

# A Framework for Application Adaptation in Mobile Computing Environments

Vaduvur Bharghavan    Vijay Gupta  
Coordinated Sciences Laboratory  
University of Illinois at Urbana-Champaign  
Urbana, IL - 61801

## Abstract

*State-of-the-art mobile computing environments must deal with scarce and dynamically varying resources - in particular, the network quality of service. Applications which execute in such environments need to adapt to the dynamic operating conditions in order to preserve the illusion of seamlessness for the end-user as far as possible. This paper proposes a framework for adaptation which provides applications with runtime support for quality of service negotiation, monitoring, and notification. Applications only need to specify the policy of adaptation at a high level, and are shielded from the mechanics of adaptation. This paper describes the design and implementation of the framework of adaptation as a part of the Prayer mobile computing environment.*

## 1 Introduction

Future indoor and outdoor mobile computing environments will need to support applications such as WWW browsers, groupware, and multimedia - which typically require sustained quality of service (QoS) at the relatively low available data rates and scarce networking/computing resources. Unfortunately, state-of-the-art wireless networking is unable to provide sustained QoS to applications for three reasons: unreliable wireless media, user mobility between cells, and migration between diverse wireless networks. In order to provide the illusion of a uniform working environment to the end-user on top of dynamically varying network QoS, applications need to adapt to the changing operating conditions. The case for application-aware adaptation in mobile computing environments has been made in related literature in order to support both the dynamics within a single wireless network, and seamless migration between wireless networks [1, 6-7].

While recent research has proposed several approaches to the problem of application-aware adaptation, the onus of monitoring QoS change and/or reacting to dynamic operating conditions has typically been with the application, thus making it hard to write adap-

tive applications. In order to facilitate writing non-trivial applications in future mobile computing environments, we perceive a need to provide a well-defined structure for adaptation-related interactions between the applications and the operating system. To this end, this paper proposes a framework to provide runtime support for adaptive applications. Our framework consists of three major components: the underlying computing and communication resources, runtime support, and applications. Resource management at the lowest layer provides primitive mechanisms to monitor quality of service (e.g. the network can provide throughput and delay measures over fixed time windows). The runtime support provides the mechanisms to monitor and adapt to changes in the QoS of the underlying resources, while the applications provide the policy of adaptation. The division of functionality along the lines of policy and mechanism allows us to shield applications from the complexity of resource management while retaining the logic to handle the network dynamics at the application level. The focus of this paper is on the framework of interaction between the systems software and the applications in order to enable QoS-aware adaptation in mobile computing environments.

We are implementing the adaptive computing framework as a part of the Prayer mobile computing environment [1]. Section 2 describes the model of the environment. Section 3 describes the framework for adaptive computing in Prayer. Section 4 describes the implementation of the framework. Section 5 discusses the limitations of the current work and suggests improvements for the future.

## 2 Mobile Computing Environment Model

The mobile computing model of Prayer is shown in Figure 1. Every mobile host has a corresponding home on the backbone network. The mobile host can connect to the home through one or more wireless networks, and has the ability to switch between these networks

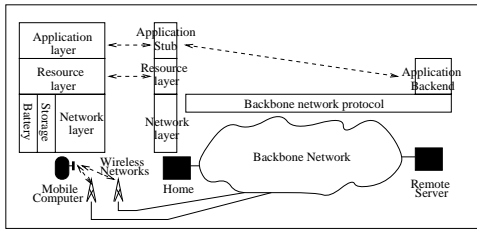


Figure 1: PRAYER Mobile Computing Model

seamlessly. For the distributed applications which run in our environment, we assume a three entity model - application entities run on the *mobile host*, *home*, and *backbone hosts*. For client-server applications, QoS-aware are clients run on the mobiles, QoS-unaware are servers on the hosts, and QoS-aware are application stubs run on the home. By QoS-aware, we mean that the clients and stubs can perform adaptation depending on the changing QoS. Servers may be oblivious to such adaptation. The stubs at the home play a key role, since they act as intermediaries between the client and the server, playing the role of the client with respect to the server, and the server with respect to the client. In Prayer, we typically address the problem of both seamless mobility and adaptive computing with reference to the mobile host and home as the two end-points, our model will work just as well for end-to-end adaptation. This restriction simplifies our prototype since it does not require every host on the backbone network to provide adaptive computing support. Similar application models have been used commonly in recent work [3, 9].

The Prayer model for adaptation consists of three layers (see Figure 1). At the lowest layer are the individual *resources* which are broadly classified into networking resources (channel capacity, buffer, etc.) and computing resources (battery power, memory disk, CPU etc). The intermediate *resource management layer* manages the different underlying resources, and provides the adaptive computing support to the *application layer*. In this paper, we are primarily concerned with networking related adaptation at the higher two layers. While the focus of this paper is adaptation to dynamic networking QoS, our framework for adaptation at the resource management layer and the application layer can support adaptation with respect to other computing resources and application-defined resources. The key feature of our framework is that *the resource management layer provides the mechanisms to manage, monitor, and react to changes in the available resources, while the application layer provides the policy for handling dynamic resource change*. The separation of mechanisms from policies enables applications

to control adaptation while not get involved in the mechanics of making adaptation happen.

Two concepts are fundamental to our adaptive computing framework - *QoS classes*, and *adaptation blocks*. We describe each below.

## 2.1 QoS classes

In mobile computing environments, QoS parameters such as throughput, delay and error rates vary dynamically, both due to the inherent nature of wireless uplink/downlink, and because of inter-cell mobility. Thus, in Prayer the applications specify a *range* of acceptable QoS values rather than single values. The application layer and resource management layer negotiate QoS bounds. The resource management layer then performs end-to-end resource reservation (and possibly, advance reservation) according to algorithms described in [4] in order to reserve the minimum required resources and provide additional resources according to the max-min fairness criterion for the network. Providing QoS bounds gives the resource management layer the flexibility to handle apparent resource conflicts caused due to transient loss in the wireless medium or new connection requests by readjusting resources in the network without violating the QoS bounds of ongoing connections.

A QoS class is defined by specifying the upper and lower bounds for resources. Currently, we use the following four parameters: *data rate* (throughput), *delay* (worst-case), *handoff dropping probability*, and *cost per byte*<sup>1</sup>. It is possible for the acceptable range of values for one or more resources to intersect between QoS classes.

## 2.2 Adaptation Blocks

An application divides its execution into adaptation blocks. An adaptation block consists of a set of alternative sequences of execution, each associated with a QoS class. The application is granted a QoS class at the start of an adaptation block, and runs the corresponding sequence of execution expecting to be provided the same QoS class throughout the execution of the block, unless otherwise notified. The application may thus trigger a QoS re-negotiation only at the start of each adaptation block.

Given the concepts of QoS classes and adaptation blocks, two events are of interest with respect to adaptive computing: (a) application-initiated QoS class (re)negotiation at the start of an adaptation block,

<sup>1</sup>In the general case, the resource parameters will include both networking-related QoS parameters, and other resources such as CPU, memory, disk, battery power, etc.

and (b) notification of QoS class violation to the application, which triggers system-initiated QoS class re-negotiation in the middle of an adaptation block. The next section describes the framework to handle both these events.

### 3 Adaptation Framework

The interaction between the application layer and the resource management layer is governed by the two concepts of QoS classes and adaptation blocks. At the start of an adaptation block, the application specifies a sequence of QoS classes in descending order of precedence. The application associates a sequence of execution (encapsulated in a procedure) for each class. The resource management layer grants the application the highest precedence QoS class in can from the sequence; the application then executes the corresponding procedure. The resource management layer tries to provide the same QoS class to the application throughout the adaptation block (using the algorithms specified in [4]). If it is unable to do so (or can provide a higher QoS class), it notifies the application and initiates a QoS class re-negotiation. We now describe the interactions involved in notification and re-negotiation.

#### 3.1 Overview of the Framework

Adaptive computing support for applications is provided through a runtime library, which contains the following calls: `QoS`, `register`, `unregister`, `build_QoS_class_list`, `clear_CAP_sequence`, `add_CAP`, and `retrieve_QoS`. These calls are described in detail in Section 4.

When an application starts, it specifies a set of acceptable QoS classes using the `build_QoS_class_list` call. A QoS class is specified by defining the lower and upper bounds for data rate (in Kbps), delay (in ms), handoff dropping probability, and cost per byte (in cents). Note, that determining the cost per byte is almost impossible in practice. One alternative is to measure cost in terms of available *credits* for a user, where a unit of credit indicates a pre-determined amount of bytes the user is allowed to transmit over the network.

The application registers the set of acceptable QoS classes to the resource management layer using the `register` call. Before the start of an adaptation block, the application specifies the adaptation block by means of a *CAP sequence*. Each CAP entry in the sequence specifies a  $\langle class, action, procedure \rangle$  triplet, where the *class* field denotes the QoS class, the *procedure* field identifies a sequence of execution, and the *action* field specifies the fallback action to be taken if the QoS class is violated during the adaptation block. The last CAP in the CAP sequence must correspond

to the DEFAULT QoS class, which the resource management layer can always satisfy<sup>2</sup>. The *action* field provides the logic to handle QoS notifications during the adaptation block. It consists of the  $\langle low\_action, high\_action \rangle$  pair, where *low\_action* specifies the fallback action if the resource management layer is unable to provide the minimum QoS bounds for the negotiated class, while *high\_action* specifies the fallback action if the resource management layer is able to satisfy a QoS class corresponding to a higher precedence CAP in the CAP sequence. There are six possibilities for the fallback action: BLOCK, BEST\_EFFORT, ABORT, ROLLBACK, SERVER, or an *application-specific handler*. The SERVER action is described in the Section 3.3.

The application starts an adaptation block by making a `QoS` call. In the `QoS` call, the CAP sequence is passed to the resource management layer, which then tries to satisfy a QoS class of the CAPs in sequential order (thus the order of the CAP sequence defines the precedence among the different sequences of execution). When the resource management layer is able to satisfy a QoS class, it notifies the application to execute the corresponding procedure, and starts monitoring the available QoS on behalf of the application.

If the QoS goes below the lower bound of the current QoS class during an adaptation block, the resource management layer will provide the mechanism to react to the QoS change depending on the fallback action specified in *low\_action*: BLOCK will cause the application to suspend execution till the resource management layer is able provide the QoS class; BEST\_EFFORT will ignore the QoS change; ABORT will abort both the current procedure and the current adaptation block; ROLLBACK will abort the current procedure but re-execute the current adaptation block (i.e. redo the `QoS` call). An application specified handler may be used in order to handle notifications in case the application has special needs (e.g. must reach a synchronization point before it can perform any of the standard actions). Also, the application may want to react in different ways depending on which QoS parameter bound was violated - this can be handled in application-specific handlers.

If the QoS value changes during the adaptation block such that the resource management layer is able to provide a QoS class corresponding to a higher precedence CAP in the CAP sequence, the resource management layer will provide the mechanism to react to the QoS change depending on the fallback action speci-

<sup>2</sup>A CAP sequence does not need to contain an entry for every QoS class acceptable to the application.

fied in *high\_action*: BEST\_EFFORT will ignore the QoS change; ROLLBACK will abort the current procedure, and notify the application to execute the procedure corresponding to the higher precedence CAP; ABORT will abort both the current procedure and current adaptation block; BLOCK is ignored (treated the same as BEST\_EFFORT); if an application specific handler is provided, it is invoked.

The QoS call provides a well-defined structure for the interactions between the application and the resource management layer. The QoS call is the only way by which the application can re-negotiate QoS classes. The resource management layer assumes the burden of monitoring the QoS during the program execution, and performs the notification and QoS class re-negotiation based on the fallback action specified by the application. The application thus retains full control of the policy of adaptation, without having to bother with the mechanics of QoS monitoring and re-negotiation.

When an application needs to adapt at a finer grain than a QoS class, it makes a `retrieve_QoS` call, which takes a QoS parameter as input and returns its current value. This enables applications to perform a two-level adaptation - at a coarse grain using adaptation blocks, and at a fine grain using explicit retrieval of the current QoS value.

### 3.2 Notification and Re-negotiation

The resource management layer has the flexibility to reassign resources among competing applications without notifying any of the applications so long as it does not violate a pre-negotiated QoS class. Thus, there are two events that are of importance in the adaptive computing framework: (a) negotiation of the QoS class at the start of an adaptation block using the QoS system call, initiated by the application, and (b) notification of a change in QoS class from the system to the application because the system is either unable to satisfy a pre-negotiated QoS class or can provide a higher QoS class. Note, that when the QoS granted goes below the lower bound, QoS class re-negotiation is always triggered. However, when the QoS that can be granted can be in a higher class, re-negotiation is triggered only if there has been no QoS change within a timeout period. This handles the pathological case of resource fluctuations about the QoS lower bound for a higher class<sup>3</sup>.

### 3.3 End-to-End Adaptation

The adaptation framework described above deals only with the interactions between the applications and the resource management layer at an end-point. It does

not deal with the adaptation-related interactions between two end-points. Two important issues arise when addressing end-point adaptation:

1. An end-to-end consistency mechanism that guarantees that both the end-points have the same QoS classes, and will take consistent actions with respect to adaptation when the QoS class changes.
2. An end-to-end consistency mechanism that guarantees that during the transition from one QoS class to another, there is no interaction between the applications. This ensures that transitory delays in QoS change detection at the two end-points do not cause the program to behave in an undesirable manner.

It is possible for only one end-point to detect QoS class changes. For example, the mobile host may perceive its battery power go below a minimum threshold. Under such circumstances, the mobile host may change QoS class even if its peer on the backbone does not notice a QoS class change.

In a client-server distributed system, one server may serve multiple clients. In this case, the server will potentially be in different QoS classes for different client connections.

In order to handle the above scenarios as well as the issues (1) and (2) above, we stipulate that only one of the two end-points is allowed to initiate the QoS adaptation (typically the client). In each CAP of the CAP sequence, the server specifies a special action, SERVER. The SERVER action is identical to the BEST\_EFFORT action, except that it can receive notification from the remote process (i.e. the client) and change CAP accordingly. When the client side detects a change in QoS class, and initiates a ROLLBACK or ABORT action, the resource management layer at the client notifies the remote side of the QoS class change and specifies the new CAP for the client. When the server receives the notification corresponding to the new CAP, it changes to the corresponding CAP sequence on the server side. Note that this approach assumes that the CAP sequences of the client and server are coordinated. Given this assumption (which is not enforced from within the adaptive computing framework), we can handle both the end-to-end issues raised above.

### 3.4 QoS Class violation

The adaptation framework described above constrains an application to behave in the same way if the QoS bounds for any of the parameter are violated. In general, the exception condition for different QoS parameters may be different. For example, for a WWW Browser, a significant reduction in data rate will disable

<sup>3</sup>The networking layer has its own checks against frequent adaptation due to fluctuating resources [5].

images, while a significant reduction in battery power will terminate the program. We can handle these differences in two ways: (a) specify QoS classes with finer granularity and (b) write customized exception handlers. Neither is a satisfactory solution: providing more QoS classes makes writing adaptive programs harder, and also constrains the types of optimization the network can perform; writing customized exception handlers for every type of QoS violation negates the whole purpose of providing runtime adaptive computing support. While finding a satisfactory solution to this problem is ongoing research, we currently advocate a middle ground using a combination of finer granularity in QoS classes and customized exception handlers.

## 4 Prototype Implementation

We have built a prototype user-level implementation of the adaptive computing framework in *Prayer*. The significant components are the *runtime library*, which provides the adaptation-related calls to applications, and the *resource manager* which implements the functionality of the resource management layer. The lower level network layer provides QoS measures currently, though ongoing implementation work seeks to improve the network resource management significantly by providing loose QoS guarantees [4].

### 4.1 QoS Classes

The *Prayer* environment supports five possible modes of connectivity for the mobile host: LAN (Ethernet: 10Mbps), Wireless LAN (Wavelan and Rangelan: 1Mbps), Slip (100Kbps), Wireless MAN (RAM: 10Kbps), and disconnection. We provide two levels of QoS support - one is simply to identify the current networking interface among the five choices above, and the other is a full fledged QoS class support [5]. While it is possible for the quality of service to vary significantly within the same mode of connection (for example, being connected to the Ethernet does not guarantee high throughput connectivity, particularly for wide area connections), just knowing the interface for the mobile host gives some applications a viable first-cut QoS class.

### 4.2 Resource Manager

The Resource Manager performs QoS class negotiation and monitoring for all the applications registered with it. It listens for application requests at a well known port. Application requests are one of four types: REGISTER, UNREGISTER, BEGIN\_QoS, and END\_QoS.

The resource management layer maintains a list of entries (`application_list`) corresponding to the applications currently registered with it. Each entry contains the application name, process id, a port at which the application can be contacted, and a list of QoS

classes the application recognizes. When the resource management layer gets a REGISTER request, it creates a new entry for the application and initializes the port and the list of QoS classes for the application. Conversely, an UNREGISTER request removes the application entry.

The Resource Manager has a fixed number of statically created threads (called *Resource Servers*) in order to service QoS system calls from applications. A Resource Server is associated with a QoS request rather than an application - once the QoS request is completed, it waits to serve other QoS requests.

When an application makes a QoS call, a BEGIN\_QoS message containing the CAP sequence is sent to the Resource Manager. The Resource Manager invokes one of the free Resource Servers to serve the QoS call, and passes the CAP sequence to it. The Resource Server provides the application with the highest priority QoS class (specified by the CAP sequence) it can, and notifies the application of this QoS class in an EXECUTE message as a response to the BEGIN\_QoS message. At the completion of the adaptation block, the application issues an END\_QoS message (from the QoS library call) to the Resource Manager, which frees up the corresponding Resource Server. During the execution of the adaptation block, the Resource Server monitors the QoS for the application. If it detects a change in QoS (e.g. ETHERNET to WMAN), it reacts depending on the fallback action specified in the CAP corresponding to the granted QoS class. For BEST\_EFFORT, it ignores QoS change; for BLOCK, it sends a SUSPEND notification to the application, followed by a RESUME notification when the QoS class can be granted again; for ABORT, it sends an ABORT notification; for ROLLBACK, it performs QoS class re-negotiation using the CAP sequence, and sends a ROLLBACK notification with the newly granted QoS class; if an application specific handler is provided, it is invoked.

Notifications from the Resource Manager to the application are one of the following: EXECUTE, SUSPEND, RESUME, ABORT, ROLLBACK, and HANDLER. The mechanism of providing notifications to applications is described below.

### 4.3 Runtime Library

The runtime library provides the functionality required by the application to interact with the Resource Manager.

Interactions between the application and Resource Manager are structured as follows. The Resource Manager listens to requests at a well known port. Upon startup (in the `register` call), the application starts

listening at an available port, and sets a signal handler to capture SIGUSR1 signals. The signal handler causes the application to read from the port. Thus, notifications from the Resource Manager to the application are performed by a combination of a SIGUSR1 signal followed by a message to the application port. This implementation was motivated by the fact that the runtime adaptive computing support should be as unintrusive as possible for the application program.

Two global data structures are defined and used by the runtime library: `_QoS_class_list` and `_CAP_sequence`. `_QoS_class_list` is used by the application to specify the set of acceptable QoS classes for the application, while `_CAP_sequence` is used by the application to specify adaptation blocks. We now describe the adaptation-related functions provided by the runtime library

- `build_QoS_class_list(QoS_class_index, lower_bound_rate, upper_bound_rate, lower_bound_delay, upper_bound_delay, lower_bound_hdp, upper_bound_hdp, lower_bound_cost, upper_bound_cost)`

This call creates a new QoS class entry in the `_QoS_class_list` with QoS class index `QoS_class_index`, and the specified lower and upper bounds for data rate, delay, handoff dropping probability and cost.

- `register()`

This call opens a Unix domain datagram port, sets the SIGUSR1 signal handler to handle Resource Manager notifications, and sends a REGISTER message to the Resource Manager with the QoS class information and the communication port.

- `clear_CAP_sequence()`

This call reinitializes the `_CAP_sequence` structure.

- `add_CAP(QoS_class_index, procedure, low_action, high_action)`

This call creates (or updates, if the entry corresponding to the `QoS_class_index` already exists) a CAP entry in the `_CAP_sequence` structure with the QoS class identified by `QoS_class_index`, the adaptation block to execute specified by `procedure`, the action to take if the QoS class falls below the QoS class specified by `low_action`, and the action to take if the QoS class can be increased beyond the QoS class specified by `high_action`. The entry is appended in sequence, to `_CAP_sequence` - thus the order of invoking `add_CAP` when building the CAP sequence is important.

- `QoS()`

This call sends a BEGIN\_QoS message to the Resource Manager with the CAP sequence information, waits for an EXECUTE notification from the Resource Manager, and executes the procedure corresponding to the

granted QoS class. At the end of the procedure, it sends an END\_QoS message to the Resource Manager. During the execution of the procedure, a change in QoS may have caused re-negotiation of the QoS class by the Resource Manager on behalf of the application. The application is then notified through the SIGUSR1 signal. The signal handler reads the communication port, and will receive one of the following messages: SUSPEND (blocks and waits for a RESUME notification), RESUME (resumes procedure execution), ABORT (returns from the QoS call), or ROLLBACK (returns from the current procedure and executes the procedure corresponding to the newly granted QoS class in the CAP sequence).

- `unregister()`

This call sends an UNREGISTER message to the Resource Manager, which removes the application class entry from the Resource Manager.

- `retrieve_QoS(resource)`

This call returns the currently available value of the `resource` to the application. It can be used for adaptation at a finer grain than a QoS class.

In [2], we describe a simple example which uses the above calls in order to perform adaptive consistency management in a file system.

## 5 Conclusion

This paper describes the design and implementation of the framework of adaptation in Prayer. The major benefit of the framework is that it structures the interactions between applications and the runtime support in a way that shields the applications from the mechanics of QoS negotiation and monitoring while retaining the adaptation policy at the application. Consequently, we believe that it becomes easier to write non-trivial adaptive applications in an environment with highly dynamic resources.

We have experimented with a number of adaptive applications in our environment, including adaptive consistency management tools for file systems using PFS, and variable frame-rate video playback and teleconferencing. During the course of experimentation, we discovered several limitations in our current design and implementation. We identify some of the unresolved issues below.

- *Extensible set of QoS parameters* The prototype implementation only provides for adaptation to network QoS. While the wireless network is the biggest bottleneck in outdoor mobile computing, applications may need to adapt to several other scarce resources, including battery power, memory, disk, etc. Besides, the application may also

need to adapt to application-defined and externally managed resources. This motivates having an extensible user-defined set of QoS parameters, and associating a low-layer Resource Manager with each parameter or subset of parameters (such as the Network Manager for data rate, delay, etc.).

- *Measurement for QoS parameters* For several QoS parameters, the unit of measure is hard to define or use in a meaningful manner. For example, cost per byte is very hard to compute since it depends on several factors such as size of the payload, pricing structure of the carrier, etc. However, cost is a major QoS parameter we need to consider for practical use of such a framework. Thus, we are considering several alternative measures to measure cost, including credits.
- *OS support* Though our implementation is currently at the user-level, we perceive a definite need for OS support for making it easier to write and use adaptive applications. In particular, the interprocess communication between the Resource Manager and the runtime libraries (linked in to the application) has unnecessary overhead because we want it to be as less intrusive on the applications as possible. Besides, executing procedures (which do not take any parameters) as a result of the QoS call makes interactions between different adaptation blocks unnecessarily hard. It would be much easier to write applications if the adaptation block is executed in the invoking environment (i.e. it could share the local variables of the calling environment).
- *End-to-end adaptation:* In the case of networking resources, QoS parameters are typically end-to-end. Communicating applications can thus cooperatively perform end-to-end adaptation based on a shared knowledge of available network QoS. However, adaptation involving local resources such as disk, memory or battery power may still require both end-points to adapt at the application layer. Notification of remote adaptation and the policy to react to such notifications is an interesting open issue.

We believe that the above issues can be resolved within the overall framework for adaptation proposed in this paper.

## References

- [1] V. Bharghavan. *Challenges and Solutions to Seamless Mobility and Adaptive Computing over Heterogeneous Wireless Networks*, **International Journal on Wireless Personal Communications**, Special Issue on Wireless and Mobile Computing, March 1997.
- [2] D. Dwyer and V. Bharghavan. *A Mobility Aware File System for Partially Connected Operation*, *ACM Operating Systems Review*, January 1997.
- [3] A. Fox, S. Gribble, E. Brewer, and E. Amir. *Adapting to Network and Client Variation via On-Demand Dynamic Distillation*, **Proceedings of ASPLOS-VII**, Boston, Massachusetts, October 1996.
- [4] S. Lu and V. Bharghavan. *Resource Management in Mobile Computing Environments*, **Proceedings of ACM SIGCOMM**, 1996.
- [5] S. Lu, K-w. Lee, and V. Bharghavan. *Adaptive Service in Mobile Computing Environments*, *Proceedings of the International Workshop on Quality of Service*, New York, NY, May 1997.
- [6] B. Noble, M. Price, and M. Satyanarayanan. *A Programming Interface for Application-Aware Adaptation in Mobile Computing*, **Computing Systems**, Volume 8, Number 5, 1995.
- [7] B. Schilit, R. Want, and N. Adams. *Context Aware Computing Applications*, **Proceedings of the Workshop on Mobile Computing Systems and Applications**, Santa Cruz, California, December 1994.
- [8] M. Satyanarayanan, B. Noble, P. Kumar, and M. Price. *Application-Aware Adaptation for Mobile Computing*, **Operating Systems Review**, Volume 29, Number 1, 1995.
- [9] B. Zenel and D. Duchamp. *Intelligent Communication Filtering for Limited Bandwidth Environments*, **Proceedings of the fifth Workshop on Hot Topics in Operating Systems**, Rosario, Washington, May 1995.