

UNIVERSITY OF CALIFORNIA

Los Angeles

Framework for Developing Adaptable Applications in Pervasive Environments

A thesis submitted in partial satisfaction  
of the requirements for the degree Master of Science  
in Computer Science

by

Shantanu Bhattacharyya

2003

© Copyright by  
Shantanu Bhattacharyya  
2003

The thesis of Shantanu Bhattacharyya is approved.

---

Richard Muntz

---

Stott Parker

---

Rajive Bagrodia, Committee Chair

---

Peter Reiher

University of California, Los Angeles

2003

## Table of Contents

|  |    |
|--|----|
| Chapter 1 Introduction .....   | 1  |
| Chapter 2 Programming Framework .....  | 6  |
| 2.1 Programming Paradigm .....   | 6  |
| 2.2 Environment Characterization .....   | 8  |
| 2.3 Adaptation: Evolution of Program Behavior .....                            | 11 |
| Chapter 3 Programming Model .....  | 14 |
| 3.1 Programming Constructs .....   | 14 |
| Chapter 4 System Design.....   | 19 |
| 4.1 Programming Language Grammar and Parser.....                               | 21 |
| 4.2 Runtime Engine: <i>Agents, Environment Object</i> Providers.....           | 33 |
| 4.3 Runtime Engine: <i>Decision System</i> .....                               | 33 |
| Chapter 5 Experiments.....   | 40 |
| 5.1 Validation – Time Trials.....  | 40 |
| 5.2 Validation – Correctness: Decision Scenarios.....                          | 45 |
| 5.3 Sample Application: Modified Pooka Java Mail Client and JAMES Server ..... | 50 |
| 5.4 Validation – Application Adaptation vs. Standard Application Behavior..... | 52 |
| Related Work .....   | 60 |
| Chapter 6 Conclusions .....  | 67 |
| REFERENCES .....   | 70 |

## List of Figures

|   |    |
|---|----|
| Figure 3.1: Example <i>Alternative Set</i> for <i>algorithmic change</i> .....        | 14 |
| Figure 3.2: Example <i>Alternative Set</i> for <i>multi-fidelity algorithms</i> ..... | 16 |
| Figure 4.1: Program View: Sample Voice Messaging Application.....                     | 19 |
| Figure 4.2: statement Grammar .....   | 22 |
| Figure 4.3: include .....   | 22 |
| Figure 4.4: <i>Environment Object Type</i> .....                                      | 24 |
| Figure 4.5: <i>Enumerated List</i> .....  | 24 |
| Figure 4.6: <i>Ordered List</i> .....   | 24 |
| Figure 4.7: <i>Unordered List</i> .....   | 24 |
| Figure 4.8: <i>Environment Object Layer Operations</i> .....                          | 25 |
| Figure 4.9: Maximum Function Definition.....  | 26 |
| Figure 4.10: <i>Layer types</i> .....   | 27 |
| Figure 4.11: Standard Library Function Layer Mapping.....                             | 27 |
| Figure 4.12: <i>Clause Statement</i> .....  | 28 |
| Figure 4.13: General Clausal Operators.....   | 28 |
| Figure 4.14: Unordered List Clausal Operators.....                                    | 29 |
| Figure 4.15: <i>Macros</i> .....  | 29 |
| Figure 4.16: <i>Correlation</i> .....   | 29 |
| Figure 4.17: <i>Programmer Comparator</i> .....                                       | 30 |
| Figure 4.18: <i>User Comparator</i> .....   | 31 |

|  |    |
|--|----|
| Figure 4.19: <i>Alternative Set</i> .....  | 32 |
| Figure 4.20: <i>Decision System APIs</i> .....   | 34 |
| Figure 5.1: Experiment #1 Decision & Parse Time vs. # of <i>Alternatives / Alternative Set</i><br>.....                  | 41 |
| Figure 5.2: Experiment #2 Decision & Parsing Time vs. # of <i>Alternatives / Alternative Set</i><br>.....                | 42 |
| Figure 5.3: Experiment #3 Decision and Parsing Time vs. # of <i>Clauses / Alternative</i> ....                           | 43 |
| Figure 5.4: Experiment #4 Decision and Parsing Time vs. # of <i>Clauses / Alternative</i> ....                           | 44 |
| Figure 5.5: Pooka Mail Client Displaying a Message of Type Image.....  | 51 |
| Figure 5.6: Experiment #5 Pooka Mail Client Voice Message Download Times.....  | 53 |
| Figure 5.7: Experiment #6 Pooka Mail Client Voice Message Download Times with<br>Server-side Adaptation.....             | 54 |
| Figure 5.8: Experiment #7 Pooka Mail Client Image Message Download Times .....   | 55 |
| Figure 5.9: Experiment #8 Pooka Mail Client Image Message Download Times with<br>Server-side Adaptation.....             | 56 |
| Figure 5.10: Experiment #9 Fidelity Biased Pooka Mail Client Voice Download Times<br>with Server-side Adaptation .....   | 57 |
| Figure 5.11: Experiment #10 Bandwidth Biased Pooka Mail Client Voice Download<br>Times with Server-side Adaptation ..... | 58 |

## List of Tables

|  |    |
|--|----|
| Table 2.1: Environment Object.....   | 10 |
| Table 4.1: Example <i>Programmer Comparator</i> .....  | 35 |
| Table 4.2: <i>User Comparator</i> .....  | 36 |
| Table 4.3: Point Allocations based on <i>User Comparator</i> .....                                   | 37 |
| Table 4.4: <i>Correlation</i> Matrix with 3 Sample <i>Environment Objects</i> .....                  | 37 |
| Table 4.5: <i>User Comparator and Correlation Matrix</i> .....                                       | 38 |
| Table 4.6: Subdivision of <i>Environment Object</i> Points to <i>Alternatives</i> .....              | 38 |
| Table 5.1: Scenario #1 <i>Programmer Comparator</i> .....  | 45 |
| Table 5.2: Scenario #1 <i>User Comparator</i> .....  | 46 |
| Table 5.3: Scenario #1 Point Allocations based on <i>User Comparator</i> .....                       | 47 |
| Table 5.4: Scenario #1 <i>Correlation</i> Matrix with 3 Sample <i>Environment Objects</i> .....      | 47 |
| Table 5.5: Scenario #1 Subdivision of <i>Environment Object</i> Points to <i>Alternatives</i> .....  | 47 |
| Table 5.6: Scenario #2 <i>Programmer Comparator</i> .....  | 48 |
| Table 5.7: Scenario #2 <i>User Comparator</i> .....  | 48 |
| Table 5.8: Scenario #2 Point Allocations based on <i>User Comparator</i> .....                       | 49 |
| Table 5.9: Scenario #2 <i>Correlation</i> Matrix with 3 Sample <i>Environment Objects</i> .....      | 49 |
| Table 5.10: Scenario #2 Subdivision of <i>Environment Object</i> Points to <i>Alternatives</i> ..... | 49 |

## Acknowledgments

I would like to thank my advisor, Rajive Bagrodia, for allowing me the freedom to explore topics until I found my calling. His guidance, wisdom, and knowledge has been quintessential in composing the body of this work. I would like to thank Richard Guy, my supervisor, friend and consummate professional who always kept me on my feet and provided much needed advice and encouragement over the years. I would like to thank Maneesh Varshney and the other students in the iMASH research lab for all the poking, prodding and aid they have provided. They really made these past few years a life experience. Finally, I would like to thank my parents. They encouraged my vision and drive from day one and have always been there for me. Their support has really allowed me to become the person I am today and has facilitated my academic accomplishments and dreams.

This work was supported by the National Science Foundation Contract ANI-9986679.

## ABSTRACT OF THE THESIS

Framework for Developing Adaptable Applications in Pervasive Environments

by

Shantanu Bhattacharyya

Master of Science in Computer Science

University of California, Los Angeles, 2003

Professor Rajive Bagrodia, Chair

The past few years have seen a dramatic drive for mobile and context aware applications. This field brings with it a drastically different set of computational constraints, and application behaviors. Traditional programming techniques and tools are not sufficient in addressing the demands of this new computational domain, as they fail to capture its sheer dynamism, the relationship between applications and devices in the surrounding computational environment, and the regularity of faults wrought by mobility.

The purpose of this work is to present a framework in which computational environmental conditions play a major role in affecting application behavior. This framework includes a programming language model which provides programming constructs and an application programming interface for realizing changes in the

environment and facilitating application adaptation; a decision and reasoning system which combines programmer, user, and device input in deciding what and how adaptations should take place; a new layered vision of the environment where user and device contexts are composed from lower physical environment characteristics; and system services to capture environmental characteristics and compose the layered environmental representation used within the computational paradigm.

To demonstrate the richness of the application programming interfaces and programming constructs to facilitate application adaptation, an existing email client is augmented to provide a fully-featured voice, image and text messaging service which adapts in the face of varying network, energy and disk conditions. The ease and naturalness of this extension demonstrates the success of this new proposed framework in the field of pervasive computing.

## Chapter 1 Introduction

Consider the life cycle of a typical application. It starts off on a workstation with a high network bandwidth connection. Later, the application moves to a laptop, because the user feels the need to move around and carry the computation with him. Doing so, he moves in and out of different kinds of networks with varying characteristics like bandwidth, jitter, etc... The application moves in and out of different security environments forcing access privileges and security paradigms to evolve with each new environment. As the user's computation is taking place, battery levels reach a critically low level. All of these environmental events occur during the single instance of this application.

The claim is that this feature of an application finding itself in changing environments will only be a typical behavior of the applications of tomorrow. Observing closely one finds there is high heterogeneity of devices and networks present today. Devices range from few pixel screens on 100MHz CPU PDAs to large plasma displays on high end workstations, while network connectivity ranges from kilobit modems to gigabit Ethernets. The situation is further aggravated when users are accustomed to a variety of device characteristics and expect to change from one type to another frequently. The trends of application use are also changing over time. The users want ubiquitous access to data and computation, and show high mobility patterns. Given this emerging trend of application behavior, there is a need to reflect this in the programming model or in

application development techniques, so the applications can cope with changing environments in a seamless and efficient manner. This prerequisite is needed not just for the sake of efficiency but for the survivability of applications in critically limited environments.

Unfortunately, the current programming models and software engineering techniques do not provide for developing applications for heterogeneous devices with a dynamic runtime environment. Historically, environmental considerations have been left on the wayside since applications were developed for environments that did not show much variation. In order to simplify application development, the developer was forced to assume a single environment, which was typically a least common denominator. Later, when the heterogeneity of clients and networks became a serious concern, application behavior started to exhibit basic environmental adaptation, however this adaptation was limited to a predefined set of possible states and the specific adaptation was often decided upon at startup. An example is a video player client which when it starts chooses between different links based on client bandwidth: "modem download", "broadband download" and "T1 download". Thus the application changes its behavior when the networking environments are different, but this is settled at boot up and does not consider any dynamic qualities of the client's link.

Lately, this problem has been identified in the research community which has produced a number of fairly similar approaches. Projects like Odyssey[Noble97],

BARWAN[Brewer98], One.world[Grimm02], PIMA[Banavar02], iMASH[Bagrodia03], etc... have come up with an approach that addresses some of the issues mentioned. A notable similarity of these approaches is that they provide an infrastructure that helps applications adapt while remaining completely outside of the application. That is, there are external entities (proxies in the case of iMASH[Bagrodia03] and BARWAN[Brewer98], viceroys in Odyssey[Noble97]) that monitor the environment, decide when it changes substantially enough to warrant an adaptation and take steps to adapt application data. Most of the time this adaptation takes the form of transcoding data objects from higher fidelity to lower fidelity representations. Unfortunately these approaches cannot leverage internal application data structures or algorithms in choosing the type of adaptation which must take place. For a synergy of infrastructure services and applications where adaptations are suited to application, programmer and user requirements, there is a need for a programming framework which can be used to develop applications targeted for frequently changing environments.

This body of work presents a programming framework which tailors application computation based on user preferences and with changes in environmental conditions. This framework exposes the runtime environment, consisting of device and network resources, processes, users and location information, through programming language constructs and a rich set of programming APIs to the application itself. The application is then free to decide which execution path it should undergo in order to best suit the environment it appears in. The framework presents a new layered view of the

environment where the lowest layer represents physical environment characteristics while higher layers represent aggregations and abstractions of these characteristics. The layered environment model provides the programmer with a number of views into the environment in order to allow them the greatest flexibility in interacting with the environment while providing for the extensibility demanded by the pervasive computation domain. Additionally, should it be required, this programming framework captures the key environmental requirements of a computation and facilitates the exposure of application requirements to infrastructure services which can then perform the appropriate adaptation. The programming framework also ensures that adaptations which are performed by the infrastructure service finally meet application and user fidelity requirements. This approach ensures that applications behave in a manner intended by the application programmer and meet the stringent demands of mobile users in varied environment contexts.

The remainder of the thesis is organized as follows. Chapter 2 discusses a new programming paradigm, application adaptation behaviors and environmental representation. Chapter 3 outlines the programming model and constructs utilized to achieve application adaptation. Chapter 4 relays system components which capture environment information, parse programming model constructs and make adaptation suggestions with application provided adaptation criteria. Chapter 5 examines the use of the programming primitives and systems services detailed in the earlier chapters in a

sample application and provides experimental results. Chapter 6 discusses related work in the field, and finally conclusions are provided in Chapter 8.

## Chapter 2 Programming Framework

### 2.1 Programming Paradigm

The aim of this study is to provide a programming framework suitable for mobile and pervasive computing. To this end we have identified two key features that must be present in this framework:

- The exposure of *environment* to applications through extensible primitives.
- The provision of user, device and programmer provided primitives for applications to *change their behavior*.

Our programming model suggests an explicit programming style for change which contrasts heavily with most of the approaches taken by research so far. The popular approach is to depend on an external system to implicitly tailor application behavior to suit a new environment. The research in the community regarding this system driven change has mainly focused on an intelligent agent making the choice of what is best for the application, leaving the application unaware of the adaptation. We instead propose a far more conservative approach, which involves the programmer's input and intimate knowledge of the application domain.

The battle between implicit and explicit techniques has existed for much of the history of computer science and has surfaced in a number of research areas. One of the classical examples exists in parallel programming. One school of thought advocated the hiding of parallelism from the programmer and wanted a compiler to automatically parallelize the program code. This approach, though convenient for the programmer, was fairly complex to implement and doubts abounded about its potential expressiveness. The alternative approach of explicit parallelism, where a MPI-like library allowed the programmer direct control over how an application would be parallelized, thus gained ground. Today, almost all parallel programming is done with explicit constructs. Similar arguments between an explicit programming API and a system adaptation of application behavior can be envisioned for the pervasive computing field. While a paradigm decreasing programmer burden is desirable, it is fairly difficult to implement, prove and may not be very expressive.

The differences between implicit and explicit programming styles becomes apparent when we consider the distinction between infrastructure services [Bagrodia03][Brewer98][Noble97][Grimm02] and the programming language model suggested so far. An infrastructure approach, with a number of components and protocols, provides services like discovery, proxy connection, distillation of data etc... These services are *external* to the application. The programming model is internal to an application, which among other things would specify how to make use of those services. In a sense the notion of a programming model is orthogonal to infrastructure services,

and in that vein our programming model solution only complements these services rather than replaces them. A programming language technique cannot effectively replicate service discovery or application data transcoding and therefore depends on some underlying infrastructure to take care of these application requirements. Similarly these external services cannot attain fine tune control over the inner-workings of an application. Thus a synergy between the two entities would be ideal, wherein the infrastructure service handles problems like discovery and transcoding while the programming language is responsible for internal application behavior, like tailoring an algorithm best suited for a given environment.

## 2.2 Environment Characterization

A pervasive application over its lifetime can travel over an inordinate number of dissimilar devices and yet is required to produce its prescribed output with an unpredictable set of resources at its disposal. In order to prepare the application programmer for such a scenario, an effective environment characterization must be utilized to best represent the changing requirements of this evolving environment. Additionally the characterization must be extensible in order to effectively represent the evolving environment and satisfy the needs of this pervasive domain.

The environment characterization can be tackled with a two-fold approach. Firstly, the physical features of the environment itself must be represented. These features include

evolving network conditions in the form of bandwidth, jitter and loss or computational resources such as a parallelized architecture and specialized floating-point units. Each physical feature is represented by a unique *environment object* which exports a common interface to provide a single effective way of interacting with the physical environment. These *environment objects* are an essential resource to the application programmer as they first allow them a snapshot of the current execution environment and then provide as primitives to express application requirements in terms of the environment. In order to make interaction with the environment as simple and hassle free as possible, programmers can construct *environment object macros* which define a relation between *environment objects* and can be utilized throughout the program code as if it were an *environment object* itself.

In order to maximize the expressiveness of the *environment object* and provide for extensibility, the *environment object* is layered in design. The lowest layer of the *environment object* contains the physical layer values and device features related to that *environment object*. For example a bandwidth *environment object* would maintain a list of bandwidth values experienced by the device while a soundcard *environment object* would maintain a history of soundcard availabilities. Additionally the soundcard *environment object* would also contain object specific device features such as the ability to play Dolby Digital 5.1 and 32 simultaneous audio streams or the current volume. On top of this layer is the first abstraction layer. Physical layer value aggregations are stored in this layer. These aggregations include minima, maxima, running averages, and a

standard deviation. A second abstraction layer is defined on top of the first which maintains physical layer value trends. These trends include the first and second derivatives or in other words how an *environment object* changes over time and how quickly that change takes place. A summary of the layer descriptions can be viewed below in Table 2.1.

| <b>Layer</b>            | <b>Layer Description</b>                           |
|-------------------------|--|
| First Abstraction Layer | Contains aggregations of physical layer values     |
| Physical Layer          | Contains physical layer values and device features |

Table 2.1: Environment Object

Application Programming Interfaces allow a programmer access into any of the layers of an *environment object* directly. The purpose of the layer design is two-fold. First, it groups operations of the same class or type together. These separate classes of operations allow different views of the essentially flat resource information available to the system, allowing the programmer to choose what operations most naturally suit his computation. Secondly, the layering allows the programmer to extend the operations in a given layer if not directly in terms of operations below it, at least in terms of operations in the same layer. An example of this operational dependency within a layer can be observed in the default second abstraction layer definition, where the second derivative is directly dependent on the first derivative definition. The programmer is free to augment the list of *environment objects* (extending the default *environment object* representation and operations provided and discussed above) by providing new operations specific to a given

*environment object* or generally for all *environment objects* accessed by an application. The programmer can modify the behavior of an existing operation by overloading its definition and the programmer can also specify or remove new *environment objects*. Lastly, *environment object macros* can have their constituent *environment objects* and relations changed dynamically. The manner in which *environment object macros* and *environment objects* are changed is discussed in more detail in Chapter 3 where programming constructs for these entities are defined and later in Chapter 4 where details regarding system components and their interaction with one another are discussed.

### 2.3 Adaptation: Evolution of Program Behavior

When a change in environment occurs, the application programmer is free to change the program behavior in a way that best suits the application domain within the constraints of the new environment. There are four common general techniques for changing programmatic behavior which include *algorithmic change*, *multi-fidelity algorithms*[Narayanan02], *interface change*, and *data-type change*.

*Algorithmic change* is the most fundamental form of environment adaptation and allows the program to tailor its computation units to specific environment domains. An example of such a change can appear in an application which has some search component. In a normal workstation environment the application could run an A\* variant to accomplish its search needs, while in a memory constrained device such as a PDA it would be

impossible to fit the required search space for A\*. In such an instance the application could switch search techniques and use a more suitable search algorithm such as Depth First Search.

*Multi-fidelity algorithms*[Narayanan02] are a special case of algorithmic change where the algorithm itself adapts its behavior based on *tunable*[Narayanan02] parameters. An example of such an adaptation is a sorting algorithm which takes a vector of values to sort and an integer k. The value of the integer k can vary from 1 to the size of the vector and represents the subset of elements that need to be sorted out of the entire vector in order to meet environmental, user or device requirements.

*Interface changes* are also fairly common forms of adaptation which consist of either adapting an application's internal or external I/O to a device's characteristics, specific environmental conditions or user requirements. These changes include tailoring Graphical User Interfaces for specific display characteristics, switching communication protocols because of network or energy considerations, like exchanging a reliable transport protocol (TCP) for a less reliable but lower maintenance protocol (UDP), and adapting internal data objects. Adapting internal data objects are a fairly common form of adaptation often called Content Adaptation or Progressive Adaptation. Examples of this type of a change include fidelity changes in Odyssey[Noble97], and the content adaptation pipeline in iMASH[Bagrodia03]. The basic notion is to adapt the data objects utilized by the application in order to simplify future computation, tailor data objects for

specific platforms and shorten potential network transmission delays for these data objects.

*Data-type changes* represent the simplest form of data object *interface changes* and constitute a special case because of their potential regularity. They consist of elementary changes to a data object's representation like converting a real valued scalar into an integer, or changing the data value of one integer from one scalar to another. These changes tend to be the most basic adaptations for tailoring computation and are most useful on architecturally constrained devices such as PDAs and in energy constrained environments.

Application programmers tend to have the above four methods for changing application's behavior including combinations of the changes. An example application which utilizes a few of the changes is a Maze search application. In a workstation environment, the application can incorporate a heavily graphical display which visually demonstrates the steps taken while running a resource intensive search algorithm like A\*. When the application migrates to or finds itself running on a PDA at some later time, it can incorporate a simpler text GUI which represents the state space that has been traversed and obstacles encountered and can change its search function to a simpler Depth First Search algorithm.

## Chapter 3 Programming Model

### 3.1 Programming Constructs

There are two adaptation constructs we wish to focus on to achieve the adaptation behaviors outlined in Section 2.3. The first adaptation construct is an *alternative*. An *alternative*, like its predecessor the alternative command[Hoare85] from Hoare's CSP, is a point where computation can branch based on the satisfaction of its constituent guards[Hoare85]. A group of *alternatives* which appear as a target of a branch are called an *Alternative Set*. An example of an Alternative Set can be viewed in Figure 3.1.

|  |
|--|
| <p><b>Traditional Code block:</b></p> <pre>Voicemail(args[]) {}</pre> <hr/>  |
| <p><b>Alternative Code block:</b></p> <pre>AlternativeSet &lt;connect&gt; {<br/>    connectVoiceStream(args[]) {}<br/>    connectTextStream(args[]) {}<br/>    connectVoicemailDownload(args[]) {}<br/>}</pre> |

Figure 3.1: Example *Alternative Set* for *algorithmic change*

displays how a traditional voicemail application is replaced by an *Alternative Set* with three distinct *alternatives*. Unlike Hoare's CSP[Hoare85] where the choice between *alternative* commands is nondeterministic, the choices between *alternatives* in an *Alternative Set* are based on environmental factors such as device constraints and user preferences. The application, with the *Alternative Set* construct, is now free stream the voice message from a server, download the voice message from a server and play it back or read a textual representation of the voice message. The benefit here is that now as the application moves from a desktop with a high bandwidth and reliable physical LAN infrastructure to a wireless PDA, using infrastructure service support for application mobility such as iMASH's client handoff[Bagrodia03], the application is free to tailor the computation to its current environment. The PDA may or may not have a speaker or soundcard to process audio, so the application can choose to display text if device constraints prevent it from considering other *alternatives*. Even when audio support is present, an audio option may not always be suitable. One must keep in mind that disk space is limited on a PDA device so the download option will have to be used sparingly. The audio streaming option may also not always be appropriate as energy and computation constraints as well as the user environment play a key role in this *alternative's* suitability. Computation complexity on a device where energy concerns are tantamount determines a very different usage profile than the high fidelity one users are accustomed to in a desktop environment. Consider also a user who wishes to check his or her messages in a conference meeting or a classroom. The device may be physically capable of playing back the user's messages however this behavior may be strictly

undesirable. In this scenario downloading audio for playback at a later time or textual representation of the messages are probably the most appropriate *alternatives*.

*Alternative Sets* are a programmatic construct created to handle *algorithmic changes* and *multi-fidelity algorithm*[Narayanan02] type adaptations. Figure 3.1 displays the *Alternative Set* connect which facilitates an *algorithmic change*. A similar approach can be taken with *multi-fidelity algorithms*[Narayanan02] as seen in Figure 3.2.

```
Traditional Code block:

sort(args[]) {}

-----

Alternative Code block:

AlternativeSet <sort> {
    sort(args[], tunable k::k1) {}
    sort(args[], tunable k::k2) {}
    sort(args[], tunable k::k3) {}
}
```

Figure 3.2: Example *Alternative Set* for *multi-fidelity algorithms*

In this example a regular sorting algorithm is replaced with a *multi-fidelity algorithm*[Narayanan02] which takes a *tunable parameter*[Narayanan02] *k* along with the array of elements to sort. *k* is an enumerated type { *k* = <*k*<sub>1</sub>, *k*<sub>2</sub>, *k*<sub>3</sub>> } and takes on the values *k*<sub>1</sub>, *k*<sub>2</sub> or *k*<sub>3</sub> in this example. The *tunable parameter*[Narayanan02] *k* determines how much of the array to sort. Each of the values *k* can take on <*k*<sub>1</sub>, *k*<sub>2</sub>,

k3> are represented in the *Alternative Set* <sort> as separate alternatives. In this way, just as in the case of *algorithmic changes*, an application is free to choose among the different *alternatives* based on environmental constraints and user preferences.

A second programmatic construct geared toward satisfying *interface* and *data-type changes* is the *Active Interface*. This construct links caller and callee code blocks together and seamlessly allows for the negotiation of data-types, interface types, and fidelity levels. Imagine a scenario where a web-browser application migrates from a desktop environment to a PDA. While on the desktop, the web browser exports a rich and highly graphically intensive interface incorporating advanced web technologies such as Flash and JavaScript. Further imagine that the application was in the middle of processing a request and rendering a web page for the user to interact with when the browser was migrated to its new environment. The application can terminate its connection, recover its state at a suitable semantic point with regards to its new environment, and reprocess the user request. The *Alternative Set* construct could be employed to achieve this functionality. However this is not a clean and natural adaptation behavior. Instead of flushing potentially crucial application state and possibly repeating computation, the application can use an *Active Interface* to adapt its current web objects for an environment where web graphics, Flash and JavaScript are not suitable. The *Active Interface* achieves this task by exporting application semantic primitives, such as fidelity levels and object dependencies to an infrastructure service such as iMASH[Bagrodia03]. iMASH's Content Adaptation Pipeline[Bagrodia03] then performs the necessary

adaptations determined by the application exported primitives, on behalf of the application, and returns the adapted objects and new interface types so the application can resume its computation. The power of this approach over a purely infrastructure driven adaptation strategy ([Noble97], [Bagrodia03]), lies within the exposure of the key semantic information regarding data object and interface characteristics. This way an adaptation is guided by an application and ensures it meets both programmer and user expectations. The notion of *Active Interfaces* is an advanced one that is currently being explored and is treated as future work for all intents and purposes in the remainder of this thesis.

## Chapter 4 System Design

Programs in this new proposed Programming Framework include both traditional code blocks and *Alternative Sets*. *Alternative Sets* as discussed earlier in Chapter 3 are points in the program code where the program flow is allowed to branch to a well defined set of alternatives based on environment characteristics and user preferences. Figure 4.1 displays a sample voicemail application highlighting the program view of the *Alternative Set* model suggested so far.

```
Class VoiceHandler {  
    void connect::voiceStream(server);  
    void connect::downloadVoice(server);  
}  
  
Class TextHandler {  
    void connect::textStream(server);  
}  
  
void main(args[]) {  
    server = getServerName();  
    connect(server);  
}
```

Figure 4.1: Program View: Sample Voice Messaging Application

The *alternatives* are the same ones found in the *Alternative Set* <connect> in Figure 3.1. The programmer is allowed to define *alternatives* in separate application objects as demonstrated in Figure 4.1. The prefix <connect> ties together these functional entities, and establishes these *alternatives* as part of the same *Alternative Set*. This methodology plays to the strengths of object oriented programming by allowing alternatives to become encapsulated in code blocks where they would naturally appear based on their functionality and object dependencies. In Figure 4.1 the sample messaging service has a class definition for voice functionality, where two *alternatives* appear, and a class definition for text messaging functionality where the remaining alternative appears. The main application just calls the *Alternative Set* <connect> expecting that an appropriate *alternative* will be chosen and invoked. As can be seen from this example a compiler component is required to pre-process the *Alternative Set* construct before an imperative language compiler is invoked. And additionally a runtime engine is required to match *alternatives* to their respective *Alternative Sets* and dynamically link the chosen *alternative* in the program flow.

For a runtime system to accurately make a choice between *alternatives* in an *Alternative Set*, three key features need to be exposed by programming language constructs. The first feature is the internal requirements of an *alternative*. These requirements can be seen as space and time complexity, functional dependencies and fidelity requirements of passed data objects. The second feature is the external requirements of an *alternative*. These are requirements on the environment such as “needs bandwidth > 50 Mbps” or represent

device characteristics such as a vector processing unit or a sound card. Both these features will allow the runtime system to determine which *alternatives* are valid for possible invocation. The programming language operator *clause* is defined to tackle both these language feature requirements. The details of the *clause* construct are provided later in the chapter with the rest of the programming language constructs. In addition to these features, a third feature is required to make accurate dynamic choices between *alternatives*. This last feature is a relational feature between *alternatives*. The runtime system must be able to create a total ordering of alternatives with respect to a given adaptation criteria, such as bandwidth for instance. This feature must not only expose which *alternatives* excel with a given computational or environmental constraint but how they compare with other *alternatives* in the *Alternative Set*, in order to find the best *alternative* across all the computational and environmental adaptation criteria captured. The operator *comparator* is defined to tackle this relational feature required to compare *alternatives*. Again the details regarding this operator are provided later in the chapter with the *clausal* operator definitions. A grammar is required to expose these three features from the programming language to the runtime system which is responsible for making *alternative* choices.

#### 4.1 Programming Language Grammar and Parser

Grammar support for *alternatives* requires two types of language features. First, in the base programming language there must be a construct to reflect our notion of *alternatives*

and *Alternative Sets*. To demonstrate the programming framework proposed a Java grammar was modified with the *Alternative Set* construct. A pre-processing compiler was not complete to undertake the responsibility of compiling these constructs, therefore *Alternative Set* constructs reflected in the sample application in Chapter 5 were hand compiled. Second, there must be a grammar which can define *environment objects* and constraints, and user preferences.

```
Statement :=      include
                  | EnvObjProperty
                  | clause
                  | macros
                  | alternativeSet
                  | correlation
                  | programmerComparator
                  | userComparator
```

Figure 4.2: statement Grammar

Figure 4.2 displays the grammar for the top level statements which appear in our decision logic language.

```
include := "use" <EnvironmentObjectName> <EnvironmentObjectType>
```

Figure 4.3: include

The first rule in the decision logic grammar is reflected in Figure 4.3. This rule is used to dynamically add *environment object* definitions for consideration by the runtime system. The *environment object* name is as one would presume, the name given to the environment object, “energy” or “bandwidth” for instance. The *environment object* type, as represented in Figure 4.4, has one of two values, an ordered or unordered list. An ordered list is of the type enumerated list or a list. An enumerated list has constants defined which represent specific run time levels for the environment object. For instance, the *environment object* “energy” could be defined as an enumerated list with the values {‘critical’, ‘low’, ‘average’, ‘high’, <higher>}. The constants can be assigned arbitrary values at *environment object* creation time, “(critical, 10%)”, where 10% represents the battery level measured within the energy *environment object*, or they derive their value from the position they hold in the list, where the first element is assigned the value 0, the second 1, etc... much like a c enumeration type. An enumeration list has a direction which is desirable, this is represented by either a “<higher>” or “<lower>” as in the above example. This direction field allows the programmer the flexibility to define how an *environment object* represents conservation of a certain resource. For example in the case of conserving battery power, higher battery levels are definitely more desirable while in the case of bandwidth preservation, lower bandwidth usage is the desirable behavior. This directionality is inherent and specific to an *environment object* itself and is therefore defined by the programmer. The other choice for ordered list representations of *environment objects* is a list. A list is ordered without any notion of specific constants or bounds defined. A bandwidth *environment object* can be represented as either an

enumerated list or a list. As an enumerated list its representation may look something like {‘low’, ‘medium’, ‘high’} however as a list its current physical values take the form {9.98 Mbps, 73 Mbps, ...}. Again the programmer is free to choose what is easiest and most natural for the *environment objects* they are defining.

```
EnvironmentObjectType := Ordered_list | Unordered_list
```

Figure 4.4: *Environment Object Type*

```
Enumerated_list := <list of elements, direction>
```

Figure 4.5: *Enumerated List*

```
Ordered_list := Enumerated_list | List
```

Figure 4.6: *Ordered List*

```
Unordered_list := Enumerated_list
```

Figure 4.7: *Unordered List*

The other type of environment object is an unordered list. Although most environment objects like bandwidth, security levels, CPU speed are measured physical values where order is important, there are several objects for which order is not important. This can be observed if we encode device type as {‘PDA’, ‘notebook’, ‘desktop’}. Device type has no inherently observable ordering.

```
EnvObjOperation := "define" return_type
property_identifier(environment_object_identifier) as layer layer_name
"{
    statement_block
}"
```

Figure 4.8: *Environment Object Layer Operations*

The next rule in the decision logic grammar reflects the definition of *environment object operations*. The `statement_block` in Figure 4.8 identified within the *environment object operation* entity above corresponds to the body of an *environment object operation*. Statement blocks are first parsed by the parser and then interpreted by Perl which is a fully interpretive language. This allows the *environment object layer* definitions to be extended dynamically during application execution with arbitrary operator complexity. The operations provided within our default *environment objects* are as follows: `value` (current physical layer value), `value_history` (all physical layer values stored in the *environment object* history), `minimum`, `maximum`, `sum`, `average`, `standard deviation`, `derivative` (rate of change), and `second derivative`. A function definition for the maximum function is provided in Figure 4.9. In the statement block, there is a *store* command which informs the parser of an internal variable's type, name and starting value. This command informs the parser to retain the value of this internal variable after function execution completes so that it may be used by this function at its next invocation. Currently the variable types supported are the Java literals `String`, `Double`, and `Integer`. A

variable is only stored the first time the parser encounters the *store* command for it. A function *return* is also provided in order to export computed values back to the runtime system. The return type must be specified in the function header so as not to inhibit the return of variables local to the *environment object* operation definition which have not been stored in the runtime engine. The types supported by the return function, like the supported variable types, are the Java literals String, Double, and Integer.

```
Double maximum(env_obj) as layer 2 {  
    #internal variable storage  
    store Double max 0.0;  
    #perl statements follow  
    temp = value(env_obj);  
    if (max < temp) max = temp;  
    #arguments returned  
    return max;  
}
```

Figure 4.9: Maximum Function Definition

As observed in the definition of the *maximum* function in Figure 4.9, a call to another *environment object operation* is possible as the local maximum is updated by accessing the current *value* of the *environment object* passed. The parser resolves these symbols before passing the statement block to the Perl interpreter. As one can see, the order of *operation* definitions is crucial as the function *value* must be defined before the function *maximum*, to properly resolve dependencies. In order to ensure this, environment object

layers were created as described in section 2.2, which ensure primitive *operations* are defined before higher order and abstract *operations*. Function dependencies can reside between functions in the same layer and it is the job of the programmer to ensure that either these do not occur by defining new layer definitions extending the three provided with the standard toolkit or by ordering the *operation* definitions so that dependencies are resolved top down by the parser. The three layer definitions in an *environment object*, detailed in section 2.2, are as follows: *layer 1* represents physical values or device characteristics key to the *environment object's* operation, *layer 2* represents aggregations of physical layer values, and *layer 3* provides a further aggregation by including physical value trends such as its rate of change.

layer := layer 1 | layer 2 | layer 3 | layer value

Figure 4.10: *Layer* types

Figure 4.11 displays the layers where the library functions provided with the programming toolkit reside.

| <b><i>Provided Library Function Layer Mapping</i></b> |                    |                   |
|---|--------------------|-------------------|
| <b>Layer 1</b>  | <b>Layer 2</b>     | <b>Layer 3</b>    |
| Value   | Average            | Derivative        |
| Value_History   | Maximum            | Second Derivative |
|   | Minimum            |                   |
|   | Standard Deviation |                   |
|   | Sum                |                   |

Figure 4.11: Standard Library Function Layer Mapping

*Clauses*, as mentioned earlier, represent a programming language construct and operator which is used to determine if the current environmental conditions are suitable for a specific alternative. Figure 4.12 displays the grammar for a *clausal* operator. Clauses are defined in terms of *environment object layer operations* and basic comparison operators which are detailed in Figure 4.13. Examples of *clauses* include “(value(bandwidth) > 50Mbps)” and “(exists(soundcard)).” Clauses are always defined such that they evaluate to a Boolean true or false. Therefore clauses can be tied together using the Boolean “AND” and “OR” operators, to represent their entire requirement specifications in terms of *environment objects*. For environment objects whose values are represented as unordered lists, comparison operators such as ‘<’ or ‘>’ hold no meaning therefore they have a more confined notion of clausal operators detailed in Figure 4.14.

```

Clause := (EnvObjOperation(EnvironmentObject) OP
          EnvObjOperation(EnvironmentObject) |
          Scalar |
          NULL)

```

Figure 4.12: *Clause Statement*

```

OP := < '<' | '>' | '<=' | '>=' | '=' | '!' | '!=' >

```

Figure 4.13: *General Clausal Operators*

```

OP := < '=' | '!' | '!=' >

```

Figure 4.14: Unordered List Clausal Operators

As mentioned in section 2.2 *environment object macros* can be defined in order to avoid repetitive definitions of complex environment requirements. As seen in Figure 4.15, *macros* are shortcuts and are resolved by the parser to their *clausal* representations at parse time.

```
macro := clause ANDOR macro | clause
i.e. car = (value (bandwidth) < 57.6Kbps) AND
           (value (device) = PDA) OR
           (value (screen) = SMALL)
```

Figure 4.15: *Macros*

```
Correlation :=
    <EnvironmentObject, EnvironmentObject, Ratio>
```

Figure 4.16: *Correlation*

*Correlations*, as seen in Figure 4.16, are relationships between environment objects and cannot be captured by clausal requirements. An example of a correlation is the relation between the *environment objects* “CPU” and “energy”. Although the definitions of these *environment objects* are separate, the high usage of “CPU” inversely affects the amount of available “energy.” The ratio parameter captures this relationship between *environment objects* through the programmer or from the device environment. The ratio

itself is a scalar between -1 and 1, where -1 indicates an inverse correlation of environment objects, 0 indicates orthogonal environment objects or no correlation, and a ratio of 1 indicates perfect correlation of environment objects. These correlations are often hard to capture, especially at programming time since they intimately involve runtime characteristics and user behavior, therefore in future work the runtime engine will pick up the role of defining these ratios between environment objects based on historical program, user and device behavior.

```
programmerComparator :=  
    EnvironmentObject:  
        <<alternative <identifier>, scalar>,  
        <alternative <identifier>, scalar>,...>
```

Figure 4.17: *Programmer Comparator*

The programmer *comparator* as discussed earlier in the chapter, is the first of two *comparators* used to define relations between *alternatives*. Once clausal bounds checking is complete, *comparators* are used to determine which *alternative* is best suited to run with the available environmental conditions. The programmer comparator in Figure 4.17 is used to rank *alternative* behaviors in terms of the available *environment objects*. For example let us presume a voicestream *alternative* utilizes more “bandwidth” than a textstream *alternative*. This is then reflected using the programmer *comparator* operator. Although a total ordering is required by the runtime system, it may be difficult to relate all alternatives’ relations as they may change based on environment variables or variables

outside the scope of the application such as caching. One of the extensions in future work which we wish to explore is using incomplete *alternative* relations to make *alternative* choices. These choices may become clear with the use of historical application and device behavior and fuzzy logic representations.

```
userComparator :=  
    <<EnvironmentObject, scalar>,  
    <EnvironmentObject, scalar>,...>
```

Figure 4.18: *User Comparator*

The user *comparator* represented in Figure 4.18 is the second *comparator* operator. The user *comparator* is used to order *environment objects* based on user preferences and requirements. For example, a specific application environment may place “energy” as the highest priority for the user while minimizing “bandwidth” utilization while important is clearly second. This runtime encoding of key relations between environment objects is fundamental in adapting an application in a manner intended by the programmer and suitable for the user. This operator combined with the programmer *comparator* allows the runtime system to create a total ordering of *environment objects* and *alternative* behaviors with regard to those objects. This allows the decision system to determine which *alternative* is best suited to meet the current environmental requirements.

```
alternativeSet :=  
    AlternativeSet connect {
```

```

    <statements: macros>

    <programmerComparator>

    alternative <identifier>:
        statements: constraints, requirements;
    }

constraints := clause | macro | constraints ANDOR constraints
requirements := requires(EnvObjOperation(EnvironmentObject))

```

Figure 4.19: *Alternative Set*

Now that the *clausal*, *comparator* and *correlation* operators are defined the *Alternative Set* definition falls out intuitively as seen in Figure 4.19. An *Alternative Set*, defined by a programmer, is composed of macros (which are later used in *alternative* requirements), the programmer *comparator* operator and *alternatives*. Each *alternative* has constraints which are represented by both the *clausal* and *macro* operators and requirements. *Requirements* are defined in terms of *environment object* layers are represent criteria which must be satisfied in order for an *alternative* to be considered in the current execution context. *Requirements* are different from *clausal* bounds in that *clauses* determine the environmental conditions which best suit the *alternative* while *requirements* determine where an alternative can and cannot possibly run. The runtime engine than uses these constructs and the notion of *environment object* defined so far in order to decide between *alternatives*.

## 4.2 Runtime Engine: *Agents, Environment Object* Providers

The runtime system is composed of two entities, a *decision system* and *agent* processes. The *agent* processes act as system processes in that they require increased system privileges in order to access environmental state information on behalf of the application. There is a one to one correspondence between *environment objects* and *agents*, therefore the *decision system* collates the currently available *environment objects* and passes them along to the application and the parsing system. *Agents* take physical layer values and construct *environment object* representations based on definitions specified by the corresponding *environment object* grammar. New *environment object* definitions can be specified by a system programmer however this cannot be done in a dynamic fashion at the moment as it requires a platform and environment entity specific view into the system. In order to provide access to entirely new *environment objects* in a dynamic fashion, the physical layer scripts which make calls into the operating system must be loaded as dynamically linked libraries. This avenue has not been explored yet and is left for future work. The *environment objects* implemented so far include “energy”, “bandwidth”, “storage” (disk space available), and “CPU” (CPU load). All of these values are easily accessible in Linux or through Cygwin under Windows through the */proc* file-system and through system utilities such as “df” and “ps.”

## 4.3 Runtime Engine: *Decision System*

The second part of the runtime engine is the *decision system*. The decision system first collates current environment object values and a list of available *environment objects* for the parsing system and application both of which are provided by the *agent* processes. As seen in the *decision system* APIs provided in Figure 4.20, the *decision system*'s main duty is to decide which alternative should run when provided an *Alternative Set* and the user preferences represented in the *user comparator*.

```
Alternative alt_decision (AlternativeSet as,  
                        userComparator uc);  
EnvironmentObjectList eolist(/*void*/);
```

Figure 4.20: *Decision System* APIs

The *decision system*'s choice of alternative is composed of two phases. First the *decision system* goes through each *alternative*'s clausal constraints and determines from current *environment object* information whether all the clauses are satisfied. *Alternatives* which have clauses that fail are removed from consideration and from herein called *failed alternatives*. An *alternative* which is chosen by the *decision system* is known as the *best alternative*. If for some reason all the *alternatives* in an *Alternative Set* are *failed alternatives*, the *decision system* will consider the possibility of utilizing the *best failed alternative* realizing that there may be suboptimal performance by the alternative. This choice to run a *failed alternative* is considerably better in most cases than terminating an application because of a lack of a suitable *alternative* for a given environment instance. The *decision system* checks for *alternative requirements* once clausal bounds checking is

complete; failure to meet *requirements* unlike *clausal* constraints result in the removal of an *alternative* from invocation consideration. An example of a *requirement* is the existence of a soundcard. If an *alternative* were to require a soundcard, it could not possibly operate without a soundcard's presence.

Once the *decision system* completes *clausal* and *requirement* checks, it enters its second phase, the determination of the *best alternative*. During this phase the *decision system* compares *alternatives* based on their relative environmental requirements. A programmer comparator is provided which determines a total ordering of *alternatives* in an *Alternative Set* with regard to a specific *environment object*. Table 4.1 displays a sample *programmer comparator*. Fidelity is not actually an *environment object* however it is represented as one since it is necessary to compare different types of *alternatives* and possible differences in fidelity values between *alternatives* cannot be exported yet through the *Active Interface* programmatic construct.

| <i>Alternative</i> | <i>Fidelity</i> | <i>Bandwidth</i> | <i>Storage</i> |
|--------------------|-----------------|------------------|----------------|
| Voicestream        | 2               | 1                | 5              |
| Textstream         | 1               | 6                | 8              |
| Download           | 2               | 3                | 2              |

Table 4.1: Example *Programmer Comparator*

Please note the scalars represented in Table 4.1 are meaningless in themselves and only attain value when compared relatively between *alternatives*. The decision system then

looks at the *user comparator*. This comparator represents the relative importance of *environment objects* to the user and as such represents the user’s preferred mode of operation. Like in the case of the *programmer comparator*, the user scores determine a total ordering however the ordering is in terms of *environment objects* instead of *alternatives*. User scores are provided on a scale of 0 to 5 where 0 represents normal consideration, 1 represents average consideration, and 5 represents critical consideration.

A total of 100 points are distributed among *environment objects* considered within an *Alternative Set* based on the user’s relative orderings. A sample *user comparator* is viewable in Table 4.2 and the point allocations based on this sample *comparator* are viewable in Table 4.3. The sample values indicate that high-fidelity application object representations are highly desirable and that network bandwidth is at a premium. The user however is not however highly concerned with the amount of storage utilized by the computation where one can presume that the device has a fairly large disk.

| <i>User Comparator</i> |                  |                |
|------------------------|------------------|----------------|
| <b>Fidelity</b>        | <b>Bandwidth</b> | <b>Storage</b> |
| {critical}             | {critical}       | {average}      |

Table 4.2: *User Comparator*

| <i>Environment Objects</i> |                  |                |
|----------------------------|------------------|----------------|
| <b>Fidelity</b>            | <b>Bandwidth</b> | <b>Storage</b> |
| 45.4545                    | 45.45454545      | 9.09091        |

Table 4.3: Point Allocations based on *User Comparator*

After the points are distributed among the *environment objects*, runtime *correlations* are calculated. These values are currently provided by the programmer however they will eventually be empirically determined based on historical relations between *environment objects* determined at runtime. Correlations are not required, and when a correlation is not provided it assumed to have a ratio of 0 or in other words the pair of environment objects in question is assumed to have an orthogonal relationship.

| <i>Correlations</i> |                 |                  |                |
|---------------------|-----------------|------------------|----------------|
|                     | <b>Fidelity</b> | <b>Bandwidth</b> | <b>Storage</b> |
| <b>Fidelity</b>     | 0               | 0.6              | 0.2            |
| <b>Bandwidth</b>    | 0.6             | 0                | -0.2           |
| <b>Storage</b>      | 0.2             | -0.2             | 0              |

Table 4.4: *Correlation Matrix* with 3 Sample *Environment Objects*

As seen in Table 4.4 above, *correlation* ratios need not be symmetric between environment objects, however for most situations that will be the case. For the purposes of this example, the *correlation* ratios are symmetric. Also note that correlation ratios are only calculated between different *environment objects*. Although an *environment object* should have a 1 to 1 *correlation* with itself, it is recorded as having an orthogonal relation with itself in order to simplify the calculation of influence between *environment objects*.

Now that the correlation values in Table 4.4 are available, they can be combined with the user preference derived point allocations from Table 4.3, to produce Table 4.5.

| <i>Environment Objects</i> |                  |                |
|----------------------------|------------------|----------------|
| <b>Fidelity</b>            | <b>Bandwidth</b> | <b>Storage</b> |
| 74.54545455                | 70.90909091      | 9.09091        |

$\text{NewFidelityValue} = \text{OriginalFidelityValue} + .6(\text{OriginalBWValue}) + .2(\text{OriginalStorageValue})$   
 $\text{NewBWValue} = \text{OriginalBWValue} + .6(\text{OriginalFidelityValue}) - .2(\text{OriginalStorageValue})$   
 $\text{NewStorageValue} = \text{OriginalStorageValue} + .2(\text{OriginalFidelityValue}) - .2(\text{OriginalBWValue})$

Table 4.5: *User Comparator and Correlation Matrix*

The total of the new *user comparator* and *correlation* matrix combined values need not be renormalized down to the 100 point scale, as the ratios between *environment objects* is the important characteristic calculated. We can now subdivide the points allocated to each *environment object* to the *alternatives* in this *Alternative Set* reflecting the *programmer comparator* ratios. This subdivision is reflected in Table 4.6.

| <i>Alternative</i> | <i>Fidelity</i> | <i>Bandwidth</i>   | <i>Storage</i>  |              |
|--------------------|-----------------|--------------------|-----------------|--------------|
|                    | <b>74.54545</b> | <b>70.90909091</b> | <b>9.090909</b> | <b>Total</b> |
| <b>Voicestream</b> | 29.81818        | 7.090909091        | 3.496503        | 40.4056      |
| <b>Textstream</b>  | 14.90909        | 42.54545455        | 5.594406        | 63.049       |
| <b>Download</b>    | 29.81818        | 21.27272727        | 1.398601        | 52.4895      |

Table 4.6: Subdivision of *Environment Object* Points to *Alternatives*

The subdivision of points to each of the alternatives allows for the immediate ascertain of which of the *alternatives* is *best*. Based on the sample values provided, the text stream option wins even though the user is not heavily concerned with storage space on their device. The reason for this is the text stream utilizes the least amount of bandwidth and wins handily in that *environment object*. The download audio *alternative* is a fairly close second as it represents the same fidelity value as the audio stream *alternative* however can operate effectively under lower bandwidth conditions.

This example demonstrates how the decision system effectively utilizes the programming language constructs provided by the programmer and user to determine the *best alternative* out of an *Alternative Set*. Calls to the decision system are currently synchronous, therefore deterministically provides program flow. An asynchronous API to the decision system will be required once *Active Interfaces* are available since a chosen *alternative* may not remain the *best alternative* as fidelity values of data objects or interface types change asynchronously. The details for such an asynchronous role for the decision system are still to be finalized and remain future work.

## Chapter 5 Experiments

The main overheads added to an application which performs adaptation as described by our programming framework, are parsing the *environment object* and *Alternative Set* descriptions and *alternative* selection decision time. The first set of experiments therefore look in detail into what these overheads are based on the number of *alternatives* and *clausal bounds* which appear in an *Alternative Set*. The next concern is how accurate the *alternative* choices are when faced with various application scenarios. We explore in detail the process of decision making and determine if the *alternative* chosen is the correct one based on the environmental requirements and user preferences. Lastly, we augment existing Java mail client and server applications with the programming constructs and decision logic described so far. We explore how our modified application behaves in comparison to the original applications without adaptation capabilities.

All tests were performed on a Pentium IV 2.2 GHz Centrino laptop with 512MB RAM under Windows XP using Cygwin to emulate Linux like services. The Java mail server in the third set of experiments was running on a Pentium IV 1.6GHz desktop with 256MB RAM running Linux 2.4.

### 5.1 Validation – Time Trials

The first experiment determines how much time it takes to parse *Alternative Set* and *environment object* descriptions and make *alternative* decisions.

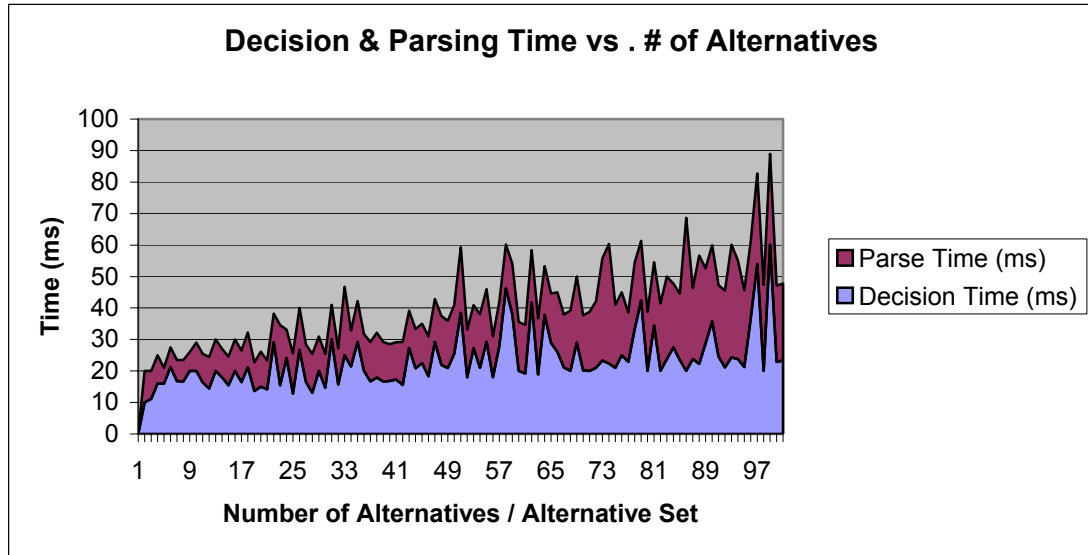


Figure 5.1: Experiment #1 Decision & Parse Time vs. # of *Alternatives / Alternative Set*

Figure 5.1 displays the average decision and parse times for a 1000 randomly generated *Alternative Sets* (each in their own separate trial). Each *Alternative Set* was randomly generated to avoid any bias in the generation of *alternatives* and *clauses*. The number of *alternatives* in the *Alternative Set* was randomly chosen and between 1 and 100. The number of *clausal* bounds for each *alternative* was fixed at 10. Each bound was set to always be satisfied in order to ensure no *alternatives* were eliminated and creating the worst case for decision time. The process of parsing and making decisions was repeated a 1000 times with a new *Alternative Set* generated for each trial. The choice to run only 1 *Alternative Set* per trial was driven by the fact that the decision system only chooses

among *alternatives* in a single *Alternative Set*. Parse times would grow to reflect the additional *Alternative Sets* if more than 1 was provided in each trial however we would not be able to compare parse and decision times with one another effectively.

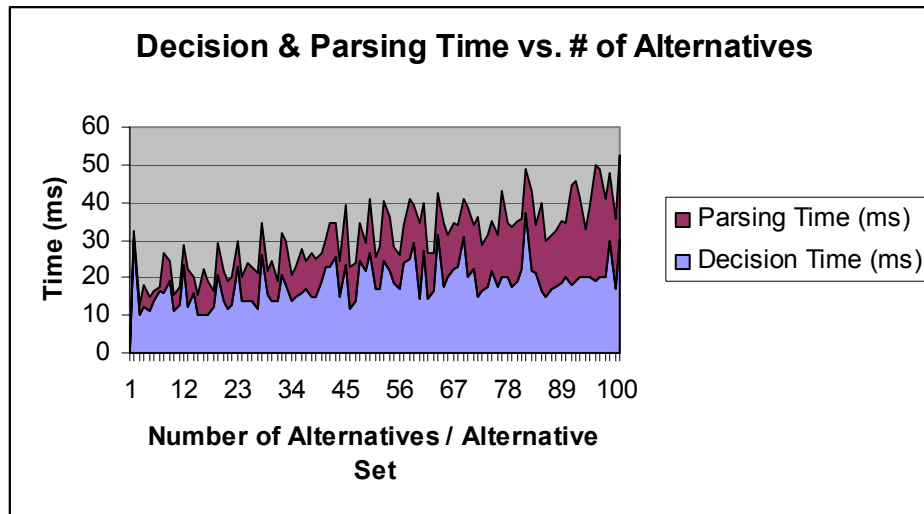


Figure 5.2: Experiment #2 Decision & Parsing Time vs. # of *Alternatives / Alternative Set*

There are a few spikes in decision times which can be attributed to the Java JVM start latencies and Java garbage collection. The difference in JVM start latencies can be observed when we compare Figure 5.1 and Figure 5.2. The experiment run in Figure 5.2 is the same as the first experiment except that the first 15 trials were pruned. The seeds are the same as the first experiment, so we are comparing similar quantities. There appears to be a slight growth in decision time with the number of alternatives and the general trend is close to linear growth. The parse times grow much faster which can be attributed to first parsing all the *environment objects* and *Alternative Set* definitions in

Java and then passing *environment object* layer definitions to the interpretive language Perl for computation. The parser need not be called each and every time a decision is made, depending on the frequency of *Alternative Sets* in the program code. The parser can also be written in a faster compiled language as it is only the language independent programming language constructs in the adaptation language we have provided that are necessary to facilitate adaptation.

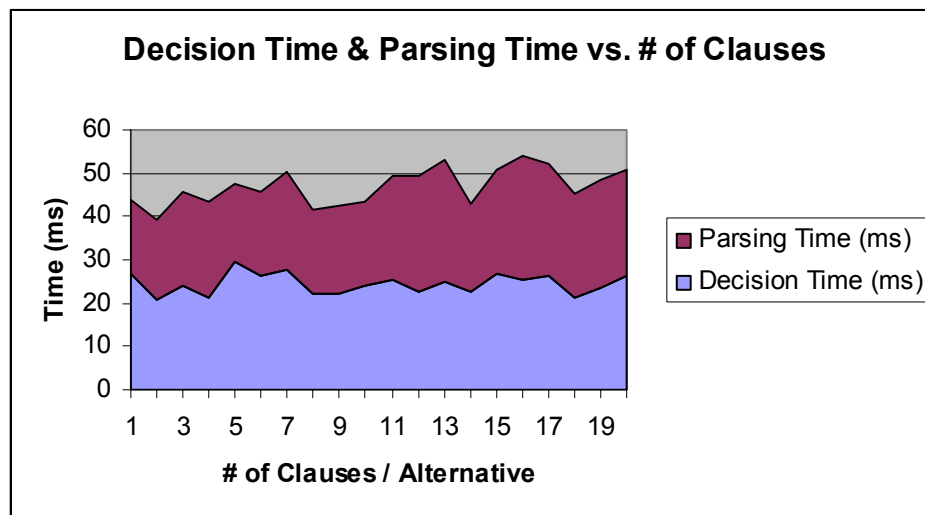


Figure 5.3: Experiment #3 Decision and Parsing Time vs. # of *Clauses / Alternative*

Again we look at decision and parsing time however this time it is in terms of number of *clauses* in each *alternative*. The number of trials in this experiment was a 1015 pruning the first 15 runs as we did in Experiment #2. The number of *alternatives* was fixed at a 100. The number of *clauses* was allowed to vary from 1 to 20. None of the *clauses* were allowed to lead to *failed alternatives*, in order to create the worst case scenario for the

decision system. In this set of trials it appears that decision and parsing time do not really vary drastically with bounds checking, although parse time grows slightly as the number of *clauses* grows to 20. This seems reasonable as each clause is looked at once in both the parser and decision system, and the *clausal* operations are fairly trivial to compute. For the sample applications we have written, 20 *clauses* seems like a fairly reasonable upper bound as it is hard to derive more *environment objects* than these. However in Figure 5.4 we vary the number of clauses between 1 and 100, keep the number of *alternatives* fixed, in order to see if this trend in parse and decision time continues.

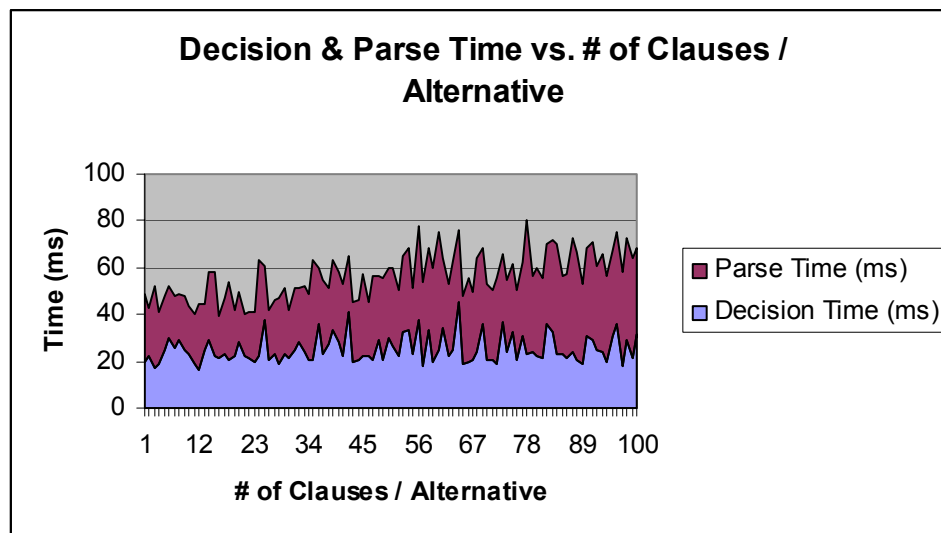


Figure 5.4: Experiment #4 Decision and Parsing Time vs. # of *Clauses / Alternative*

Again decision time does not change much with the increase in *clauses*. Parse time however does slowly increase as the number of *clauses* reaches a 100.

## 5.2 Validation – Correctness: Decision Scenarios

The next set of experiments is geared toward verifying the accuracy of the decision system. To that effect we shall work through scenarios as we did in explaining the inner workings of the decision system. The *Alternative Set* will be <connect> from Figure 4.1, the application is a messaging service and the alternatives are a voice stream, a text stream and downloading voice. The four scenarios will focus on a few environment object definitions {“security”, “energy”, CPU”, “BW”} in order to keep it simple enough for human verification. The scenarios will include competing relations between environment objects, similar relations between environment objects, and both types of relations.

The first scenario relates competing interests between the environment objects. As was done before, Fidelity will be modeled as an environment object since this is not reflected with the *Active Interface* construct yet.

| <i>Alternative</i> | <i>Fidelity</i> | <i>Security</i> | <i>Energy</i> |
|--------------------|-----------------|-----------------|---------------|
| Voicestream        | 4               | 4               | 2             |
| Textstream         | 1               | 8               | 9             |
| Download           | 6               | 8               | 3             |

Table 5.1: Scenario #1 *Programmer Comparator*

In the first scenario the programmer comparator reflects the fact that the fidelity of a voice stream is slightly less than downloading voice since real time playback constraints are the highest priority for voice streams which may inversely affect quality. Text streams although reflecting the content disseminated in the voice stream equivalents, lose qualities such as a speaker's tone which are important in expression. Both text stream and download voice options have equivalent security structures as they both use block cipher encryption strategies. Voice stream on the other hand uses a stream cipher so that bits can be dropped on the floor without having to re-request packets. Stream ciphers are inherently less secure than block ciphers as they do not benefit from block chaining. Text streams use the least amount of energy, while voice streams and downloaded voices are fairly close, with downloaded voices winning slightly as the system is free to schedule the download optimizing for bandwidth since there is no real time playback requirement.

| <i>User Comparator</i> |                 |               |
|------------------------|-----------------|---------------|
| <b>Fidelity</b>        | <b>Security</b> | <b>Energy</b> |
| {critical}             | {critical}      | {critical}    |

Table 5.2: Scenario #1 *User Comparator*

The user ratings for all *environment objects* are assumed to be critical in this case to ensure simplicity.

| <i>Environment Objects</i> |                 |               |
|----------------------------|-----------------|---------------|
| <b>Fidelity</b>            | <b>Security</b> | <b>Energy</b> |
| 33.33                      | 33.33           | 33.33         |

Table 5.3: Scenario #1 Point Allocations based on *User Comparator*

| <i>Correlations</i> |                 |                 |               |
|---------------------|-----------------|-----------------|---------------|
|                     | <b>Fidelity</b> | <b>Security</b> | <b>Energy</b> |
| <b>Fidelity</b>     | 0.00            | 0.00            | 0.00          |
| <b>Security</b>     | 0.00            | 0.00            | -0.80         |
| <b>Energy</b>       | 0.00            | -0.80           | 0.00          |

Table 5.4: Scenario #1 *Correlation Matrix* with 3 Sample *Environment Objects*

The *correlation* matrix reflects that security and energy have high inverse *correlation* or in other words, high security levels usually involve higher computation requirements thus inversely affecting energy levels.

| <i>Alternative</i> | <i>Fidelity</i> | <i>Security</i> | <i>Energy</i> |              |
|--------------------|-----------------|-----------------|---------------|--------------|
|                    | <b>33.33</b>    | <b>6.67</b>     | <b>6.67</b>   | <b>Total</b> |
| <b>Voicestream</b> | 12.12           | 1.33            | 0.95          | 14.41        |
| <b>Textstream</b>  | 3.03            | 2.67            | 4.29          | 9.98         |
| <b>Download</b>    | 18.18           | 2.67            | 1.43          | 22.28        |

Table 5.5: Scenario #1 Subdivision of *Environment Object* Points to *Alternatives*

Based on the *comparator* and *correlation* matrix definitions, we arrive at download voice as the *best alternative*. The fidelity *environment object* dominates as the negative relations between security and energy, and high user priority in both take them out of contention. The choice makes sense as download voice as it mediates between the two

other extreme alternatives. If fidelity were not in question, text would win handily which again is as expected.

| <i>Alternative</i> | <i>Fidelity</i> | <i>CPU</i> | <i>Security</i> |
|--------------------|-----------------|------------|-----------------|
| Voicestream        | 4               | 3          | 4               |
| Textstream         | 1               | 10         | 8               |
| Download           | 6               | 3          | 8               |

Table 5.6: Scenario #2 *Programmer Comparator*

In the second scenario the programmer comparator the fidelity and security comparisons between alternatives remain the same. Now we have two environment objects which have a positive correlation with one another, “CPU” and “security.” A voice stream and a downloaded voice use the same amount of CPU in playback while a text stream has fairly low CPU utilization.

| <i>User Comparator</i> |            |                 |
|------------------------|------------|-----------------|
| <b>Fidelity</b>        | <b>CPU</b> | <b>Security</b> |
| {critical}             | {critical} | {critical}      |

Table 5.7: Scenario #2 *User Comparator*

Again all the *user comparator* values are all critical for simplification purposes.

| <i>Environment Objects</i> |            |                 |
|----------------------------|------------|-----------------|
| <b>Fidelity</b>            | <b>CPU</b> | <b>Security</b> |
| 33.33                      | 33.33      | 33.33           |

Table 5.8: Scenario #2 Point Allocations based on *User Comparator*

| <i>Correlations</i> |                 |            |                 |
|---------------------|-----------------|------------|-----------------|
|                     | <b>Fidelity</b> | <b>CPU</b> | <b>Security</b> |
| <b>Fidelity</b>     | 0.00            | 0.00       | 0.00            |
| <b>CPU</b>          | 0.00            | 0.00       | 0.80            |
| <b>Security</b>     | 0.00            | 0.80       | 0.00            |

Table 5.9: Scenario #2 *Correlation Matrix* with 3 Sample *Environment Objects*

CPU utilization and security have a high correlation as the most secure protocols require a considerable amount of CPU.

| <i>Alternative</i> | <i>Fidelity</i> | <i>CPU</i>   | <i>Security</i> |              |
|--------------------|-----------------|--------------|-----------------|--------------|
|                    | <b>33.33</b>    | <b>60.00</b> | <b>60.00</b>    | <b>Total</b> |
| <b>Voicestream</b> | 12.12           | 11.25        | 12.00           | 35.37        |
| <b>Textstream</b>  | 3.03            | 37.50        | 24.00           | 64.53        |
| <b>Download</b>    | 18.18           | 11.25        | 24.00           | 53.43        |

Table 5.10: Scenario #2 Subdivision of *Environment Object* Points to *Alternatives*

With highly *correlated environment objects*, text stream wins since it has the least CPU utilization by far and is one of the more secure *alternatives*. Download voice is a close second which also makes sense, as it performed well with all *environment objects*. If the degree of correlation was reduced between CPU and security, download voice alternative would win as both CPU and security would become relatively less important to the *Alternative Set*.

### 5.3 Sample Application: Modified Pooka Java Mail Client and JAMES Server

Pooka is an email client written in Java, using the Javamail API. It supports email through the IMAP (connected and disconnected) and POP3 protocols. Outgoing mail is sent using SMTP. To demonstrate the feature set of the programming framework explored so far, we have expanded Pooka into a general messaging service with support for text, images and voice. In order to fit the messaging service into the email structure of Pooka, we encoded each of the messages as MIME attachments. The Pooka client was changed to optimize email access and remove unnecessary, beta, and buggy features such as the message preview window and toolbars. A screenshot of the working mail client can be viewed in Figure 5.5.



Figure 5.5: Pooka Mail Client Displaying a Message of Type Image

Apache JAMES or the Java Apache Mail Enterprise Server is a fully pure Java SMTP, POP3, and IMAP Mail server and NNTP News server. The mail server software although not very stable (IMAP support is alpha for instance), is fairly modular and easy to interface with and augment. James is also a *mail application platform*. There is a built in Java API for adding Java code to process emails called the mailet API. A mailet can generate an automatic reply, update a database, prevent spam, build a message archive, or in our case determine the alternative type for mail attachments taking into consideration varying email client conditions. A matcher service determines whether a mailet should

process a given email in the server. The mailet and matcher interface makes a call to our decision system which returns which adaptation to perform. The mailet interface then changes the MIME attachment to the email message based on the adaptation specified by the decision system. The protocol between the modified Pooka client and the JAMES mail server was the connected IMAP service. The modified client actually has no adaptation supported added to it. The client device reports to the server device what its current environment physical layer values are through the *Agent* interfaces specified earlier in Chapter 4, and the server performs the necessary adaptations based on these values. This ensures that the client only downloads *alternatives* that are suited for its environment and avoids downloading all the *alternatives* and deciding once the messages are on the client.

#### 5.4 Validation – Application Adaptation vs. Standard Application Behavior

Figure 5.6 displays the modified Pooka mail client downloading voice message files from the modified JAMES mail server. The voice file was a 19MB mp3 and bandwidth was varied from just over 57.6Kbps to 100Mbps. The same voice file was repeatedly downloaded 1000 times with varying bandwidth conditions. We used a bandwidth throttle script built into Linux called “tc” or “traffic controller”. This system utility can be used to change the available bandwidth perceived on a network connection by changing network packet queue sizes very much like traffic shapers built into firewall packages. A

pseudo random number representing available bandwidth was generated by the bandwidth Agent which was then used to call the “tc” scripts and reported to the decision system.

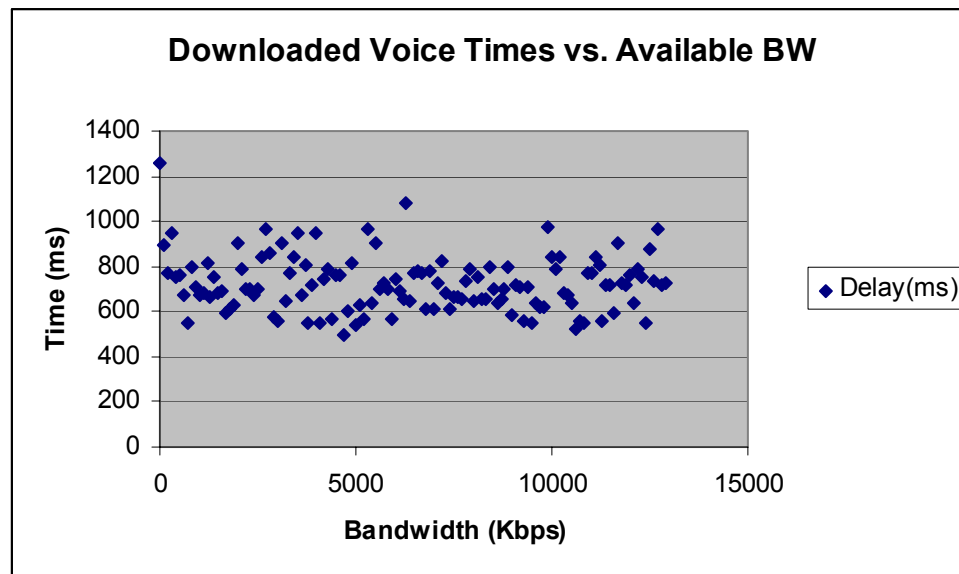


Figure 5.6: Experiment #5 Pooka Mail Client Voice Message Download Times

Figure 5.7 displays voice messages adapted on the server-side. The adaptation type used here was a multi-fidelity adaptation varying the fidelity of the voice audio. The *Content Adaptation Pipeline*[Bagrodia03] was used to create 10 separate fidelity values of the same voice message. The *Content Adaptation Pipeline*[Bagrodia03] uses the Java Media Framework and data object meta-data exported in XML documents to perform data content adaptation. Since the *Active Interface* feature was not available, these XML meta-data descriptions were hand generated and the adapted data objects were cached. Adaptation was performed by changing the voice bit rate and adding low pass and high

pass filters to remove noise. The adapted voice messages ranged from 3.7MB to the completely un-adapted 19 MB message. *Clauses* were not added for each of the *alternatives*, in order to allow the decision system to choose from the whole range of provided in the *Alternative Set*. Figure 5.7 displays a much cleaner and more uniform download behavior than the un-adapted downloads found in Figure 5.6. The overall performance gain with adaptation is well more than 50% improved in some cases. With adaptation the modified Pooka mail client is allowed to expect far more consistent message accesses even with vastly varying bandwidth conditions.

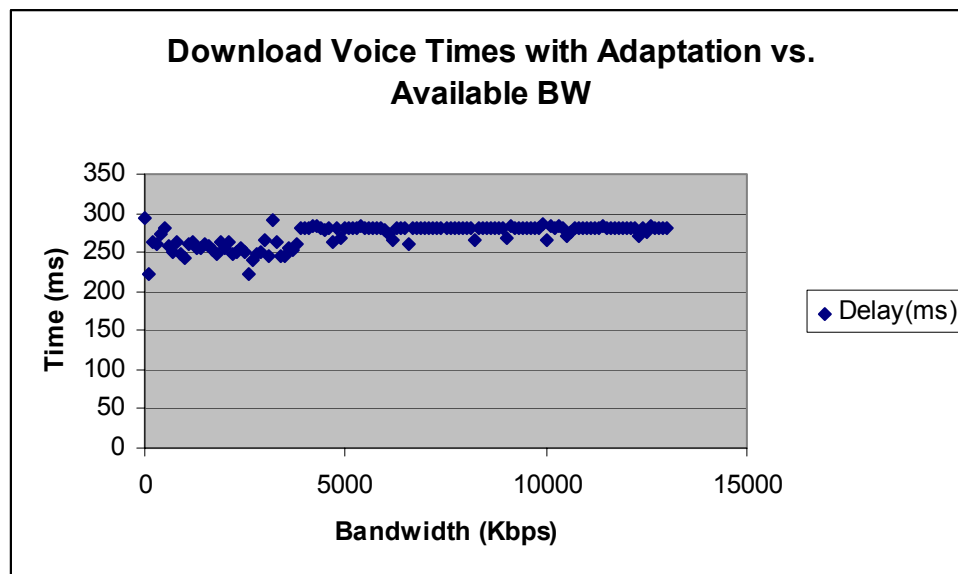


Figure 5.7: Experiment #6 Pooka Mail Client Voice Message Download Times with Server-side Adaptation

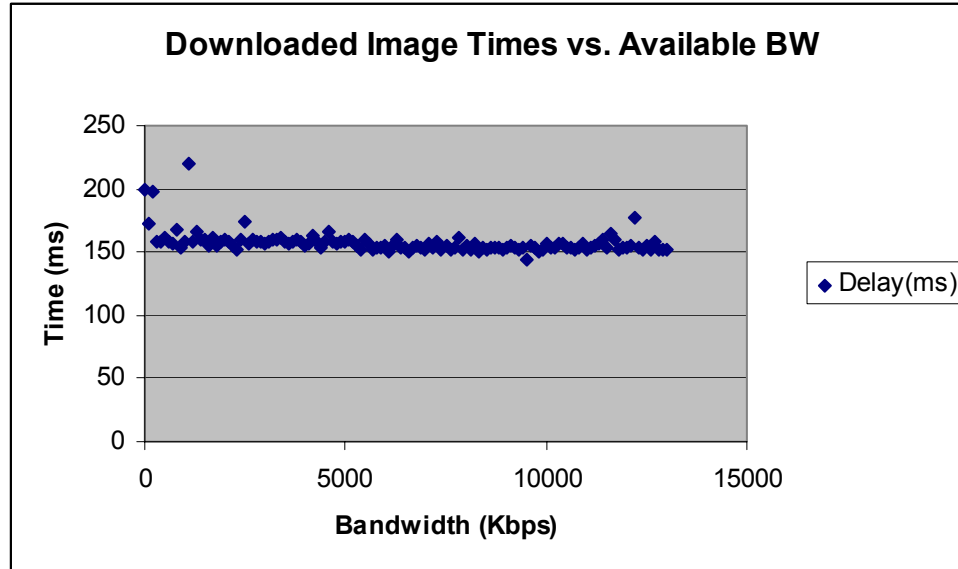


Figure 5.8: Experiment #7 Pooka Mail Client Image Message Download Times

Experiment #7 documented in Figure 5.8 displays an image message downloaded over various bandwidth conditions. A single jpeg image of the size 5.2 MB (displayed earlier in Figure 5.5) was repeatedly downloaded a 1000 times. The download behavior is far more consistent for the un-adapted client and server as compared to the voice message download behavior, because of the image's relatively smaller size. Figure 5.9, displays the modified Pooka client download times with image adaptations. The adaptations were again *multi-fidelity*[Narayanan02] adaptations which were created by the *Content Adaptation Pipeline*[Bagrodia03] and then cached. The 10 adapted image messages, created by the *Content Adaptation Pipeline*[Bagrodia03], ranged in size from 119KB to the un-adapted 5.2MB original image. Although there are a few outliers in Figure 5.9, these can be attributed to inconsistencies in network delivery times and Java object

decoding times. Again there is over a factor of 7 gain in using the adaptation strategy we have proposed.

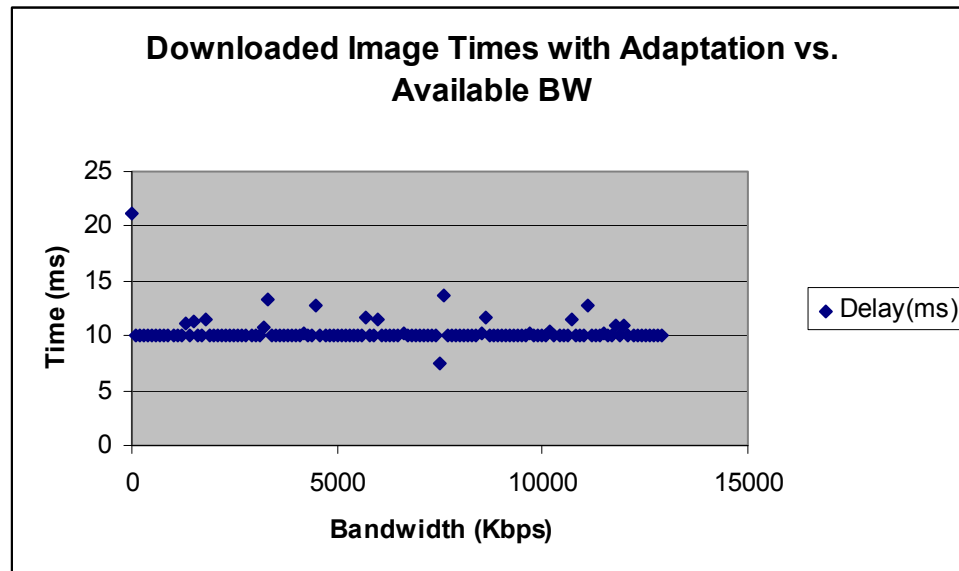


Figure 5.9: Experiment #8 Pooka Mail Client Image Message Download Times with Server-side Adaptation

In order to display capture an *Alternative Set* with different types of alternatives, we took the voice message file used so far and created a text message transcript. As the *Content Adaptation Pipeline*[Bagrodia03] does not perform speech to text at this time, the written transcript was created by hand. The text message created from this transcript was 17KB.

Figure 5.10 displays the performance of the mail service with an *Alternative Set* comprised of 2 *multi-fidelity* adaptations of the voice message (5.3MB, 19MB) in an *alternative* and an entirely separate the text message *alternative* as described above.

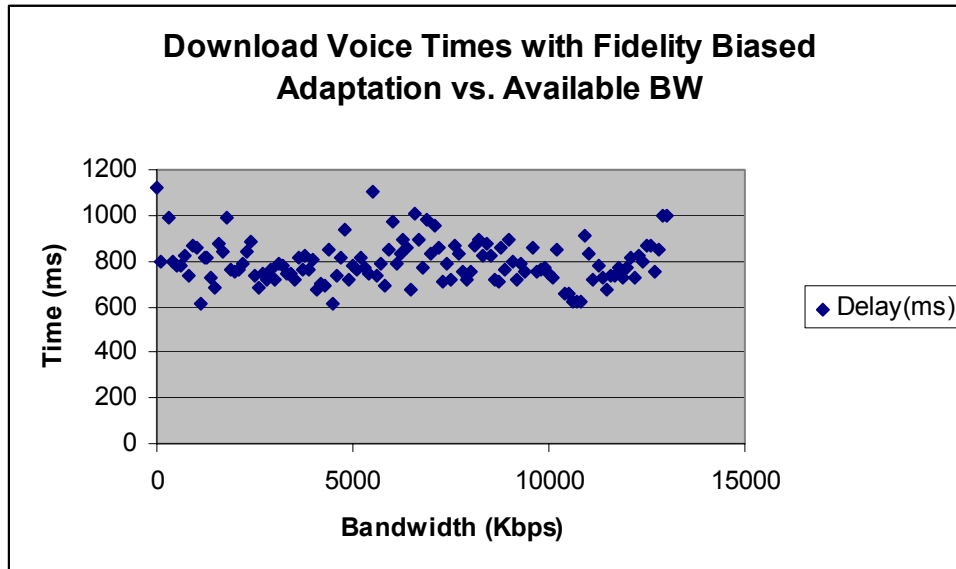


Figure 5.10: Experiment #9 Fidelity Biased Pooka Mail Client Voice Download Times with Server-side Adaptation

There is a high bias in Experiment #9 for voice message fidelity in order to capture the worst case scenario for delivering the adapted messages. The user comparator, which relates user preferences regarding environment objects, had an 8/1.7 ratio between fidelity and bandwidth. The two voice message alternatives had a 2/1 ratio in fidelity while the ratio between the un-adapted voice message (19MB) and the text representation of that message was set to 10/1. As can be expected, the decision system nearly always selected the high fidelity representations, leading to high download times.

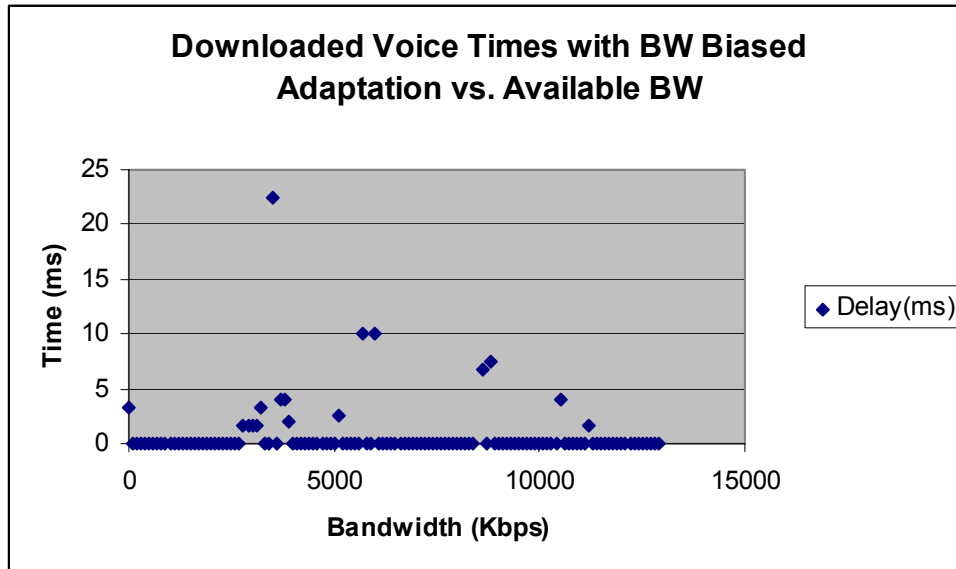


Figure 5.11: Experiment #10 Bandwidth Biased Pooka Mail Client Voice Download Times with Server-side Adaptation

Figure 5.11 displays the best case scenario for the adaptation scenario with the user placing a higher priority for minimizing bandwidth usage than data fidelity. In Experiment #10 the ratio from Experiment #9 was reversed 8/1.7 in favor of bandwidth over fidelity. As can be seen the decision system predominantly chose the lower fidelity text representation. The outliers in the graph can again be explained by Java object decode, inconsistent stream parsing delays and variations in network delay.

The power of the Alternative Set representation is clear from the experimental findings provided. The greatest testament to its success however is the fact that the entire set of library functions, parsing, decision-making and agent utilities comes to merely 51,342

lines of code and that adding adaptation support to the extensive modified Pooka mail client and the modified JAMES mail server code bases was completed in under 8 hours.

## Related Work

An application in Odyssey[Noble97] starts with registering triggers with a Viceroy. When a change in environment is detected, the Viceroy sends an up-call to the application. The Odyssey[Noble97] approach requires the application to request via system calls a distillation for data objects in the file system. As discussed in the introductory chapters, Odyssey[Noble97] is an infrastructure service and as such can only provide external adaptation and service support. Our approach focuses on internal application behavior and fundamental exposure of environmental characteristics into the application environment. Our approach mainly tackles the choice of adaptation strategies, the representation of these strategies in application objects. Our approach allows the application to change the algorithm used, the data type of data objects and the representation of both internal and external application interfaces. This is a difficult paradigm to achieve with the base Odyssey[Noble97] design.

Multi-fidelity Odyssey[Narayanan02] is an extension to the original Odyssey[Noble97] work which is geared toward the development of *multi-fidelity algorithms* which when provided *tunable* parameters[Narayanan02] adapt their behavior according to those parameters. *Multi-fidelity algorithms* are a novel idea and a necessity in algorithmic development where programmers are aware of pervasive environments with drastic resources changes over short time intervals. The short coming of the treatment provided

is the mainly in the view of resources as flat metrics or values. The environment model prescribed requires a very static definition of environment and does not allow programmers a rich and extensible access into the application's environment. The predictive resource system that underlies the Odyssey[Noble97] engine makes choices regarding adaptations autonomously and without any programmer or user input. This approach reduces the burden on the application developer however in doing so alienates the one entity firmly aware of the problem domain and adaptation characteristics. Another problem with an AI driven adaptation strategy is the sheer number and variety of contexts and possible adaptations forced by a pervasive computing domain. Unfortunately the size of the domain severely hinders the possibility of training artificial intelligence based systems. Additionally this research is limited to changing fidelity values which is one of the many possible ways an application programmer can facilitate a change application behavior. The fascination with fidelity driven approaches has been motivated by infrastructure services (of which Odyssey[Noble97] is one) and the multimedia application domain. Although these appear to be very important domains based on the types applications users make use of today, a programming framework must be extensible enough to tackle general computation with new application trends and user behaviors as they arise in the immediate future.

BARWAN[Brewer98] and iMASH[Bagrodia03] are typically suited for client-server type applications. They position a proxy layer between both entities which snoops the transactions between clients and servers. This layer monitors the environment and

transcodes multimedia data objects that pass by based on the availability of resources. For the proxy to successfully intercept data objects it needs to understand the protocol followed by the application. Thus application level computation must extend to the proxy servers. This work is different from ours as there is no programmatic control of the changing behavior of applications. Like the base Odyssey[Noble97] strategy, these approaches are suited to distilling data objects and cannot effectively capture other application behavioral changes. As was suggested in the introductory sections, it is possible to extend these infrastructure services by exposing programming constructs through Active Interfaces. These services will then have a view into an application's data and interface representations which when combined with user and application fidelity requirements, can effectively determine an adaptation strategy.

one.world[Grimm02] project at University of Washington discusses, as we have, the notion of exposing environments into the application domain. Environments are containers for tuples which represent data and service relations in a Linda like mailbox paradigm. Services such as fault tolerance can then be written in terms of tuples allowing for composable services. The problem with one.world[Grimm02] is its close proximity of infrastructure services to the application development process. The process of debugging or even design is considerably complicated when the code base for an infrastructure service is fundamentally involved in the development process. Furthermore, one.world[Grimm02] forces the tuple-space data representation semantics on the developer. This may or may not be optimal for the problems the application is suited to

tackle, not to mention the sheer complexity that is added to the development process. The worst problem however is finding an effective implementation of the Linda tuple space strategy. Although the Linda model is one of the cleanest computational models for distributed and parallel computation, application programmers have been trying for years to create an efficient representation of its primitives. Projects like Jini[Waldo99], which is also based on the Linda model, have struggled to meet scalability requirements which is crucial for pervasive environments where although device capabilities may be limited, device proliferation more than makes up for it.

PIMA[Banavar02], a model suggested in a challenge paper on system design in pervasive computing, talks about designing applications using an abstract notion of environment. The mapping of the abstract environment to the physical one happens at runtime. Thus a programmer wishing to output something, would simply use an abstract function *output()*. During runtime, depending on the environment in which the application is running, it could be mapped to GUI or console printing or even audio output. Thus the behavior of the application would always be suitable for the current context. The decision behind selecting the proper semantics of abstract calls is made by the system and outside the realm of the application. Although the model is a refreshing approach to tackling device heterogeneity, the application writer becomes alienated from the interface description or the application object representations to the user resulting in fairly limited control over the behavior of the application. Additionally it is not clear if such a programming paradigm is even feasible as it implies that data storage semantics and data

representation semantics can be easily separable and seamlessly remain disjoint until application instantiation. For instance one must create a message service, as we have done in demonstrating our programming framework, without knowing what message characteristics should be maintained to facilitate the eventual data representation to the user. Such an approach is conceivable, however it is not immediately obvious how it can succeed without a fairly robust representation of application meta-data, abstract object representations and efficient and possibly lossless application object transcoding.

The PCL[Ensink02] project at UIUC is geared toward providing a programming framework (programming model, programming language, compiler and runtime environment) that enables programmers to design, develop, and optimize the performance of adaptive distributed applications. The type of adaptations PCL[Ensink02] is intended to handle includes internal changes to applications such as parameter settings, algorithms, and data representations, and external changes such as task migration. One example provided in their paper describes a long running parallel code for stochastic optimization which adapts to processor availability by creating parallel tasks dynamically. Their framework is used to balance the degree parallelization in the face of reduced efficiency and potentially worse convergence behavior. The approach is implemented by adding adaptation objects to each application object. These adaptation objects then adapt their respective application objects asynchronously based on operators and behaviors the application programmer provides. The purpose of this type of an approach is to reduce development complexity as the base application remains unchanged and unaware of

adaptation. However this approach produces an adaptive application which is in fact considerably more complicated as base classes and adaptor classes manipulate shared data objects in and out of synchronization primitives. Determining program flow and debugging can become fairly complicated with this approach, necessitating that the debugger must be as familiar with the base application as the original application developer. This can undoubtedly lead to the original application programmer having to develop both the base application and the adaptation classes, which would be much more complicated than envisioning an adaptive application from the start. This approach is fundamentally opposite to the approach we have taken. Our focus has been to reflect environmental change to the application, allowing the programmer to facilitate adaptation with the use of rich programming primitives. In contrast the PCL[Ensink02] approach takes the opposite approach by reflecting application behavior outside of the application to the environment. Although this approach seems better suited to facilitating adaptation than infrastructure services discussed earlier, the focus of the work has been on adaptive parallel and distributed computation and therefore has not been extended out of the realm of parallelizing or distributing tasks.

The SoNS[Saif03] project at MIT describes an infrastructure system approach to adaptation which utilizes application exported primitives to tailor adaptation. Instead of determining alternative behavior within the programming language framework as we have done, SoNS[Saif03] exports clausal parameters for alternative satisfaction to a service discovery interface. As a user then migrates from one environment to another, the

service discovery backbone forwards the application data from node to node within the network, basically composing and performing services as required. The application in the SoNS[Saif03] model does not adapt its internal behavior, rather the environment is asked to adapt on its behalf or at least the environment is responsible to find where the application can suitably run. In other words, adaptation in SoNS[Saif03] is transformed into a resource discovery problem allowing applications to remain static and unaware of environmental conditions. This requires a fairly extensive infrastructure system with intimate knowledge of application behavior as data representations and interfaces will have to be adapted without application knowledge. This introduces latencies and inefficiency in general as the application maintains and performs operations on state which may have to undergo complete transcoding in order to meet the environmental requirements of nodes. The simple video application which they have built does not perform transcoding, so what in fact happens now is a decision on where an application should migrate with dynamic resource discovery. SoNS[Saif03] in fact is more similar to the distributed Content Adaptation Pipeline in the IMASH[Bagrodia03] infrastructure service project and service discovery protocols in general, than the programming framework we propose.

## Chapter 6 Conclusions

The contributions claimed from this thesis are two-fold: firstly identifying the emerging trend of applications which run in environments that change drastically and frequently. This will be the norm for applications rather than the exception, which justifies the need to search for models wherein applications can modify their behavior with external environment changes. The second contribution is identifying two key features for achieving this end: the exposure of the environment to the application and providing primitives for changing application behavior. We have focused our vision on a programming framework and require that the programmer explicitly envision a pervasive application when committing to a programmatic design. This explicit programming language approach does not place the entire burden of application adaptation in the face of pervasive and mobile computation on the programmer. Rather, it provides a rich set of programming language primitives and support infrastructure to facilitate application adaptation.

Applications are modeled as a synergy of traditional code blocks and *Alternative Sets*. An application can change its behavior in four ways: algorithmic change, multi-fidelity algorithms, interface change and data-type change. The *Alternative Set* programming construct handles the first two types of application behavioral changes while the *Active Interface* construct detailed tackles the change in interfaces and data-types specifically. In

order to provide a system extensible enough to tackle a pervasive programming environment, we have proposed a layered *environment object* view of the application environment. This layered approach allows for a program to fundamentally change its view of the environment during execution while still providing a feature rich API for exporting environmental conditions. This thesis also proposed and demonstrated a flexible parsing and runtime system which interprets environment object definitions and composes alternatives for adaptation dynamically and utilizes programmer and user input in making alternative choices. As demonstrated in the experiment section, the parsing and runtime systems are fast and provide a deterministic and accurate model for adaptation.

This work however is only the beginning for research in this field. There are several key issues which need to be detailed and explored including: Active Interfaces and an asynchronous application adaptation strategy; exposure of programming primitives and application meta-data to infrastructure services for the purposes of service-discovery and composition, and content adaptation; user trust in the face of infrastructure driven fidelity changes; security models in a system where environment state is easily accessible in the application domain; trust models between runtime agents and the decision system where incorrect or malicious environment information may drastically affect application behavior possibly resulting in failure. This is really is only the tip of the ice berg when it comes to pervasive and context aware computing. This thesis aims at showing the need for a new programming language framework and provides a basic implementation to validate the proposed framework.



## REFERENCES

- [Bagrodia03] R. Bagrodia, S. Bhattacharyya, F. Cheng, S. Gerding, G. Glazer, R. Guy, Z. Ji, J. Lin, T. Phan, E. Skow, M. Varshney, G. Zorpas, “iMASH: Interactive Mobile Application Session Handoff.” Proceedings of the First International Conference on Mobile Systems, Applications, and Services, May 2003.
- [Brewer98] E.A. Brewer, R.H. Katz, E. Amir, H. Balakrishnan, Y. Chawathe, A. Fox, S.D. Gribble, T. Hodes, G. Nguyen, V.N. Padmanabhan, M. Stemm, S. Seshan, and T. Henderson, “A Network Architecture for Heterogeneous Mobile Computing.” IEEE Personal Communications Magazine, October 1998.
- [Noble97] B.D. Noble, M. Satyanarayanan, D. Narayanan, J.E. Tilton, J. Flinn, and K.R. Walker, “Agile Application-Aware Adaptation for Mobility.” Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles, October 1997.
- [Narayanan02] D. Narayanan, “Operating System Support for Mobile Interactive Applications.” PhD thesis, CMU, August 2002.
- [Ensink02] B. Ensink, J. Stanley, and V. Adve, “Program Control Language: A Programming Language for Adaptive Distributed Applications.” Journal of Parallel and Distributed Computing, (accepted Nov. 2002).
- [Banavar02] G. Banavar, J. Beck, E. Gluzberg, J. Munson, J. Sussman, and D. Zukowski, “Challenges: An Application Model for Pervasive Computing.” Proceedings of the Sixth Annual International Conference on Mobile Computing and Networking, August 2000.
- [Saif03] U. Saif and J.M. Paluska, “Service-oriented Network Sockets.” Proceedings of the First International Conference on Mobile Systems, Applications, and Services, May 2003.
- [Grimm02] R. Grimm, “System Support for Pervasive Applications.” PhD thesis, University of Washington, 2002.
- [Hoare85] C.A.R. Hoare, *Communicating Sequential Processes*. Prentice Hall International, 1985.

[Amano98] N. Amano and T. Watanabe, "LEAD: A Language for Dynamically Adaptable Applications." IEICE Trans on Fundamental of Electronics, Communications and Computer Science, Vol.E81-A N0.6, pp.992-1000, 1998.

[Amano99] N. Amano and T. Watanabe, "LEAD++: An Object-Oriented Reflective Language for Dynamically Adaptable Software Model." IEICE Trans on Fundamental of Electronics, Communications and Computer Science, Vol.E82-A N0.6, pp.1009-1016, 1999.

[Waldo99] J. Waldo, "The Jini Architecture for Network-centric Computing." Communications of the ACM, pp76-82, July 1999.