resource/MVI_8065
```

Figure 5.24: An example execution script for binding UDP relay services.

Binding with the robust pervasive streaming service. As mentioned earlier, the Service Clouds infrastructure supports the video stream atop an overlay network to clients of various types (e.g., desktop, laptop, PDA) with different connections (e.g., wired LAN, 802.11 wireless). The overlay service path, which is composed and maintained dynamically, has to provide robustness through different mechanisms as the stream traverses different environments. The Service Clouds infrastructure supports continuous streaming by the dynamic instantiation of transient proxies, while users roam along different subnets.

For example, user *M1* requests to watch the video stream, and the Service Clouds infrastructure connects to the video server and successfully multicasts the stream through a transient proxy *W1* in subnet *A*, where the user *M1* is located. Next, user *M2* requests to watch the video stream being broadcasted from the server. Upon receiving the request, the infrastructure identifies that the video is already being multicast in user *M2* location, that is, subnet *A*. Thus, no extra configuration is necessary in the service path, except registering user *M2* as a service receiver. Finally, when user *M1* goes to wireless via subnet *B*, the infrastructure detects this change and branches off a copy of the stream through another transient proxy *W2*, practically constructing an overlay multicast tree, which delivers the stream at subnet *B*.

Figure 5.25 shows an example SSD binding section for robust pervasive streaming services. The interactive activities between the system developer, the application developer, and the service developer as well as the involved software components are illustrated in Figure 5.26. Specifically, the application developer specifies the multicast information and the service developer provides the information about the candidate transient proxies. When the running host of the application *app.fecc* changes its IP address, the robust pervasive streaming service *sev.per* will compare the client's new IP address with the IP addresses of the candidate transient proxies (i.e., the one running on *W1.cse.msu.edu* and the one running on *W2.egr.msu.edu*), and check if there is a transient proxy within the same subnet as the client's new IP address. If not, the transient proxy will be instantiated and configured to multicast to the multicast group of *228.5.6.7*, to which the application *app.fecc* listens. At the same time, the application *app.fecc* will get a COCA *reConnect* message notification of re-joining the multicast group. Here, only one candidate transient proxy is available for each subnet. If there were multiple candidates, similar selection rules as those of primary proxies could be applied. Remembering that we have demonstrated how the application *app.fecc* should react to the COCA *reConnect* message, this operation demonstrates that the Service Clouds infrastructure builds an overlay service path for multicasting toward the wireless edge and handles change of IP address on the mobile client transparently to the end user.

```
<bindingSection>
  <transientProxy-1>W1.cse.msu.edu</transientProxy-1>
  <transientProxy-2>W2.egr.msu.edu</transientProxy-2>
  <multicastGroup>228.5.6.7</multicastGroup>
</bindingSection>
```

Figure 5.25: An example SSD binding section for robust pervasive streaming services.

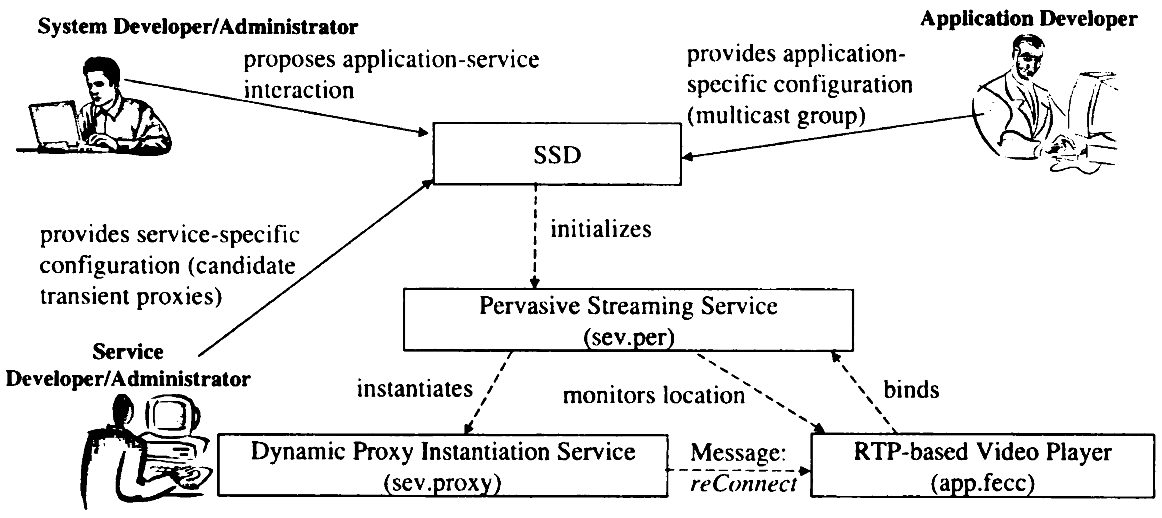

System Developer/Administrator
proposes application-service interaction
Application Developer
provides application-specific configuration (multicast group)
SSD
initializes
provides service-specific configuration (candidate transient proxies)
Service Developer/Administrator
Pervasive Streaming Service (sev.per)
instantiates
monitors location
binds
Dynamic Proxy Instantiation Service (sev.proxy)
Message: reConnect
RTP-based Video Player (app.fecc)

Figure 5.26: Interactive activities for binding with the robust pervasive streaming service.

5.5.3 Run-Time Service-Application Interaction

Having shaped the applications and configured the services, the autonomic system is ready for use. To manage the run-time service-application interaction, the end user should be able to specify the user preferences according to the parameterized services resource and run-time conditions. Meanwhile, the system developer should be able to specify how the underlying services provide supporting information for adaptation decisions, which are either made by the end user or made by the system automatically according to the pre-defined rules. The system developer should also be able to specify how the applications react to the adaptation-specific events generated by the underlying services. All this information can be reflected in the SSD interaction section.

In the interaction section, the SSD defines the conditions under which the system should adjust its behaviors and the corresponding concrete actions. The interactive activities between the system developer, the application developer, the service developer, and the end user as well as the involved software components are illustrated in Figure 5.27. Specifically, the SSD groups concrete interaction activities of the autonomic system in response to the run-time environment, identifies the events that will trigger the interactions, defines an action list that will guide the system behaviors in response to the trigger events, and specifies any constraints that validate the policy rules. An example SSD interaction section is illustrated in Figure 5.28. In this example, the user specifies the tolerable loss rate of the video streaming at *20%*; when the network packet loss rate is higher than this threshold, the application *app.fecc* should request the instantiation of the FEC services (*sev.fec*) on the transient proxies by sending COCA *insertFEC* messages. Figure 5.29 shows the generated "glue code" skeleton and the concrete implementation for the FEC service *sev.fec* to react to the adaptation request of compensating for network packet loss. Specifically, after receiving a COCA message and if this message is for inserting FEC facilities, the FEC service *sev.fec* will use the FEC encoder to process the incoming data packets with FEC (n, k) parameters. Finally, the FEC service *sev.fec* will multicast the FEC encoded packets. Correspondingly, the application *app.fecc* will insert the FEC decoder to decode the incoming FEC data packets. With the help of the FEC services, the quality of the video stream is improved.

In the demonstration example, the multimedia stream sends audio and video UDP packets over separate UDP sockets. When *M1* moves to the wireless subnet, the connection becomes prone to high loss rate. Whenever the application *app.fecc* detects intolerable loss rate (higher than 20% in our test), the transient proxy on the wireless edge uses FEC to encode the stream by breaking each packet into four packets and sending them across the wireless link along with four extra parity packets. Figure 5.30 plots the packet loss rates for audio and video at *M1*. We have applied the FEC encoding only on the audio stream. In

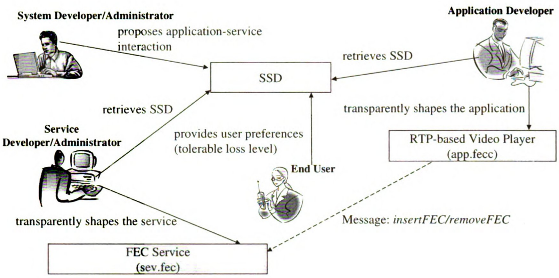

System Developer/Administrator
proposes application-service interaction
Application Developer
retrieves SSD
SSD
retrieves SSD
transparently shapes the application
Service Developer/Administrator
provides user preferences (tolerable loss level)
RTP-based Video Player (app.fecc)
End User
transparently shapes the service
Message: insertFEC/removeFEC
FEC Service (sev.fec)

Figure 5.27: Interactive activities for run-time service-application interaction.

our experiment, video packets are a few times bigger than MTU (maximum transmission unit). This results in the fragmentation of UDP packets and significantly increases packet loss rate due to a lack of MAC-layer retransmission in multicasting. Typical FEC encoding has little advantage for the high quality video packet streaming on wireless channels. Thus, more complicated adaptation techniques to transcode the video are needed. However, this is not our focus in this work. As the plots show, at the time slot 5 the user switched from the wired subnet to the wireless subnet. Thus, the network loss rate raised significantly. Accordingly, based on the feedback from the application *app.fecc*, the system instantiated or terminated the FEC service at the time slot 11 and 26, respectively. This adaptation effectively mitigated the packet loss rate observed by the application *app.fecc*.

```
<interactionSection>
  <InteractionItem>
    <name>interactLossRate</name>
    <description>reduce high loss rate</description>
    <YourTolerableLossRate>20</YourTolerableLossRate>
    <event>
      <name>high_loss_rate_alert</name>
    </event>
    <action>
      <source>app.fecc</target>
      <target>sev.fec</target>
      <message>insertFEC</message>
    </action>
  </InteractionItem>
</interactionSection>
```

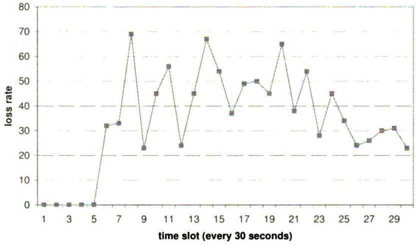
Figure 5.28: An example SSD interaction section for compensating for the network packet loss.

5.6 Conclusions

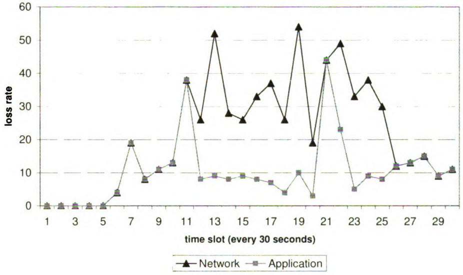
In this chapter, we propose ASSL, an XML-based technique that provides comprehensive specification of an autonomic system, focusing on the system integration, configuration, and run-time interaction management. ASSL is an extension of COCA specification with more visualization and extensibility, and it provides a unified platform to support the interactions among different parties in the orchestration and execution of autonomic systems. We illustrate the use of ASSL to specify interaction between the applications and the underlying autonomic services. Using aspect-oriented techniques, we can generate and weave in "glue code" into an existing application to make it ready for interaction with the autonomic services and other applications. Meanwhile, the specified information included in an SSD can be used to facilitate the deployment, configuration, and run-time management of autonomic services.

```
public aspect Bridging_AutoMMC {

// connects to the COCA infrastructure
// and register reconfiguration interfaces
   ......

// process reconfiguration message
// this code skeleton is automatically generated based on COCA framework
  public autommcmobilenode.update(Observable arg0, Object arg1) {
    // receive and interpret  message
    MsqMessage msg = (MsqMessage) arg1;
    if (msg.checkMsgName(MadaptMessage.MSG_NAME_RECONF)) {
      ReconfCmd reconfCmd = new ReconfCmd(msg.getMsgParams());
      String cmdName = reconfCmd.getCmdName();

      if (cmdName.equalsIgnoreCase("insertFEC")) {
       // here, the concrete implementation should be completed
       // by the application developer

       // create a RTPpacket object from the DP
       RTPpacket rtp_packet = new RTPpacket(pktRcvd.getData(),
                                            pktRcvd.getLength());

       // get the header and payload of the received RTP packet
       int payload_length = rtp_packet.getpayload_length();
       byte[] payload = new byte[payload_length];
       rtp_packet.getpayload(payload);
       int rtpHeaderSize = RTPpacket.HEADER_SIZE;
       byte[] rtpHeader = new byte[rtpHeaderSize];
       System.arraycopy(rtp_packet.header,0,rtpHeader,0,rtpHeaderSize);

       // encode the payload with FEC(n,k)
       FECPacket[] fecPacketList =
         currentRelay.fecEncoder.encodePacket(payload);
       for (int j = 0; j < fecPacketList.length; j++) {
         // Builds an RTPpacket object containing the frame
         int fecPacketSize = fecPacketList[j].getPacketSize();
         byte[] fecPacket = fecPacketList[j].getPacket();
         int rtpPacketSize = rtpHeaderSize + fecPacketSize;
         byte[] rtpPacket = new byte[rtpPacketSize];
         System.arraycopy(rtpHeader,0,rtpPacket,0,rtpHeaderSize);
         System.arraycopy(fecPacket,0,rtpPacket,
                          rtpHeaderSize,fecPacketSize);
         // construct the packet to send
         DatagramPacket aDP = new DatagramPacket(
           rtpPacket,
           rtpPacketSize,
           currentRelay.receiverInetAddress,
           currentRelay.receiverPortNum);
         try {
           currentRelay.outSocket.send(aDP);
         }catch (Exception ex) {
           ex.printStackTrace();
         }
       }
     }
   }
  }
}
```

Figure 5.29: An example of "glue code" skeleton generated for the FEC service *sev.fec* to react the adaptation request of compensating for the network packet loss.



(a) video



(b) audio

Figure 5.30: Packet loss rate at the mobile node M1.

Chapter 6

CONCLUSIONS AND FUTURE RESEARCH

Our studies have established a solid understanding of adaptation characteristics and the use of expressive orchestration in adaptive software design for mobile systems. By providing a means to specify system requirements, manage the interaction between systems and users, support interoperability among system compositional components, and encapsulate the adaptation logic, expressive orchestration offers an effective solution to the design, development, and run-time management of adaptive mobile systems. The applicable domain of expressive orchestration expands from individual applications to composite systems and fully distributed systems. In the rest of this chapter, we summarize our specific contributions and discuss the future work.

6.1 Summary of Contributions

In summary, this research makes several contributions [12–17]:

1. This dissertation provides a comprehensive investigation of the necessary techniques (including basic adaptation characteristics, language and architecture support for collaborative adaptation, and adaptation decision reasoning) for building an adaptive mobile system. These preliminary and experimental investigations can be used as a basis for the development of adaptive software mechanisms that attempt to manage adaptation tradeoffs in the presence of highly dynamic wireless environments. As a case study, we evaluate the energy consumption of FEC as used to improve QoS on wireless devices, where encoded audio streams are multicast to multiple mobile computers. Our results quantify the tradeoff between improved QoS, due to FEC, and additional energy consumption, delay, and bandwidth usage caused by receipt and decoding of redundant packets.

2. Based on the preliminary studies on adaptation characteristics, we investigate the use of message-based communication to facilitate the integration and collaboration of adaptive/non-adaptive components. As a proof of concept, we develop COCA (**CO**mposing **C**ollaborative **A**daptation), an infrastructure for collaborative adaptation among components that were not necessarily designed to interoperate in the composite systems. COCA provides a set of development utilities to aid system designers in specifying system architecture and adaptation logic and automatically generating the corresponding code to realize collaborative adaptation among existing components. COCA provides a set of run-time utilities to enforce the collaborative

adaptation execution. COCA also provides a Web services infrastructure to support the corresponding interaction among components. The methods used in COCA are general and can be extended to other distributed computing models that require collaborative adaptation. For example, we apply COCA in a service-oriented infrastructure, called Service Clouds, providing interactive design support and run-time adaptation management.

3. This dissertation addresses specification techniques that can help design, development, deployment, and management of fully distributed service-oriented autonomic systems. We propose ASSL (**A**utonomic **S**ervice **S**pecification **L**anguage), an XML-based technique that provides comprehensive specification of an autonomic system, focusing on system integration, configuration, and run-time interaction management. ASSL is an extension of COCA specification with more visualization and extensibility, and it provides a unified platform to support the interactions among different parties in the development and execution of autonomic systems.

6.2 Future Research

Several investigations complementary to the research presented in this dissertation may be pursued in future work.

6.2.1 Modeling Adaptive Systems with Patterns

Given the potentially critical nature of adaptive systems in which system faults could lead to significant loss, methods for modeling and analyzing adaptive systems before starting the design and development phase are increasingly important. However, currently many adaptive systems use ad hoc development approaches that emphasize implementation over analysis, often causing conceptual errors to be propagated from prototyping design to system execution. To model adaptive systems, we first need to understand those basic characteristics (in both design and execution aspects) of adaptive systems. The concept of design patterns can help on this issue.

Patterns are a way of documenting experience by capturing successful solutions to recurring problems. Therefore, they are best suited for describing proven solutions of design problems in adaptive systems. Although patterns are well-known in software engineering, they have successfully been applied to other domains as well, including patterns for organizations, processes, analysis, customer interaction, and many more. Because patterns are rooted in practice, this dissertation, as well as other related works conducted by the Software Engineering and Network Systems (SENS) Laboratory in Michigan State University, has investigated different aspects of adaptive systems and implemented several running adaptive systems. Thus, it is possible to generalize patterns that cover most aspects of design, development, and management of adaptive systems. These patterns could provide a solid basis for further modeling adaptive systems.

6.2.2 Contract-Based QoS Specification

As discussed earlier, in many recent studies [10, 60, 79, 91, 92, 94–96, 169, 209], contracts have been used in the management of adaptation. Techniques have been proposed for contract description, contract reasoning, and contract enforcement. However, the correctness of adaptation contracts has not yet been studied extensively. In order to illustrate the problems that we plan to investigate, let us consider the MetaSocket enabled audio conferencing testbed described in Chapter 3. Components in such a system may expose two types of interfaces: adaptation interfaces (which can be used to reconfigure the application behaviors) and constraint interfaces (which can be used to inspect the pre- or post-condition of the adaptation). For example, adaptation interfaces include methods to insert/remove FEC filters; whereas obtaining the processing overhead (time) is a constraint interface. The application may provide both of these two types of interfaces or only the adaptation interfaces.

As shown in Chapter 4, in order to integrate the above audio application into a collaborative multimedia conferencing system, we can use COCA to specify the architecture composition and adaptation policies. However, some quality of service aspects are still missing. First, how can we ensure that the adaptation policies are not obviated by other constraints? For example, an adaptation rule may indicate that an FEC filter should be inserted when the observed loss rate is high. This rule itself is executable. However, in a particular conferencing system, the user may also have specified real-time constraints. From this example we can see that while each component is correct, some constraint logic on their composition may need to be considered. The system developer needs a means to express such QoS concerns and formally specify those constraints with the format of contract at design time.

For adaptive component design, if a component provides a constraint interface, we may use model checking to test if it meets the overall system real-time requirements. On the other hand, if the component does not provide such an interface to support model checking, we may generate testing code from the formal constraint specification in the contract. Using the testing code, we can check if a component satisfies the overall system requirements. If it does, the component is allowed to connect to the system, otherwise a negotiation between the system developer and component developer (system and its compositional components) is required for integration purposes. Open questions in this work include the following: How do we use the "contract" to specify these constraints? How do we verify (statically and dynamically) the component design and the generated code against the above contracts? How can we manage the contract negotiation and enforcement?

BIBLIOGRAPHY

Bibliography

[1] Jacob R. Lorch and Alan Jay Smith. Software strategies for portable computer energy management. *IEEE Personal Communications Magazine*, 5(3):60–73, June 1998.

[2] Erik P. Harris, Steven W. Deep, William E. Pence, Scott Kirkpatrick, M. Sri-Jayantha, and Ronald R. Troutman. Technology directions for portable computers. *Proceedings of the IEEE*, 83(4):636–658, April 1995.

[3] Gartner Inc. `http://www.gartner.com`.

[4] Sanjay Udani and Jonathan Smith. Power management in mobile computing. Technical Report MS-CIS-98-26, Distributed Systems Laboratory, Department of Computer Information Science, University of Pennsylvania, August 1996.

[5] Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile application-aware adaptation for mobility. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 276–287, Saint Malo, France, 1997.

[6] Sarita V. Adve, Albert F. Harris, Christopher J. Hughes, Douglas L. Jones, Robin H. Kravets, Klara Nahrstedt, Daniel Grobe Sachs, Ruchira Sasanka, Jayanth Srinivasan, and Wanghong Yuan. The Illinois GRACE Project: Global Resource Adaptation through CoopEration. In *Proceedings of the ACM Workshop on Self-Healing, Adaptive and Self-Managed Systems (SHAMAN)*, New York City, June 2002.

[7] W. Yuan, K. Nahrstedt, S. Adve, D. Jones, and R. Kravets. Design and evaluation of a cross-layer adaptation framework for mobile multimedia systems. In *Proceedings of the SPIE/ACM Multimedia Computing and Networking Conference (MMCN'03)*, pages 1–13, Santa Clara, CA, January 2003.

[8] B. D. Noble and M. Satyanarayanan. Experience with adaptive mobile applications in Odyssey. *Mobile Networks and Applications*, 4(4):245–254, 1999.

[9] Christian Poellabauer and Karsten Schwan. Kernel support for the event-based cooperation of distributed resource managers. In *Proceedings of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2002)*, San Jose, California, September 2002.

[10] John Keeney and Vinny Cahill. Chisel: A policy-driven, context-aware, dynamic adaptation framework. In *Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks*, pages 3–14, 2003.

[11] Component-Based Development of Adaptable and Dependable Middleware. http://www.cse.msu.edu/ mckinley/rapidware/, accessed November 2005. Computer Science and Engineering Department of Michigan State University.

[12] Z. Zhou, P. K. McKinley, and S. M. Sadjadi. On quality-of-service and energy consumption tradeoffs in FEC-enabled audio streaming. In *Proceedings of the 12th IEEE International Workshop on Quality of Service (IWQoS 2004)*, Montreal, Canada, June 2004.

[13] Philip K. McKinley, E. P. Kasten, S. M. Sadjadi, and Zhinan Zhou. Realizing multi-dimensional software adaptation. In *Proceedings of the ACM Workshop on Self-Healing, Adaptive and self-Managed Systems (SHAMAN)*, held in conjunction with the *16th Annual ACM International Conference on Supercomputing*, New York City, June 2002.

[14] S. M. Sadjadi, Philip K. McKinley, Eric P. Kasten, and Zhinan Zhou. Metasockets: Design and operation of run-time reconfigurable communication services. *The special issue on Auto-adaptive and Reconfigurable Systems of the Wiley InterScience Software-Practice and Experience (SP&E) journal*, 2006.

[15] Zhinan Zhou, Ji Zhang, Philip K. McKinley, and Betty H. C. Cheng. TA-LTL: Specifying adaptation timing properties in autonomic systems. In *Proceedings of the 3rd IEEE Workshop on Engineering of Autonomic and Autonomous Systems (EASe 2006)*, Columbia, MD, USA, April 2006.

[16] Zhenxiao Yang, Zhinan Zhou, Betty H. Cheng, and Philip K. McKinley. Enabling collaborative adaptation across legacy components. In *Proceedings of the 3rd Workshop on Reflective and Adaptive Middleware (RM 2004)*, 2004.

[17] Zhinan Zhou and Philip K. McKinley. COCA: A contract-based infrastructure for composing adaptive multimedia systems. In *Proceedings of the 8th International Workshop on Multimedia Network Systems and Applications (MNSA 2006)*, Lisboa, Portugal, July 2006.

[18] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng. Composing adaptive software. *IEEE Computer*, 37(7):56–64, 2004.

[19] David P. Helmbold, Darrell D. E. Long, and Bruce Sherrod. A dynamic disk spin-down technique for mobile computing. In *Mobile Computing and Networking*, pages 130–142, 1996.

[20] David P. Helmbold, Darrell D. E. Long, Tracey L. Sconyers, and Bruce Sherrod. Adaptive disk spin-down for mobile computers. *Mobile Networks and Applications*, 5(4):285–297, 2000.

[21] Fred Douglis, Padmanabhan Krishnan, and Brian Bershad. Adaptive disk spin-down policies for mobile computers. In *Proceedings of the 2nd USENIX Symposium on Mobile and Location-Independent Computing*, 1995.

[22] P. Krishnan, Philip M. Long, and Jeffrey Scott Vitter. Adaptive disk spindown via optimal rent-to-buy in probabilistic environments. In *Proceedings of the 12th International Conference on Machine Learning (ML95)*, pages 322–330, 1995.

[23] Y. Lu and G. De Micheli. Adaptive hard disk power management on personal computers. *IEEE Great Lakes Symposium on VLSI*, pages 50–53, 1999.

[24] Fred Douglis, P. Krishnan, and Brian Marsh. Thwarting the power-hungry disk. In *USENIX Winter*, pages 292–306, 1994.

[25] Alexey Rudenko, Peter Reiher, Gerald J. Popek, and Geoffrey H. Kuenning. Saving portable computer battery power through remote process execution. *Mobile Computing and Communications Review*, 2(1):19–26, January 1998.

[26] Dietmar A. Kottmann, Ralph Wittmann, and Markus Posur. Delegating remote operation execution in a mobile computing environment. *Mobile Networks and Applications*, 1(4):387–397, 1996.

[27] Kester Li, Roger Kumpf, Paul Horton, and Thomas E. Anderson. A quantitative analysis of disk drive power management in portable computers. In *USENIX Winter*, pages 279–291, 1994.

[28] B.T. Zivkov and A.J. Smith. Disk caching in large database and timeshared system. In *Proceedings of the 5th International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS 97)*, pages 184–195, Haifam Israel, 1997.

[29] Mark Weiser, Brent Welch, Alan J. Demers, and Scott Shenker. Scheduling for reduced CPU energy. In *Operating Systems Design and Implementation*, pages 13–23, 1994.

[30] Kinshuk Govil, Edwin Chan, and Hal Wasserman. Comparing algorithm for dynamic speed-setting of a low-power CPU. In *Mobile Computing and Networking*, pages 13–25, 1995.

[31] Jacob R. Lorch and Alan Jay Smith. Operating system modifications for task-based speed and voltage scheduling. In *Proceedings of the 1st International Conference on Mobile Systems, Applications, and Services (MobiSys 2003)*, pages 215–230, San Francisco, CA, USA, 2003.

[32] J. Lorch and A.J. Smith. Reducing processor power consumption by improving processor time management in a single-user operating system. In *Proceedings of the 2nd ACM International Conference on Mobile Computing and Networking (MOBICOM)*, page 143C154, Rye Brook, NY, 1996.

[33] Trevor Pering, Tom Burd, and Robert Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *Proceedings of the 1998 International Symposium on Low Power Electronics and Design*, pages 76–81. ACM Press, 1998.

[34] D. Lee. Energy management issues for computer systems, http://www.cs.washington.edu/homes/dlee/frontpage/mypapers/generals.ps.gz.

[35] J. Lorch. A complete picture of the energy consumption of a portable computer, Master Thesis, Computer Science, University of California at Berkeley, 1995.

[36] K. Werner. Flat panels fill the color bill for laptops. *Circuits and Devices*, 10(4):21–29, July 1994.

[37] Subu Iyer, Lu Luo, Robert Mayo, and Parthasarathy Ranganathan. Energy-adaptive display system designs for future mobile environments. In *Proceedings of the 1st International Conference on Mobile Systems, Applications, and Services (MobiSys2003)*, San Francisco, California, May 2003.

[38] M. Stemm and R. H. Katz. Measuring and reducing energy consumption of network interfaces in hand-held devices. *IEICE Transactions on Communications*, E80-B(8):1125–31, 1997.

[39] R. Xu, Z. Li, C. Wang, and P. Ni. Impact of data compression on energy consumption of wireless-networked handheld devices. In *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems (ICDCS'03)*, Providence, Rhode Island, May 2003.

[40] Michele Zorzi and Ramesh R. Rao. Error control and energy consumption in communications for nomadic computing. *IEEE Transactions on Computers*, 46(3):279–289, 1997.

[41] CORBA and IIOP Specification. `http://www.omg.org/technology/documents/formal/corbaiiop.htm`, accessed July 2004.

[42] Microsoft .NET Homepage. `http://www.microsoft.com/net/`, accessed July 2005.

[43] Java RMI Homepage. `http://java.sun.com/products/jdk/rmi/`, accessed July 2005.

[44] IBM and Cisco. *Adaptive Services Framework*, October 2003.

[45] Barry Redmond and Vinny Cahill. Supporting unanticipated dynamic adaptation of application behaviour. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, London, UK, 2002.

[46] Shivajit Mohapatra, Radu Cornea, Nikil Dutt, Alex Nicolau, and Nalini Venkatasubramanian. Integrated power management for video streaming to mobile handheld devices. In *Proceedings of the 11th ACM International Conference on Multimedia*, pages 582–591. ACM Press, 2003.

[47] M. Satyanarayanan. Fundamental challenges in mobile computing. In *Symposium on Principles of Distributed Computing*, pages 1–7, 1996.

[48] M. Satyanarayanan, Brian Noble, Puneet Kumar, and Morgan Price. Application-aware adaptation for mobile computing. In *Proceedings of the 6th ACM SIGOPS European Workshop*, pages 1–4. ACM Press, 1994.

[49] Jason Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, Kiawah Island Resort, SC, December 1999.

[50] Jason Flinn and M. Satyanarayanan. Powerscope: a tool for profiling the energy usage of mobile applications. In *Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications*, pages 2–10, New Orleans, LA, February 1999.

[51] Jason Flinn, Eyal de Lara, M. Satyanarayanan, Dan S. Wallach, and Willy Zwaenepoel. Reducing the energy usage of office applications. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, Heidelberg, Germany, November 2001.

[52] Wanghong Yuan and Klara Nahrstedt. ReCalendar: Calendaring and scheduling applications with CPU and energy resource guarantees for mobile devices. In *Proceedings of the 1st IEEE International Conference on Pervasive Computing and Communications (PerCom'03)*, Fort Worth,Texas, March 2003.

[53] Shang-Wen Cheng, An-Cheng Huang, David Garlan, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self adaptation with reusable infrastructure. *IEEE Computer*, 37(10), 2004.

[54] H. Liu, M. Parashar, and S. Hariri. A component-based programming framework for autonomic applications. In *Proceedings of the 1st IEEE International Conference on Autonomic Computing (ICAC)*, New York, USA, May 2004.

[55] Fabio Kon, Roy H. Campbell, M. Dennis Mickunas, and Klara Nahrstedt. 2K: A distributed operating system for dynamic heterogeneous environments. In *Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing*, Pittsburgh, 2000.

[56] J. Appavoo et al. Enabling autonomic behavior in systems software with hot swapping. *IBM Systems Journal, Special Issue on Autonomic Computing*, 42(1), 2003.

[57] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee. The design of the TAO real-time object request broker. *Computer Communications*, 21(4), 1997.

[58] Fabio Kon, Manuel Román, Ping Liu, Jina Mao, Tomonori Yamane, Luiz Claudio Magalhães, and Roy H. Campbell. Monitoring, security, and dynamic configuration

with the dynamicTAO reflective ORB. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, number 1795 in LNCS, pages 121–143, New York, April 2000. Springer-Verlag.

[59] G. S. Blair, G. Coulson, A. Andersen, M. Clarke, F. M. Costa, H. A. Duran, R. Moreira, N. Paralavantzas, and K. B. Saikoski. The design and implementation of open ORB version 2. *IEEE Distributed Systems Online*, 2(6), 2001.

[60] Partha P. Pal, Joseph.P. Loyall, Richard E. Schantz, John A. Zinky, and Franklin Webber. Open implementation toolkit for building survivable applications. In *Proceedings of the DARPA Information Survivability Conference and Exposition*, January 2000.

[61] IONA Technologies Inc. *ORBacus for C++ and Java version 4.1.0*, 2001.

[62] R. Koster, A. P. Black, J. Huang, J. Walpole, and C. Pu. Thread transparency in information flow middleware. In *Proceedings of the International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer Verlag, November 2001.

[63] R. Baldoni, C. Marchetti, A. Termini. Active software replication through a three-tier approach. In *Proceedings of the 22th IEEE International Symposium on Reliable Distributed Systems (SRDS02)*, pages 109–118, Osaka, Japan, October 2002.

[64] Martin Geier, Martin Steckermeier, Ulrich Becker, Franz J. Hauck, Erich Meier, and Uwe Rastofer. Support for mobility and replication in the AspectIX architecture. Technical Report TR-I4-98-05, Univ. of Erlangen-Nuernberg, IMMD IV, 1998.

[65] S. M. Sadjadi and P. K. McKinley. ACT: An adaptive CORBA template to support unanticipated adaptation. In *Proceedings of the 24th IEEE International Conference on Distributed Computing Systems (ICDCS)*, Tokyo, Japan, March 2004.

[66] Common Lisp Object System. http://www.dreamsongs.com/CLOS.html, accessed November 2005.

[67] Python Programming Language. http://www.python.org/, accessed November 2005.

[68] Ian Welch and Robert J. Stroud. Kava - a reflective java based on bytecode rewriting. In *Proceedings of the 1st OOPSLA Workshop on Reflection and Software Engineering*, pages 155–167, London, UK, 2000. Springer-Verlag.

[69] S. M. Sadjadi, P. K. McKinley, B. H. C. Cheng, and R. E. K. Stirewalt. TRAP/J: Transparent generation of adaptable java programs. In *Proceedings of the 2004 International Symposium on Distributed Objects and Applications*, Agia Napa, Cyprus, October 2004.

[70] Eric Wohlstadter, Stoney Jackson, and Premkumar T. Devanbu. DADO: Enhancing middleware to support crosscutting features in distributed, heterogeneous systems. In *International Conference on Software Engineering ICSE*, pages 174–186, 2003.

[71] P. David, T. Ledoux, and M. Bouraqadi-Saadani. Two-step weaving with reflection using AspectJ. In *Proceedings of the OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, 2001.

[72] E. Tanter, J. Noye, D. Caromel, and P. Cointe. Partial behavioral reflection: spatial and temporal selection of reification. In *Proceedings of the 18th ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications (OOPSLA 2003)*, pages 27–46, Anaheim, California, 2003. ACM Press.

[73] Power aware distributed systems. `http://pads.east.isi.edu/`.

[74] V. Raghunathan, C. Pereira, M. B. Srivastava, and R. Gupta. Energy aware wireless systems with adaptive power-fidelity tradeoffs. *IEEE Transactions on VLSI Systems*, February 2005.

[75] Jan-Peter Richter and Hermann de Meer. Towards formal semantics for QoS support. In *Proceedings of INFOCOM 1998.*

[76] Cristian Koliver, Klara Nahrstedt, Jean-Marie Farines, Joni da Silva Fraga, and Sandra Aparecida Sandri. Specification, mapping and control for QoS adaptation. *Real-Time Systems*, 23(1-2), 2002.

[77] Tarek F. Abdelzaher and Kang G. Shin. QoS provisioning with qContracts in web and multimedia servers. In *Proceedings of the IEEE Real-Time Systems Symposium*, New York,USA, December 1999.

[78] Tarek Abdelzaher, Kang G. Shin, and Nina Bhatti. User-level QoS-adaptive resource management in server end-systems. *IEEE Transactions on Computers*, 52(5), 2003.

[79] Eric Wohlstadter, Stefan Tai, Thomas Mikalsen, Isabelle Rouvellou, and Premkumar Devanbu. GlueQoS: Middleware to sweeten quality-of-service policy interactions. In *Proceedings of the 26th International Conference on Software Engineering*, pages 189–199. IEEE Computer Society, 2004.

[80] Baochun Li and Klara Nahrstedt. Dynamic reconfiguration for complex multimedia applications. In *Proceedings of IEEE International Conference on Multimedia Computing and Systems*, Florence, Italy, June 1999.

[81] E. Gelenbe, M. Gellman, and Pu Su. Self-awareness and adaptivity for QoS. In *Proceedings of IEEE International Symposium on Computers and Communication*, June 2003.

[82] R.R.-F. Liao and A.T. Campbell. A utility-based approach for quantitative adaptation in wireless packet networks. *ACM Journal on Wireless Networks*, 7(5), 2001.

[83] Baochun Li, Dongyan Xu, Klara Nahrstedt, and Jane W.S. Liu. End-to-end QoS support for adaptive applications over the internet. In *SPIE Proceedings on Internet Routing and Quality of Service*, November 1998.

[84] Stefan Fischer, Abdelhakim Hafid, Gregor von Bochmann, and Hermann de Meer. Cooperative QoS management for multimedia applications. In *Proceedings of IEEE International Conference on Multimedia Computing and Systems*, 1997.

[85] Bobby Vandalore, Raj Jain, Sonia Fahmy, and Sudhir Dixit. QuaFWiN: Adaptive QoS framework for multimedia in wireless networks and its comparison with other QoS frameworks. In *Proceedings of the 24th IEEE Conference on Local Computer Networks (LCN)*, October 1999.

[86] Baochun Li, Dongyan Xu, and Klara Nahrstedt. Towards integrated runtime solutions in QoS-aware middleware. In *Proceedings of ACM Multimedia Middleware Workshop*, Ottawa, Canada, 2001.

[87] M. Shaw and D. Garlan, editors. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1989.

[88] G. S. Blair, L. Blair, V. Issarny, P. Tuma, and A. Zarras. The role of software architecture in constraining adaptation in component-based middleware platforms. In *Proceedings of the 2nd International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, New York, April 2000.

[89] Ziyang Duan, Arthur Bernstein, Philip Lewis, and Shiyong Lu. A model for abstract process specification, verification and composition. In *Proceedings of the Second International Conference on Service Oriented Computing (ICSOC)*, New York, November 2004.

[90] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

[91] Anand R. Tripathi, Tanvir Ahmed, Richa Kumar, and Shremattie Jaman. Design of a policy-driven middleware for secure distributed collaboration. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*. IEEE Computer Society, 2002.

[92] B. N. Jorgensen, E. Truyen, F. Matthijs, and W. Joosen. Customization of object request brokers by application specific policies. In *Proceedings of the 2nd International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, New York, April 2000.

[93] Romain Rouvoy and Philippe Merle. Abstraction of transaction demarcation in component-oriented platforms. In *Proceedings of the fourth ACM/IFIP/USENIX International Middleware Conference (Middleware'2003)*, Rio de Janeiro, Brazil, June 2003.

[94] H. Gimpel, H. Ludwig, A. Dan, and B. Kearney. PANDA: Specifying policies for automated negotiations of service contracts. In *Proceedings of the 1st International Conference on Service Oriented Computing (ICSOC)*, Trento, Italy, December 2003.

[95] Svend Frolund and Jari Koisten. QML: A language for quality of service specification. Technical report, HP Laboratories, Palo Alto, 1998.

[96] Xiaohui Gu, Klara Nahrstedt, Wanghong Yuan, Duangdao Wichadakul, and Dongyan Xu. An XML-based quality of service enabling language for the web. Technical report, Department of Computer Science University of Illinois at Urbana-Champaign, Urbana, April 2001.

[97] Orlando Loques, Alexandre Sztajnberg, Romulo Curty Cerqueira, and Sidney Ansaloni. A contract-based approach to describe and deploy non-functional adaptations in software architectures. *Journal of the Brazilian Computer Society*, 2004.

[98] Nicolas Le Sommer and F. Guidec. A contract-based approach of resource-constrained software deployment. In *Proceedings of the IFIP/ACM Working Conference on Component Deployment (CD 2002)*, pages 15–30, London, UK, 2002. Springer-Verlag.

[99] Arun Mukhija and Martin Glinz. CASA - a contract-based adaptive software architecture framework. In *Proceedings of the 3rd Workshop on Applications and Services in Wireless Networks*, pages 275–286, 2003.

[100] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making components contract aware. *IEEE Computer*, 32(7):38–45, 1999.

[101] Bertrand Meyer. Applying 'Design by Contract'. *IEEE Computer*, 25(10):40–51, 1992.

[102] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transaction of Software Engineering Methodolgy*, 6(3):213–249, 1997.

[103] Jeff Magee and Jeff Kramer. Dynamic structure in software architectures. In *Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering*, pages 3–14. ACM Press, 1996.

[104] Eric M. Dashofy, Andr Van der Hoek, and Richard N. Taylor. A highly-extensible, xml-based architecture description language. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, page 103. IEEE Computer Society, 2001.

[105] K. Appleby, S. B. Calo, J. R. Giles, and K.-W. Lee. Policy-based automated provisioning. *IBM Systems Journal*, 43(1), 2004.

[106] Eiffel Software. http://www.eiffel.com/, accessed July 2004.

[107] L. Andrade and J. Fiadeiro. Evolution by contract. In *ECOOP'00 Workshop on Object-Oriented Architectural Evolution*, 2000.

[108] L. Rizzo. Effective erasure codes for reliable computer communication protocols. *ACM Computer Communication Review*, April 1997.

[109] Jutta Degener and Carsten Bormann. The GSM 06.10 lossy speech compression library and its applications, 2000. available at http://kbs.cs.tu-berlin.de/ jutta/toast.html.

[110] Christine E. Jones, Krishna M. Sivalingam, Prathima Agrawal, and Jyh-Cheng Chen. A survey of energy efficient network protocols for wireless networks. *Wireless Networks*, 7(4):343–358, 2001.

[111] Krishna M. Sivalingam, Jyh-Cheng Chen, Prathima Agrawal, and Mani B. Srivastava. Design and analysis of low-power access protocols for wireless and mobile ATM networks. *Wireless Networks*, 6(1):73–87, 2000.

[112] B. Burns and J.-P. Ebert. Power consumption, throughput and packet error measurements of an IEEE 802.11 WLAN interface. Technical report, Telecommunication Networks Group, Technische University Berlin, August 2001.

[113] Suresh Singh and C. S. Raghavendra. PAMAS: power aware multi-access protocol with signaling for ad hoc networks. *ACM SIGCOMM Computer Communication Review*, 28(3):5–26, 1998.

[114] W. Ye, J. Heidemann, and D. Estrin. An energy-efficient MAC protocol for wireless sensor networks. In *Proceedings of the INFOCOM 2002*, 2002.

[115] Bob O'Hara and Al Petrick, editors. *The IEEE 802.11 Handbook: A Designer's Companion*. Standards Information Network and IEEE Press, January 2000.

[116] Tijs van Dam and Koen Langendoen. An adaptive energy-efficient MAC protocol for wireless sensor networks. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, pages 171–180. ACM Press, 2003.

[117] P. Havinga and G. Smit. Energy-efficient TDMA medium access control protocol scheduling. In *Asian International Mobile Computing Conference (AMOC 2000)*, pages 1–9, 2000.

[118] Rajgopal Kannan, Ram Kalidindi, S. S. Iyengar, and Vijay Kumar. Energy and rate based MAC protocol for wireless sensor networks. *ACM SIGMOD Record*, 32(4):60–65, 2003.

[119] D. Xu, B. Li, and K. Nahrstedt. QoS-Directed error control of video multicast in wireless networks. Technical Report Computer Science Dept., UIUC, August 1999.

[120] Dongyan Xu, Baochun Li, Klara Nahrstedt, and Jane W.-S. Liu. Providing seamless QoS for multimedia multicast in wireless packet networks. In *Proceedings of SPIE Multimedia Systems and Applications*, pages 352–361, Boston, MA, USA, 1999.

[121] X. Xu, A. Myers, H. Zhang, and R. Yavatkar. Resilient multicast support for continuous-media applications. In *Proceedings International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, St. Louis, Missouri, May 1997.

[122] N. Maxemchuk, K. Padmanabhan, and S. Lo. A cooperative packet recovery protocol for multicast video. In *Proceedings International Conference on Network Protocols*, October 1997.

[123] Bert J. Dempsey, Jorg Liebeherr, and Alfred C. Weaver. A new error control scheme for packetized voice over high-speed local area networks. In *Proceedings of the 18th IEEE Local Computer Networks Conference*, pages 91–100, Minneapolis, MN, 1993.

[124] M.Luby, L.Vicisano, J.Gemmell, L.Rizzo, M.Handley, and J.Crowcroft. RFC 3452 Forward Error Correction (FEC) Building Block.

[125] M.Luby, L.Vicisano, J.Gemmell, L.Rizzo, M.Handley, and J.Crowcroft. RFC 3453 The Use of Forward Error Correction (FEC) in Reliable Multicast.

[126] D. Rubenstein, J. Kurose, and D. Towsley. Real-time reliable multicast using proactive forward error correction. Technical Report UM-CS-1998-019, 1998.

[127] M. Podolsky, C. Romer, and S. McCanne. Simulation of FEC-based error control for packet audio on the Internet. In *Proceedings of IEEE INFOCOM'98*, San Francisco, California, March 1998.

[128] Jonathan Rosenberg, Lili Qiu, and Henning Schulzrinne. Integrating packet FEC into adaptive voice playout buffer algorithms on the Internet. In *Proceedings of IEEE INFOCOM 2000*, pages 1705–1714, 2000.

[129] Paul Lettieri, Christina Fragouli, and Mani B. Srivastava. Low power error control for wireless links. In *Proceedings of ACM/IEEE MobiCom'97*, pages 139–150, 1997.

[130] Paul J. M. Havinga. Energy efficiency of error correction on wireless systems. In *Proceedings of the IEEE Wireless Communications and Networking Conference*, September 1999.

[131] A. Nadgir, M. Kandemir, and G. Chen. An access pattern based energy management strategy for instruction caches. In *Proceedings of 2003 IEEE International SOC Conference*, Portland, Oregon, 2003.

[132] D. Duarte, N. Vijaykrishnan, M. J. Irwin, and Y.F. Tsai. Impact of technology scaling and packaging on dynamic voltage scaling techniques. In *Proceedings of the 15th Annual IEEE International ASIC/SOC Conference*, 2002.

[133] A. Vahdat, A. R. Lebeck, and C. S. Ellis. Every joule is precious: A case for revisiting operating system design for energy efficiency. In *Proceedings of the 9th ACM SIGOPS European Workshop*, 2000.

[134] H. Zeng, X. Fan, C. Ellis, A. Lebeck, and A. Vahdat. Ecosystem: Managing energy as a first class operating system resource. In *Proceedings of ASPLOS 2002*, 2002.

[135] H. Zeng, C. Ellis, A. Lebeck, and A. Vahdat. Currentcy: Unifying policies for resource management. In *Proceedings of USENIX 2003 Annual Technical Conference*, 2003.

[136] Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. Information and Control in Gray-box Systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 43–56, 2001.

[137] Manish Anand, Edmund B. Nightingale, and Jason Flinn. Self-tuning wireless network power management. In *Proceedings of the 9th Annual International Conference on Mobile Computing and Networking (MOBICOM '03)*, 2003.

[138] Surendar Chandra and Amin Vahdat. Application-specific network management for energy-aware streaming of popular multimedia formats. In *Proceedings of USENIX Annual Technical Conference*, 2002.

[139] Surendar Chandra, Carla Schlatter Ellis, and Amin Vahdat. Managing the storage and battery resources in an image capture device (digital camera) using dynamic transcoding. In *Proceedings of the Third ACM International Workshop on Wireless and Mobile Multimedia (WoWMoM'00)*, 2000.

[140] Blackdown Project. Java platform 2 version 1.3.x for Linux. available at `http://www.blackdown.com/java-linux/java2-status/jdk1.3-status.html`, 2001.

[141] Joel M. Vincent. iPAQ H3100/H3600/H3700 series Pocket PC battery white paper. Technical report, Compaq Computer Corporation, October 2001.

[142] S. M. Sadjadi, P. K. McKinley, and E. P. Kasten. Architecture and operation of an adaptable communication substrate. In *Proceedings of the Ninth IEEE International Workshop on Future Trends in Distributed Computing*, San Juan, Puerto Rico, May 2003.

[143] E. Kasten, P. K. McKinley, S. Sadjadi, and R. Stirewalt. Separating introspection and intercession in metamorphic distributed systems. In *Proceedings of the IEEE Workshop on Aspect-Oriented Programming for Distributed Computing (with ICDCS'02)*, Vienna, Austia, July 2002.

[144] P. K. McKinley and S. Gaurav. Experimental evaluation of forward error correction on multicast audio streams in wireless LANs. In *Proceedings of ACM Multimedia 2000*, pages 416–418, Los Angeles, California, November 2000.

[145] Jean-Chrysotome Bolot and Andres Vega-Garcia. Control mechanisms for packet audio in Internet. In *Proceedings of IEEE INFOCOM'96*, pages 232–239, San Francisco, California, April 1996.

[146] Philip K. McKinley, Chiping Tang, and Arun P. Mani. A study of adaptive forward error correction for for wireless collaborative computing. *IEEE Transactions on Parallel and Distributed Systems*, September 2002.

[147] E.O. Elliot. Estimates of error rates for codes on burst-noise channels. *Bell System Technology Journal*, 42:1977–1997, September 1963.

[148] David A. Eckhardt and Peter Steenkiste. A trace-based evaluation of adaptive error correction for a wireless local area network. *Mobile Networks and Applications*, 4(4):273–287, 1999.

[149] Yu-Chee Tseng, Chih-Shun Hsu, and Ten-Yueng Hsieh. Power-saving protocols for IEEE 802.11-based multi-hop ad hoc networks. In *Proceedings of the IEEE INFOCOM 2002*, New York, June 2002.

[150] ITU-T Rec. P.862. Perceptual evaluation of speech quality (PESQ): An objective method for end-to-end speech quality assessment of narrow-band telephone networks and speech codecs, February 2001.

[151] Spirent Communications. Using PESQ to test voice quality - white paper, 2002.

[152] Mehmet Aksit and Zièd Choukair. Dynamic, adaptive and reconfigurable systems overview and prospective vision. In *Proceedings of the 23rd International Conference on Distributed Computing Systems Workshops (ICDCSW'03)*, Providence, Rhode Island, May 2003.

[153] G.S. Blair, G. Coulson, L. Blair, H. Duran-Limon, P. Grace, R. Moreira, and N. Parlavantzas. Reflection, self-awareness and self-healing in OpenORB. Charleston, SC, November 2002.

[154] Thorsten Kramp and Rainer Koster. A service-centered approach to QoS-supporting middleware (Work-in-Progress Paper). In *IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, The Lake District, England, September 1998.

[155] Anind K. Dey and Gregory D. Abowd. The Context Toolkit: Aiding the development of context-aware applications. In *Proceedings of the Workshop on Software Engineering for Wearable and Pervasive Computing*, Limerick, Ireland, June 2000.

[156] Eddy Truyen, Bo N. Jörgensen, Wouter Joosen, and Pierre Verbaeten. Aspects for run-time component integration. In *Proceedings of the ECOOP 2000 Workshop on Aspects and Dimensions of Concerns*, Sophia Antipolis and Cannes, France, 2000.

[157] F. Akkai, A. Bader, and T. Elrad. Dynamic weaving for building reconfigurable software systems. In *Proceedings of OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, Tampa Bay, Florida, October 2001.

[158] Z. Yang, B. H.C. Cheng, R. E. K. Stirewalt, J. Sowell, S. M. Sadjadi, and P. K. McKinley. An aspect-oriented approach to dynamic adaptation. In *Proceedings of the ACM SIGSOFT Workshop On Self-healing Software (WOSS'02)*, pages 85–92, November 2002.

[159] S. M. Sadjadi. *Transparent Shaping to Support Adaptation in Pervasive and Autonomic Computing*. PhD thesis, Department of Computer Science and Engineering, Michigan State University, August 2004.

[160] S. M. Sadjadi and P. K. McKinley. Using transparent shaping and web services to support self-management of composite systems. In *Proceedings of the Second IEEE International Conference on Autonomic Computing*, Seattle, Washington, June 2005.

[161] Michiaki Tatsubori, Shigeru Chiba, Kozo Itano, and Marc-Olivier Killijian. OpenJava: A class-based macro system for Java. In *Proceedings of OORaSE*, pages 117–133, 1999.

[162] Jean Charles Fabre and Tanguy Perennou. A metaobject architecture for fault-tolerant distributed systems: The FRIENDS approach. *IEEE Transactions on Computers*, 47(1):78–95, 1998.

[163] Raymond Klefstad, Douglas C. Schmidt, and Carlos O'Ryan. Towards highly configurable real-time object request brokers. In *Proceedings of the 5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, April - May 2002.

[164] Z. Yang, Z. Zhou, P. K. McKinley, and B. H. C. Cheng. Enabling collaborative adaptation across legacy components. In *Proceedings of the Third Workshop on Reflective and Adaptive Middleware (with Middleware'04)*, Toronto, Ontario, Canada, October 2004.

[165] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003.

[166] Web Services Policy Framework. `http://www-128.ibm.com/developerworks/library/specification/ws-polfram`.

[167] Arun Mukhija and Martin Glinz. CASA - a contract-based adaptive software architecture framework. In *Proceedings of the 3rd IEEE Workshop on Applications and Services in Wireless Networks (ASWN 2003)*, Berne, Switzerland, july 2003.

[168] Arun Mukhija and Martin Glinz. A framework for dynamically adaptive applications in a self-organized mobile network environment. In *Proceedings of the 4th International Workshop on Distributed Auto-adaptive and Reconfigurable Systems at the 24th International Conference on Distributed Computing Systems (ICDCS 2004)*, Tokyo, Japan, march 2004.

[169] Arun Mukhija and Martin Glinz. Runtime adaptation of applications through dynamic recomposition of components. In *Proceedings of the 18th International Conference on Architecture of Computing Systems (ARCS 2005)*, Innsbruck, Austria, March 2005.

[170] Scott D. Fleming, Betty H. C. Cheng, R. E. Kurt Stirewalt, and Philip K. McKinley. An approach to implementing dynamic adaptation in c++. In *DEAS '05: Proceedings of the 2005 workshop on Design and evolution of autonomic application software*, pages 1–7, New York, NY, USA, 2005. ACM Press.

[171] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. Design patterns: abstraction and reuse of object-oriented design. In Oscar M. Nierstrasz, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 707, pages 406–431, Berlin, Heidelberg, New York, Tokyo, 1993. Springer-Verlag.

[172] S. M. Sadjadi, P. K. McKinley, and E. P. Kasten. Architecture and operation of an adaptable communication substrate. In *Proceedings of the 9th IEEE International Workshop on Future Trends of Distributed Computing Systems (FTDCS'03)*, pages 46–55, San Juan, Puerto Rico, May 2003.

[173] The Sphinx Project. `http://cmusphinx.sourceforge.net/html/cmusphinx.php`.

[174] The FreeTTS Project. `http://freetts.sourceforge.net/docs/index.php`.

[175] The Microsoft Visual Studio .NET. `http://msdn.microsoft.com/vstudio/`, accessed January 2005. The Microsoft Visual Studio .NET.

[176] ALTOVA XML Spy. `http://www.xmlspy.com/`, accessed January 2005. The XML Spy.

[177] The Jess Project. `http://herzberg.ca.sandia.gov/jess/`.

[178] Ji Zhang, Zhenxiao Yang, Betty H.C. Cheng, and Philip K. McKinley. Adding safeness to dynamic adaptation techniques. In *Proceedings of the ICSE 2004 Workshop on Architecting Dependable Systems*, Edinburgh, Scotland, May 2004.

[179] Chiping Tang and Philip K. McKinley. Modeling multicast packet losses in wireless LANs. In *Proceedings of ACM International Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM'03) (in conjunction with ACM Mobicom)*, San Diego, September 2003.

[180] W. Asprey, et al. Conquer system complexity: Build systems with billions of parts. In *CRA Conference on Grand Research Challenges in Computer Science and Engineering*, pages 29–33, 2002.

[181] Farshad A. Samimi, Philip K. McKinley, and S. Masoud Sadjadi. Mobile Service Clouds: a self-managing infrastructure for autonomic mobile computing services. In *Proceedings of the Second International Workshop on Self-Managed Networks, Systems & Services (SelfMan 2006)*, Dublin, Ireland, June 2006. Springer (LNCS).

[182] Steve R. White, James E. Hanson, Ian Whalley, David M. Chess, and Jeffrey O. Kephart. An architectural approach to autonomic computing. In *Proceedings of the First International Conference on Autonomic Computing (ICAC 2004)*, pages 2–9, 2004.

[183] OASIS SOA Reference Model group. OASIS Reference Model for Service Oriented Architecture V 1.0. Technical report, OASIS, July 2006.

[184] Extensible Markup Language (XML) 1.1. `http://www.w3.org/TR/2004/REC-xml11-20040204/`, accessed July 2004. W3C Recommendation.

[185] Simple Object Access Protocol (SOAP) 1.1. `http://www.w3.org/TR/2000/NOTE-SOAP-20000508/`, accessed July 2004. W3C Note 08.

[186] Web Services Description Language (WSDL) 1.1. `http://www.w3.org/TR/wsdl`, accessed July 2004. W3C Note 15.

[187] UDDI: Universal Description, Discovery and Integration. `http://www.uddi.org/`, accessed July 2004.

[188] The Nixes Tool Set. http://www.aqualab.cs.northwestern.edu/nixes.html, accessed May 2006.

[189] Smart Framework for Object Groups. www.smartfrog.org, accessed May 2006.

[190] Management Software: HP OpenView. http://www.novadigm.com/, accessed May 2006.

[191] S. Vestal. A cursory overview and comparison of four architecture description languages. Technical report, Honeywell Technology Center, February 1993.

[192] D. Garlan, R. Monroe, and D. Wile. ACME: An architectural interconnection language. Technical report, CMU-CS-95-219, Carnegie Mellon University, 1997.

[193] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering*, pages 336–355, 1995.

[194] S. Vestal. MetaH Programmer's Manual, Version 1.09. Technical report, Honeywell Technology Center, April 1996.

[195] Nenad Medvidovic, Peyman Oreizy, Jason E. Robbins, and Richard N. Taylor. Using object-oriented typing to support architectural design in the c2 style. In *SIGSOFT '96: Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering*, pages 24–32, New York, NY, USA, 1996. ACM Press.

[196] Mary Shaw and David Garlan. Formulations and formalisms in software architecture. In Jan van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *Lecture Notes in Computer Science*, pages 307–323. Springer-Verlag, 1995.

[197] Jeremy S. Bradbury, James R. Cordy, Juergen Dingel, and Michel Wermelinger. A survey of self-management in dynamic software architecture specifications. In *WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, pages 28–33, New York, NY, USA, 2004. ACM Press.

[198] Daniel Le Metayer. Describing software architecture styles using graph grammars. *IEEE Trans. Softw. Eng.*, 24(7):521–533, 1998.

[199] Dan Hirsch, Paolo Inverardi, and Ugo Montanari. Graph grammars and constraint solving for software architecture styles. In *ISAW '98: Proceedings of the third international workshop on Software architecture*, pages 69–72, New York, NY, USA, 1998. ACM Press.

[200] Gabriele Taentzer, Michael Goedicke, and Torsten Meyer. Dynamic change management by distributed graph transformation: Towards configurable distributed systems. In *TAGT'98: Selected papers from the 6th International Workshop on Theory and Application of Graph Transformations*, pages 179–193, London, UK, 2000. Springer-Verlag.

[201] Michel Wermelinger, Ant&#243;nia Lopes, and Jos&#233; Luiz Fiadeiro. A graph based architectural (re)configuration language. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 21–32, New York, NY, USA, 2001. ACM Press.

[202] Michel Wermelinger. A simple description language for dynamic architectures. In *ISAW '98: Proceedings of the third international workshop on Software architecture*, pages 159–162, New York, NY, USA, 1998. ACM Press.

[203] Carlos Canal, Ernesto Pimentel, and José M. Troya. Specification and refinement of dynamic software architectures. In *Software Architecture, TC2 First Working IFIP Conference on Software Architecture (WICSA1)*, pages 107–126, 1999.

[204] Carlos E. Cuesta, Pablo de la Fuente, and Manuel Barrio-Solorzano. Dynamic coordination architecture through the use of reflection. In *SAC '01: Proceedings of the 2001 ACM symposium on Applied computing*, pages 134–140, New York, NY, USA, 2001. ACM Press.

[205] Nazareno Aguirre and Tom Maibaum. A temporal logic approach to the specification of reconfigurable component-based systems. In *ASE '02: Proceedings of the 17th IEEE international conference on Automated software engineering*, page 271, Washington, DC, USA, 2002. IEEE Computer Society.

[206] Virginia C. de Paula, G. R. Ribeiro Justo, and P. R. F. Cunha. Specifying dynamic distributed software architectures. In *Proceedings of XII Brazilian Symposium on Software Engineering*, 1998.

[207] XUI Rich Client Framework. http://xui.sourceforge.net/, accessed June 2006.

[208] Zhinan Zhou and Philip K. McKinley. COCA: A contract-based infrastructure for composing adaptive multimedia systems. In *Proceedings of the 8th International Workshop on Multimedia Network Systems and Applications (MNSA 2006)*, Lisboa, Portugal, July 2006. to appear.

[209] Fei Yu, Vincent W.S. Wong, and Victor C.M. Leung. Efficient QoS provisioning for adaptive multimedia in mobile communication networks by reinforcement learning. In *Proceedings of the SPIE/ACM Multimedia Computing and Networking Conference (MMCN'04)*, 2004.